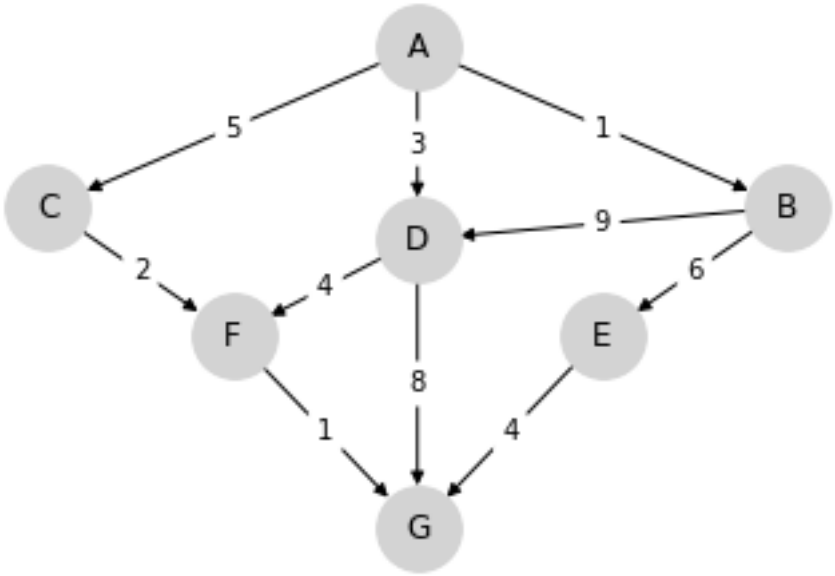
Семинар 11

Если не оговорено особо, то рисунки взяты из книги Саржента и Стачурски «Intermediate Quantitative Economics with Python»

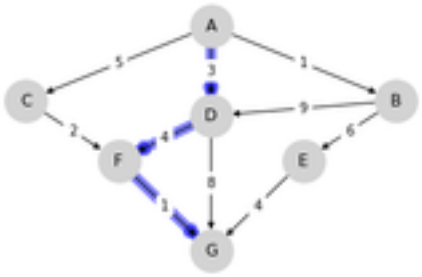
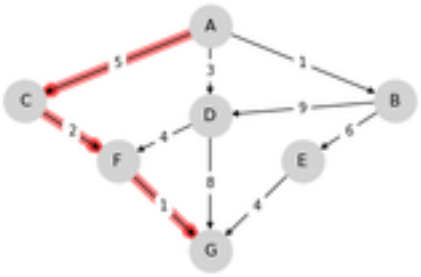
1. Поиск оптимального пути

Рассмотрим следующую задачу. Имеется некоторый направленный граф, ребра которого имеют некоторый вес.



От нас требуется найти оптимальный путь — в нашем случае имеющий наименьшую длину — от вершины A до вершины G.

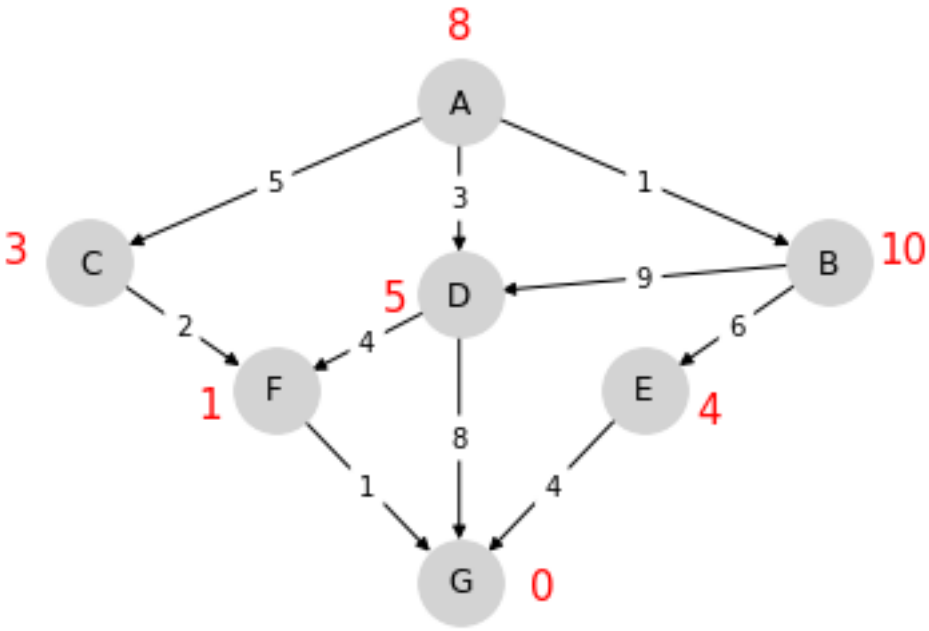
Несложно увидеть, что имеется **два** оптимальных решения:

Решение 1	Решение 2
	

Попробуем найти алгоритм для *произвольного* графа.

1.1. Уравнение Беллмана

Посчитаем длину кратчайшего пути из **каждой** вершины графа до вершины G. Обозначим эту длину как $J(v)$.



Заметим, что $J(G) = 0$, что очевидно, так как оптимальный путь из G в G — оставаться на месте!

Алгоритм поиска кратчайшего пути состоит из единственного повторяющегося шага. Начнем с вершины $v = A$. На каждом шаге мы должны переходить в вершину $v \to w$ такую, что

$$w = \arg\min_{w \in F_v} \{c(v, w) + J(w)\}$$

где $c(v, w)$ — расстояние между вершинами v и w, F_v — множество вершин, непосредственно достижимых из вершины v, то есть непосредственно соединенных с v.

Данный алгоритм подразумевает, что нам известна функция $J(v)$. Как же ее найти?

Рассмотрим вершину A. Так как следующая вершина (C или D) является $\arg\min_{w \in F_A}$, то

$$J(A) = \min_{w \in F_A} \{c(A, w) + J(w)\}$$

Это справедливо для **любой** вершины.

Полученное уравнение

$$J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\}$$

называется **уравнением Беллмана**.

Функция $J(v)$ может быть оценена при помощи следующей процедуры. Пусть стартовым значением $J(v)$ будет

$$J_0(v) = 0, \quad \forall v$$

1. Рассчитайте следующее значение $J_1(v)$ как

$$J_1(v) = \min_{w \in F_v} \{c(v, w) + J_0(w)\}, \quad \forall v$$

2. Повторяйте шаг (1) до сходимости процедуры, то есть до достижения $J_{n+1}(v) = J_n(v), \ \forall v$.

Данная процедура в рассмотренном нами случае обязательно сойдется. Подробнее в:

> Bellman R. On the Theory of Dynamic Programming / R. Bellman // Proceedings of the National Academy of Sciences. – 1952. – T. 38. – № 8. – C. 716-719.

1.2. Алгоритм Дейкстры

Подход, использующий уравнение Беллмана, обладает рядом недостатков. Ключевых можно выделить два:

1. В зависимости от того, ищем ли мы наилучший путь из «точки A в точку B» или же нет, сложность алгоритма равна $O(N^3)$ и $O(N^2)$, соответственно.
2. Для поиска пути нам надо *de facto* найти все оптимальные пути от начала до всех других узлов.

Алгоритм Дейкстры является более быстрым способом поиска оптимального пути. Предложен он был, как ни странно, Дейкстрой в 1959 году.

> Dijkstra E.W. A note on two problems in connexion with graphs / E.W. Dijkstra // Numerische Mathematik. – 1959. – Vol. 1. – № 1. – P. 269-271.

В чем основная идея алгоритма? Рассмотрим сначала алгоритм в его первозданном виде, в котором он был предназначен для поиска оптимального пути от начального узла графа до **всех остальных**.

Основное ограничение, накладываемое на граф, состоит в том, что он должен иметь ребра со строго положительными весами! Обозначим вес ребра, идущего от узла x к узлу y как $c(x, y)$.

Присвоим каждому узлу графа «метку» $d[x]$, равную длине кратчайшего пути от стартового узла. Метка стартового узла считается равной 0, всех остальных — ∞.

Также для каждого узла будем хранить оптимальный путь до него; обозначим его как $p[x]$. Для стартового узла (допустим A) $p[A] \coloneqq A$.

Пусть мы находимся в узле v.

1. Рассмотрим все узлы, которые
   - являются соседями узла v, то есть напрямую связанные с узлом v с учетом направленности ребра. То есть, если **от** v **к** узлу идет ребро, то этот узел является соседом. Если же ребро **к** узлу не идет, то он соседом не является, даже если **от** узла к v ребро идет!
   - мы еще не посещали (на первом шаге мы еще не посетили ни одного узла).
2. Для каждого узла-соседа x мы проверяем следующее условие

   $$d[x] \geqslant d[v] + c(v, x)$$

   Если данное неравенство истинно, то метка узла x меняется на $d[v] + c(v, x)$.
   - Если нас интересует не только длина пути, но и сам путь, то

   $$p[x] \coloneqq (p[v], x)$$

3. Отметим узел v как **посещенный**.
4. Перейдем в непосещенный ранее узел с наименьшим значением $d[x]$.

В виде псевдокода это будет иметь вид:

```
d := Словарь(узел: расстояние)
p := Словарь(узел: путь)
visited = Список

для каждого узла x:
    d[x] := inf
    p[x] := ()
d[A] = 0
p[A] := (A)

CUR := A

для i от 1 до N:
    F := [x | x сосед CUR и x не в visited]
    для x в F:
        если d[x] ≥ d[CUR] + c(CUR, x) то:
            d[x] := d[CUR] + c(CUR, x)
            p[x] := p[CUR] + (x,)
    добавить CUR в visited
    CUR := x | d[x] = min(d[x]) и x не в visited
    если CUR неопределен, то выход
```

Если нас не интересуют пути до **всех** узлов, а лишь путь до одного конкретного, то как только мы переходим в него — останавливаем алгоритм.

В описанной выше реализации мы все еще должны знать все узлы на момент работы алгоритма, что увеличивает требования к памяти и не всегда возможно. Можно модифицировать алгоритм так, чтобы это нам было не нужно.

Пусть мы находимся в стартовом узле A. Создадим объект типа **очередь с приоритетом** — очередь, в которой элементы расположены в порядке возрастания некоторой величины, например расстояния до начала графа. В начале добавим в очередь стартовый узел A с приоритетом 0.

1. Удалим узел A из очереди.
2. Рассмотрим все узлы, которые являются соседями узла A.
3. Для каждого узла-соседа x мы проверяем следующее условие

$$d[x] \geqslant d[v] + c(v, x)$$

   Если данное неравенство истинно, то
   - метка узла x меняется на $d[v] + c(v, x)$;
   - узел x добавляется в очередь с приоритетом, равным новому значению метки;
   - если нас интересует не только длина пути, но и сам путь, то

$$p[x] \coloneqq (p[v], x)$$

4. Перейдем в узел из очереди с наименьшим приоритетом.

Повторяем эти шаги до тех пор, пока очередь на момент шага (4) не пуста.

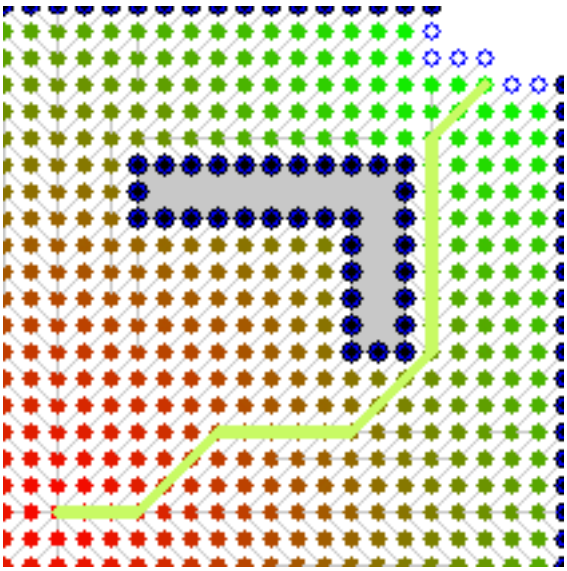
В виде псевдокода это будет иметь вид:

```
CUR := A
d := Словарь(узел: расстояние)
d[A] := 0

p := Словарь(узел: путь)
p[A] := (A)

QUEUE := ОчередьСПриоритетом(узел, расстояние)
добавить (A, 0) в QUEUE

пока QUEUE не пуста:
    CUR := извлечь первый элемент QUEUE
    F := [x | x сосед CUR]
    для x в F:
        если d[x] ≥ d[CUR] + c(CUR, x) или x не в d то:
            d[x] := d[CUR] + c(CUR, x)
            p[x] := p[CUR] + (x,)
            добавить (x, d[x]) в QUEUE
```

Если нас не интересуют пути до **всех** узлов, а лишь путь до одного конкретного, то как только мы переходим в него — останавливаем алгоритм.

1.3. Алгоритм A*

Алгоритм Дейкстры, тем не менее, в ряде задач не самый оптимальный. Если перед нами стоит задача поиска оптимального пути между двумя **конкретными** узлами, то он будет выполнять достаточно большое количество ненужных проверок.



Subh83, CC BY 3.0 https://creativecommons.org/licenses/by/3.0, via Wikimedia Commons

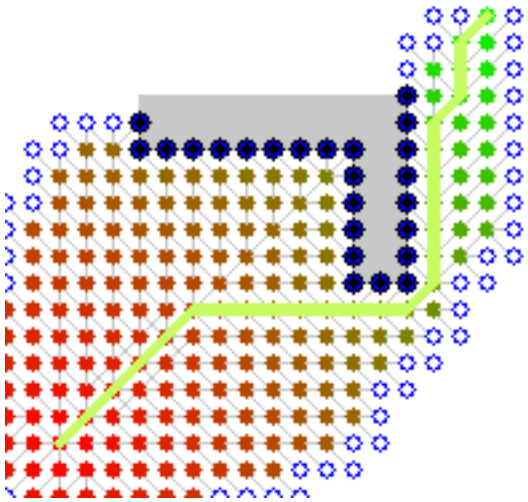
Небольшая модификация, называемая **алгоритмом A***, позволяет ускорить нахождение результата. Для этого в метот вводится некая (зависящая от решаемой задачи) эвристическая функция. Ее задача — модифицировать процедуру выбора следующего узла для обработки. Модификация происходит путем прибавления значения эвристической функции к значению приоритета (или расстояния в нашем случае) потенциального узла. Таким образом, следующим узлом будет узел с минимальной суммой значений приоритета-расстояния и эвристической функции.

В виде псевдокода это будет иметь вид:

```
CUR := A
d := Словарь(узел: расстояние)
d[A] := 0

p := Словарь(узел: путь)
p[A] := (A)

QUEUE := ОчередьСПриоритетом(узел, расстояние)
добавить (A, 0 + heuristic(A)) в QUEUE

пока QUEUE не пуста:
    CUR := извлечь первый элемент QUEUE
    если CUR = целевой узел:
        выход из цикла
    F := [x | x сосед CUR]
    для x в F:
        если d[x] ≥ d[CUR] + c(CUR, x) или x не в d то:
            d[x] := d[CUR] + c(CUR, x)
            p[x] := p[CUR] + (x,)
            добавить (x, d[x] + heuristic(x)) в QUEUE
```

Можно увидеть, что данная версия отличается от исходного алгоритма Дейкстры только тем, что приоритет узла в очереди равен не расстоянию от узла до старта, но сумме этого расстояния и значения некоторой эвритсической функции.

Выбор эвристики зависит от задачи. В задачах поиска кратчайшего пути (например на карте) в качестве таковой можно взять евклидово расстояние от узла до финиша. Правильно подобранная эвристика может существенно ускорить поиск решения.



Subh83, CC BY 3.0 https://creativecommons.org/licenses/by/3.0, via Wikimedia Commons

2. Эксперименты

2.1. Уравнение Беллмана

2.1.1. Простой пример

```
import numpy as np
from numpy import inf
```

```
Q = np.array(
    [
        [inf, 1, 5, 3, inf, inf, inf],
        [inf, inf, inf, 9, 6, inf, inf],
        [inf, inf, inf, inf, inf, 2, inf],
        [inf, inf, inf, inf, inf, 4, 8],
        [inf, inf, inf, inf, inf, inf, 4],
        [inf, inf, inf, inf, inf, inf, 1],
        [inf, inf, inf, inf, inf, inf, 0],
    ]
)
```

```
nodes = list(range(7))
J = np.zeros_like(nodes, dtype=int)
next_J = np.zeros_like(nodes, dtype=int)

iters = 500
```

```
for _ in range(iters):
    for v in nodes:
        next_J[v] = np.min(Q[v, :] + J[:])
    if np.equal(next_J, J).all():
        break
    J[:] = next_J[:]

print(f"J равно = {J}")
```

```
J равно = [ 8 10  3  5  4  1  0]
```

```
current_node = 0
destination_node = 6
len_path = 0

path = []

while current_node != destination_node:
    path.append(current_node)
    next_node = np.argmin(Q[current_node, :] + J)
    len_path += Q[current_node, next_node]
    current_node = int(next_node)

path.append(destination_node)

print(path, len_path)
```

```
[0, 2, 5, 6] 8.0
```

2.1.2. Пример посложнее

```
%%file graph.txt
node0, node1 0.04, node8 11.11, node14 72.21
node1, node46 1247.25, node6 20.59, node13 64.94
node2, node66 54.18, node31 166.80, node45 1561.45
node3, node20 133.65, node6 2.06, node11 42.43
node4, node75 3706.67, node5 0.73, node7 1.02
node5, node45 1382.97, node7 3.33, node11 34.54
node6, node31 63.17, node9 0.72, node10 13.10
node7, node50 478.14, node9 3.15, node10 5.85
node8, node69 577.91, node11 7.45, node12 3.18
node9, node70 2454.28, node13 4.42, node20 16.53
node10, node89 5352.79, node12 1.87, node16 25.16
node11, node94 4961.32, node18 37.55, node20 65.08
node12, node84 3914.62, node24 34.32, node28 170.04
node13, node60 2135.95, node38 236.33, node40 475.33
node14, node67 1878.96, node16 2.70, node24 38.65
node15, node91 3597.11, node17 1.01, node18 2.57
node16, node36 392.92, node19 3.49, node38 278.71
node17, node76 783.29, node22 24.78, node23 26.45
```

```
node18, node91 3363.17, node23 16.23, node28 55.84
node19, node26 20.09, node20 0.24, node28 70.54
node20, node98 3523.33, node24 9.81, node33 145.80
node21, node56 626.04, node28 36.65, node31 27.06
node22, node72 1447.22, node39 136.32, node40 124.22
node23, node52 336.73, node26 2.66, node33 22.37
node24, node66 875.19, node26 1.80, node28 14.25
node25, node70 1343.63, node32 36.58, node35 45.55
node26, node47 135.78, node27 0.01, node42 122.00
node27, node65 480.55, node35 48.10, node43 246.24
node28, node82 2538.18, node34 21.79, node36 15.52
node29, node64 635.52, node32 4.22, node33 12.61
node30, node98 2616.03, node33 5.61, node35 13.95
node31, node98 3350.98, node36 20.44, node44 125.88
node32, node97 2613.92, node34 3.33, node35 1.46
node33, node81 1854.73, node41 3.23, node47 111.54
node34, node73 1075.38, node42 51.52, node48 129.45
node35, node52 17.57, node41 2.09, node50 78.81
node36, node71 1171.60, node54 101.08, node57 260.46
node37, node75 269.97, node38 0.36, node46 80.49
node38, node93 2767.85, node40 1.79, node42 8.78
node39, node50 39.88, node40 0.95, node41 1.34
node40, node75 548.68, node47 28.57, node54 53.46
node41, node53 18.23, node46 0.28, node54 162.24
node42, node59 141.86, node47 10.08, node72 437.49
node43, node98 2984.83, node54 95.06, node60 116.23
node44, node91 807.39, node46 1.56, node47 2.14
node45, node58 79.93, node47 3.68, node49 15.51
node46, node52 22.68, node57 27.50, node67 65.48
node47, node50 2.82, node56 49.31, node61 172.64
node48, node99 2564.12, node59 34.52, node60 66.44
node49, node78 53.79, node50 0.51, node56 10.89
node50, node85 251.76, node53 1.38, node55 20.10
node51, node98 2110.67, node59 23.67, node60 73.79
node52, node94 1471.80, node64 102.41, node66 123.03
node53, node72 22.85, node56 4.33, node67 88.35
node54, node88 967.59, node59 24.30, node73 238.61
node55, node84 86.09, node57 2.13, node64 60.80
node56, node76 197.03, node57 0.02, node61 11.06
node57, node86 701.09, node58 0.46, node60 7.01
node58, node83 556.70, node64 29.85, node65 34.32
node59, node90 820.66, node60 0.72, node71 0.67
node60, node76 48.03, node65 4.76, node67 1.63
node61, node98 1057.59, node63 0.95, node64 4.88
node62, node91 132.23, node64 2.94, node76 38.43
node63, node66 4.43, node72 70.08, node75 56.34
node64, node80 47.73, node65 0.30, node76 11.98
node65, node94 594.93, node66 0.64, node73 33.23
node66, node98 395.63, node68 2.66, node73 37.53
node67, node82 153.53, node68 0.09, node70 0.98
```

```
node68, node94 232.10, node70 3.35, node71 1.66
node69, node99 247.80, node70 0.06, node73 8.99
node70, node76 27.18, node72 1.50, node73 8.37
node71, node89 104.50, node74 8.86, node91 284.64
node72, node76 15.32, node84 102.77, node92 133.06
node73, node83 52.22, node76 1.40, node90 243.00
node74, node81 1.07, node76 0.52, node78 8.08
node75, node92 68.53, node76 0.81, node77 1.19
node76, node85 13.18, node77 0.45, node78 2.36
node77, node80 8.94, node78 0.98, node86 64.32
node78, node98 355.90, node81 2.59
node79, node81 0.09, node85 1.45, node91 22.35
node80, node92 121.87, node88 28.78, node98 264.34
node81, node94 99.78, node89 39.52, node92 99.89
node82, node91 47.44, node88 28.05, node93 11.99
node83, node94 114.95, node86 8.75, node88 5.78
node84, node89 19.14, node94 30.41, node98 121.05
node85, node97 94.51, node87 2.66, node89 4.90
node86, node97 85.09
node87, node88 0.21, node91 11.14, node92 21.23
node88, node93 1.31, node91 6.83, node98 6.12
node89, node97 36.97, node99 82.12
node90, node96 23.53, node94 10.47, node99 50.99
node91, node97 22.17
node92, node96 10.83, node97 11.24, node99 34.68
node93, node94 0.19, node97 6.71, node99 32.77
node94, node98 5.91, node96 2.03
node95, node98 6.17, node99 0.27
node96, node98 3.32, node97 0.43, node99 5.87
node97, node98 0.30
node98, node99 0.33
node99,
```

```
Overwriting graph.txt
```

```python
def file_to_Q(filename, fin, N):
    Q = np.full((N, N), np.inf)

    with open(filename, "r", encoding="utf8") as f:
        for line in f:
            chunks = line.split(",")
            node = chunks.pop(0)
            node = int(node[4:])

            if node != fin:
                for el in chunks:
                    dest, dist = el.strip().split()
                    dest = int(dest[4:])
                    Q[node, dest] = dist
        Q[fin, fin] = 0
```

```
    return Q
```

```
def compute_J(Q, iters=500):
    N = Q.shape[0]
    J = np.zeros(N)
    next_J = np.zeros(N)

    for _ in range(iters):
        for v in range(N):
            next_J[v] = np.min(Q[v, :] + J)
        if np.allclose(next_J, J):
            return J
        J[:] = next_J[:]

    return J
```

```
def compute_path(J, Q, beg=0, fin=99):
    path = []
    cur_node = beg
    len_path = 0

    while cur_node != fin:
        path.append(cur_node)
        next_node = np.argmin(Q[cur_node, :] + J)
        len_path += Q[cur_node, next_node]
        cur_node = int(next_node)

    path.append(fin)

    return (path, len_path)
```

```
Q1 = file_to_Q("graph.txt", 99, 100)
```

```
%%time
J1 = compute_J(Q1)
path, len_path = compute_path(J1, Q1)
print(f"Путь: {path}\nДлина пути: {len_path:2.2f}\nПервый␣
  ↪элемент J: {J1[0]}")
```

```
Путь: [0, 8, 11, 18, 23, 33, 41, 53, 56, 57, 60, 67, 70, 73, 76,␣
  ↪85, 87, 88, 93,
94, 96, 97, 98, 99]
Длина пути: 160.55
Первый элемент J: 160.55
CPU times: total: 15.6 ms
Wall time: 16 ms
```

2.2. Алгоритм Дейкстры

Модуль, содержащий методы для работы со списками как с очередью с приоритетом.

```
import heapq as hq
```

Метод, который возвращает список соседей узла.

```
def next_nodes(cur, Q):
    N = Q.shape[0]
    res = []
    for node in range(N):
        if node == cur:
            continue
        if Q[cur, node] != np.inf:
            res.append(node)
    return res
```

```
def dijkstra(Q, beg=0, fin=99):
    QUEUE = []
    DIST = {}
    PATH = {}

    hq.heappush(QUEUE, (0, beg))
    DIST[beg] = 0
    PATH[beg] = (beg,)

    while QUEUE:
        dist, cur = hq.heappop(QUEUE)
        if cur == fin:
            return PATH[fin], DIST[fin]
        next_step = next_nodes(cur, Q)
        for node in next_step:
            if (node not in DIST) or (DIST[node] >= Q[cur, node]
 ↪+ dist):
                DIST[node] = Q[cur, node] + dist
                PATH[node] = PATH[cur] + (node,)
                hq.heappush(QUEUE, (DIST[node], node))
    return None, None
```

```
path, len_path = dijkstra(Q, 0, 6)
print(f"Путь: {path}\nДлина пути: {len_path:2.2f}")
```

```
Путь: (0, 2, 5, 6)
Длина пути: 8.00
```

```
%%time
path, len_path = dijkstra(Q1, 0, 99)
print(f"Путь: {path}\nДлина пути: {len_path:2.2f}")
```

```
Путь: (0, 8, 11, 18, 23, 33, 41, 53, 56, 57, 60, 67, 70, 73, 76,␣
  ↪85, 87, 88, 93,
94, 96, 97, 98, 99)
Длина пути: 160.55
CPU times: total: 0 ns
Wall time: 0 ns
```