Маршрутизация и безопасность

In [1]:

```
import numpy as np
```

Задача 1: Маршруты из S в F

In [14]:

```
N = 935
M = 873
```

In [16]:

```
numWays = []

for i in range(M):
    numWays.append([0] * N)

for i in range(0, M):
    numWays[i][0] = 1

for i in range(0, N):
    numWays[0][i] = 1

for i in range(1, M):
    for j in range(1, N):
        #if (i > 0):
            numWays[i][j] += numWays[i-1][j]
        #if (j > 0):
            numWays[i][j] += numWays[i][j-1]
        #if (j > 0 and i > 0):
            numWays[i][j] += numWays[i-1][j-1]


print("Number of ways = ", numWays[-1][-1])
```

```
Number of ways =  8306841929738506110061239661017700147933092763872198529804
6638490994357495673106698894919131603175869259518082308603733204631896015509
2411406485725890732653360496338565395451759432581274542613985221759456501753
7133807193796647208580361820610621958645924413643982091237950058572270407565
0486028149474357901117716038030727270970467873450402653882506118969844872926
5759938301155493249852112445625865350592519167403465638281410346939785980247
6717625836416956819397698895014806086027192682088497250502762754625429419429
3176531821837421726759834089499519981727870550870216776834410434730586984582
7298764017333029387049907422310341291834677105100357550043572414131799543873
68026871166930820019025
```

Выше мы рассматриваем таблицу M на N, где весь самый левый столбец - единицы, и вся верхняя строка тоже. Вообще, изначально я делала, где только самый верхний левый элемент = 1, а дальше через "if", но так быстрее (убрана проверка условий).

В каждой ячейке массива храним число путей. И исходя из схемы в задании видим формулу:

число путей до клетки = (число путей до клетки сверху) + (число путей до клетки слева) + (число путей до клетки по диагонали слева сверху)

Это и написано в коде выше.

Проходимся по каждой клетке матрицы по разу (точнее, один раз считаем в ней + 1 раз используем ее для клетки снизу + один раз используем ее для клетки справа + один раз используем ее для клетки справа снизу). Итого, каждую клетку построенной матрицы используем не более 4 раз. Значит, сложность: $O(4 \cdot N \cdot M) = O(N \cdot M)$

Задача 2: Маршруты в (не)безопасной сети (NORMAL)

In [81]:

```
# вектор связей
nets = [[1, 2], [1, 3], [1, 4], [1, 5], [1, 7], [1, 8], [1, 9], [1, 12], [1, 16], [2, 9], [
        [3, 10], [3, 11], [3, 13], [3, 18], [4, 6], [4, 22], [5, 15], [5, 21], [5, 29], [6,
        [7, 18], [7, 22], [8, 9], [8, 20], [8, 29], [9, 24], [10, 11], [10, 13], [10, 25],
        [12, 31], [12, 35], [13, 28], [13, 36], [14, 17], [14, 22], [14, 26], [14, 33], [15
        [15, 39], [16, 19], [16, 30], [16, 33], [16, 35], [17, 21], [17, 27], [17, 28], [17
        [19, 25], [19, 32], [19, 33], [20, 25], [20, 34], [21, 42], [21, 45], [22, 25], [23
        [24, 26], [24, 28], [24, 41], [25, 50], [26, 44], [27, 30], [28, 37], [29, 39], [29
        [31, 40], [32, 44], [33, 36], [33, 44], [34, 37], [34, 40], [34, 47], [34, 49], [35
        [38, 39], [38, 49], [39, 49], [39, 50], [40, 46], [41, 43], [41, 44], [41, 48], [42
        [43, 46], [44, 46], [45, 47], [45, 50], [46, 47], [46, 48], [47, 50], [48, 49], [48

# коэффициенты эффективности файрвола
coef = [1336, 783, 1025, 612, 578, 1125, 1583, 1837, 1509, 364, 1635, 1531, 1356, 799, 1600
        777, 1787, 1849, 1768, 1715, 1948, 339, 841, 523, 320, 1967, 1300, 1360, 306, 1093,
        1028, 712, 1399, 1348, 1318, 1563, 1541, 1762, 1684, 375, 302, 735, 1202, 255, 324,
        1241, 1976, 428, 1247, 492, 1624, 602, 294, 688, 1321, 539, 1301, 1760, 285, 1832,
        766, 900, 756, 336, 485, 1250, 1904, 1747, 446, 369, 333, 1790, 1119, 740, 1750, 15
        922, 383, 322, 1970, 1295, 1592, 1287, 1829, 972, 1437, 1923]

# время анализа в миллисекундах -> в секундах
t_analysis = 0.2941 / 1000
```

In [82]:

```
def count_efficiency(t, a):
    res = np.exp((-1) * t * a)
    return res
```

In [83]:

```
# есть ли связь между сетями
ways_exist = np.zeros((50, 50), dtype=float)

for i, pair in enumerate(nets):
    ways_exist[pair[0] - 1][pair[1] - 1] = count_efficiency(t_analysis, coef[i])
```

In [84]:

```
# вероятности для каждой сети храним тут, заполняем динамически
probWays =  np.zeros((50, 50), dtype=float)

reachable_from_first_net = np.zeros(50, dtype=float)
reachable_from_first_net[0] = 1

# максимальная вероятность попасть в сеть
max_probs = np.zeros(50, dtype=float)
max_probs[0] = 1
```

In [85]:

```
def find_prob():
    for i in range(50):
        # идем по строкам
        for j in range(50):
            if ((ways_exist[i, j]>0) and (reachable_from_first_net[i]>0)):
                cur_prob = max_probs[i] * ways_exist[i, j]
                if (cur_prob > max_probs[j]):
                    max_probs[j] = cur_prob
                reachable_from_first_net[j] = 1
```

In [86]:

```
find_prob()
```

In [87]:

```
print("Искомая вероятность: ", max_probs[-1])
```

```
Искомая вероятность:  0.573922562672252
```

В этой задаче небольшие пояснения:

ways_exist - массив, в котором 0 стоит, если связи между сетями нет, а если она есть, то стоит положительное число, равное значению p(x).

reachable_from_first_net - достижима ли уже эта сеть из первой.

max_probs - для каждой сети храним только максимальную вероятность попадания в эту сеть.

Задача 3: Маршруты в безопасной сети (HARD)

Данные из условия:

In [2]:

```
nets = [[1, 2], [1, 3], [1, 4], [2, 6], [2, 3], [2, 4], [3, 5], [3, 6], [4, 6], [4, 7], [5,
        [6, 8], [7, 8], [7, 9], [8, 9], [7, 10], [8, 10], [9, 10]]

coefs = [1900.25857029435, 462.277027148576, 1213.68516708357, 971.9649374186, 1782.5979322
         912.9353303366829, 37.0072864964488, 1642.8143285905098, 889.4067287063881, 1230.8
         1583.8740748540702, 1843.6259414896099, 1476.41449162133, 352.532288989236, 811.41
         1870.93939821521, 1833.80887982682, 820.540413981891, 1787.29906182707]

times = [1.05789130478427, 1.352868132217, 1.81316649730376, 1.00986130066092, 1.1388908819
         1.19872174266149, 1.60379247919382, 1.27218792496996, 1.19881426776106, 1.01527392
         1.44509643228795, 1.93181457846166, 1.46599434167542, 1.41864946772751, 1.84622141
         1.20264735765039, 1.67213746847429]

max_time = 10
```

In [38]:

```
def count_efficiency(t, a):
    t = t/1000
    res = np.exp((-1) * t * a)
    return res
```

In [39]:

```
# есть ли связь между сетями
ways_exist = np.zeros((10, 10), dtype=float)

for i, pair in enumerate(nets):
    ways_exist[pair[0] - 1][pair[1] - 1] = 1 #count_efficiency(t_analysis, coef[i])
```

Вообще, в клетке ниже я использую то, что в задании связи идут всегда от сети с меньшим номером к сети с большим номером.

In [48]:

```
# backprop
# в массивах храним веса из матрицы ways_exist для ребер (или их последовательностей),
# которые ведут из указанной вершины до последней
# ways = как раз матрица с весами

def fill_weights(ways):
    con = []
    for i in range (9):
        con.append([])

    for i in reversed(range(9)):
        if (ways[i][9]):
            con[i].append(ways[i][9])
        for j in range(i+1, 9):
            if (ways[i][j]):
                for way in con[j]:
                    con[i].append(ways[i][j] * way)

    return con
```

In [55]:

```
def max_weighted_way(ways):
    cons = fill_weights(ways)
    #print(cons[0])
    max_prob = np.max(cons[0])
    return max_prob
```

In [56]:

```
def find_max_from_times(cur_times):
    ways_existt = np.zeros((10, 10), dtype = float)
    for i, pair in enumerate(nets):
        ways_existt[pair[0] - 1][pair[1] - 1] = count_efficiency(cur_times[i], coefs[i])
    #print(ways_exist)
    m_prob = max_weighted_way(ways_existt)
    return m_prob
```

In [121]:

```
def find_my_best_times():
    # начальное приближение, сумма даст как раз 10
    b_times = np.ones(20) * 1/2

    m_prob = find_max_from_times(b_times)

    # просто чтобы не сошлось на первой же итерации
    m_prob_prev = m_prob + 0.5

    # точность
    eps = 1e-30

    # шаг
    h = 0.0003

    while (abs(m_prob - m_prob_prev) > eps):
        m_prob_prev = m_prob
        for i in range(20):
            for j in range(20):
                if ((b_times[i] + h) < times[i]):
                    if (b_times[j] > h):
                        b_times[i] += h
                        b_times[j] -= h
                        if (find_max_from_times(b_times) > m_prob):
                            # откатываем обратно
                            b_times[i] -= h
                            b_times[j] += h
                        else:
                            m_prob = find_max_from_times(b_times)

    return m_prob, b_times
```

In [122]:

```
find_my_best_times()
```

Out[122]:

```
(0.05003742890446188,
 array([4.34650000e-01, 1.35286000e+00, 8.51420000e-01, 9.99999998e-06,
        9.99999998e-06, 1.36080000e-01, 1.16553000e+00, 9.99999998e-06,
        9.99999998e-06, 7.37710000e-01, 2.58940000e-01, 2.00000000e-05,
        5.77170000e-01, 9.36580000e-01, 9.04000000e-01, 9.99999998e-06,
        9.99999998e-06, 7.11910000e-01, 1.20264000e+00, 7.30430000e-01]))
```

Пару слов о том, для чего нужны все функции выше:

- find_my_best_times(): ищет оптимальный набор времен (и вероятности соответственно). Для начала все времена равны и их сумма равна 10. Далее, мы для каждой пары времен одно из них уменьшаем на h, а другое увеличиваем на h (если не превосходим при этом лимит времени). Если вероятность уменьшилась, оставляем как есть. В противном случае производим обратные изменения.
- find_max_from_times(): ищет по набору времен вероятность при помощи следующей функции:
- max_weighted_way(): тут мы ищем максимум из массива вероятностей для самой первой сети. Сам массив получаем из следующей функции:
- fill_weights(): мы постепенно для каждой вершины (вершина - это номер сети) заполняем массив вероятностей - это вероятности добраться до заключительной (10-й) сети. Каждая вероятность соответствует какому-либо маршруту до 10-й сети.

Поскольку результат из функции выше получился довольно большой, то ниже будет представлен еще один итерационный подход, работающий более точно.

Проблема первого подхода заключалась в том, что h остается неизменным, причем оно довольно большое (если же его сразу сделать маленьким, то работать будет безумно долго). Таким образом, h надо постепенно уменьшать. Итак, если в ходе одной итерации по всем парам связей вероятность уменьшилась, то считаем, что это хорошо и можем h увеличить. Если же этого не произошло, то мы, наоборот, уменьшаем h.

По поводу коэффициентов, на сколько увеличиваем h: подбирались экспериментально.

Итак, в подходе будет несколько стадий (3 штуки):

Стадия 1: идем по парам связей и, как и ранее, увеличиваем одно время на h и уменьшаем другое. Если вероятность уменьшилась - оставляем так. Если не уменьшилась - откатываем назад. Кроме того, если в ходе одной итерации по всем парам связей вероятность уменьшилась, то считаем, что это хорошо и можем h увеличить. Если же этого не произошло, то мы, наоборот, уменьшаем h.

In [194]:

```
def find_my_best_times_PAIRS():
    # начальное приближение, сумма даст как раз 10
    b_times = np.ones(20) * 1/2

    m_prob = find_max_from_times(b_times)

    # просто чтобы не сошлось на первой же итерации
    m_prob_prev = m_prob + 0.5

    # точность
    eps = 1e-30

    # шаг, его будем постепенно менять
    h = 0.5

    for i in range(6000):
        if (h < 1e-45):
            return m_prob, b_times

        m_prob_prev = m_prob
        for i in range(20):
            for j in range(20):
                if ((b_times[i] + h) < times[i]):
                    if (b_times[j] > h):
                        b_times[i] += h
                        b_times[j] -= h
                        if (find_max_from_times(b_times) > m_prob):
                            # откатываем обратно
                            b_times[i] -= h
                            b_times[j] += h
                        else:
                            m_prob = find_max_from_times(b_times)
        if (m_prev > m_prob):
            h *= 1.12
        else:
            h *= 0.88

    return m_prob, b_times
```

In [195]:

```
prob_1, bests_1 = find_my_best_times_PAIRS()
print("Вероятность на стадии 1: ", prob_1)
```

```
Вероятность на стадии 1:  0.047888280068922255
```

Стадия 2: идем по тройкам, то есть рассматриваем три связи и:

- для одной уменьшаем время на h (если не уйдем при этом в ноль)
- для двух других увеличиваем времена на h/2 для каждой (если влезаем в лимит)

h также меняется в зависимости от того, хороший ли результат был получен на итерации или плохой.

In [198]:

```
def find_my_best_times_TRIPLES(b_times):
    m_prob = find_max_from_times(b_times)

    # просто чтобы не сошлось на первой же итерации
    m_prob_prev = m_prob + 0.5

    # точность
    eps = 1e-30

    # шаг
    h = 0.01

    for i in range(200):
        if (h < 1e-45):
            return m_prob, b_times, h

        m_prob_prev = m_prob
        for i in range(20):
            for j in range(20):
                for k in range(20):
                    if ((b_times[i] + h/2) < times[i] and (b_times[j] + h/2) < times[j]):
                        if (b_times[k] > h):
                            b_times[i] += h/2
                            b_times[j] += h/2
                            b_times[k] -= h
                            if (find_max_from_times(b_times) > m_prob):
                                # откатываем обратно
                                b_times[i] -= h/2
                                b_times[j] -= h/2
                                b_times[k] += h
                            else:
                                m_prob = find_max_from_times(b_times)
        if (m_prev > m_prob):
            h *= 1.1
        else:
            h *= 0.9

    return m_prob, b_times
```

In [199]:

```
prob_2, bests_2 = find_my_best_times_TRIPLES(bests_1)
print("Вероятность на стадии 2: ", prob_2)
```

```
Вероятность на стадии 2:  0.04556055339139869
```

Стадия 3: идем по четверкам, то есть рассматриваем четыре связи и:

- для двух из них уменьшаем время на h/2 для каждой (если не уйдем при этом в ноль)
- для двух других увеличиваем времена на h/2 для каждой (если влезаем в лимит)

In [200]:

```
def find_my_best_times_QUARTETS(b_times):

    m_prob = find_max_from_times(b_times)

    # просто чтобы не сошлось на первой же итерации
    m_prob_prev = m_prob + 0.5

    # точность
    eps = 1e-30

    # шаг
    h = 0.01

    for i in range(100):
        if (h < 1e-45):
            return m_prob, b_times, h

        m_prob_prev = m_prob
        for i in range(20):
            for j in range(20):
                for k in range(20):
                    for t in range(20):
                        if ((b_times[i] + h/2) < times[i] and (b_times[j] + h/2) < times[j]
                            if (b_times[k] > h/2 and b_times[t] > h/2):
                                b_times[i] += h/2
                                b_times[j] += h/2
                                b_times[k] -= h/2
                                b_times[t] -= h/2
                                if (find_max_from_times(b_times) > m_prob):
                                    # откатываем обратно
                                    b_times[i] -= h/2
                                    b_times[j] -= h/2
                                    b_times[k] += h/2
                                    b_times[t] += h/2
                                else:
                                    m_prob = find_max_from_times(b_times)
        if (m_prev > m_prob):
            h *= 1.1
        else:
            h *= 0.9

    return m_prob, b_times
```

In [202]:

```
prob_3, bests_3 = find_my_best_times_QUARTETS(bests_2)
print("Вероятность на стадии 3: ", prob_3)
```

```
Вероятность на стадии 3:  0.04346682175943892
```

Важно

На сайте в тесте я сначала ввела не 0.0435, а 0.0434 (ниже представлено кратко, откуда это взялось, последняя функция для поиска по четверкам была написана иначе). Но решение, с помощью которого это было получено, изменила, так что стало 0.0435. На сайте тот ответ сбросила и ввела 0.0435, тоже засчиталось.

Все-таки, ответ выше (0.04346) честнее округлить в большую сторону.

In [187]:

```
find_my_best_times_QUARTETS0(bests25)
```

Out[187]:

```
(0.043418502038081765,
 [0.5510730869999945,
  1.3497550400000788,
  0.8826114539999856,
  -0.004310000000009848,
  -0.004440000000010064,
  0.01576707700000002,
  1.1942043700000178,
  8.098333859951621e-06,
  9.999999989851459e-06,
  1.1987947300000128,
  0.35411464499999884,
  0.2672205020000001,
  0.8209176810000004,
  1.0336536500000015,
  0.03582829710000018,
  -0.003765048125979772,
  1.935428319832521e-06,
  0.5450140260000282,
  1.2026376600000002,
  0.5609027950000066],
 1303.9238970822203)
```

Вот тут 0.043418 можно было бы округлить до 0.0434 (тут были другие коэффициенты + еще одна более точная дополнительная проверка, но работало намного дольше, так что оставлен был вариант, который представлен выше).