МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное Государственное Бюджетное Образовательное
Учреждение Высшего Образования

«МОСКОВСКИЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ
ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ (МАДИ)»

Кафедра «Высшая математика»

ОТЧЕТЫ ПО ЛАБАРАТОРНЫМ РАБОТАМ

по дисциплине **«Теория графов и математическая логика»**

Выполнил:

Серов П.Г.

Группа 3бПМ

Подпись _____

МОСКВА 2022

Алгоритм Краскала

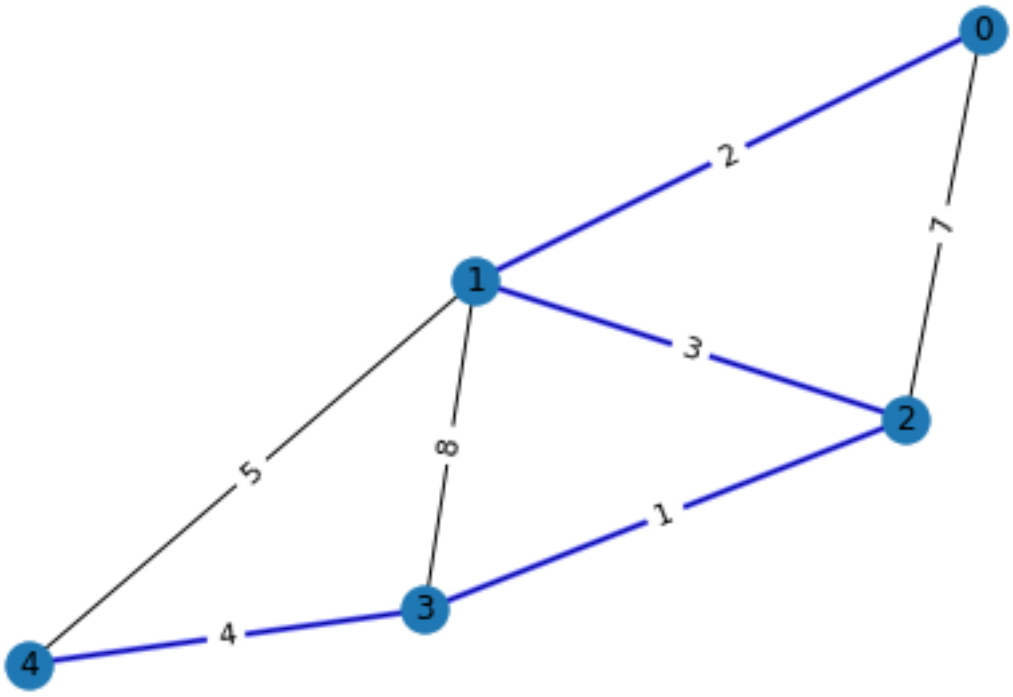
Входные данные: матрица размерности 5x5

*input – Блокнот

Файл Правка Формат Вид

```
5
  0 2 9 -3 -3
  5 0 8 1 7
  9 4 0 3 -8
  -4 8 3 0 8
  -1 3 -2 7 0
```

Результат работы



Листинг

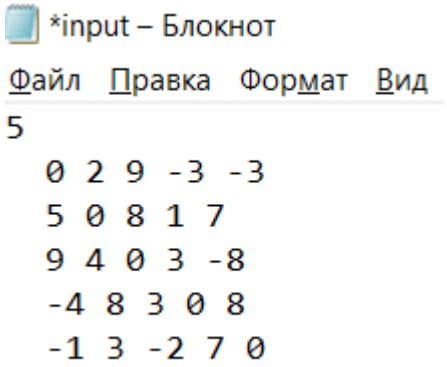
```
# A utility function that return the smallest unprocessed edge
def getMin(G, mstFlag):
    min = sys.maxsize  # assigning largest numeric value to min
    for i in [(u, v, edata['length']) for u, v, edata in G.edges(data=True)
if 'length' in edata]:
        if mstFlag[i] == False and i[2] < min:
            min = i[2]
            min_edge = i
    return min_edge

# A utility function to find root or origin of the node i in MST
def findRoot(parent, i):
```

```
    if parent[i] == i:
        return i
    return findRoot(parent, parent[i])


# A function that does union of set x and y based on the order
def union(parent, order, x, y):
    xRoot = findRoot(parent, x)
    yRoot = findRoot(parent, y)
    # Attach smaller order tree under root of high order tree
    if order[xRoot] < order[yRoot]:
        parent[xRoot] = yRoot
    elif order[xRoot] > order[yRoot]:
        parent[yRoot] = xRoot
    # If orders are same, then make any one as root and increment its order
by one
    else:
        parent[yRoot] = xRoot
        order[xRoot] += 1


# function that performs Kruskals algorithm on the graph G
def kruskals(G, pos):
    eLen = len(G.edges())  # eLen denotes the number of edges in G
    vLen = len(G.nodes())  # vLen denotes the number of vertices in G
    mst = []  # mst contains the MST edges
    mstFlag = {}  # mstFlag[i] will hold true if the edge i has been
processed for MST
    for i in [(u, v, edata['length']) for u, v, edata in G.edges(data=True)
if 'length' in edata]:
        mstFlag[i] = False

    parent = [None] * vLen  # parent[i] will hold the vertex connected to i,
in the MST
    order = [None] * vLen  # order[i] will hold the order of appearance of
the node in the MST
    for v in range(vLen):
        parent[v] = v
        order[v] = 0
    while len(mst) < vLen - 1:
        curr_edge = getMin(G, mstFlag)  # pick the smallest egde from the set
of edges
        mstFlag[curr_edge] = True  # update the flag for the current edge
        y = findRoot(parent, curr_edge[1])
        x = findRoot(parent, curr_edge[0])
        # adds the edge to MST, if including it doesn't form a cycle
        if x != y:
            mst.append(curr_edge)
            union(parent, order, x, y)
    # Else discard the edge
    # marks the MST edges with red
    for X in mst:
        if (X[0], X[1]) in G.edges():
            nx.draw_networkx_edges(G, pos, edgelist=[(X[0], X[1])],
width=2.5, alpha=0.6, edge_color='r')
    return


# takes input from the file and creates a weighted graph
def CreateGraph():
    G = nx.Graph()
    f = open('KruskalsInput.txt')
    n = int(f.readline())
    wtMatrix = []
```

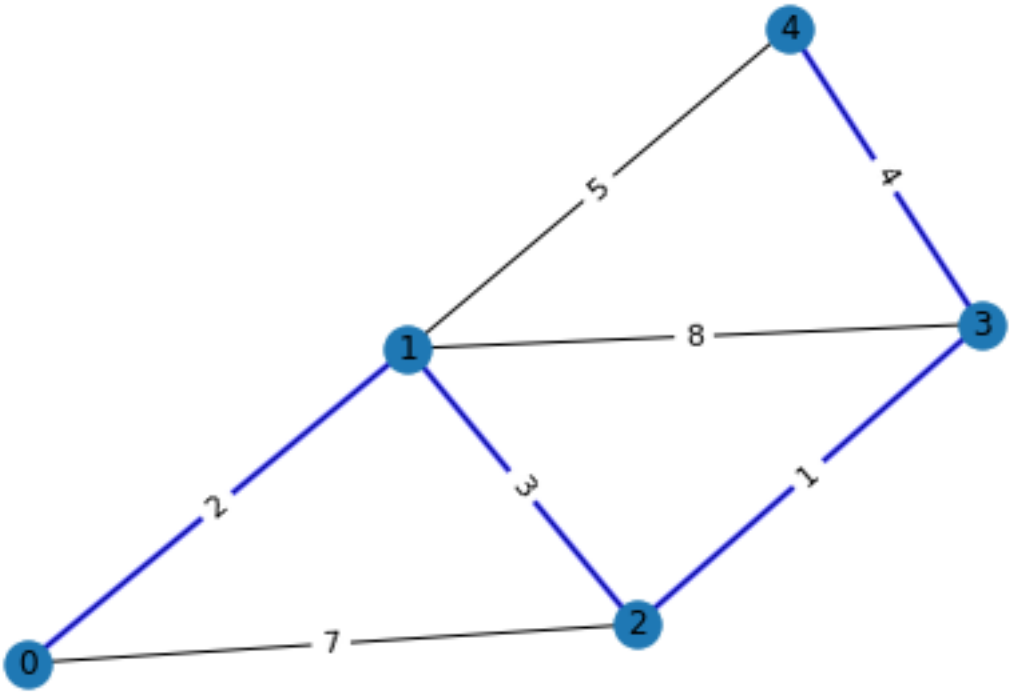
```
    for i in range(n):
        list1 = list(map(int, (f.readline()).split()))
        wtMatrix.append(list1)
    # Adds egdes along with their weights to the graph
    for i in range(n):
        for j in range(n)[i:]:
            if wtMatrix[i][j] > 0:
                G.add_edge(i, j, length=wtMatrix[i][j])
    return G


# draws the graph and displays the weights on the edges
def DrawGraph(G):
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True)  # with_labels=true is to show the node
number in the output graph
    edge_labels = nx.get_edge_attributes(G, 'length')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
font_size=11)  # prints weight on all the edges
    return pos
if __name__ == "__main__":
    G = CreateGraph()
    pos = DrawGraph(G)
    kruskal(G, pos)
    plt.show()

 # Kruskal
```

Алгоритм Прима

Входные данные: матрица размерности 5x5



Результат работы:



Листинг

```
def minDistance(dist, mstSet, V):
    min = sys.maxsize  # assigning largest numeric value to min
    for v in range(V):
        if mstSet[v] == False and dist[v] < min:
            min = dist[v]
            min_index = v
    return min_index


# function that performs prims algorithm on the graph G
def prims(G, pos):
    V = len(G.nodes())  # V denotes the number of vertices in G
    dist = []  # dist[i] will hold the minimum weight edge value of node i to
be included in MST
    parent = [None] * V  # parent[i] will hold the vertex connected to i, in
the MST edge
    mstSet = []  # mstSet[i] will hold true if vertex i is included in the
MST
    # initially, for every node, dist[] is set to maximum value and mstSet[]
is set to False
    for i in range(V):
        dist.append(sys.maxsize)
        mstSet.append(False)
    dist[0] = 0
    parent[0] = -1  # starting vertex is itself the root, and hence has no
parent
    for count in range(V - 1):
        u = minDistance(dist, mstSet, V)  # pick the minimum distance vertex
from the set of vertices
        mstSet[u] = True
        # update the vertices adjacent to the picked vertex
        for v in range(V):
            if (u, v) in G.edges():
```

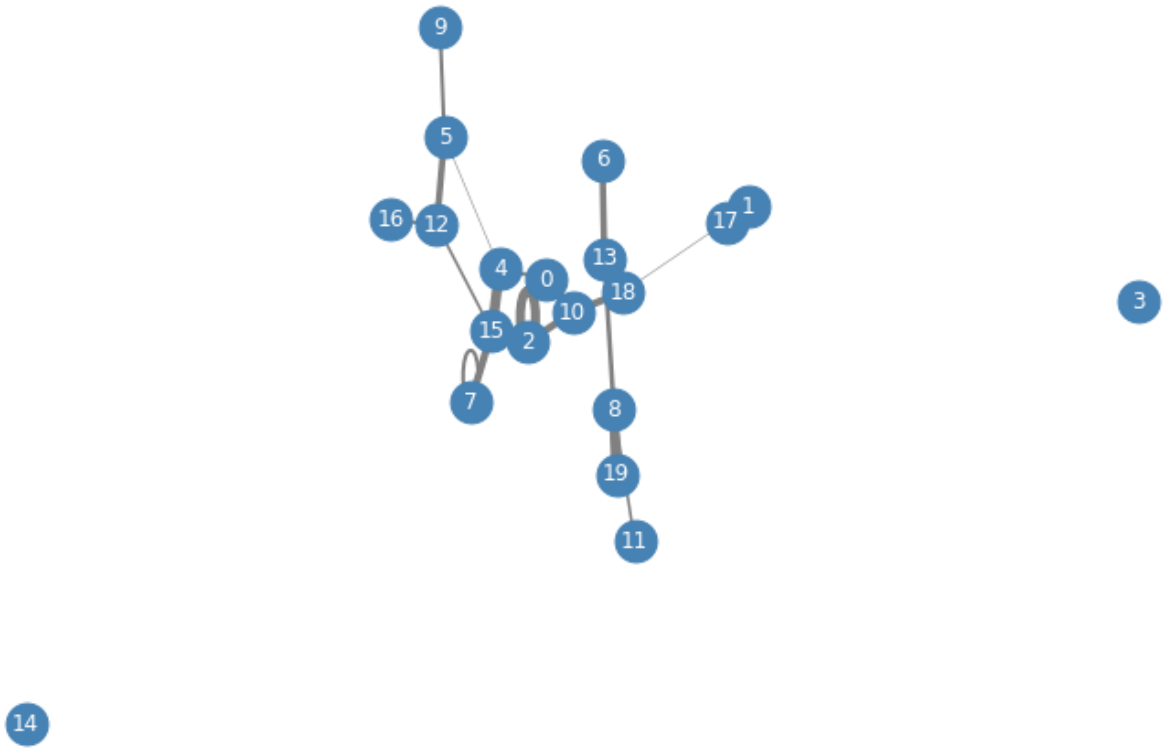
```
                if mstSet[v] == False and G[u][v]['length'] < dist[v]:
                    dist[v] = G[u][v]['length']
                    parent[v] = u
    for X in range(V):
        if parent[X] != -1:  # ignore the parent of the starting node
            if (parent[X], X) in G.edges():
                nx.draw_networkx_edges(G, pos, edgelist=[(parent[X], X)],
width=2.5, alpha=0.6, edge_color='r')
    return


# takes input from the file and creates a weighted graph
def CreateGraph():
    G = nx.Graph()
    f = open('KruskalsInput.txt')
    n = int(f.readline())
    wtMatrix = []
    for i in range(n):
        list1 = list(map(int, (f.readline()).split()))
        wtMatrix.append(list1)
    # Adds egdes along with their weights to the graph
    for i in range(n):
        for j in range(n)[i:]:
            if wtMatrix[i][j] > 0:
                G.add_edge(i, j, length=wtMatrix[i][j])
    return G


# draws the graph and displays the weights on the edges
def DrawGraph(G):
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True)  # with_labels=true is to show the node
number in the output graph
    edge_labels = nx.get_edge_attributes(G, 'length')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
font_size=11)  # prints weight on all the edges
    return pos

if __name__ == "__main__":
    G = CreateGraph()
    pos = DrawGraph(G)
    prims(G, pos)
    plt.show()
```

Алгоритм Дийкстры

Входной поток:

```
3] n = 20
   graph = nx.Graph()
   graph.add_nodes_from(range(n))
   for u, v in np.random.randint(0, n, (n, 2)):
       graph.add_edge(u, v, weight=abs(u - v))
```

Результат работы:



Дистанция

```
(1, ..., 1) = 0
(1, ..., 17) = 16
(1, ..., 18) = 17
(1, ..., 10) = 25
(1, ..., 2) = 33
(1, ..., 0) = 35
(1, ..., 4) = 39
(1, ..., 5) = 40
(1, ..., 9) = 44
(1, ..., 15) = 46
(1, ..., 12) = 47
(1, ..., 16) = 51
(1, ..., 7) = 54
```

Листинг

```
from heapq import heappush, heappop
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

def dijkstra(graph, source):
    distance = {}
    queue = [(0, source)]
```

```
    while queue:
        # shortest unexplored path
        p, v = heappop(queue)
        if v in distance:
            continue

        # shortest path (source, ..., v)
        print('({}, ..., {}) = {}'.format(source, v, p))
        distance[v] = p

        # extend path to (source, ..., v, u)
        for _, u, e in graph.edges(v, data=True):
            heappush(queue, (p + e['weight'], u))

    return distance

n = 20

graph = nx.Graph()
graph.add_nodes_from(range(n))
for u, v in np.random.randint(0, n, (n, 2)):
    graph.add_edge(u, v, weight=abs(u - v))

weights = [e['weight'] / n * 10  for (u, v, e) in graph.edges(data=True)]

plt.figure(figsize=(12, 8))
plt.axis('off')

layout = nx.spring_layout(graph)
nx.draw_networkx_nodes(graph, layout, node_color='steelblue', node_size=52
0)
nx.draw_networkx_edges(graph, layout, edge_color='gray', width=weights)
nx.draw_networkx_labels(graph, layout, font_color='white')

distances = dijkstra(graph, 1)
```

Алгоритм Флойда-Фалкерсона

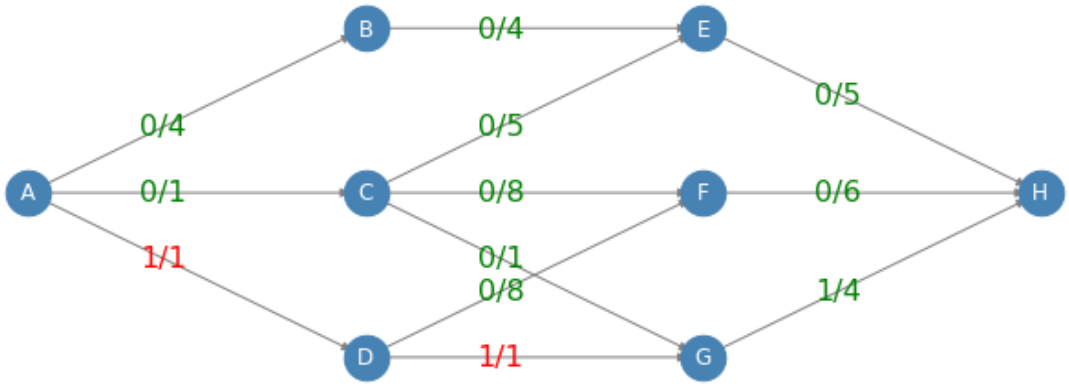
Входной поток:

```
graph = nx.DiGraph()
graph.add_nodes_from('ABCDEFGH')
graph.add_edges_from([
    ('A', 'B', {'capacity': 4, 'flow': 0}),
    ('A', 'C', {'capacity': 1, 'flow': 0}),
    ('A', 'D', {'capacity': 1, 'flow': 0}),
    ('B', 'E', {'capacity': 4, 'flow': 0}),
    ('C', 'E', {'capacity': 5, 'flow': 0}),
    ('C', 'F', {'capacity': 8, 'flow': 0}),
    ('C', 'G', {'capacity': 1, 'flow': 0}),
    ('D', 'F', {'capacity': 8, 'flow': 0}),
    ('D', 'G', {'capacity': 1, 'flow': 0}),
    ('E', 'H', {'capacity': 5, 'flow': 0}),
    ('F', 'H', {'capacity': 6, 'flow': 0}),
    ('G', 'H', {'capacity': 4, 'flow': 0}),
])
```

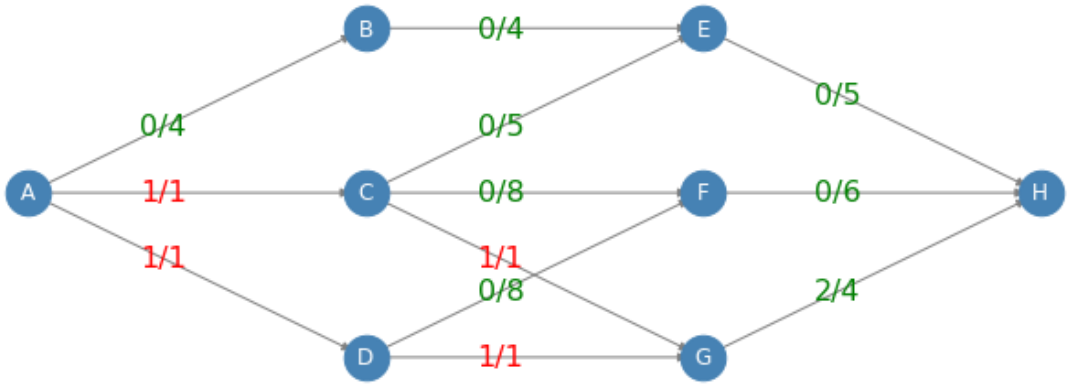
```
[26] layout = {
        'A': [0, 1], 'B': [1, 2], 'C': [1, 1], 'D': [1, 0],
        'E': [2, 2], 'F': [2, 1], 'G': [2, 0], 'H': [3, 1],
    }
```

Результат работы:

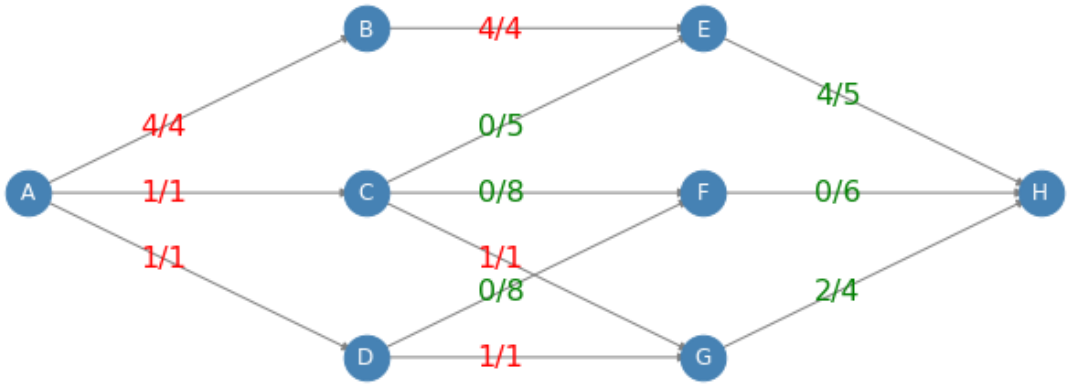
```
flow increased by 1 at path ['A', 'D', 'G', 'H'] ; current flow 1
```



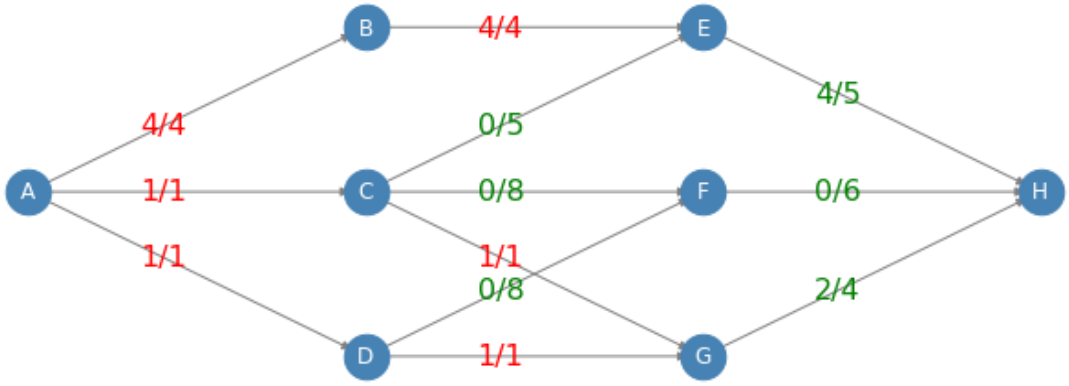
```
flow increased by 1 at path ['A', 'C', 'G', 'H'] ; current flow 2
```




```
flow increased by 4 at path ['A', 'B', 'E', 'H'] ; current flow 6
```




```
flow increased by 0 at path [] ; current flow 6
```



\

Листинг

```
import networkx as nx
import matplotlib.pyplot as plt


def ford_fulkerson(graph, source, sink, debug=None):
    flow, path = 0, True

    while path:
        # search for path with flow reserve
        path, reserve = depth_first_search(graph, source, sink)
        flow += reserve

        # increase flow along the path
        for v, u in zip(path, path[1:]):
            if graph.has_edge(v, u):
                graph[v][u]['flow'] += reserve
            else:
                graph[u][v]['flow'] -= reserve

        # show intermediate results
        if callable(debug):
            debug(graph, path, reserve, flow)


def depth_first_search(graph, source, sink):
    undirected = graph.to_undirected()
    explored = {source}
    stack = [(source, 0, dict(undirected[source]))]

    while stack:
        v, _, neighbours = stack[-1]
        if v == sink:
            break

        # search the next neighbour
        while neighbours:
            u, e = neighbours.popitem()
            if u not in explored:
                break
        else:
            stack.pop()
            continue

        # current flow and capacity
        in_direction = graph.has_edge(v, u)
        capacity = e['capacity']
        flow = e['flow']
        neighbours = dict(undirected[u])

        # increase or redirect flow at the edge
        if in_direction and flow < capacity:
            stack.append((u, capacity - flow, neighbours))
            explored.add(u)
        elif not in_direction and flow:
            stack.append((u, flow, neighbours))
            explored.add(u)

    # (source, sink) path and its flow reserve
    reserve = min((f for _, f, _ in stack[1:]), default=0)
    path = [v for v, _, _ in stack]

    return path, reserve
```

```
graph = nx.DiGraph()
graph.add_nodes_from('ABCDEFGH')
graph.add_edges_from([
    ('A', 'B', {'capacity': 4, 'flow': 0}),
    ('A', 'C', {'capacity': 1, 'flow': 0}),
    ('A', 'D', {'capacity': 1, 'flow': 0}),
    ('B', 'E', {'capacity': 4, 'flow': 0}),
    ('C', 'E', {'capacity': 5, 'flow': 0}),
    ('C', 'F', {'capacity': 8, 'flow': 0}),
    ('C', 'G', {'capacity': 1, 'flow': 0}),
    ('D', 'F', {'capacity': 8, 'flow': 0}),
    ('D', 'G', {'capacity': 1, 'flow': 0}),
    ('E', 'H', {'capacity': 5, 'flow': 0}),
    ('F', 'H', {'capacity': 6, 'flow': 0}),
    ('G', 'H', {'capacity': 4, 'flow': 0}),
])
layout = {
    'A': [0, 1], 'B': [1, 2], 'C': [1, 1], 'D': [1, 0],
    'E': [2, 2], 'F': [2, 1], 'G': [2, 0], 'H': [3, 1],
}


def draw_graph():
    plt.figure(figsize=(12, 4))
    plt.axis('off')

    nx.draw_networkx_nodes(graph, layout, node_color='steelblue',
node_size=600)
    nx.draw_networkx_edges(graph, layout, edge_color='gray')
    nx.draw_networkx_labels(graph, layout, font_color='white')

    for u, v, e in graph.edges(data=True):
        label = '{}/{}'.format(e['flow'], e['capacity'])
        color = 'green' if e['flow'] < e['capacity'] else 'red'
        x = layout[u][0] * .6 + layout[v][0] * .4
        y = layout[u][1] * .6 + layout[v][1] * .4
        t = plt.text(x, y, label, size=16, color=color,
                     horizontalalignment='center',
verticalalignment='center')

    plt.show()


draw_graph()

def flow_debug(graph, path, reserve, flow):
    print('flow increased by', reserve,
          'at path', path,
          '; current flow', flow)
    draw_graph()

ford_fulkerson(graph, 'A', 'H', flow_debug)
```