Lista 2 - modelowanie niezawodności sieci

Paweł Kubala

1 Problem

Rozważmy model sieci $S =< G, H >$. Przez $N = [n(i, j)]$ będziemy oznaczać macierz natężeń strumienia pakietów, gdzie element $n(i, j)$ jest liczbą pakietów przesyłanych (wprowadzanych do sieci) w ciągu sekundy od źródła $v(i)$ do ujścia $v(j)$.

1. Zaproponuj topologię grafu G ale tak aby żaden wierzchołek nie był izolowany oraz aby: |V|=20, |E|<30. Zaproponuj N oraz następujące funkcje krawędzi ze zbioru H: funkcję przepustowości 'c' (rozumianą jako maksymalną liczbę bitów, którą można wprowadzić do kanału komunikacyjnego w ciągu sekundy), oraz funkcję przepływu 'a' (rozumianą jako faktyczną liczbę pakietów, które wprowadza się do kanału komunikacyjego w ciągu sekundy). Pamiętaj aby funkcja przeplywu realizowała macierz N oraz aby dla każdego kanału 'e' zachodziło: $c(e) > a(e)$.

2. Niech miarą niezawodności sieci jest prawdopodobieństwo tego, że w dowolnym przedziale czasowym, nierozspójniona sieć zachowuje $T < T_{max}$, gdzie: $T = \frac{1}{G} * \sum_e (\frac{a(e)}{\frac{c(e)}{m} - a(e)})$, jest średnim opóźnieniem pakietu w sieci, $\sum_e$ oznacza sumowanie po wszystkich krawędziach 'e' ze zbioru E, 'G' jest sumą wszystkich elementów macierzy natężeń, a 'm' jest średnią wielkością pakietu w bitach. Napisz program szacujący niezawodność takiej sieci przyjmując, że prawdopodobieństwo nieuszkodzenia każdej krawędzi w dowolnym interwale jest równe 'p'. Uwaga: 'N', 'p', 'T_{max}' oraz topologia wyjsciowa sieci są parametrami.

3. Przy ustalonej strukturze topologicznej sieci i dobranych przepustowościach stopniowo zwiększaj wartości w macierzy natężeń. Jak będzie zmieniać się niezawodność zdefiniowana tak jak punkcie poprzednim ($Pr[T < T_{max}]$).

4. Przy ustalonej macierzy natężeń i strukturze topologicznej stopniowo zwiększaj przepustowości. Jak będzie zmieniać się niezawodność zdefiniowana tak jak punkcie poprzednim ($Pr[T < T_{max}]$).

5. Przy ustalonej macierzy natężeń i pewnej początkowej strukturze topologicznej, stopniowo zmieniaj topologię poprzez dodawanie nowych krawędzi o przepustowościach będących wartościami średnimi dla sieci początkowej. Jak będzie zmieniać się niezawodność zdefiniowana tak jak punkcie poprzednim ($Pr[T < T_{max}]$).

2 Przebieg symulacji

2.1 Topologia grafu

Najważniejsze na początek jest stworzenie grafu. W tym celu użyłem biblioteki *networkx*, która zawiera wszystkie potrzebne narzędzia do prostego tworzenia grafów oraz operowania na nich. Dodatkowo do wyświetlania otrzymanych grafów użyłem biblioteki *pyplot*.

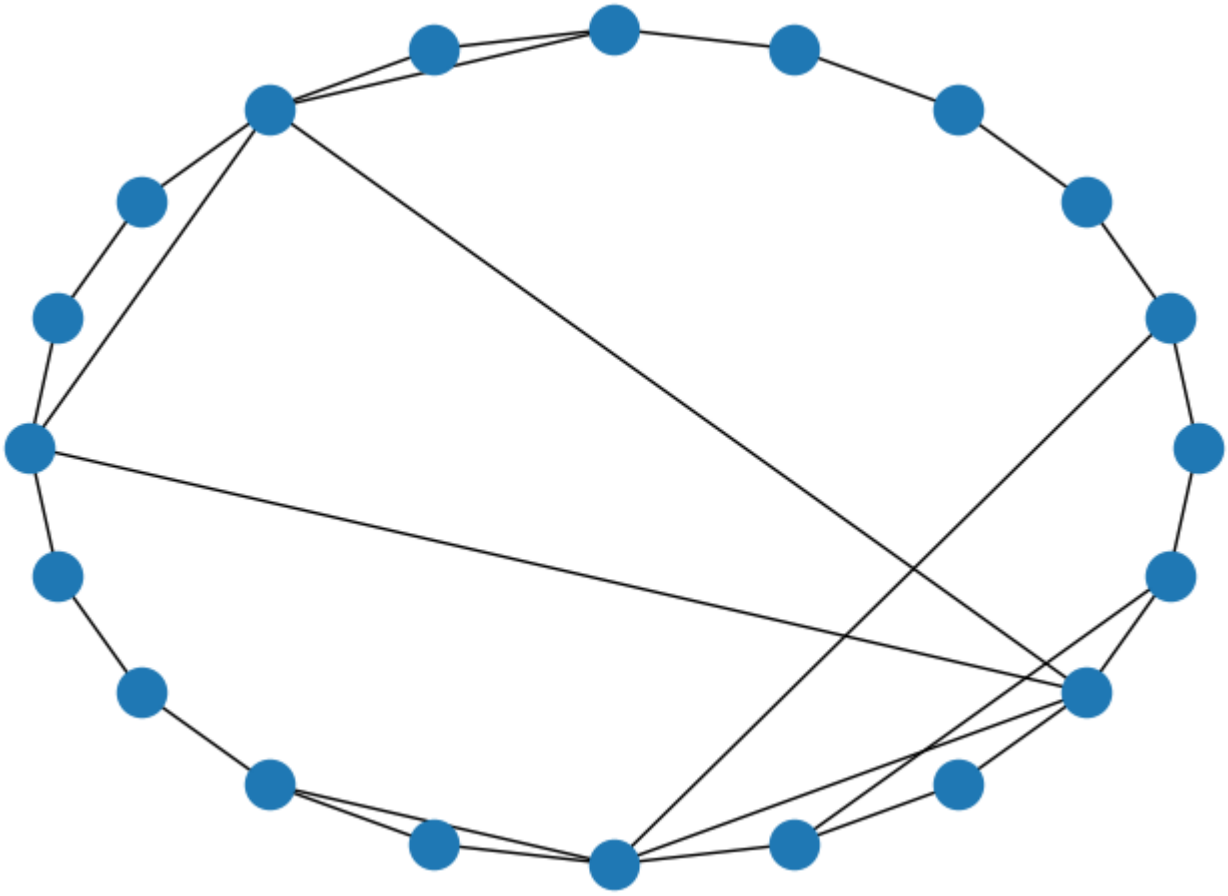
2.1.1 Graf cykliczny

Dosyć oczywistym wyborem na sam początek był graf cykliczny o $|V| = 20$ - posiada on 20 krawędzi, więc chcąć zmaksymalizować niezawodność pozostałe 9 krawędzi dodałem losowo.
Kod stworzenia takiego grafu prezentuje się następująco:

```
def czy_krawedz(G, i, j):
    return i in G.neighbors(j)

def tworz_graf_cykliczny(v, e, p, c):
    G = nx.cycle_graph(v)
    for i in range(e-v):
        krawedz1 = random.randint(0,v-1)
        krawedz2 = random.randint(0,v-1)
        while(czy_krawedz(G,krawedz1, krawedz2)):
            krawedz2 = random.randint(0,v-1)
        G.add_edge(krawedz1, krawedz2)
    nx.set_edge_attributes(G, p, 'p')
    nx.set_edge_attributes(G, c, 'c')
    nx.set_edge_attributes(G, 0, 'a')
    return G
```

Otrzymany graf po wywołaniu powyższej funkcji prezentuje się następująco:

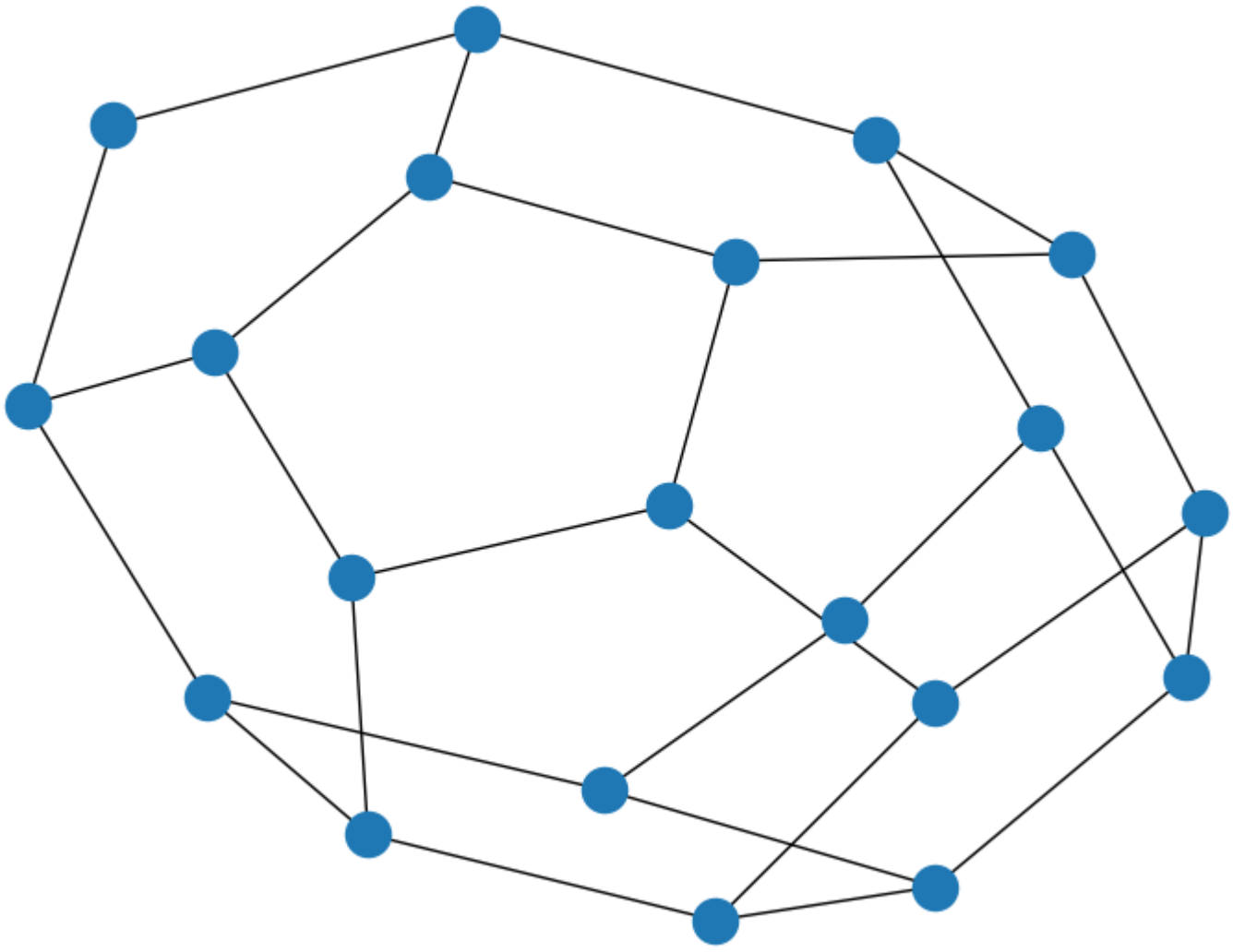


2.1.2 Graf Dwunastościenny

W powyższym przykładzie widzimy, że w wielu przypadkach wystarczy awaria w 2 liściach, aby rozspójnić ten graf. Lepszym wyborem będzie graf, który będzie miał jak najwięcej wierzchołków stopnia > 2. Graf dwunastościenny to najmniejszy graf o 20 wierzchołkach, który jest 3 stopnia. Posiada on 30 krawędzi, więc jedną usunąłem. Tym sposobem mamy graf spójny o 20 wierzchołkach i 29 krawędziach.
Kod stworzenia takiego grafu wygląda następująco:

```
G = nx.dodecahedral_graph()
for e in G.edges:
    G.remove_edge(*e)
    break
nx.draw_(G)
plt.show()
```

Otrzymany graf po wywołaniu powyższej funkcji prezentuje się następująco:



2.2 Tworzenie funkcji

2.2.1 Tworzenie tablicy natężeń

```
def tworz_n(size, max_n):
    " size - wielkość tablicy, max_n - największy element tablicy "
    N = np.zeros((size,size),dtype=int)
    for i in range(size):
        for j in range(size):
            if i != j:
                N[i][j] = random.randint(0,max_n)
    return N
```

2.2.2 Tworzenie funkcji $a(e)$

Wiemy, że w przeciągu sekundy z $v(i)$ do $v(j)$ wysyłamy $N[i][j]$ pakietów. Załóżmy, że chcemy to zrobić tak, aby te pakiety pokonały jak najkrótszą drogą. Zauważmy, że aby pakiety doszły z $v(i)$ do $v(j)$, to wierzchołki znajdujące się w najkrótszej ścieżce z $v(i)$ do $v(j)$ muszą być w stanie przetworzyć te pakiety, ale mają też swoje pakiety do wysłania. Czyli $a(e)$ to suma wszystkich $N[i][j]$, takich, że e znajduje się w najkrótszej ścieżce z $v(i)$ do $v(j)$.

```
def tworz_a(G, N):
    "G - Graf, N - tablica natężeń"
    nx.set_edge_attributes(G,0,'a')
    for i in range(len(N)):
        for j in range(len(N[i])):
            min_path = nx.shortest_path(G,i,j)
            for k in range(len(min_path)-1):
                G[min_path[k]][min_path[k+1]]['a'] += N[i][j]
```

2.2.3 Tworzenie funkcji T

```
def T(C, N, m):
    "C - graf, N - tablica natężeń, m - średnia wielkość pakietu w bitach"
    G = N.sum()
    sum2 = np.sum([C.get_edge_data(*e).get('a') / (C.get_edge_data(*e).get('c')/m - C.get_edge_data(*e).ge
    return (1/G*sum2)
```

2.3 Testowanie niezawodności modeli za pomocą metody Monte Carlo

Mając już przykładowe grafy i funkcje możemy przejść do testów. Metoda Monte Carlo polega na tym, że n razy usuwamy losowe krawędzie z prawdopodobieństwem $1 - p$:

```
def usun_losowe_krawedzie(G):
    "funkcja pomocnicza do usuwania losowych krawędzi z grafu na podstawie niezawodności"
    g = nx.Graph(G)
    for e in g.edges:
        if random.random() > g.get_edge_data(*e).get('p'):
            g.remove_edge(*e)
    return g
```

a następnie sprawdzamy, czy otrzymany graf $G^{'}$ posiada następujące własności:

a) jest spójny;

b) $(\forall e \in V^{'})(a(e) < c(e))$

Zliczamy ile z n wygenerowanych grafów posiada takie właściwości i naszą niezawodność określamy jako $\frac{udane}{n} * 100\%$, natomiast średnie opóźnienie to iloraz sumy opóźnień wszystkich "udanych"do liczby wszystkich "udanych".
Kod do algorytmu testującego wygląda następująco:

```
def testuj_model(G, N, powtorzenia = 1000):
    "G - graf, N - tablica natężeń, powtorzenia - ile mamy sprawdzić grafów"
    t = []
    for i in range(powtorzenia):
        g = usun_losowe_krawedzie(G)
        if not nx.is_connected(g):
            continue
        tworz_a(g, N)
        for e in g.edges():
            if g.get_edge_data(*e).get('a') >= g.get_edge_data(*e).get('c'):
                break
        else:
            t.append(T(g,N,1))
    if len(t) == 0:
        print("Sukces w 0%")
        return False
    print("Sukces w ",len(t)/powtorzenia * 100,"%, średnie opóźnienie: ",sum(t)/len(t),"s")
    return t
```

Przetestujmy teraz modele grafów z podpunktu 1 przyjmując następujące parametry: $p = 0.9$, $N[i][j] <= 30$, $c = 8192$ dla liczby testów równej 10^5:

```
N = tworz_n(20,30)
G = nx.dodecahedral_graph()
for e in G.edges:
    G.remove_edge(*e)
    break
nx.set_edge_attributes(G,0.9,'p')
nx.set_edge_attributes(G,0,'a')
nx.set_edge_attributes(G,8192, 'c')
G2 = tworz_graf_cykliczny(20,29,0.9,8192)
print("Wyniki dla grafu dwunastościennego:")
testuj_model(G,N)
print("Wyniki dla grafu cyklicznego:")
testuj_model(G2,N)
```

1. Wyniki dla grafu dwunastościennego:
   Sukces w 96.39999999999999 %, średnie opóźnienie: 0.0004085023363858766s

2. Wyniki dla grafu cyklicznego:
   Sukces w 84.7 %, średnie opóźnienie: 0.00046614154689356754s

2.3.1 Dodanie warunku $T < T_{max}$

Nasza metoda potrafi już testować graf od części technicznej i wyliczać średnie opóźnienie. Chcemy, aby dodatkowo sprawdzała, które z wyliczonych T spełniają nam równanie $T < T_{max}$:

```
def niezawodnosc(G, N, Tmax, p=0.95, count=10000):
    g = nx.Graph(G)
    nx.set_edge_attributes(g,p,'p')
    t = testuj_model(g, N,count) or [1]
    c = 0
    for d in t:
        if d < Tmax:
            c+=1
    print("niezawodność w ", c/len(t)*100,"%")
    return c/len(t)*100
```

Dla $T_{max} = 0.0005$ otrzymujemy następujące wartości:

1. Wyniki dla grafu dwunastościennego:
   sukces w 95.72%, średnie opóźnienie: 0.0003976026733433603s
   niezawodność w 97.02256581696615%
2. Wyniki dla grafu cyklicznego: Sukces w 81.84%, średnie opóźnienie: 0.00048071360147955596s
   niezawodność w 71.07771260997067%

Jak mogliśmy się spodziewać niezawodność grafu dwunastościennego jest dużo większa, niż niezawodność grafu cyklicznego. Wynika to z faktu, że w naszej wersji grafu dwunastościennego tylko 2 wierzchołki mają stopień równy 2.

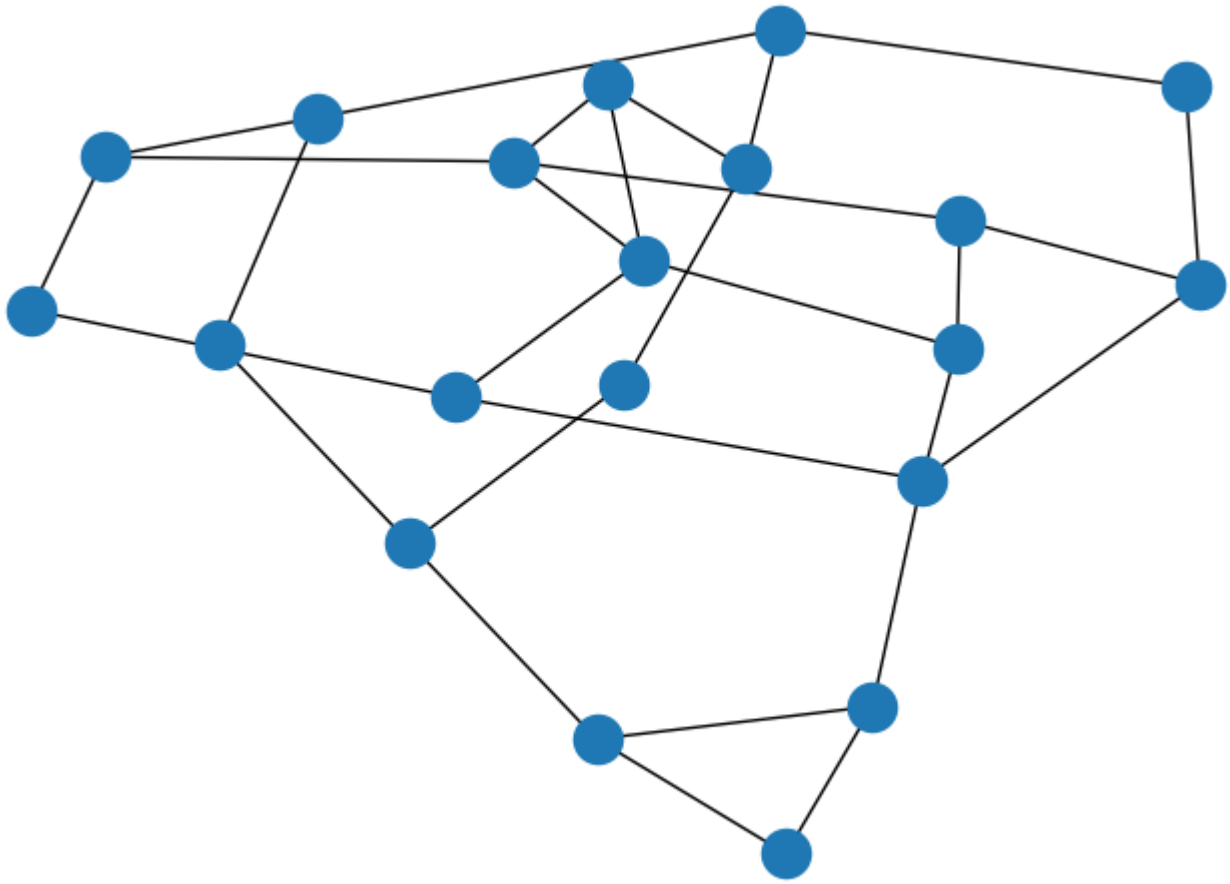
2.3.2 Generowanie najlepszego grafu

Skoro możemy już testować niezawodność każdego grafu, to mamy możliwość napisania programu, który znajdzie nam najlepszy graf na podstawie uzyskanego sukcesu. Kod algorytmu szukającego wygląda następująco:

```
def generuj_najlepszy_graf(v,e,p=0.95,c=1024, count = 1000, count2=1000):
"v - liczba wierzchołków, e - liczba krawędzi, c - przepustowość"
"p - prawdopodobieństwo na nieuszkodzenie krawędzi"
"count - ile grafów chcemy wygenerować, count2 - wielkość testu dla pojedynczego grafu"
    print("Start")
    Grafy = []
    def izomorficzny(g1):
        for g in Grafy:
            if(nx.is_isomorphic(g,g1)):
                return True
        return False
    N = tworz_n(v, 30)
    min_sukces = 0
    for i in range(count):
        g = generuj_losowy_graf(v,e,p,c)
        while izomorficzny(g):
```

```
            g = generuj_losowy_graf(v,e,p,c)
        print("graf wygenerowany")
        sc = testuj_model(g,N)
        print("Graf ",i+1,"najlepszy sukces: ",min_sukces, "%")
        if sc > min_sukces:
            min_g, min_sukces = g, sc
            nx.write_gexf(g, "graf.gexf")
    return min_g
```

Niestety złożoność algorytmu testującego jest zbyt duża, aby wygenerować wszystkie możliwe grafy, dlatego ograniczyłem się do generowania najlepszego losowego grafu przez **12 godzin**.
Otrzymany graf wygląda następująco:



Sukces w 91.9%, średnie opóźnienie: 0.00044496337384440155s
niezawodność w 100.0%
Otrzymany graf cechuje się niższym sukcesem od dwunastościennego oraz maksymalną niezawodnością.

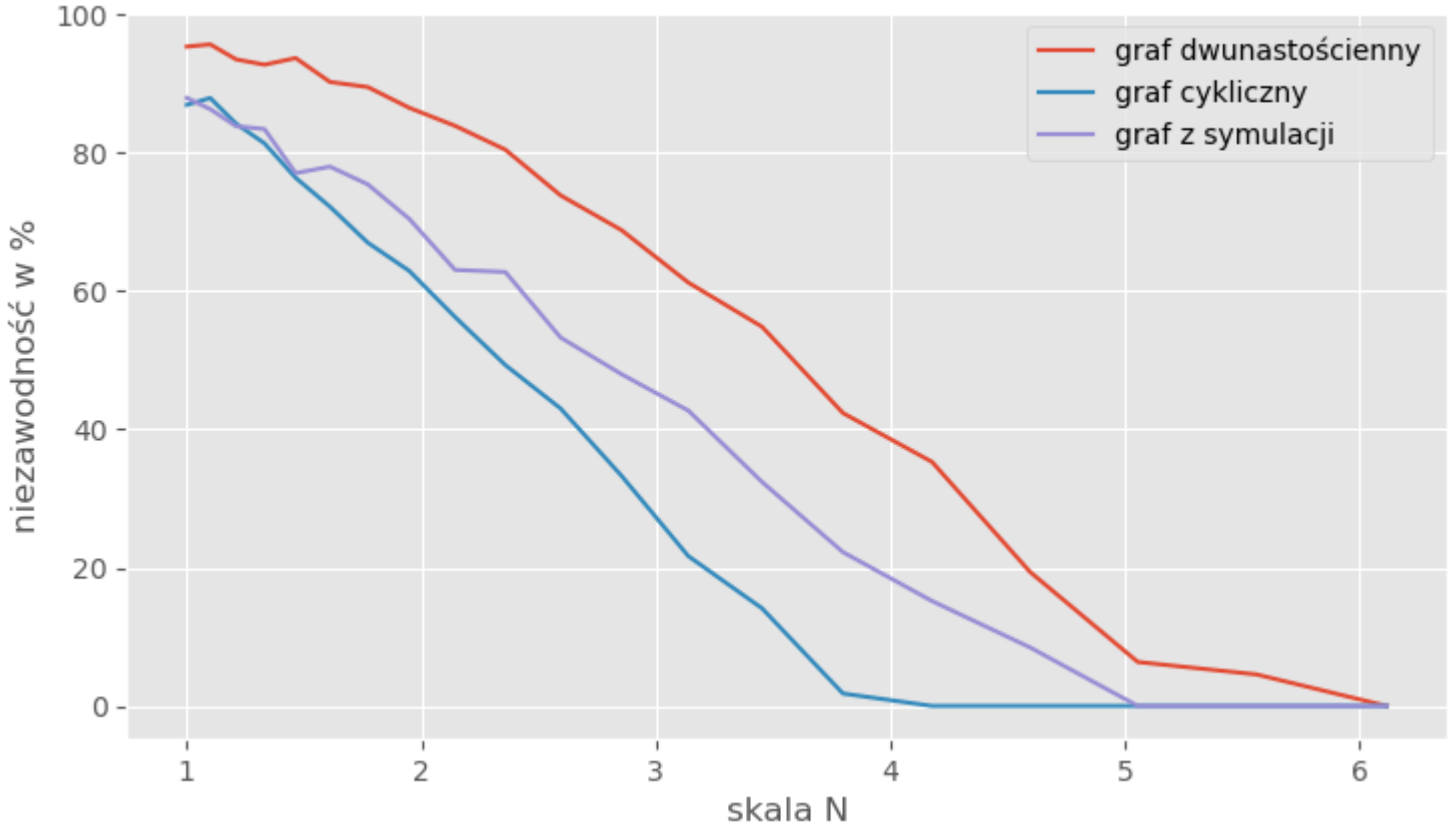
2.4 Zwiększanie wartości

2.4.1 Tablica natężeń

```
def zwieksz_n(N, stala):
    "N - tablica natężeń, stala - stała, przez którą skalujemy tablicę"
    for i in range(len(N)):
```

```
        for j in range(len(N[i])):
            N[i][j] = int(N[i][j] * stala)
```

Przeskalujmy sobie 20 razy naszą tablicę natężeń przez 1.1 i zobaczmy, jakie otrzymamy wyniki:

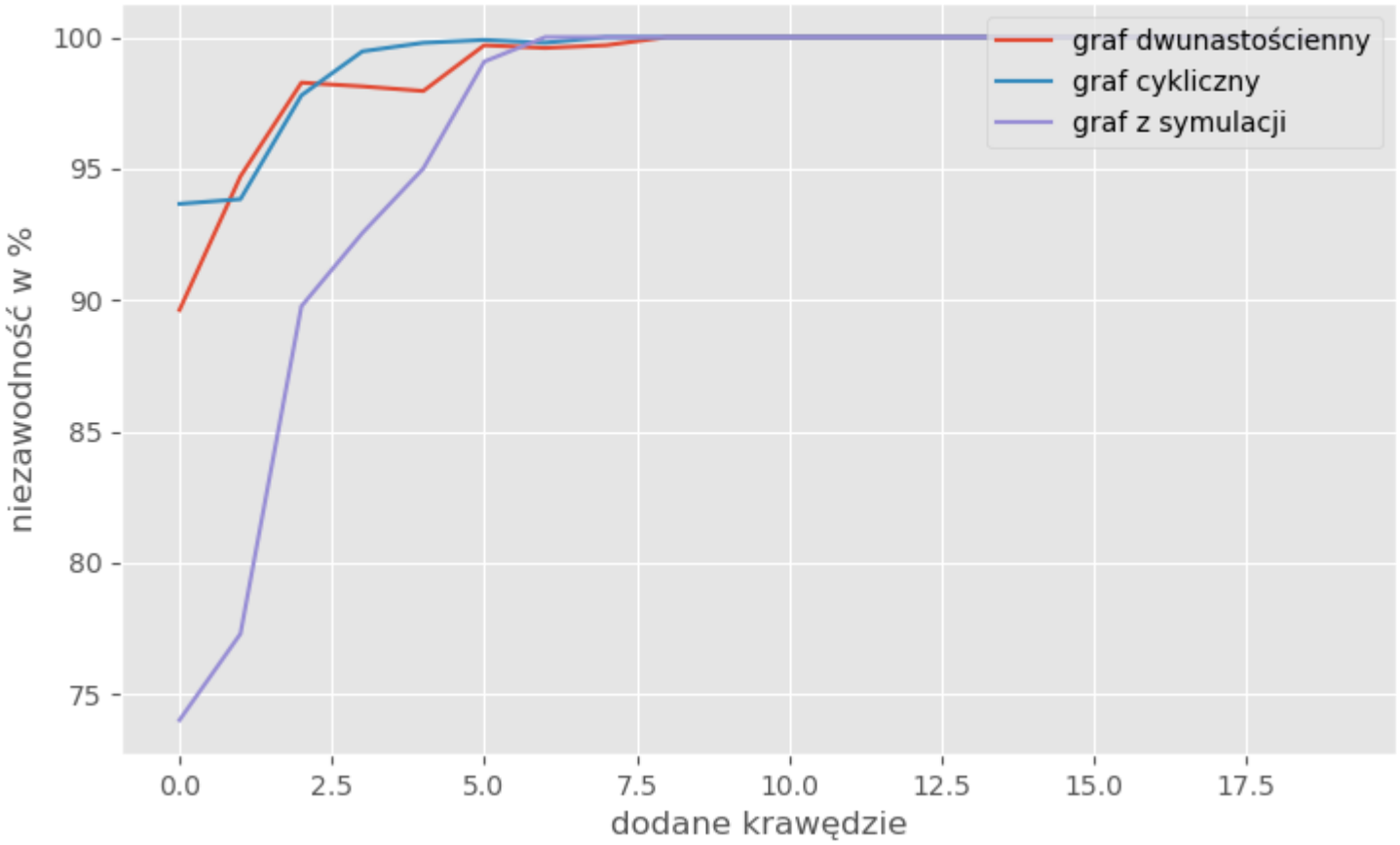


Jak widać na powyższym wykresie - graf cykliczny nie daje rady już dla skali 4, graf z symulacji natomiast zawodzi przy stałej równej 5, natomiast graf dwudziestościenny - przy stałej równej 6.

2.4.2 Krawędzie

```
def dodaj_krawedz(G):
    i = random.randint(0,len(G.nodes)-1)
    j = random.randint(0, len(G.nodes)-1)
    while(czy_krawedz(G,i,j)):
        j = random.randint(0, len(G.nodes)-1)
    G.add_edge(i,j)
```

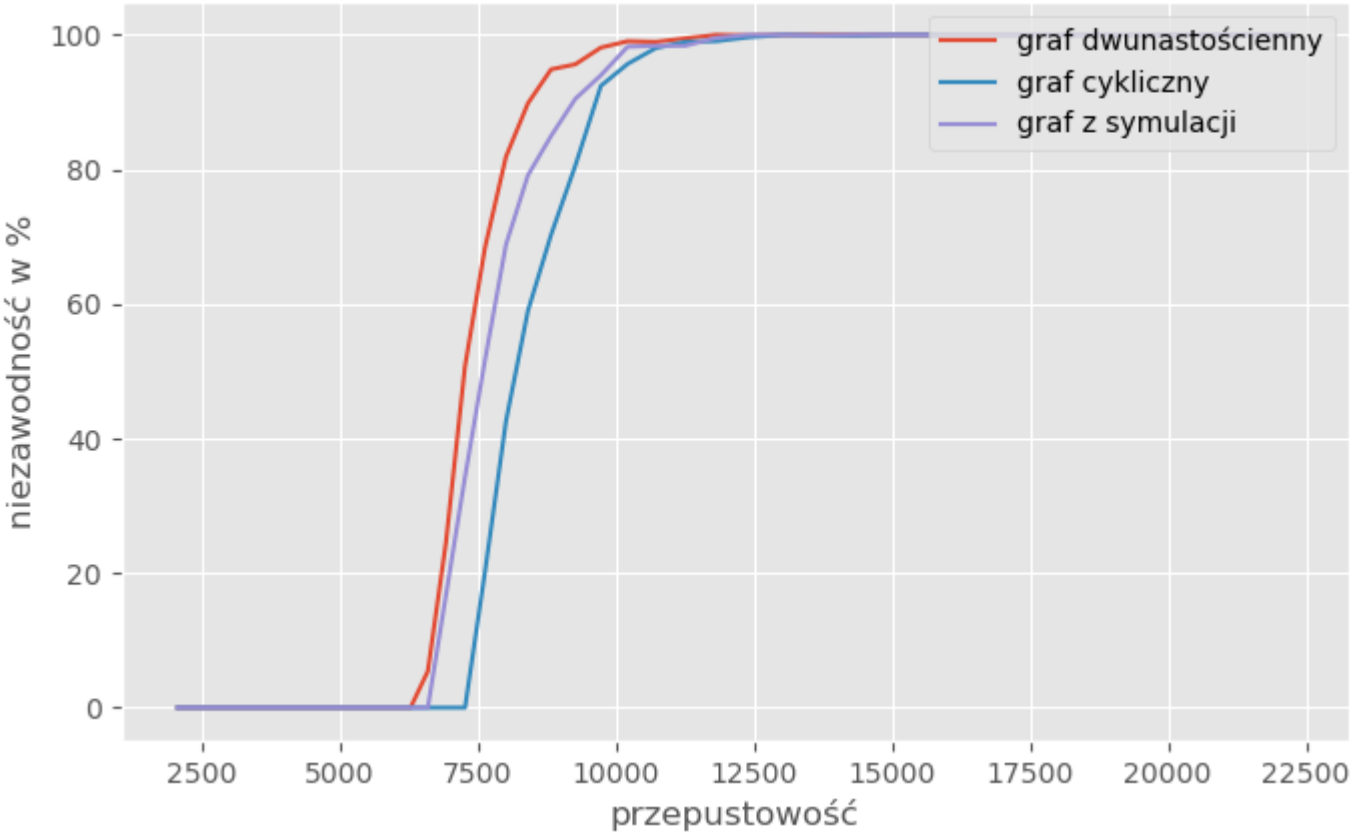
Stwórzmy teraz wykres z założeniem, że dodamy od 1 do 20 krawędzi do każdego grafu. W tym przypadku załóżmy, że $N[i][j] \leqslant 50$, a c jest stałe:



Widzimy, że wystarczy dodać jedną krawędź do każdego grafu, aby niezawodność znacząco wzrosła. Po dodaniu 5 dodatkowych krawędzi niezawodność wszystkich grafów wynosiła praktycznie 100%.

2.4.3 Przepustowość

Ustalmy $N[i][j] \leqslant 50$ oraz $c_{pocz} = 1024$ Przeskalujmy sobie 20 razy nasze przepustowości przez 1.1:



Jak widzimy - dla podanych $N[i][j]$ potrzebujemy przepustowość wynoszącą minimalnie 7000b/s, aby nasza sieć nie została przeciążona. Widzimy również, że nie potrzebujemy wartości większych, niż 12000b/s, ponieważ przy tych wartościach nasza sieć jest praktycznie niezawodna.

3 Podsumowanie

Na podstawie powyższych symulacji możemy wywnioskować, że:

1. najmniej zawodnym modelem sieci jest taki, w którym każdy wierzchołek jest mniej więcej tego samego, jak największego stopnia - do takiego modelu sieci dążył model z 10-godzinnej symulacji, natomiast takim modelem jest przedstawiony graf dwunastościenny.

2. Dodanie nawet jednej krawędzi do sieci może znacząco wpłynąć na niezawodność.

3. Jeżeli zależy nam na przepustowości, to chcemy, żeby dla wszystkich i,j nasza ścieżka z $v(i)$ do $v(j)$ była jak najmniejsza.