TECHNOLOGIE SIECIOWE

Sprawozdanie

Lista 2

Mikołaj Szarapanowski 208865
grupa Poniedziałkowa 7:30

1. Cel

Celem niniejszego sprawozdania jest zbadanie niezawodności pewnych modeli sieci.
Do testowania grafów używamy metody Monte Carlo: Zostały usunięte losowe krawędzie, a następnie została sprawdzona spójność grafu (zadanie 1 oraz 2) oraz przesyły i opóźnienie (zadanie 2). Przy odnoszeniu się do linii kodu w odnośnikach, należy zwrócić uwagę, że są to numery linii w klasie, w której dane pole, metoda czy funkcja się znajduje.

2. Zadanie pierwsze

a) Opis programu

Program ma za zadanie sprawdzenie niezawodności sieci. Program został napisany w języku Java, natomiast biblioteką, która okażę się niezbędna do tworzenia grafów jest *Jgrapht*[1].

Opis klas:

-public interface GraphFactory

Na początku w programie został zaimplementowany interfejs publiczny *GraphFactory*[2], który posiada metodę *createGraph*[2]. Według tego interfejsu mają zostać zaimplementowane grafy do odpowiednich podpunktów.

-public class *A,B,C,D*

W klasie *A* został zaimplementowany interfejs *GraphFactory* do stworzenia grafu *g*[3]. W pierwszej pętli for zostało dodanych 20 wierzchołków. Metoda *addVertex*[4] jest metodą z biblioteki *Jgrapht*. Z tej samej biblioteki w kolejnej pętli użyto metod *setEdgeWeight*[5] i *addEdge*[6]. Odpowiadały one za stworzenie 20 krawędzi o wadze 0.95. Analogicznie do klasy *A*, w klasach *B*, *C* oraz *D* użyto tych metod do dodania krawędzi opisanych w specyfikacji zadania.

-public class Exercise1

W klasie *Exercise1* zaczynam od implementowania metody *start*[7], która wywoła się w chwili uruchomienia zadania pierwszego. Na początku zostanie stworzona tablica grafów, a następnie dodane do niej zostaną 4 grafy (*A*, *B*, *C* oraz *D*). Zmienne *all*[8] oraz *passed*[9] odpowiadają odpowiednio za liczenie ile prób usuwania się odbyło i ile razy graf się po próbie rozerwania zachował spójność. Aby zbadać niezawodność sieci na początku uruchamiam generator liczb losowych *Random generator*[10] z przedziału (0,1) i jeśli liczba losowa jest większa niż waga krawędzi, to krawędź zostanie usunięta. W ten sposób jest sprawdzana każda krawędź grafu. Po zbadaniu wszystkich grafów sprawdzam metodą *isConnectivity*[11], czy graf zachował spójność, jeśli tak, to zwiększam licznika *passed*. Na końcu wypisuje wyniki mojego eksperymentu

1)*Jgrapht*- Linie: 3-4 2)*GraphFactory*- Linia: 6 3) *g*- Linia:10 4) *addVertex*- Linia: 14
5)setEdgeWeight- Linia: 18 6)addEdge- Linia: 18 7)start- Linia: 17 8)all- Linia: 30
9)passed- Linia: 31 10)Random generator- Linia: 15 11)isConnectivity- Linia: 78

b) Przykładowe wywołania, analiza oraz interpretacja wyników

Przykładowy wynik wywołania:
A: 37% 376/1000

Każdą linijkę programu należy odczytywać w ten sposób:
<litera podzadania> <niezawodność> <ilość pomyślnych prób>/<Ilość wszystkich prób>

Oto tabela niezawodności poszczególnych grafów przy próbie wykonania 1000 prób pięciokrotnie:

	A	**B**	**C**	**D**
1	37%	75%	89%	94%
2	35%	74%	90%	93%
3	39%	75%	90%	94%
4	37%	74%	88%	94%
5	36%	75%	88%	94%

Obserwując powyższe wyniki łatwo zauważyć że w przypadku A (gdy graf jest w postaci linii) mamy najmniejsze szanse na spójność grafu. Usunięcie już nawet jednej krawędzi z grafu powoduje jego niespójność.
W przypadku B po dodaniu krawędzi można zauważyć blisko dwukrotny wzrost na uzyskanie grafu spójnego.
W podpunkcie C dodajemy dwie dodatkowe krawędzie i tutaj również zostaje zauważony znaczący wzrost szans na uzyskanie spójności grafu.
W przypadku D do grafu dodajemy 4 krawędzie między losowo wybranymi wierzchołkami, co powoduje większy procent przypadków gdy graf po usunięciu krawędzi zachował spójność.

3. Zadanie drugie

a) Opis programu

-public class MyEdge
Klasa *MyEdge* posiada pola *capacity*[12], *flow*[13] oraz *weight*[14], które zawierają w sobie odpowiednio informacje o wadze krawędzi, przepływie oraz przepustowości strumienia pakietów. W klasie znajdują się metody, które są odpowiedzialne za ustalanie i pobieranie informacji o tych polach.

12)capacity- Linia: 7 13)flow- Linia:8 14)weight- Linia: 9

-public class Exercise2

Lwia część całego zadania drugiego znajduję się w klasie *Exercise2*, a dokładnie w metodzie *makeExercises*[15], która wywołuje inne zaimplementowane w klasie funkcje.

```
163   public static void makeExercises(){
164     SimpleGraph<Integer, MyEdge> graph;
165     boolean c, isConnective;
166       G=0;
167       graph = createGraph();
168       N = createMatrix();
169       isConnective = testGraph(graph);
170       if (isConnective){
171       searchPath(graph, N);
172       c = check(graph);
173       avgDelay(graph, c);
174       }
175   }
```

W metodzie *makeExercises* stworzono graf oraz macierz natężeń, a następnie zbadano spójność (podobnie jak w zadaniu 1). Jeśli uzyskano spójny graf, to wykonuje się kolejno szukanie ścieżki, sprawdzenie czy przepływ jest mniejszy od przepustowości oraz obliczenie średniego opóźnienia pakietu.

W metodzie *createGraph*[16] każdy element *ArrayList graphArray*[17] rozdzielamy odpowiednio do nowej tablicy, a następnie dodajemy wierzchołki do grafu i tworzymy nową krawędź, której ustawiamy wagę (domyślnie na 0.95). Dodatkowo dla każdej krawędzi ustalany jest jej przepływ (domyślnie na 0), a także losowana jest wartość przepustowości tej krawędzi (losowana z przedziału od 500 do 700). Ustalamy też wagę krawędzi domyślnie na 0.95. Następnie losujemy 7 dodatkowych wierzchołków wykorzystując metodę *addRandomEdge(graph)*[18].

Funkcja testująca spójność grafu (*testGraph*[19]), gdy graf nie jest spójny zwiększy licznik *failGraphIsNotConnective*[20] o 1 i zwróci wartość *false*[21]. W tym przypadku kolejne instrukcje z funkcji *makeExercises*[22] nie będą wykonywane.

Macierz natężeń N tworzona jest w sposób losowy. Wartości macierzy na przekątnej (czyli przepływ od wierzchołka np. 1 do 1) są ustawiane na zero – graf nie posiada pętli. Podczas tworzenia macierzy wyliczana jest również wartość *G*[23], która jest sumą wartości w macierzy. Wartość *m*[24], czyli średnia wielkość pakietu w bitach ustalana jest ze wzoru *G/100*.

15)makeExercises- Linia: 163
16)CreateGrapg- Linia: 26
17) graphArray- Linia: 28
18)addRandomEdge- Linia: 51
19)testGraph- Linia: 142
20) failGraphIsNotConnective- Linia: 156
21)false- Linia: 157
23)G- Linie 78 oraz 88
24)m-Linia: 88

Kolejnym krokiem jest poszukiwanie dla każdego połączenia wierzchołków najkrótszej ścieżki między nimi. W tym celu zaimplementowano funkcję *searchPath*[25]. Do znalezienia ścieżki wykorzystano algorytm Dijkstry, zaimplementowany w bibliotece *Jgrapht*. Następnie dla każdej krawędzi w znalezionej ścieżce, pobierana jest wartość przepływu dla danej krawędzi, a potem zwiększanie jej o wartość z macierzy.

Funkcja *check*[26] polega na wybraniu z grafu zbioru krawędzi tworzących graf, a następnie dla każdej krawędzi sprawdzana jest czy warunek, dotyczący wartości przepustowości jest większy od wartości przepływu pomnożonej przez średnią wartość pakietu w bitach. Jeśli wynik jest niższy zmiennej zwiększany jest licznik niepowodzeń dla *failLowBandwidth*[27] i zwracamy wartość *true*[28].

W przypadku gdy wartość *fail*[29] wyświetlamy komunikat o błędzie - sieć chce przesłać więcej pakietów niż sieć jest w stanie udźwignąć. W przeciwnym wypadku pobieramy zbiór krawędzi przechodząc po każdej obliczamy średnie opóźnienie pakietu, korzystając ze wzoru:

$$T = \frac{1}{G} * \sum_{e \in edge}^{edge} \left(\frac{e.getFlow()}{\frac{e.getCapacity()}{m} - e.getFlow()} \right)$$

Gdzie:
G – suma wszystkich elementów macierzy.
m – średnia wielkość pakietu w bitach.

Również w tej funkcji sprawdzany jest jeszcze ostatni warunek czy średnie opóźnienie pakietu jest mniejsze od *TMAX*[30] . Jeśli tak jest, to zwiększany jest licznik sukcesu, w przeciwnym przypadku zwiększany jest licznik niepowodzenia.

25)searchPath- Linia: 94 26)check-107 27)failLowBandwidth- Linia: 112
28)true- Linia: 113 29)fail- Linia: 120 30)Tmax- Linia 132

b) Przykładowe wywołania, analiza oraz interpretacja wyników

Tworzymy graf dla |V| = 10 oraz |E| = 17 oraz macierz natężeń:

OUTPUT:

```
0 3 1 7 0 2 5 5 4 8
3 0 9 1 1 3 1 5 2 0
5 4 0 5 2 0 1 3 9 5
6 6 7 0 2 5 4 7 3 7
8 8 2 5 0 5 8 9 6 1
9 9 4 9 8 0 4 1 3 8
4 0 5 5 7 0 0 7 9 5
8 3 5 1 3 7 0 0 0 7
0 5 8 0 7 0 6 8 0 4
8 7 3 5 4 1 6 6 7 0
```

Dla powyższej macierzy wartość G wynosi 409 oraz m wynosi 4. Dodane zostały krawędzie (8,2), (5,8), (2,7), (1,7), (6,10), (10,8), (1,8). Po utworzeniu grafu sprawdzana jest jego spójność. W kolejnym kroku dla każdej pary wierzchołków poszukiwana jest najkrótsza ścieżka i zwiększana wartość flow. Następnie badany jest warunek czy nie doszło do przeciążenia sieci.

Po tym dla każdej krawędzi z grafu wyświetlana jest wartość przepływu oraz przepustowości:

OUTPUT

```
(1 : 2) : Flow -> 33, Capacity -> 578
(2 : 3) : Flow -> 80, Capacity -> 581
(3 : 4) : Flow -> 42, Capacity -> 588
(4 : 5) : Flow -> 59, Capacity -> 525
(5 : 6) : Flow -> 64, Capacity -> 554
(6 : 7) : Flow -> 52, Capacity -> 594
(7 : 8) : Flow -> 29, Capacity -> 515
(8 : 9) : Flow -> 63, Capacity -> 603
(9 : 10) : Flow -> 18, Capacity -> 585
(1 : 10) : Flow -> 43, Capacity -> 644
(8 : 2) : Flow -> 49, Capacity -> 580
(5 : 8) : Flow -> 61, Capacity -> 585
(2 : 7) : Flow -> 28, Capacity -> 551
(1 : 7) : Flow -> 24, Capacity -> 538
(6 : 10) : Flow -> 36, Capacity -> 661
(10 : 8) : Flow -> 13, Capacity -> 560
(1 : 8) : Flow -> 28, Capacity -> 521
```

Średni czas przepływu T: 0.01956721349591049

Następnie wykonujemy powyższe kroki dla ustalonych *quantity*[31] *= 10 000*, *TMAX*[32] *= 0.025* oraz *h*[33]*= 0.95*:

OUTPUT:

```
Connection reliability: 87.46%
Success: 8784
Fail - low bandwidth: 146 | time is greater than Tmax: 1012  | graph isn't
connective: 58
Connection reliability: 87.84%
Success: 8820
Fail - low bandwidth: 150 | time is greater than Tmax: 979  | graph isn't
connective: 51
Connection reliability: 88.2%
Success: 8787
Fail - low bandwidth: 141 | time is greater than Tmax: 1017  | graph isn't
connective: 55
Connection reliability: 87.87%
```

Możemy zauważyć, że niezawodność sieci w tym przypadku jest duża. Czynnikiem który spowodował najwięcej błędów jest przekroczenie *TMAX* , a najmniej szansa na rozspojenie grafu.

Weźmy teraz dane: *quantity = 10 000*, *TMAX* = 0.02 a *h* = 0.5:

OUTPUT:

```
Success: 140
Fail - low bandwidth: 1193 | time is greater than Tmax: 780  | graph isn't
connective: 7887
Connection reliability: 1.4%
Success: 134
Fail - low bandwidth: 1197 | time is greater than Tmax: 791  | graph isn't
connective: 7878
Connection reliability: 1.34%
Success: 118
Fail - low bandwidth: 1138 | time is greater than Tmax: 815  | graph isn't
connective: 7929
Connection reliability: 1.18%
Success: 125
Fail - low bandwidth: 1153 | time is greater than Tmax: 798  | graph isn't
connective: 7924
Connection reliability: 1.25%
```

Możemy zauważyć ogromny spadek niezawodności sieci. Spowodowany jest on bardzo dużą liczbą przypadków uzyskania niespójnego grafu, a także wzrostem błędów w przypadku przekroczenia ilości bitów przepływających przez daną krawędź.

31)quantity- Linia: 15 32)Tmax- Linia: 24 33)h- Linia: 19

OUTPUT:

```
Success: 1509
Fail - low bandwidth: 150 | time is greater than Tmax: 8287  | graph isn't
connective: 54
Connection reliability: 15.09%
Success: 1557
Fail - low bandwidth: 128 | time is greater than Tmax: 8264  | graph isn't
connective: 51
Connection reliability: 15.57%
Success: 1522
Fail - low bandwidth: 156 | time is greater than Tmax: 8266  | graph isn't
connective: 56
Connection reliability: 15.22%
Success: 1544
Fail - low bandwidth: 153 | time is greater than Tmax: 8248  | graph isn't
connective: 55
Connection reliability: 15.44%
```

Dla *quantity = 10 000, T MAX = 0.012* oraz *h = 0.95* niezawodnośd sieci również jest bardzo mała. Tym razem głownym czynnikiem niepowodzenia jest przekroczenie *TMAX* w 80% przypadków

4. Podsumowanie i wnioski

Na podstawie zadań wykonanych przy tym sprawozdaniu można dojść do wniosku, że w modelu sieci wraz ze zwiększaniem liczbą krawędzi, rośnie jego niezawodność. Z tego wynika, że mając wiele różnych dróg przez krawędzie, łączące wierzchołki, można wybrać kilka alternatywnych tras, więc usunięcie jednej krawędzi z modelu sieci nie musi wpływać znacząco na niezawodność modelu sieci. W modelu sieci należy też zwrócić na ilość przepływających bitów przez krawędź, która zbyt wysoka powoduje przeciążenie sieci. Wynika to z ograniczonej przepustowości sieci.