Marcel Jerzyk

Technologie Sieciowe

04.04.2019

Model Sieci

1. Cel listy

Zadanie dotyczy zbadania sieci przedstawionych jako grafy – dokładniej: szacowania niezawodności oraz badania natężeń (opóźnień). Programy testujące oba problemy zostały napisane przeze mnie w Javie z pomocą biblioteki JGraphT.

2. Szacowanie niezawodności

1) Przygotowanie do badań

```
// Holds edges to be deleted further on
ArrayList<DefaultWeightedEdge> edges = new ArrayList<>();

// Loop through the edges of graph
for (DefaultWeightedEdge e : graph.edgeSet()) {
    if (Math.random() > graph.getEdgeWeight(e)) {
        if(laud) System.out.println("Edge [" + e + "] has broken!");
        edges.add(e);
    }
}

// Remove them
for(DefaultWeightedEdge e : edges){
    graph.removeEdge(e);
}
```

Sprawdzenie spójności

Dla każdej krawędzi: jeżeli wylosowana liczba jest większa niż "wytrzymałość" krawędzi to markujemy ją jako zawodną (usuwamy). Po pętli, sprawdzamy czy graf jest spójny.

```
// Test if the graph is reliable.
ConnectivityInspector<Integer, DefaultWeightedEdge> inspectorTest =
        new ConnectivityInspector<>(graph);
boolean connectedGraph = inspectorTest.isConnected();
if(connectedGraph) {
    if(laud) System.out.println("> Graph [" + i + "] has proved to be reliable.");
    testPassed++;
}
```

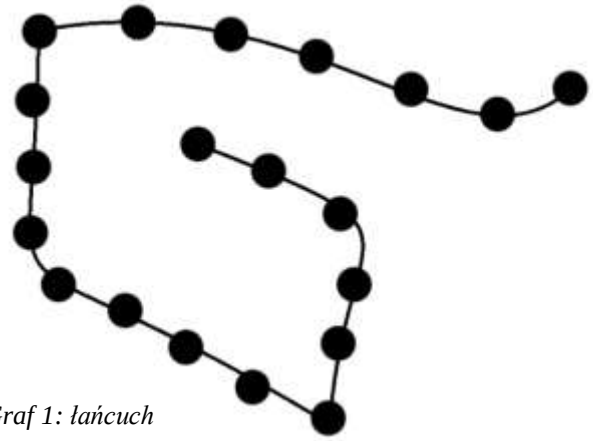
Odtwarzanie

Usunięcie pozostałych krawędzi, by móc stworzyć graf na nowo na początku pętli. Po wyjściu - print wyniku.

```
        // Remove every single edge (part 1 of remaking a graph)
        ArrayList<DefaultWeightedEdge> edgesLeft = new ArrayList<>(graph.edgeSet());

        // Remove them ^
        for(DefaultWeightedEdge e : edgesLeft){
            graph.removeEdge(e);
        }
    }
    System.out.println("> END: Tests passed: [" + testPassed + "/"
                       + testNum + "]\n> Version: " + testVersion);
}
```

Kod jest odpowiednio modyfikowany podczas tworzenia grafu przez dorzucenie if statement (za pomocą testVersion), tak by pasował do omawianego zagadnienia.

2) Graf „liniowy” (łańcuch)



Graf 1: łańcuch

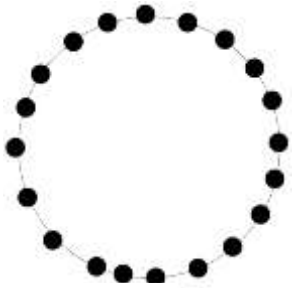
Zaczniemy testowanie od grafu, który ma **20 wierzchołków** i w którym każdy wierzchołek jest połączony z następnym (można sobie to wyobrazić, jako LinkedList wierzchołków). Warto zauważyć, że w tym wypadku kod jest gotowy do sprawdzenia tej sytuacji bez modyfikacji.

Na próbie 10000 powtórzeń dostajemy wyniki, które średnio dadzą wynik około **3750** udanych połączeń. Jest to wynik poprawny, ponieważ można go oczekiwać mając podstawową wiedzę nt. statystyki czy kombinatoryki. Mianowicie wiedząc, że mamy do czynienia z listą, w której przerwanie połączenia pomiędzy pierwszym i następnym wierzchołkiem prowadzi do niespójności grafu oraz szansa na zerwanie jest stała przy każdym kroku, zauważyć można, że prawdopodobieństwo, że taki graf będzie niezawodnym można opisać wzorem **(p)^e = (0,95)^19 ~ 0,3774**.

```
Graphs
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
> END: Tests passed: [3759/10000]
> Version: 1

Process finished with exit code 0
```

Obrazek 2: graf liniowy - test

3) Graf cykliczny



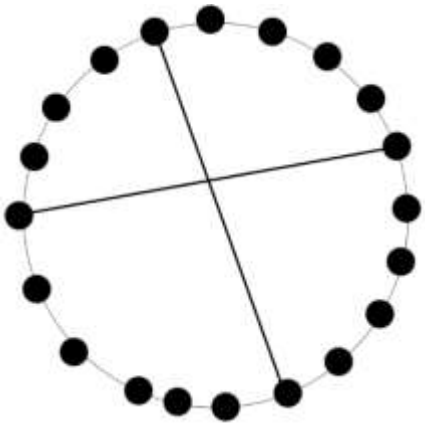
Graf 2: cykliczny

Kolejną sytuacją jest połączenie krawędzią ostatniego wierzchołka z pierwszym (domyślną wagą niezawodności = 95%). W ten sposób otrzymujemy graf cykliczny. Dzięki temu, w porównaniu do grafu z pkt 2), nasz graf ma aż dwukrotnie większą szansę na to, że będzie spójny. Analogicznie można też zauważyć, że teraz nie można pozwolić na rozerwanie dwa razy większej liczby krawędzi w porównaniu do poprzedniego przypadku – dwie zawodne krawędzie zniszczą spójność grafu.

```
Graphs
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
> END: Tests passed: [7385/10000]
> Version: 2

Process finished with exit code 0
```

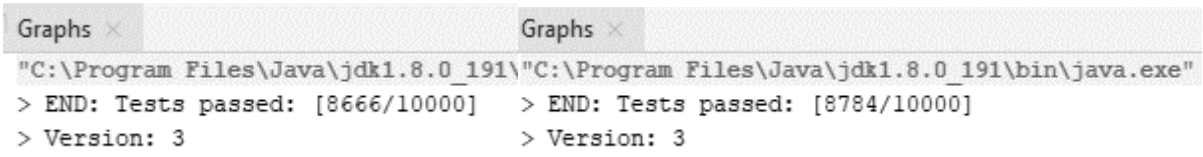
Obrazek 2: graf cykliczny - test

4) Graf z połączeniami w ćwiartkach



Graf 3: (1,10) i (5,15)

Zadaniem było połączenie krawędzi (1, 10) (niezawodność: 80%) i (5, 15) (niezawodność: 70%). Zapewne chodziło o połączenie grafu tak, aby pomiędzy dodatkowym mostem na drugą stronę grafu były dokładnie 4 wierzchołki przerwy, lecz wtedy należałoby połączyć graf w miejscach (1, 11) oraz (6, 16). W taki sposób otrzymamy dwa równomiernie ustawione przejścia na drugą stronę.
Zróbmy test dla obu wersji tego zadania.



Obrazek 3: oryginalny test

Obrazek 4: test zmodyfikowany przeze mnie

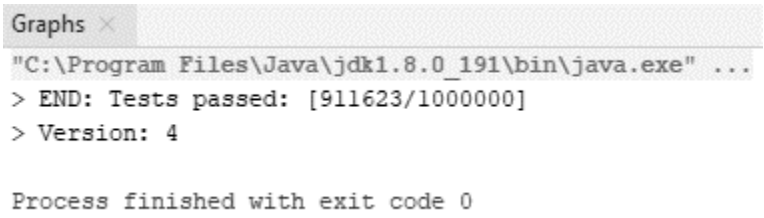
Pomiędzy jednym a drugim wynikiem jest lekka różnica. Dla upewniania się w rezultatach poddałem to jeszcze dwóm próbom, na próbie 1mln powtórzeń. Różnica wynosi 851 i 1549 na korzyść wersji zmodyfikowanej. Potwierdza to przypuszczenia, że wersja zmodyfikowana jest lepsza w utrzymaniu spójności niż ta początkowa, która jest odrobinę bardziej losowa.

Jak to się ma do grafu z punktu 3)? Dorzucenie dwóch dodatkowych krawędzi w wyżej wymieniony sposób poprawiło niezawodność z 74% na 87% co daje, aż 13 punktów procentowych, lub inaczej – 18% wzrost niezawodności. Jest to kolejna spora poprawa, choć skok nie jest tak duży jak z 1) na 2).

5) Graf czwarty z czteroma krawędziami dodatkowymi

Można spróbować dorzucić kolejne symetryczne połączenia w grafie, lecz nie będą one już tak „satysfakcjonująco” symetryczne jak te, które wykorzystaliśmy w poprzednim przypadku. W rzeczywistości sieci również nie są idealnie rozłożone – dużo bardziej przypomina to chaos.

Użyję grafu z punktu 4 (zmodyfikowanego, ponieważ działa lepiej) to zbadania jak zwiększy się niezawodność grafu po dodaniu 4 losowych krawędzi (przy każdej iteracji) o niezawodności 40% z warunkiem, że żadna krawędź nie będzie się powtarzać (tzn. jeżeli już jest krawędź z 4 do 5, to nie stworzymy kolejnej z 4 do 5) oraz krawędź szła z wierzchołka do tego samego wierzchołka. Zostawię również próbę jednego miliona.



Obrazek 5: graf z losowymi krawędziami - test

Otrzymany wynik wynosi 91%. Jest to już całkiem dużo „in total", lecz jeżeli pomyślimy o tym, że co dziesiąty dzień nasz Internet będzie nie działał, to wtedy wynik nie jest ani trochę satysfakcjonujący. Cztery krawędzie losowe zwiększyły niezawodność o tylko 4%. Wzrost bardzo mały, ledwo zauważalny w praktyce.

6) Wnioski i podsumowanie

Graf	Udane próby	Niezawodność
Łańcuch	375/1000	38%
Cykl	738/1000	74%
Cykl + "Ćwiartka"	866/1000	87%
Cykl + Ćwiartka	878/1000	88%
Cykl + Ćwiartka + 4x rand(e)	911/1000	91%

Tabela 1: porównanie grafów

Jak widać, nasza poprawa niezawodności grafu rosła logarytmicznie. Z każdą kolejną próbą niezawodność grafu rosła coraz wolniej. Użyję analogii do włókna: największe różnice wytrzymałości zauważyć można przy dodaniu do pierwszej nitki kolejnej, a przy kolejnych dwóch-trzech i mamy wytrzymałość jak przy koszulce z sieciówki. Jest to problemem dla ISP, ponieważ konsument spodziewa się 100% niezawodności przez cały czas, więc infrastruktura musi być gęsta oraz bardzo niezawodna. Podczas tworzenia

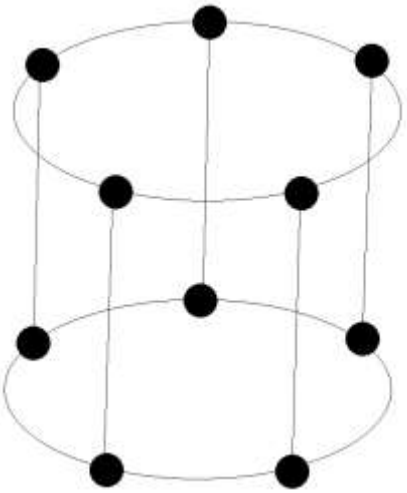
takich sieci warto próbować łączyć najbardziej odlegle końce (graf 2), robić to regularnie (graf 3) i nie liczyć na łut szczęścia (graf 4).

3. Badanie natężeń

1) Opis badania

W tym dziale przeanalizujemy nie tylko wytrzymałość, ale również jakość grafów poprzez zmierzenie potencjalnego średniego czasu działania, przepływu pakietów na krawędziach.

2) Moje propozycje topologii



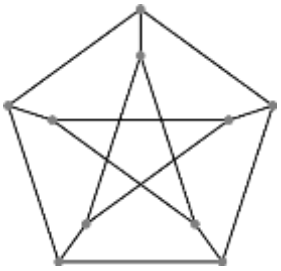
Graf 4: moja propozycja grafu

Posiadając wiedze z obserwacji z poprzedniego zadania sugeruje taki graf (graf 4).

Idąc faktem o regularności oraz łączeniem odległych punktów uznałem, że dobrze będzie zrobić z wierzchołków dwa małe grafy kołowe i połączyć je symetrycznie. Spełnia on wymagania – jest 10 wierzchołków <20 krawędzi (15) oraz żaden wierzchołek nie jest izolowany. Do tego weźmy jeszcze graf cykliczny podobny do Graf 2. oraz graf Petersena.

Graf Petersena jest często używany w Teorii Grafów. Ma ciekawe właściwości, m.in.: jest silnie regularny, trójspójny, posiada Ścieżkę Hamiltona oraz jest krawędziowo i wierzchołkowo tranzytywny, czyli po prostu symetryczny.

Osobiście się trochę zdziwiłem, że na podstawie moich oczywistych wniosków utworzyłem graf bardzo podobny do Grafa Petersena.



Graf 5: Graf Petersena

3) Test topologii

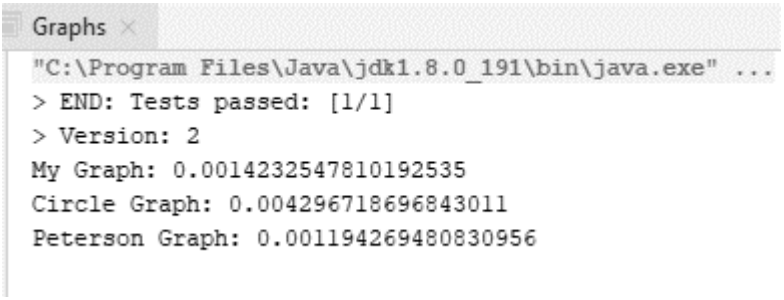
```
private static double averageTime
        (int[][] matrixIntensity,
         WeightedMultigraph<Integer, DefaultWeightedEdge> graphSecond){

    double matrixSum = 0;
    for(int k = 1; k <= 10; k++){
        for(int n = 1; n <= 10; n++){
            matrixSum += matrixIntensity[k][n];
        }
    }

    double sumEdgesEquation = 0;
    for (DefaultWeightedEdge e : graphSecond.edgeSet()) {

        int i = graphSecond.getEdgeSource(e);
        int j = graphSecond.getEdgeTarget(e);

        double a = calculateA(i, j, graphSecond);
        double c = capacity;

        sumEdgesEquation += (a/((c/bytesInPacket)-a));
    }

    return (1/matrixSum) * sumEdgesEquation;
}
```

Obrazek 6: średni czas opóźnienia

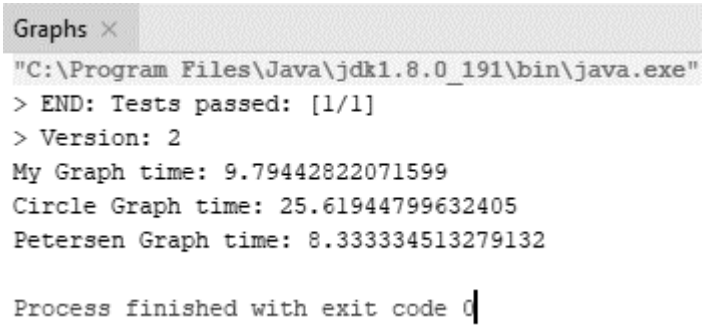
```
// Sum of intensity on shortest paths
private static double calculateA(int i, int j,
                WeightedMultigraph<Integer, DefaultWeightedEdge> graphSecond, int[][] matrix){

    // Get all packets that will go through that edge in this graph
    int packetsInEdge = 0;
    for(int a = 1; a < 10; a++){
        for(int b = 1; b < 10; b++) {
            GraphPath<Integer, DefaultWeightedEdge> shortest_path = DijkstraShortestPath.findPathBetween(graphSecond, a, b);

            if (shortest_path.getEdgeList().contains(graphSecond.getEdge(i, j))) {
                packetsInEdge += matrix[a][b];
            }
        }
    }

    if(capacity < packetsInEdge){
        System.out.println("[Error] Edge (" + i + "," + j + ") overloaded.");
        return 999999999;
    }

    return packetsInEdge;
}
```

Obrazek 7: obliczanie przepływu na krawędzi

Graf cykliczny nie dał rady przy capacity = 1000, więc przeprowadziłem jeden test na c = 1500. Graf Petersena okazał się być najlepszym grafem pod względem osiąganego średniego opóźnienia oraz co (dla mnie) zaskakujące, graf zaproponowany przeze mnie był niedaleko za nim. Liczby te są dosyć małe i człowiekowi za wiele nie powiedzą, ponieważ można nazwać je marginalnymi. Ale czy na pewno? Zwiększę teraz capacity na 1900, żeby mieć pewność powodzenia testu dla grafu kołowego, ponieważ na pojemności 1500 zdarza mu się przepełniać.



Obrazek 7: średni czas opóźnienia - test



Obrazek 8: średni czas opóźnienia – test x10000

Po lewej stronie na obrazku 8. podane są wyniki zsumowanego średniego opóźnienia na podstawie 10000 testów dla każdego z grafów. Teraz liczby są już bardziej klarowne, a sam eksperyment potwierdza wcześniej wysunięte wnioski.

Ostatnie, co pozostało sprawdzić to miary niezawodności sieci. Eksperyment będzie przeprowadzony na prawdopodobieństwie nie uszkodzenia krawędzi wynoszącym 95% oraz czasie do sukcesu = 20s oraz 10s na 100 próbach.

Przy 20s graf cykliczny nie daje sobie prawie kompletnie rady, a przy 10s (czas nim krawędzie się rozspójnią bądź ilość pakietów przechodzących przez krawędź > capacity) prawdopodobieństwo jest jak rzut monetą. Pozostałe dwa grafy wypadają nienajgorzej i całkiem podobnie przy 10s, lecz po podwojeniu Petersen jest wyraźnie lepszy.

Graf	Udane próby	Niezawodność	Sukces po T
Cykl	3/100	3%	20s
Cykl	55/100	55%	10s
Zaproponowany	68/100	68%	20s
Zaproponowany	92/100	92%	10s
Petersen	77/100	77%	20s
Petersen	98/100	98%	10s

Tabela 2: próby przejścia testu niezawodności

4. Podsumowanie

Można z tego wyciągnąć wnioski, że sieć powinna zostać skonstruowana odpowiednio do możliwości, a także i potrzeb. Niekoniecznie najszybsze rozwiązanie będzie najbardziej niezawodnym. Generalnie należy stawiać na dużą przepustowość, silne kable oraz przemyślaną topologię. Graf Petersena okazuje się być bardzo dobrym grafem pod każdym względem i można go brać za wzór podczas konstruowania sieci, zwłaszcza pod względem regularności i spójności.