Обход графов в Ширину Breadth first search (BFS)

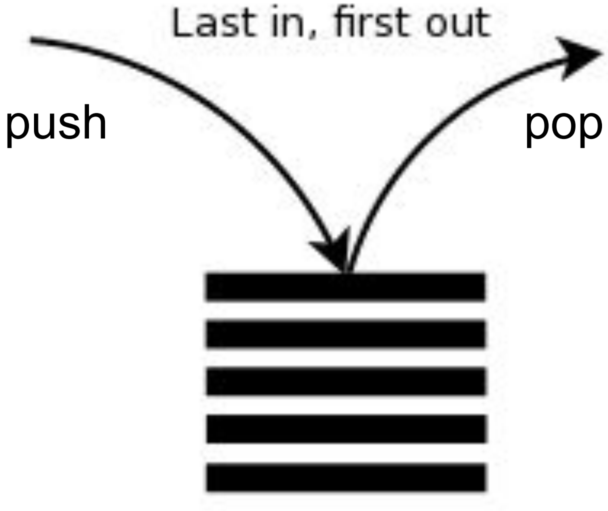
Рязанов Василий Владимирович

План лекции

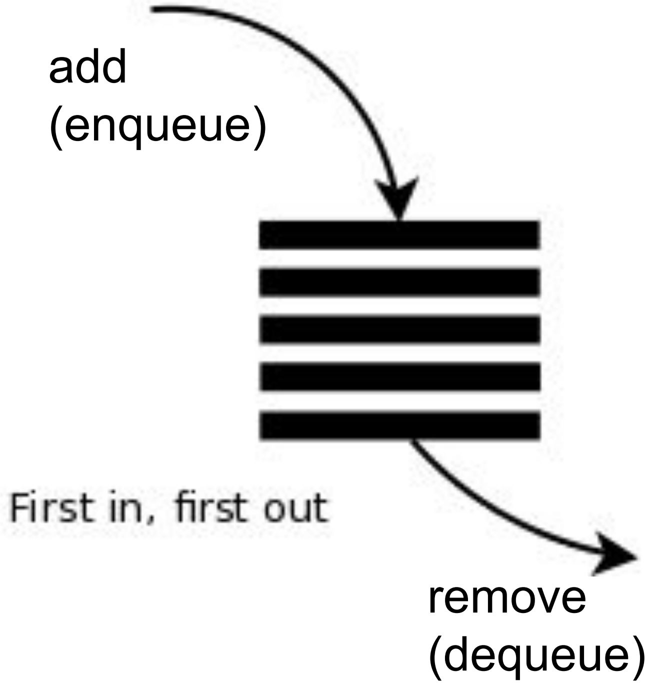
1) Очередь (структура обращения к данным)
2) Обход графов в Ширину
3) Приложения алгоритма

Очередь (и немного стек)

Stack (стек)



Queue (очередь)



Очередь в python

```python
# МЕДЛЕННАЯ реализация
queue = []
queue.append(5) # add
queue.append(10) # add
queue.append(7) # add
print(queue)
```

```
[5, 10, 7]
```

```python
x = queue.pop(0) # remove
print(x)
print(queue)
```

```
5
[10, 7]
```

```python
queue = list(range(100000000))
```

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

```
CPU times: user 325 ms, sys: 77
Wall time: 1.1 s
```

O(N) по времени!

```python
# не эффективная по памяти реализация
queue = list(range(100000000))
q_start = 0
```

```python
%%time
# remove превращается в 2 операции
# мы не удаляем элемент из памяти, а просто сдвигаемся
x = queue[q_start]
q_start += 1
```

```
CPU times: user 9 µs, sys: 10 µs, total: 19 µs
Wall time: 47 µs
```

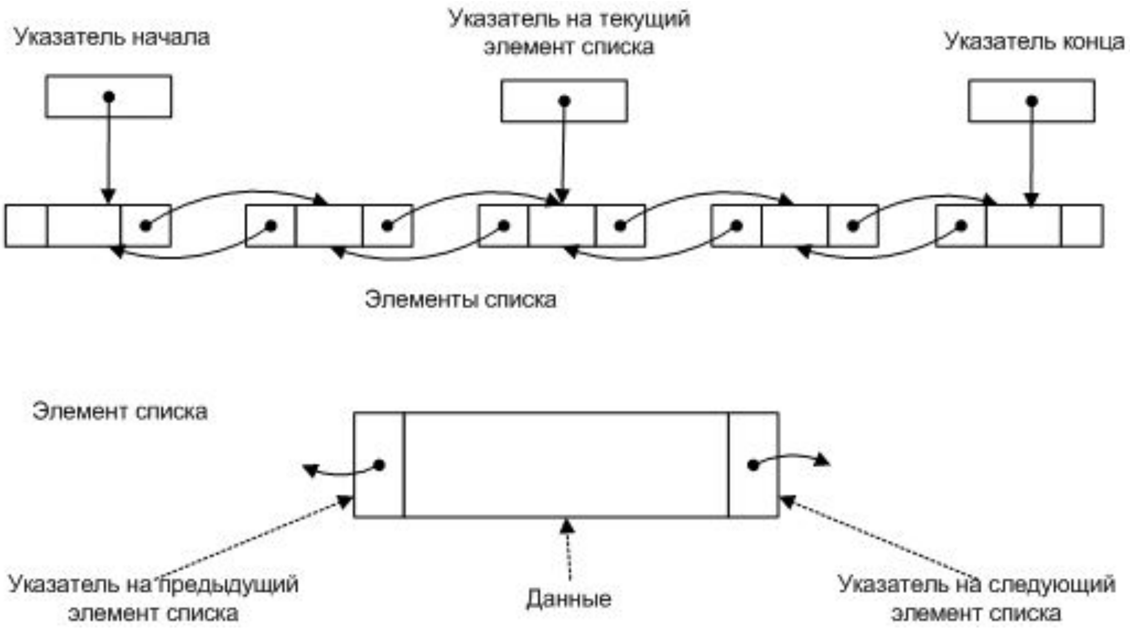
Память не освобождаем!

```python
# наиболее оптимальная реализация
from collections import deque
queue = deque(range(100000000))
queue.append(1) # add
```

```python
%%time
x = queue.popleft() # remove
```

```
CPU times: user 5 µs, sys: 1e+03 ns, total: 6 µs
Wall time: 17.2 µs
```

Очередь через двусвязный список

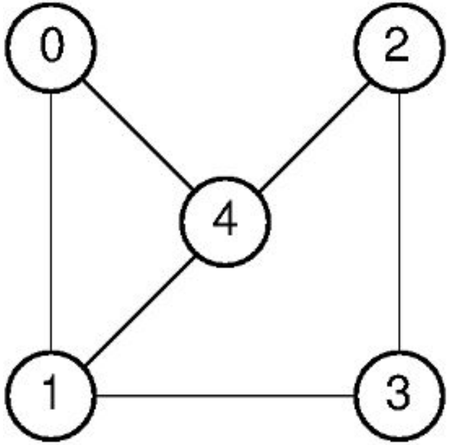


https://docs.python.org/3/library/collections.html#collections.deque
https://pythontips.com/2014/07/02/an-intro-to-deque-module/

Обход графа в ширину

Сегодня мы рассматриваем **невзвешенные** графы. Алгоритм работает и на ориентированных и на неориентированных графах.

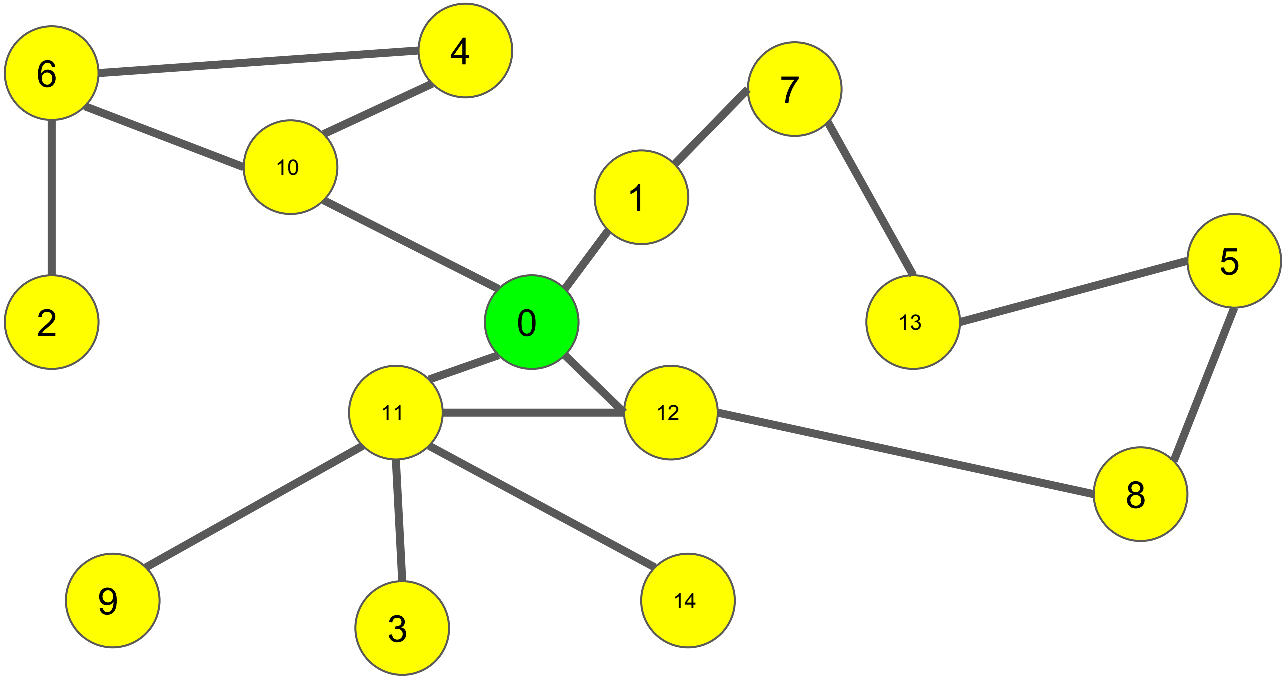
Граф будем хранить в виде списка смежности




```
{0: {1, 4},
 1: {0, 3, 4},
 2: {3, 4},
 3: {1, 2},
 4: {0, 1, 2}}
```

Обход графа в ширину

Очередь обхода: [0]

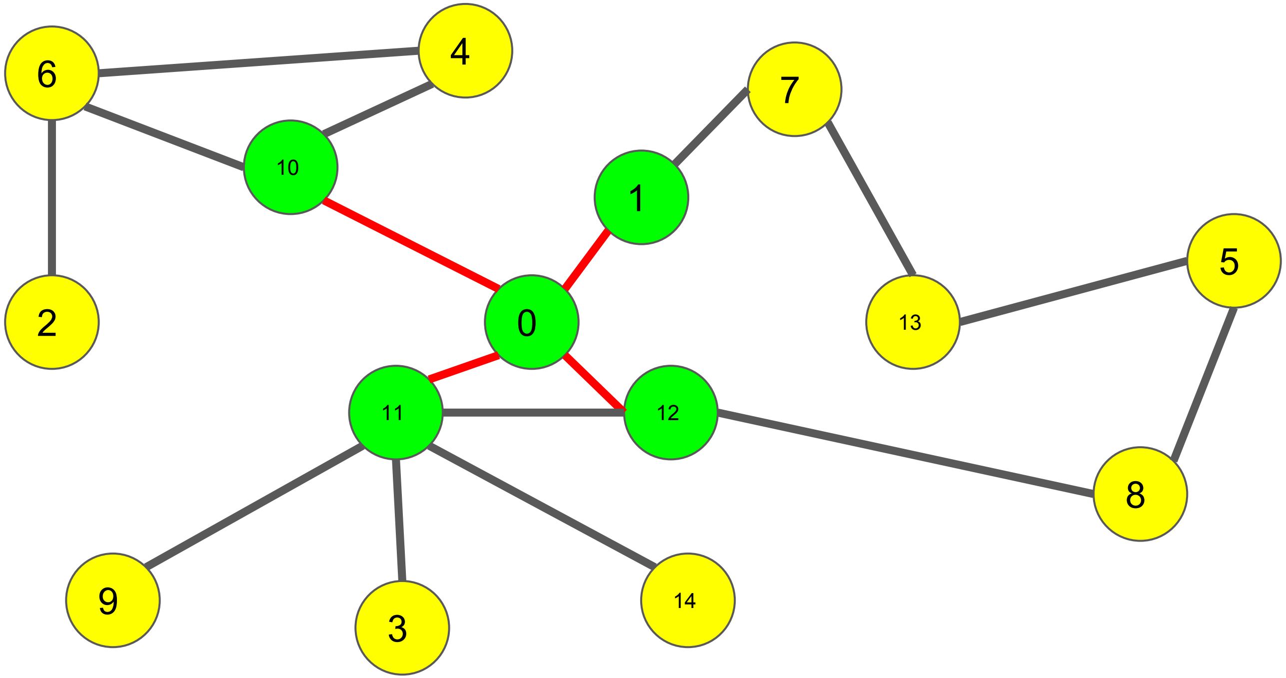


Расстояния от 0 вершины до остальных

0	**0**
1	**-**
2	**-**
3	**-**
4	**-**
5	**-**
6	**-**
7	**-**
8	**-**
9	**-**
10	**-**
11	**-**
12	**-**
13	**-**
14	**-**

Обход графа в ширину

Очередь обхода: [1, 12, 11, 10]

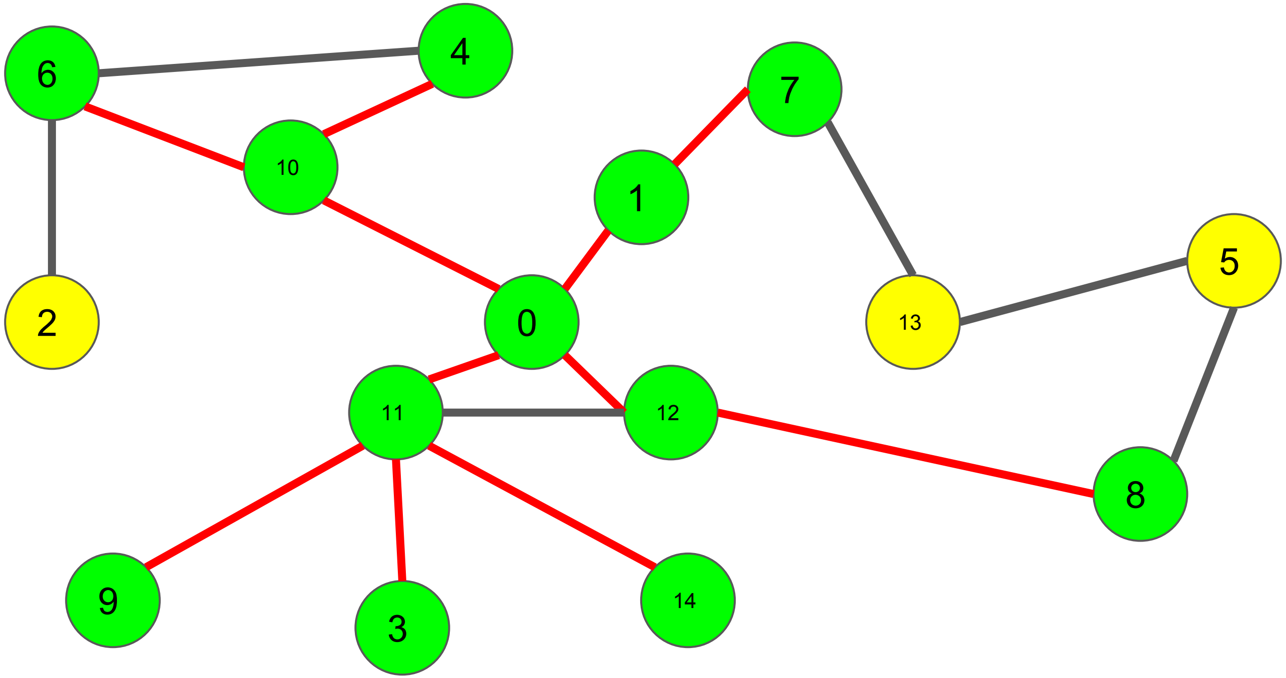


Расстояния от 0 вершины до остальных

0	**0**
1	**1**
2	**-**
3	**-**
4	**-**
5	**-**
6	**-**
7	**-**
8	**-**
9	**-**
10	**1**
11	**1**
12	**1**
13	**-**
14	**-**

Обход графа в ширину

Очередь обхода: [6, 4, 7, 8, 14, 3, 9]

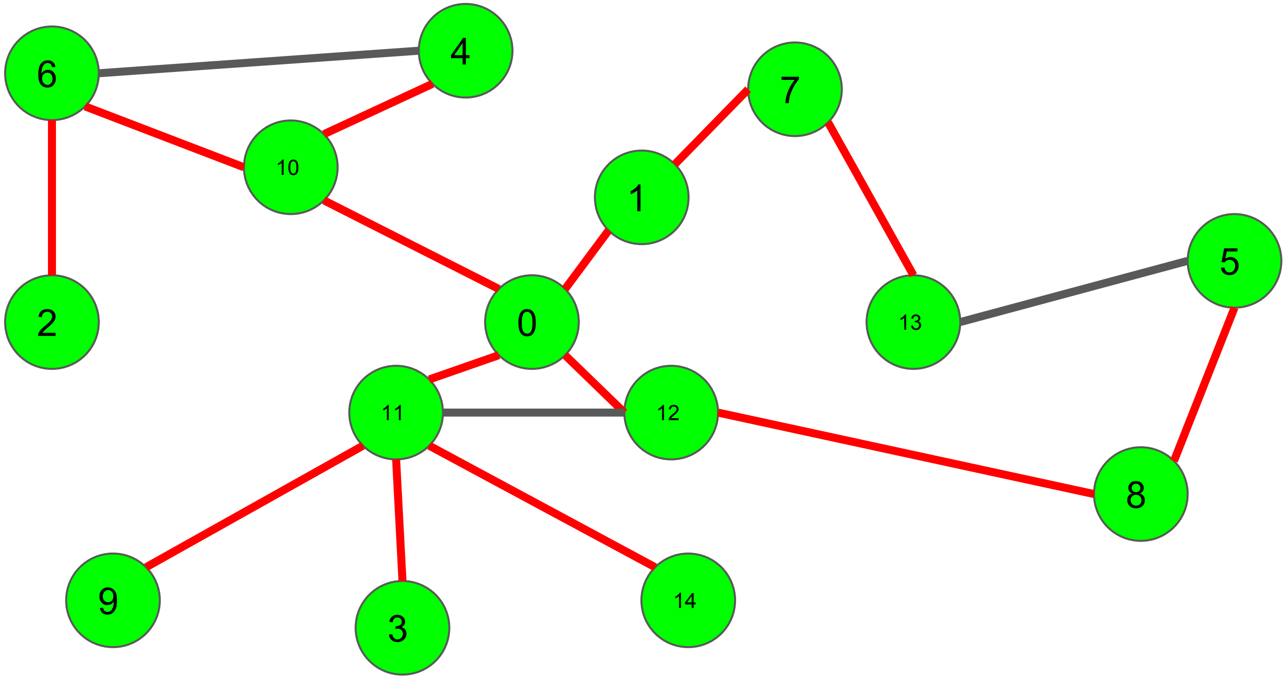


Расстояния от 0 вершины до остальных

0	**0**
1	**1**
2	**-**
3	**2**
4	**2**
5	**-**
6	**2**
7	**2**
8	**2**
9	**2**
10	**1**
11	**1**
12	**1**
13	**-**
14	**2**

Обход графа в ширину

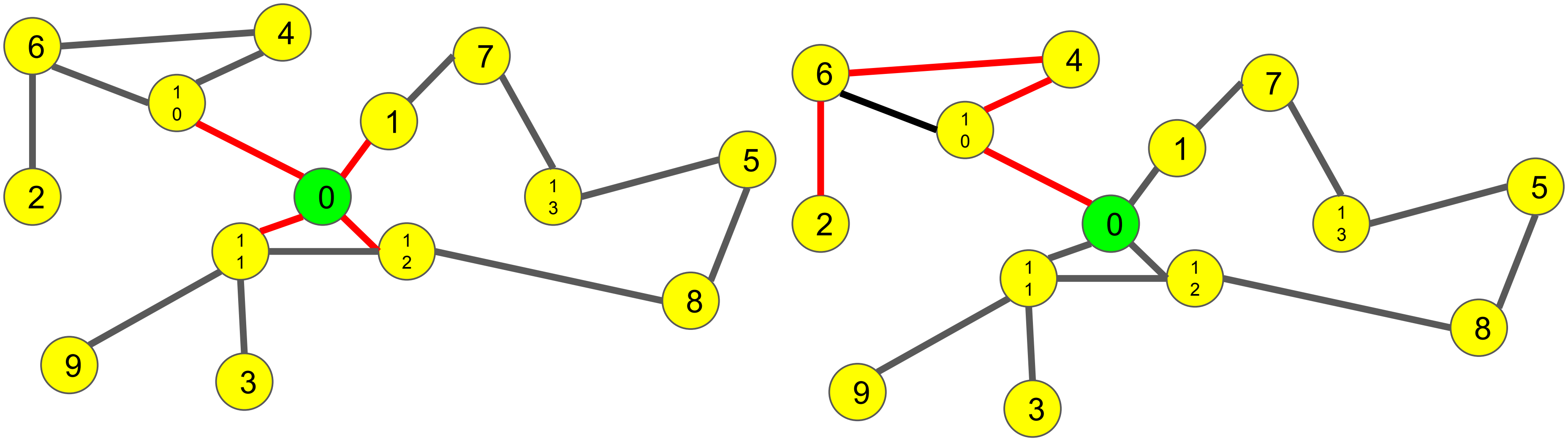
Очередь обхода: [2, 5, 13]



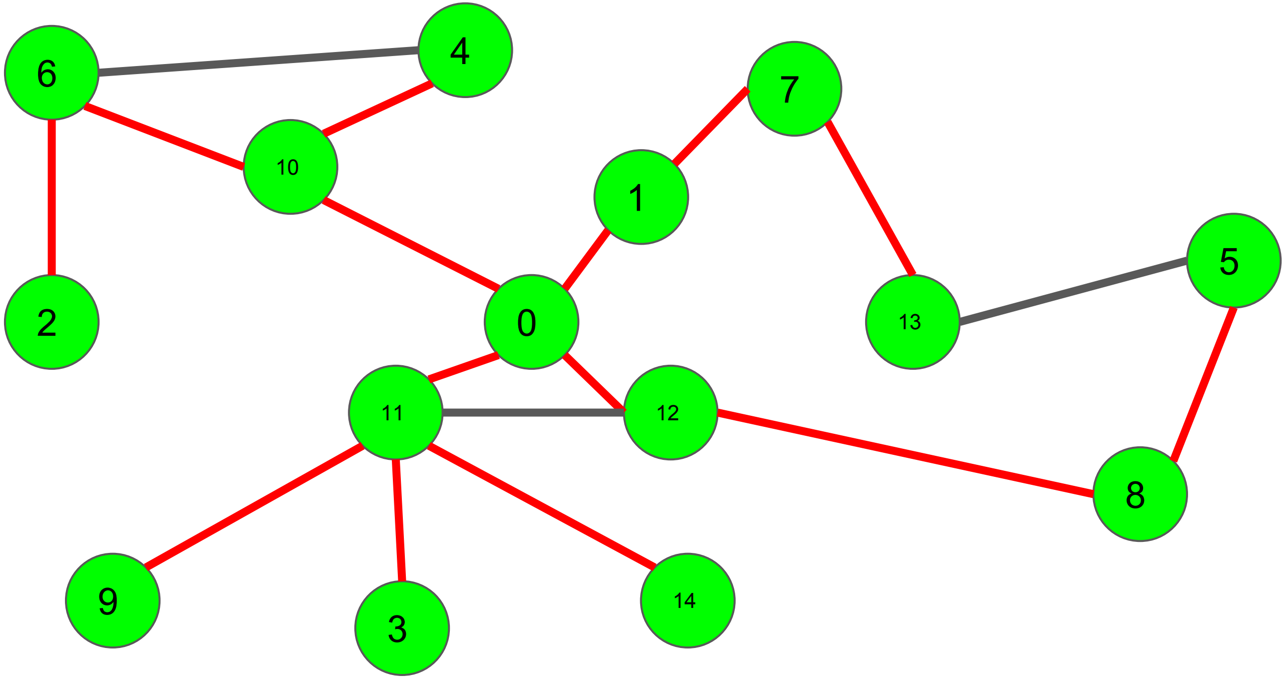
Расстояния от 0 вершины до остальных

0	**0**
1	**1**
2	**3**
3	**2**
4	**2**
5	**3**
6	**2**
7	**2**
8	**2**
9	**2**
10	**1**
11	**1**
12	**1**
13	**3**
14	**2**

Отличие обхода в ширину и в глубину



Обход графа в ширину



Расстояния от 0 вершины до остальных

0	**0**
1	**1**
2	**3**
3	**2**
4	**2**
5	**3**
6	**2**
7	**2**
8	**2**
9	**2**
10	**1**
11	**1**
12	**1**
13	**3**
14	**2**

Трудоемкость: O(N+M)

Обход в ширину - код

```
N, M = map(int, input().split()) # считываем кол-во вершин и кол-во ребер
```

```
graph = {i:set() for i in range(N)} # будем хранить в виде словаря с множествами
for i in range(M):
    v1, v2 = map(int, input().split()) # считываем ребро
    graph[v1].add(v2) # добавляем смежность двух вершин
    graph[v2].add(v1)
```

```
from collections import deque
distances = [None] * N # массив расстояний, по умолчанию неизвестны
start_vertex = 0 # начинаем с 0 вершины
distances[start_vertex] = 0 # расстояния до себя же равно 0
queue = deque([start_vertex]) # создаем очередь

while queue: # пока очередь не пуста
    cur_v = queue.popleft() # достаем первый элемент
    for neigh_v in graph[cur_v]: # проходим всех его соседей
        if distances[neigh_v] is None: # если сосед еще не посещен(=>расстояние None)
            distances[neigh_v] = distances[cur_v] + 1 # считаем расстояние
            queue.append(neigh_v) # добавляем в очередь чтобы проверить и его соседей
print(distances) # смотрим что получилось
```

Обход графа в ширину - приложения

- Выделение **компонент связности** в графе за O(n+m)
- Поиск **кратчайшего пути** в невзвешенном графе
- **Восстановление кратчайшего пути**
- Нахождение **кратчайшего цикла** в ориентированном невзвешенном графе
- Найти все **рёбра**, лежащие на каком-либо **кратчайшем пути** между заданной парой вершин (a,b)
- Найти все **вершины**, лежащие на каком-либо **кратчайшем пути** между заданной парой вершин (a,b)
- Найти **кратчайший чётный путь** в графе (т.е. путь чётной длины)
- *Бонус

Выделение компонент связности

1. Полагаем кол-во компонент связности равное 0
2. Начинаем обход в ширину из произвольной вершины
3. Когда обход завершается увеличиваем кол-во компонент связности на 1
4. Если остались еще непосещенные вершины, повторяем шаги 2-3, сохраняя при этом массив посещенных вершин *used*
5. Если все вершины посещены, завершаем. Время по-прежнему O(N+M)

Восстановление кратчайшего пути

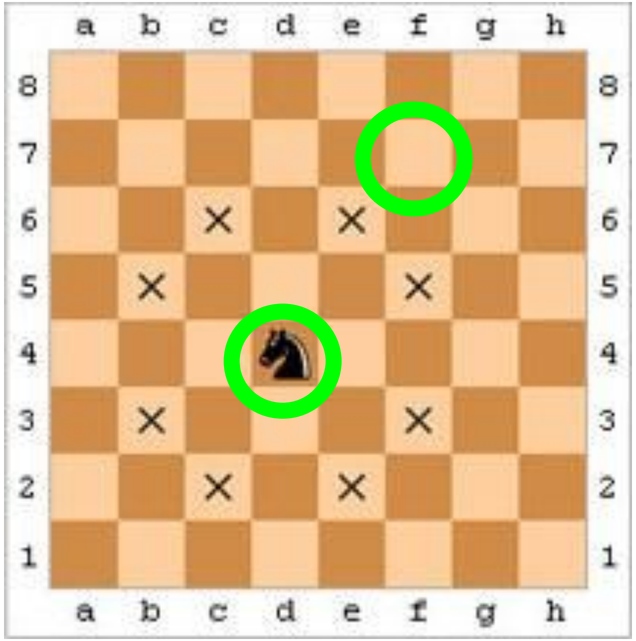
Совершаем обход в ширину с подсчетом расстояний. Для каждой вершины запоминаем предка

```
start_vertex = 0
end_vertex = 2

parents = [None] * N
distances = [None] * N
distances[start_vertex] = 0
queue = deque([start_vertex])

while queue:
    u = queue.popleft()
    for v in graph[u]:
        if distances[v] is None:
            distances[v] = distances[u] + 1
            parents[v] = u
            queue.append(v)
path = [end_vertex]
parent = parents[end_vertex]
while not parent is None:
    path.append(parent)
    parent = parents[parent]
print(path[::-1])
```

Восстановление траектории шахматного коня

По каким клеткам должен пройти конь, чтобы попасть из d4 в f2 наиболее быстро?

1) Сводим задачу к графу
2) Обход в ширину из одной точки в другую



Конь: создаем граф

```
letters = 'abcdefgh'
numbers = '12345678'
graph = dict()
graph = {l+n:set() for l in letters for n in numbers}

def add_edge(v1, v2):
    graph[v1].add(v2)
    graph[v2].add(v1)

for i in range(8):
    for j in range(8):
        v1 = letters[i] + numbers[j]
        v2 = ''
        if 0 <= i + 2 < 8 and 0 <= j + 1 < 8:
            v2 = letters[i+2] + numbers[j+1]
            add_edge(v1, v2)
        if 0 <= i - 2 < 8 and 0 <= j + 1 < 8:
            v2 = letters[i-2] + numbers[j+1]
            add_edge(v1, v2)
        if 0 <= i + 1 < 8 and 0 <= j + 2 < 8:
            v2 = letters[i+1] + numbers[j+2]
            add_edge(v1, v2)
        if 0 <= i - 1 < 8 and 0 <= j + 2 < 8:
            v2 = letters[i-1] + numbers[j+2]
            add_edge(v1, v2)
```

Выглядит граф как-то так

```
graph
```

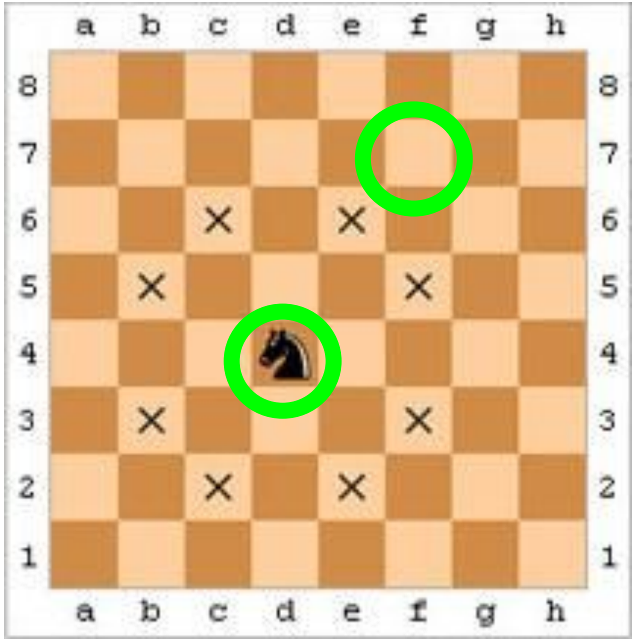
```
{'a1': {'b3', 'c2'},
 'a2': {'b4', 'c1', 'c3'},
 'a3': {'b1', 'b5', 'c2', 'c4'},
 'a4': {'b2', 'b6', 'c3', 'c5'},
 'a5': {'b3', 'b7', 'c4', 'c6'},
 'a6': {'b4', 'b8', 'c5', 'c7'},
 'a7': {'b5', 'c6', 'c8'},
 'a8': {'b6', 'c7'},
 'b1': {'a3', 'c3', 'd2'},
 'b2': {'a4', 'c4', 'd1', 'd3'},
 'b3': {'a1', 'a5', 'c1', 'c5', 'd2', 'd4'},
 'b4': {'a2', 'a6', 'c2', 'c6', 'd3', 'd5'},
 'b5': {'a3', 'a7', 'c3', 'c7', 'd4', 'd6'},
 'b6': {'a4', 'a8', 'c4', 'c8', 'd5', 'd7'},
 'b7': {'a5', 'c5', 'd6', 'd8'},
 'b8': {'a6', 'c6', 'd7'},
 'c1': {'a2', 'b3', 'd3', 'e2'},
 'c2': {'a1', 'a3', 'b4', 'd4', 'e1', 'e3'},
 'c3': {'a2', 'a4', 'b1', 'b5', 'd1', 'd5', 'e2', 'e4'},
 'c4': {'a3', 'a5', 'b2', 'b6', 'd2', 'd6', 'e3', 'e5'},
 'c5': {'a4', 'a6', 'b3', 'b7', 'd3', 'd7', 'e4', 'e6'},
 'c6': {'a5', 'a7', 'b4', 'b8', 'd4', 'd8', 'e5', 'e7'},
 'c7': {'a6', 'a8', 'b5', 'd5', 'e6', 'e8'},
 'c8': {'a7', 'b6', 'd6', 'e7'},
 'd1': {'b2', 'c3', 'e3', 'f2'},
 'd2': {'b1', 'b3', 'c4', 'e4', 'f1', 'f3'},
```

Восстановление траектории шахматного коня

Стандартный поиск в ширину: из d4 в f7

```
start_vertex = 'd4'
end_vertex = 'f7'

parents = {v: None for v in graph}
distances = {v: None for v in graph}

distances[start_vertex] = 0
queue = deque([start_vertex])

while queue:
    u = queue.popleft()
    for v in graph[u]:
        if distances[v] is None:
            distances[v] = distances[u] + 1
            parents[v] = u
            queue.append(v)
path = [end_vertex]
parent = parents[end_vertex]
while not parent is None:
    path.append(parent)
    parent = parents[parent]
print(path[::-1])
```

```
['d4', 'c6', 'd8', 'f7']
```



Нахождение кратчайшего цикла

1. Запускаем обход в ширину из каждой вершины
2. Как только пытаемся попасть в посещенную вершину - значит есть цикл
3. Запустив обход из каждой вершины выбираем кратчайший

Нахождение всех вершин на кратчайшем пути (a,b)

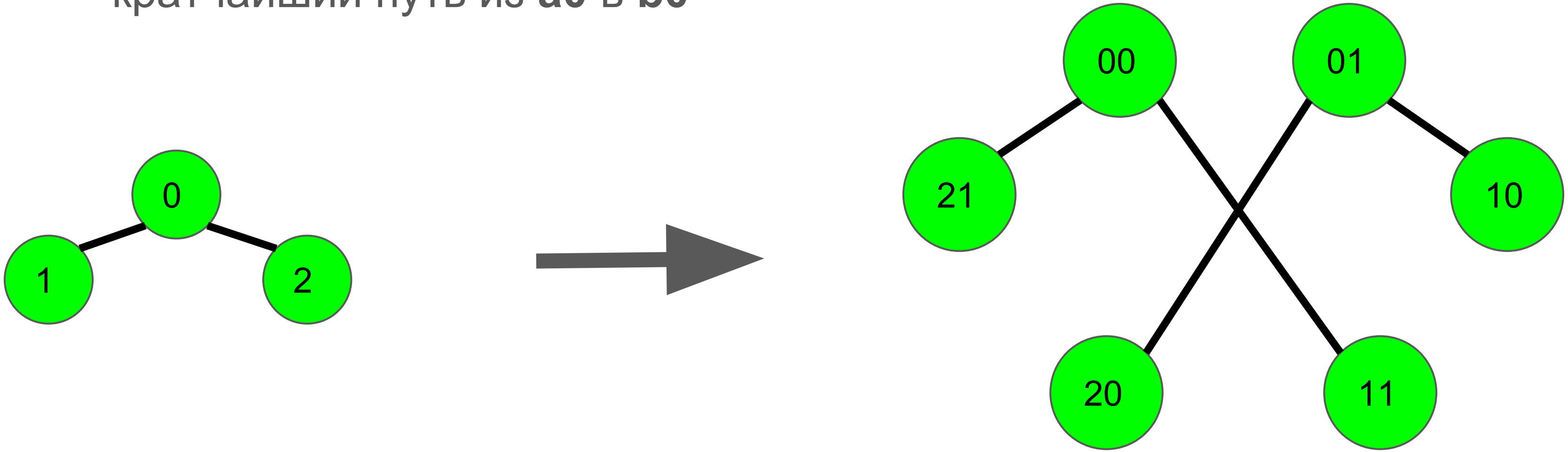
1. Запускаем обходы в ширину из **a** и из **b** с **подсчетом расстояний**
2. Расстояния до вершины x храним как d_a[x] и d_b[x]
3. Если d_a[x] + d_b[x] = d_a[b], то вершина лежит на кратчайшем пути

Нахождение всех ребер на кратчайшем пути (a,b)

1. Запускаем обходы в ширину из **a** и из **b** с **подсчетом расстояний**
2. Расстояния до вершины x храним как d_a[x] и d_b[x]
3. Для ребра (u,v) проверяем d_a[u] + 1 + d_b[v] = d_a[b]
4. Если равенство выполнено, то ребро лежит на кратчайшем пути

Кратчайший путь четной длины

1. Строим вспомогательный граф, каждая вершина e превращается в 2 вершины: e0 и e1
2. Каждое ребро (u, v) превращается в 2 ребра: (u0, v1) и (u1, v0)
3. Найти кратчайший путь четный путь из **a** в **b** == найти в новом графе кратчайший путь из **a0** в **b0**



Бонус!

Нахождение наиболее короткой цепочки друзей между двумя пользователями ВКонтакте.

Получение данных: HTTP-запросы + VK API

Алгоритм:

1) Задаем id_start, id_end
2)

Построение цепочки друзей - код

```
import requests # делать запросы
import time # делать задержки между запросами
from tqdm import tqdm # progress bar

HOST = 'https://api.vk.com/method/'
VERSION = '5.74'
access_token = 'TOKEN HERE'

def get_friends_id(user_id):
    r = requests.get(HOST + 'friends.get', params={'user_id': user_id,
                                                    'access_token': access_token,
                                                    'v': VERSION})
    if 'response' in r.json():
        return r.json()['response']['items']
    return []
```

Построение цепочки друзей - код

```
queue = deque(get_friends_id(id_start))

parents = {user:id_start for user in queue}
distances = {user:1 for user in queue}

while id_end not in distances:
    cur_user = queue.popleft()
    new_users = get_friends_id(cur_user)
    time.sleep(0.5)
    for u in tqdm(new_users):
        if u not in distances:
            queue.append(u)
            distances[u] = distances[cur_user] + 1
            parents[u] = cur_user
```

Спасибо за внимание!