

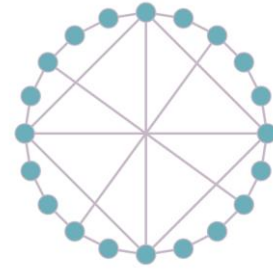
# Laboratoria z Technologii Sieciowych

Damian Kokot

## 1. Zadanie 1

### 1.1. Tworzenie modelu sieci

W tym zadaniu chciałbym spróbować stworzyć model jak najbardziej symetrycznej sieci i go zbadać pod kątem przepustowości. Stworzę do tego funkcję w języku *Python*.



```
import networkx as nx

graphSize = 20
G = nx.cycle_graph(graphSize)

# Dodawanie krawędzi sąsiedzkich
for index in range(graphSize - 1):
    G.add_edge(index, index + 1)
# Dodawanie krawędzi przekątnych
diagonals = [(source, source + 10) for source in [0, 2, 5, 7]]
diagonals += [(source, (source + 5) % graphSize) for source in [0, 12, 17, 2, 6, 5, 10, 15]]

for (source, target) in diagonals:
    G.add_edge(source, target)
```

Następnie wyznaczę macierz N losując wartości z pewnego przedziału.

```
import numpy as np
import random

# Generowanie najkrótszych ścieżek
graphSize = G.number_of_nodes()
N = np.zeros((graphSize, graphSize))
for i in range(graphSize):
    for j in range(graphSize):
        if i != j:
            N[i][j] += random.randrange(30, 50)
```

Kolejnym krokiem jest ustalenie funkcji a(e) i c(e). Starając sobie wyobrazić model sieci założymy że mamy dwa rodzaje łącza o prędkościach 100Mb/s i 300Mb/s. Założymy, że łącza szybsze stosujemy w 30% przypadków, tam gdzie model ma największe obciążenie

Dodałem również funkcję, która po obliczeniu ścieżki pomiędzy punktami uaktualnia poszczególne obciążenia, tzn. jeżeli między punktami „i” i „j” pakiety muszą przejść przez krawędź „k”, łączy jest obciążone nieco mocniej.

```
def getSummaric(graph, N):
    graphSize = graph.number_of_nodes()
    NSumaric = np.zeros((graphSize, graphSize), dtype=int)
    fastEdges = {}

    for sourceIndex, sourceValues in enumerate(N):
        for targetIndex, weight in enumerate(sourceValues):
            nodeFrom = min(sourceIndex, targetIndex)
            nodeTo = max(sourceIndex, targetIndex)

            path = nx.shortest_path(graph, sourceIndex, targetIndex)
            for pathIndex in range(len(path) - 1):
                NSumaric[path[pathIndex]][path[pathIndex + 1]] += weight

            if N[nodeFrom][nodeTo] >= 8 / 9 * max(map(max, N)):
                fastEdges[(nodeFrom, nodeTo)] = N[nodeFrom][nodeTo]
    return NSumaric, fastEdges
```

Dzięki temu mogłem uwzględnić te wartości w tworzeniu ogólnego obciążenia, wtedy niech:

```
def c(edge):
    if edge in fastEdges.keys():
        return 3 * 10**8
    else:
        return 1 * 10**8

def a(edge, NSumaric):
    nodeFrom = min(edge)
    nodeTo = max(edge)

    return int(c(edge) / NSumaric[nodeFrom][nodeTo]) if NSumaric[nodeFrom][nodeTo] else 0
```

## 2. Zadanie 2

### 2.1. Mierzenie niezawodności sieci

Do tego celu postanowiłem utworzyć nową funkcję, która dla pewnych ustalonych parametrów, tj. wielkości pakietów, grafu i macierzy obciążeń oblicza obciążenie na każdej z krawędzi, po czym oblicza opóźnienie średnie opóźnienie. W dalszym etapie testowania otrzymywałem bardzo dziwne wyniki, tj. dla pakietu większego niż 1b średni czas był mniejszy od 0. Dlatego postanowiłem przeanalizować działanie algorytmu i odkryłem, że jednostki się nie zgadzają. Po przeanalizowaniu wzoru, jego zasady działania zauważyłem, że

$$\sum_{e \in E} \frac{\frac{b}{s}}{\frac{b}{s} * b - \frac{b}{s}} = \sum_{e \in E} \frac{1}{b - 1}$$

Po czym uświadomiłem sobie, że mając maksymalną przepustowość (a(e)) możemy zrobić:

$$\frac{\sum_{e \in E} \frac{m}{a(e)}}{G}$$

Gdzie „m” i „G” są ustalone w zadaniu. Dla sprawdzenia jednostek:

$$\frac{b}{\frac{b}{s}} = b * \frac{s}{b} = s$$

Implementacja:

```
def averageWaitTime(G, N, averageSize):
    if not nx.is_connected(G):
        return -1
    sumOfIntensity = sum(sum(N))
    NSumaric, _ = getSummaric(G, N)
    return sum([averageSize / a(e, NSumaric) for e in G.edges]) / sumOfIntensity
```

Po przetestowaniu modelu otrzymywałem dosyć sensowne wyniki.

Następnie utworzyłem funkcję do testowania:

```
def getRandom(edgeIndex):
    rangeOfRandom = 1000
    return (random.random() * edgeIndex ** 2) % rangeOfRandom / rangeOfRandom

def reliability(graph, N, Tmax, p, packetSize=10**4, attempts=100):
    passedAttempts = 0
    delaysTotal = averageWaitTime(graph, N, packetSize)
    inConnectedCases = 0

    for _ in range(attempts):
        G = nx.Graph(graph)

        # Remove edges
        for edgeIndex, edge in enumerate(G.edges):
            if getRandom(edgeIndex) > p:
                G.remove_edge(*edge)
            if not nx.is_connected(G):
                break

        # Check if delay in each edge is higher or smaller than current
        waitTime = averageWaitTime(G, N, packetSize)
        if nx.is_connected(G) and waitTime < Tmax:
            passedAttempts += 1
            delaysTotal += waitTime
        elif not nx.is_connected(G):
            inConnectedCases += 1

    delaysAvg = delaysTotal / passedAttempts if passedAttempts else delaysTotal
    if passedAttempts != attempts:
        inConnectedCases = inConnectedCases / (attempts - passedAttempts) * 100
    else:
        inConnectedCases = 0
    return passedAttempts / attempts * 100, delaysAvg, inConnectedCases
```

Uruchamia program określoną ilość razy i losowo usuwa połączenia, po czym testuje model pod kątem niezawodności, czyli:

- Czy można przejść z punktu A do B
- Czy czas jest mniejszy od  $T_{\max}$

### 3. Obserwacje

Obserwacji dokonywano przy ustaleniu  $p=50\%$  i  $T_{\max}=0,000135$

#### 3.1. Zwiększanie macierzy natężeń

Aby zrealizować to zadanie należy stopniowo zwiększać ilość pakietów w macierzy N.

range (x, y)	Output
x=0, y=30	Reliability: 97.00% and average time passed: 0.000122 where in 98.00% cases there was inconnectCases
x=30, y=60	Reliability: 86.00% and average time passed: 0.000125 where in 92.86% cases there was inconnectCases
x=60, y=90	Reliability: 86.00% and average time passed: 0.000120 where in 92.86% cases there was inconnectCases
x=90, y=120	Reliability: 89.00% and average time passed: 0.000127 where in 81.82% cases there was inconnectCases
x=120, y=150	Reliability: 76.00% and average time passed: 0.000128 where in 77.27% cases there was inconnectCases
x=1000, y=1500	Reliability: 15.00% and average time passed: 0.013410 where in 1.27% cases there was inconnectCases

Jak przy wyborze  $T_{\max}$  kierowałem się średnim czasem, który ustaliłem na 0,000135. Jak widzimy, przy zwiększaniu macierzy ilość przypadków otrzymania „Timeouta” zwiększała się (procentowa ilość rozspójnienia sieci zmniejsza się). Oczywiście zmniejszyła się również niezawodność sieci ze względu na ilość połączeń. Po ostatnim teście możemy wywnioskować, że mając duże obciążenie bardzo łatwo doprowadzić do „connection timeout”.

#### 3.2. Zwiększanie przepustowości łącza


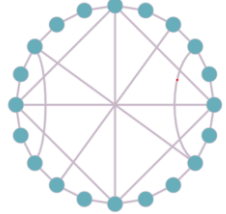
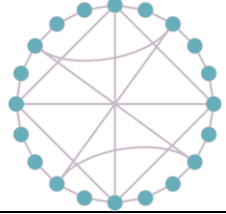

Tu postanowiłem zmieniać tylko funkcję c(e). Niech „s” oznacza wolne łącze, a „f” szybkie. Zaznaczmy, że wielkość pakietu to 10kb

Prędkość ( $\frac{Mb}{s}$ )	Output
s=100, f=300	Reliability: 91.00% and average time passed: 0.000119 where in 89.00% cases there was inconnectCases
s=200, f=300	Reliability: 92.00% and average time passed: 0.000064 where in 100.00% cases there was inconnectCases
s=300, f=300	Reliability: 92.00% and average time passed: 0.000048 where in 100.00% cases there was inconnectCases
s=100, f=1000	Reliability: 94.00% and average time passed: 0.000107 where in 100.00% cases there was inconnectCases
s=250, f=1000	Reliability: 93.00% and average time passed: 0.000040 where in 100.00% cases there was inconnectCases
s=500, f=1000	Reliability: 94.00% and average time passed: 0.000025 where in 100.00% cases there was inconnectCases
s=1000, f=3000	Reliability: 94.00% and average time passed: 0.000038 where in 100.00% cases there was inconnectCases
$s=10\frac{Gb}{s}$ , $f=100\frac{Gb}{s}$	Reliability: 93.00% and average time passed: 0.000001 where in 100.00% cases there was inconnectCases

Nie zmieniając  $T_{\max}$  możemy zauważyć, że prędkości maleją wraz ze zwiększeniem przepustowości. Warty zaznaczenia jest fakt, że pomiary były przeprowadzane niejednokrotnie, a wynik jest tylko uśrednieniem obrazu. Problemem był generator liczb losowych, gdyż okazało się, że prawdopodobieństwo wylosowania liczby większej od 0,9 z przedziału [0, 1] wynosiło w niektórych przypadkach nawet 80%. Poprawiłem to dodając swoją funkcję dla „urandomizowania” wyników. To pozwoliło na większą stabilizację wyników.

### 3.3. Zwiększanie ilości połączeń w sieci

W tym zadaniu postanowiłem dodać krawędzie do grafu, tak, aby najlepiej ulepszyć graf. Zmieniłem również  $p=30\%$  aby zobaczyć lepsze wyniki rozpójnienia się sieci.

Model	Output
	Reliability: 0.10% and average time passed: 0.000101 where in 100.00% cases there was inconnectCases
	Reliability: 36.00% and average time passed: 0.000110 where in 100.00% cases there was inconnectCases
	Reliability: 29.00% and average time passed: 0.000108 where in 100.00% cases there was inconnectCases
	Reliability: 45.00% and average time passed: 0.000120 where in 100.00% cases there was inconnectCases

Dodane krawędzie mają znaczenie dla poprawności działania sieci. Zauważmy, że im bardziej uzależnimy jeden wierzchołek od całości tym lepiej dla naszego modelu, bo trudniej jest rozpójnić graf sieci. Ciekawym faktem jest, że poprzez dodanie krawędzi nie łączącym kluczowy punkt jak w teście 1 a jak w teście 2 uzyskujemy większą spójność.

Wniosek nasuwa się taki, że mając wierzchołek st. 2 prawdopodobieństwo wyeliminowania go z grafu jest równe  $\frac{1}{30} * \frac{1}{29}$ , a dla wierzchołka st. 3 prawdopodobieństwo wyeliminowania go to  $\frac{1}{30} * \frac{1}{29} * \frac{1}{28}$ , a zatem nieco mniejsze.

### 4. Wnioski

Budowanie modelu sieci to nie prosta sprawa. Na poprawność działania może wpływać wiele czynników. Jednak bardzo ciekawym doświadczeniem jest wyciągnięcie wniosków jak ważne jest uniezależnianie całości od jednego punktu, tzn. gdyby wszystkie pakiety z Europy miałyby wędrować przez jeden punkt np. w Londynie, a wszystkie pakiety w Ameryce przez np. Waszyngton, to w przypadku awarii jednego z ośrodków albo połączenia, moglibyśmy stracić całkowicie łączność z Stanami Zjednoczonymi.