Michał Budnik – sprawozdanie

1. Zadanie 1

1.1 Opis modelu sieci

Rozważamy model sieci, w którym czas działania jest podzielony na interwały. Model taki oznaczamy jako S = <G, H>, gdzie:
- S to model sieci,
- G to graf reprezentujący naszą sieć (G = <V, E>),
- H to zbiór funkcji niezawodności przyporządkowującej każdej krawędzi funkcję niezawodności ‘h’.

W celu wyjaśnienia – funkcja niezawodności ‘h’ przypisuje danej krawędzi prawdopodobieństwo nierozerwania węzła komunikacyjnego w dowolnym przedziale czasowym. Warto również wspomnieć, że nie bierzemy pod uwagę możliwości uszkodzenia wierzchołka.

W rozważanym dla naszego zadania modelu (w wersji podstawowej) będziemy przyjmować:
- |V| = 20, V = {1, 2, ..., 20},
- |E| = 19, E = {$e_{i, i+1}$: i należy do [1, 19]},
- |H| = |E|, H = {h($e_{i, i+1}$) = 0.95 ∧ i należy do [1, 19]}.

1.2 Program szacujący niezawodność danej sieci

Założenia programu:
- niezawodność sieci definiowana jako istnienie ścieżki (kanału komunikacyjnego) pomiędzy każdą parą wierzchołków należących do tej sieci,
- sprawdzamy niezawodność używając metody Monte Carlo.

Program został napisany w języku Python, pełen kod znajduje się w załączniku 1. Tutaj pokrótce przedstawię i opiszę funkcję symulującą cykle i sprawdzającą spójność sieci.

```
def check(interval, vertices, edges):
    def cycle():
        for edge in edges:
            if randint(1, 100) / 100 > edge[3]: edge[2] = False
    for i in range(0, interval): cycle()
    def connectivity(checked, toCheck):
        if len(toCheck) != 0:
            for edge in edges:
                if (edge[0] == toCheck[0]) and edge[2] and (edge[1] not in checked) and (edge[1] not in
toCheck):
                    toCheck += [edge[1]]
                elif (edge[1] == toCheck[0]) and edge[2] and (edge[0] not in checked) and (edge[0] not in
toCheck):
                    toCheck += [edge[0]]
            checked += [toCheck[0]]
            toCheck.remove(toCheck[0])
            return connectivity(checked, toCheck)
        else:
            if len(checked) == len(vertices):
                return True
            return False
    return connectivity([], [vertices[0]])
```

Snippet 1: Sprawdzanie spójności grafu

Funkcja ‘check(interval, vertices, edges)’ sprawdza, czy graf o wierzchołkach ‘vertices’ i krawędziach ‘edges’ jest spójny po ‘interval’ cyklach. Struktura krawędzi ‘edge’ wygląda następująco:

edge = [(int)vertex 1, (int)vertex 2, (boolean)active, (float)h]

Pierwsza funkcja wewnętrzna ‘cycle’ prowadzi symulację rozspójnienia dla wszystkich krawędzi. Losuje ona cyfrę z przedziału 1-100 i sprawdza, czy wylosowania cyfra podzielona przez 100 jest większa od ‘h’ danej krawędzi. Zakładając pełną losowość losowanych liczb, prawdopodobieństwo niezawodności krawędzi jest równe dokładnie h. ((100-h)/100 szansy na wylosowanie liczby wyłączającej daną krawędź z użytku).

Druga funkcja wewnętrzna ‘connectivity’ sprawdza spójność grafu metodą ‘rozlewania wody’. Zaczynając od dowolnego wierzchołka (w przypadku mojego programu zawsze od pierwszego na liście wierzchołków), sprawdzamy do jakich wierzchołków możemy z niego dojść. Po sprawdzeniu, dany wierzchołek jest wypisywany z listy wierzchołków ‘do sprawdzenia’ – jest natomiast dopisywany do listy wierzchołków ‘sprawdzonych’. Wszystkie wierzchołki, do których mogliśmy dojść, a które nie są jeszcze w liście ‘sprawdzonych’ lub ‘do sprawdzenia’, są dodawane do listy ‘do sprawdzenia’. Powtarzamy ten algorytm tak długo, aż lista ‘do sprawdzenia’ nie będzie pusta. Gdy to nastąpi, oznacza to, że przeszliśmy przez wszystkie możliwe wierzchołki w grafie, który jest spójny z pierwszym wierzchołkiem, i wszystkie zostały dodane do listy ‘sprawdzone’. Finalnie, porównujemy listę sprawdzone z listą wszystkich wierzchołków naszej sieci. Jeżeli są one równe, to znaczy, że sieć jest dalej spójna. Jeżeli natomiast nie są sobie równe, to sieć się rozspójniła.

Teraz, używając metody Monte Carlo (a więc powtarzając wywołanie funkcji ‘check’ dla tych samych parametrów dość dużą ilość razy) uśredniamy uzyskane wyniki czy sieć się rozspójniła czy też nie. Otrzymujemy w ten sposób dosyć dobre przybliżenie, czy po X interwałach sieć o danych połączeniach i funkcjach niezawodności się rozspójni.

1.2.1 Podstawowy model

W przypadku pierwszego podpunktu, a więc sprawdzenia prawdopodobieństwa nierozspójnienia podstawowego modelu rozważanego w tym sprawozdaniu (linii), prawdopodobieństwo to wynosi 37.785%, co dosyć dobrze pokrywa się z teorią matematyczną ((0.95^19)*100 = 37.735: 19 krawędzi z których każda ma prawdopodobieństwo nierozspójnienia 0.95).

```
"/home/michal/Desktop/Semestr 3/TS/Lab2/venv/bin/python"
"/home/michal/Desktop/Semestr 3/TS/Lab2/Zad 1.py"

For 1 intervals, probability of problem a is: 0.37785
For 1 intervals, probability of problem b is: 0.73611
For 1 intervals, probability of problem c is: 0.87078
For 1 intervals, probability of problem d is: 0.91064
Process finished with exit code 0
```

Output 1: Przykładowy wynik wszystkich problemów z zadania 1

1.2.2 Uzupełnienie do cyklu

W problemie b zadania pierwszego dodajemy krawędź $e_{1, 20}$ z funkcją h = 0.95. Graf sieci staje się więc cyklem. Sprawdzając niezawodność tej sieci tym samym programem wzrosła ona do 73.611% - co jest prawie podwojeniem jej niezawodności w porównaniu z podstawowym modelem.

1.2.3 Dodanie kolejnych krawędzi

W problemie c dodajemy kolejne krawędzie:
- $e_{1, 10}$ o funkcji h = 0.8,
- $e_{5, 15}$ o funkcji h = 0.7.

Po dodaniu tych krawędzi do sieci z problemu b jej niezawodność wzrasta do 87.078%.

1.2.4 Dodanie 4 losowych krawędzi

W ostatnim problemie dodajemy do sieci z problemu c cztery losowe krawędzie, każda o funkcji niezawodności h = 0.4. Znów można zauważyć wzrost niezawodności sieci – tym razem do 91.064%.

1.3 Wnioski

Da się zauważyć wyraźną korelację ilości krawędzi z prawdopodobieństwem jej nierozspójnienia. Jak łatwo wywnioskować najłatwiej jest rozspójnić sieć, gdy wszystkie wierzchołki są połączone w linie – wystarczy, żeby jedna krawędź się rozspójniła, a cała sieć stanie się niespójna. Prawdopodobieństwo, że sieć będzie spójna po jednym cyklu wynosi tylko około 38%!

Gdy naszą sieć uzupełnimy o krawędź $e_{1, 20}$ można zauważyć, że tworzymy cykl. Wynik nierozspójnienia na poziomie 74% ma całkowity sens, gdyż możemy myśleć o cyklu jako o dwóch liniach, przez które możemy przejść aby dostać się do danego wierzchołka. Wyjaśniając w inny sposób, nie mniej niż dwie krawędzie muszą się zepsuć, żeby rozspójnić graf (uszkodzenie jednej tworzy nam linię).

Uzupełnienie o kolejne wierzchołki – $e_{1, 10}$ oraz $e_{5, 15}$ – jest strategicznie dobrym rozwiązaniem, łączymy w ten sposób najodleglejsze końce sieci, oraz połączenia te są oddalone od siebie o dokładnie taką samą ilość wierzchołków z każdej strony, dzieląc nam graf na 4 równe części. Jak wynika z obserwacji niezawodność takiej sieci wzrasta do poziomu 87%, co jest wciąż dużym skokiem w porównaniu do poprzednich 74% (wzrost niezawodności o 13%!).

Dodanie do tej sieci losowych czterech wierzchołków nie zwiększa już niezawodności tej sieci o bardzo znaczący procent. Niezawodność po dodatkowych połączeniach rośnie do 91%, czyli jest o 4% większa niż wcześniej. Da się więc zauważyć, że strategiczne zbudowanie sieci, połączone z tworzeniem dodatkowych cykli (szczególnie gdy żadne w nim nie występują) dają duży skok w niezawodności w tej sieci. Jednak nawet przy dodaniu nawet kilku wierzchołków losowo, podczas gdy sieć jest już dosyć niezawodna, nie skutkuje znaczącym wzrostem tej niezawodności.

2. Zadanie 2

2.1 Opis modelu

Rozważamy taki sam model sieci jak w przypadku zadania pierwszego. Jednakże tym razem będziemy sprawdzać przepływ danych przez daną sieć. Musimy więc zdefiniować dodatkowe zmienne:
- macierz natężeń strumienia pakietów N, gdzie $n_{i,j}$ jest liczbą wprowadzanych pakietów ze źródła (v_i) do ujścia (v_j),
- dla każdej krawędzi funkcję przepustowości ‘c’, czyli maksymalną liczbę bitów, którą można wprowadzić do kanału komunikacyjnego,
- dla każdej krawędzi funkcję przepływu ‘a’, czyli faktyczną liczbę pakietów przepuszczaną przez kanał w trakcie sekundy.

Na potrzeby zadań przyjmuję wielkość jednego pakietu jako 64bity.

2.2 Podzadania

2.2.1 Propozycje topologii

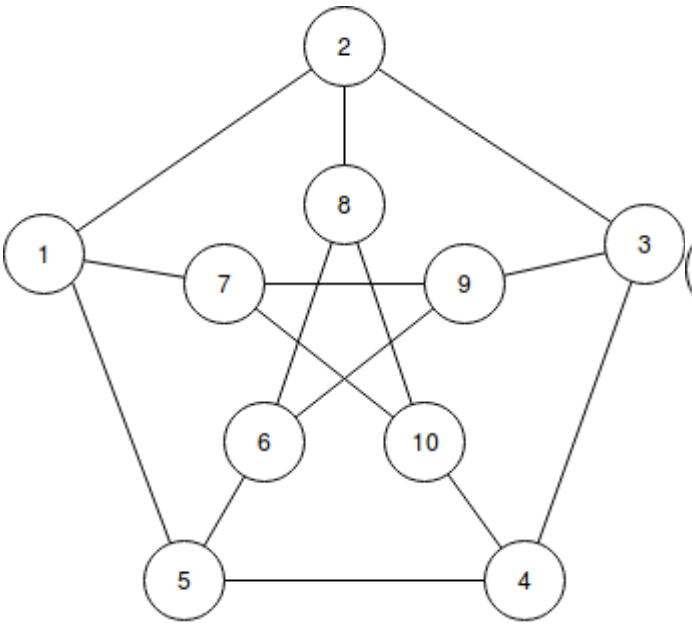

Illustration 1: Graf Petersena

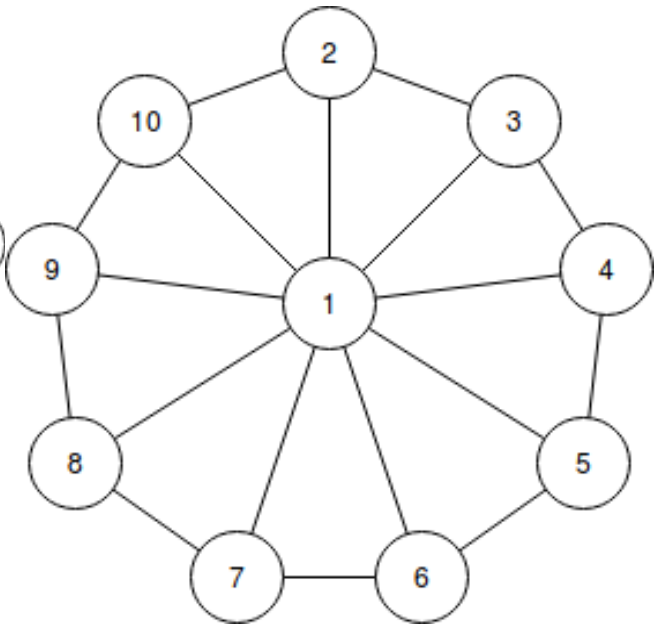

Illustration 2: Koło$_{10}$

Dla widocznych topologii proponuję prostą macierz N = [$n_{i,j}$ = {10 if i!=j; else 0], czyli każdy wierzchołek chce wysłać 10 pakietów do każdego innego wierzchołka.

Jeżeli chodzi o drogę, którą ma przebyć pakiet, szukam najkrótszej możliwej ścieżki. W przypadku grafu Petersena każda krawędź jest równo obciążona. Dla zadanej macierzy N, a(e) = 100 pakietów. Więc aby sieć mogła funkcjonować każda krawędź musi mieć maksymalną przepustowość większą niż 100 pakietów (c(e) > 100 pakietów * 64 bity/pakiet = 6400 bitów.)

Natomiast dla koła sytuacja wygląda różnie, w zależności od obranej ścieżki – istnieje więcej niż jedna najkrótsza ścieżka między dwoma wierzchołkami. Zakładam więc, że pakiety przechodzą przez centralny wierzchołek, kiedy istnieją przynajmniej dwie najkrótsze ścieżki. Wtedy obciążenie kanałów dzieli się na dwie

grupy:
- zewnętrzny cykl, gdzie dla każdej krawędzi a(e) = 20 pakietów ,
- wewnętrzne ‘szprychy’, gdzie dla każdej krawędzi a(e) = 140 pakietów.

Tak więc dla zewnętrznych krawędzi c(e) > 20 pakietów * 64 bity/pakiet = 1280 bitów, natomiast dla wewnętrznych krawędzi c(e) > 140 pakietów * 64 bity/pakiet = 8960 bitów.

2.2.2 Obliczanie średniego opóźnienia pakietu

Kod programu można znaleźć w załączniku 2. Postaram się tutaj przybliżyć działanie programu. Mamy zadeklarowaną macierz natężeń N, dla wszystkich krawędzi definiujemy maksymalną liczbę bitów (c) przepuszczanych przez kanał. Utworzona została klasa grafu, która przechowuje informacje o grafie, takie jak:
- wierzchołki i krawędzie grafu,
- początkową listę najkrótszych ścieżek w grafie,
- funkcję znajdowania wszystkich najkrótszych ścieżek w grafie (nadpisującą początkową listę) oraz funkcję znajdowania najkrótszej ścieżki pomiędzy dwoma wybranymi wierzchołkami
- funkcję obliczania ilości pakietów przepływających dla każdej krawędzi (a) używającą listy najkrótszych ścieżek w grafie.

Używamy wzoru T = 1/G * SUM_e(a(e)/(c(e)/m - a(e))), gdzie ‘G ‘to suma wszystkich elementów macierzy natężeń, a ‘m’ to średnia wielkość pakietu.

Następnie obliczane są ‘G’ - na podstawie macierzy natężeń, ‘SUM_e’, oraz średnie opóźnienie korzystając ze wcześniejszych obliczeń, oraz zadeklarowaniej średniej wielkości pakietu ‘m’ (w moim przypadku 64 bity.)

Zwiększając ilość możliwych bitów do przesłania przez dany kanał o wielkość jednego pakietu (zaczynając od najniższej możliwej wartości dla faktycznego natężenia ‘a’), otrzymałem taki wykres zależności dodatkowej przepustowości od średniego opóźnienia (czyste dane w załączniku 3):

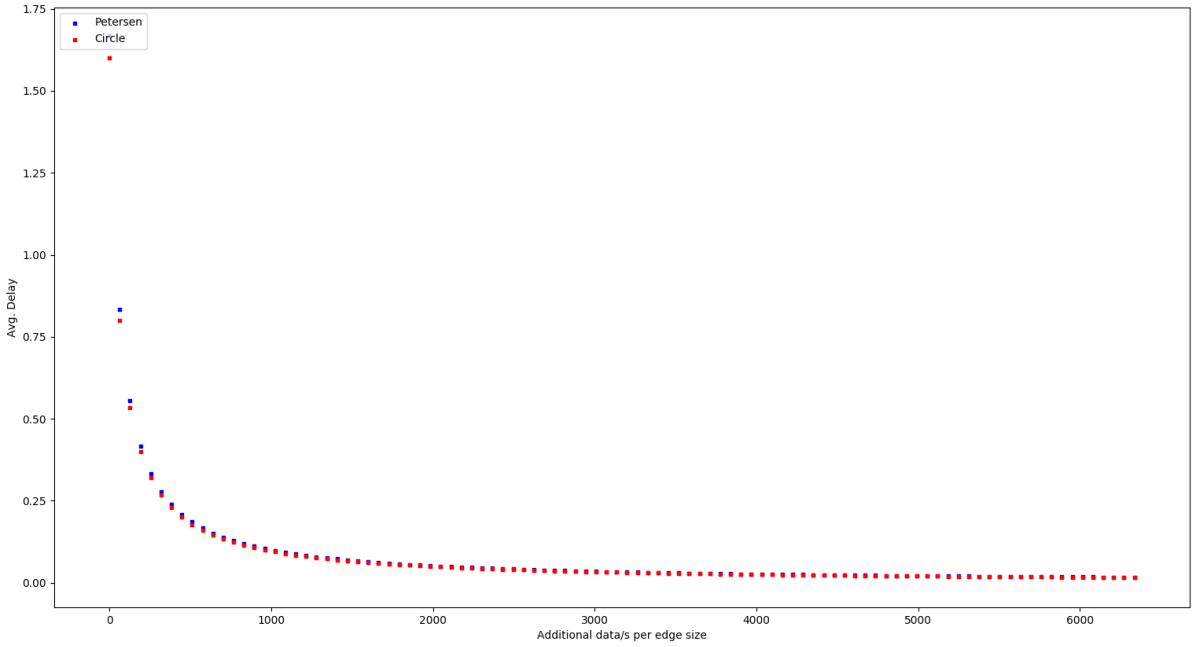

Illustration 3: Opóźnienie a dodatkowa wielkość pakietu

2.2.3 Niezawodność sieci

Program użyty do tego zadania jest tym samym programem co program w zadaniu 2.2.2 (załącznik 2). Opiszę tutaj główną funkcję testującą niezawodność sieci (która jest zmodyfikowaną funkcją z zadania 1):

```
def checkIntegrity(graph, cycles):
    def cycle():
        global p
        for i in range(0, len(graph.edges)):
            if randint(1, 100) / 100 > p:
                graph.edges[i] = False
        graph.edges = [x for x in graph.edges if x!= False]
    for i in range(0, cycles):
        cycle()
    graph.shortestPaths()
    for p in graph.paths:
        if graph.paths[p] == None:
            return False
    if graph.id == 'p':
        if avgDelayPetersen() > T_MAX:
            return False
    if graph.id == 'c':
        if avgDelayCircle() > T_MAX:
            return False
    return True
```

Snippet 2: Sprawdzanie niezawodności grafu

Do funkcji ‘checkIntegrity’ przekazujemy dany graf oraz liczbę czykli, którą chcemy przesymulować. W zależności od ‘p’ mamy (1-’p’)*100% szansy na wyłączenie dowolnej krawędzi z grafu (zakładając pełną losowość generatora pseudolosowego). Po zasymulowaniu uszkodzeń krawędzi aktualizujemy listę najkrótszych ścieżek danego grafu, wywołując funkcję ‘graph.shortestPaths()’. Następnie sprawdzamy, czy dla którejkolwiek pary wierzchołków znaleziona ścieżka jest nullem. Jeżeli tak, to oznacza, że graf się rozspójnił i nie da się przesłać pakietu, więc zwracamy False (sieć rozspójniona). Jeżeli tak nie jest, to sprawdzamy średnie opóźnienie pakietu. Jeżeli przekracza ono parametr T_MAX, to również przyjmujemy, że sieć zawiodła.

Sprawdzałem niezawodność sieci dla różnych parametrów ‘T_MAX’ oraz ‘p’ zarówno dla grafu Petersena, jak i koła. Wynika z nich jednoznacznie, że koło ogólnie zachowuje większą niezawodność niż graf petersena.

2.3 Wnioski

Postanowiłem wybrać graf Petersena oraz koło jako moje topologie z kilku względów:
- w pewnym sensie grafy są sobie podobne – większość wierzchołków ma stopień 3, najkrótsza ścieżka między dwoma wierzchołkami zawsze jest krótsza niż 3 krawędzie, oraz mogą dobrze reprezentować sieć istniejącą w faktycznym świecie – są również fundamentalnie różne – graf Petersena jest całkowicie zdecentralizowany (nie ma ‘najważniejszego’ wierzchołka lub krawędzi, natężenie na każdej krawędzi jest dokładnie takie samo), gdy koło jest prawie idealnie scentralizowane (tylko połączenia bezpośrednie nie muszą przechodzić przez ‘główny’ wierzchołek).

Myślę, że może to dosyć dobrze oddawać sytuację w realnym świecie, gdyż graf Petersena może być np. siecią łączącą różne uczelnie (każda z nich jest

jednakowo ważna, a te bliższe sobie geograficznie są bezpośrednio połączone), a koło jest dosyć dobrze widocznym przykładem głównego serwera, który reguluje większość ruchu sieci. W sprawozdaniu chciałem zaobserwować podobieństwa i różnice tych topologii, dlatego postanowiłem też możliwie ujednolicić testy. Sprawdzam więc wszystkie wyniki dla macierzy natężeń takiej, że pomiędzy każdą parą wierzchołków wymieniana jest taka sama ilość pakietów. Dodatkowo ustawiłem wartości c(e) na najmniejsze możliwe, aby sprawdzić ‘najgorsze’ przypadki działającej sieci.

Już w pierwszym eksperymencie da się zauważyć różnicę pomiędzy scentralizowaną a zdecentralizowaną siecią. W przypadku grafu Petersena wszystkie wierzchołki miały dokładnie taką samą faktyczną wartość przepływu – równą 100 pakietom. Dla koła natomiast widoczny jest podział na mniej używane ścieżki zewnętrzne (dla których wartość przepływu wyniosła 20 pakietów) a ścieżki wewnętrzne prowadzące do ‘serwera’ (gdzie wartość przepływu wyniosła 140 pakietów). Jednakże w żaden sposób nie wskazuje to na przewagę jednej topologii nad drugą – wybranie odpowiedniej struktury sieci w dużym stopniu zależy od potrzeb oraz przeznaczenia sieci, możliwości finansowych oraz logistycznych. Struktura grafu Petersena wymaga, aby każde połączenie pozwalało na średniej wielkości połączenie, podczas gdy dla koła wymagane są silne połączenia do serwera, oraz słabe dla zewnętrznych krawędzi. Warto jeszcze wspomnieć, że suma obciążeń w grafie Petersena jest większa niż dla koła – w moim eksperymencie 1500 pakietów do 1440 pakietów.

W drugim eksperymencie zacząłem pomiary dla c(e) minimalnego, niezbędnego do prawidłowego działania sieci. Następnie, w każdej iteracji zwiększałem możliwy przepływ dla każdej krawędzi o stałą ilość bitów (o wielkość jednego pakietu). W ten sposób chciałem sprawdzić, jak takie samo polepszenie wszystkich krawędzi dla obu topologii wpływa na średnie opóźnienie pakietu. Jak można odczytać z danych graf kołowy osiąga niższe średnie opóźnienie w porównaniu z grafem Petersena – najprawdopodobniej wynika to z większej ilości krawędzi. Jednakże przy znaczącym zwiększeniu wartości ‘c’ różnice te zacierają się i średnie opóźnienie pakietu staje się praktycznie takie samo dla tych topologii.

W ostatnim zadaniu sprawdzałem niezawodność sieci dla różnych parametrów wejściowych ‘p’ (prawdopodobieństwa zepsucia się krawędzi), ‘T_MAX’ (maksymalnego dopuszczalnego opóźnienia paketu) oraz ‘m’ (średniej wielkości pakietu – dla moich eksperymentów używałem m = 64bity). Postanowiłem raz jeszcze porównać zaproponowane przeze mnie topologie. Używając metody Monte Carlo, sprawdzałem prawdopodobieństwo niezawodności sieci (przyjmowanej jako jej nierozspójnienie oraz nieprzekroczenie T_MAX). Pomiary znów wykazały, że graf kołowy miał większą szansę na zachowanie spójności. Myślę, że głównym czynnikiem przekładającym się na dany wynik jest ilość krawędzi w grafie. Im więcej krawędzi w grafie tym trudniej go rozspójnić oraz średnie opóźnienie pakietu powinno zwiększyć się o mniejszą część w porównaniu z grafem o mniejszej ilości krawędzi.