Dacjan Naumowicz
WPPT
Informatyka
Technologie Sieciowe
Lista nr 2

Zadanie 1.

Rozważmy model sieci, w którym czas działania podzielony jest na interwały. Niech S = < G, H > będzie modelem sieci takim, że zbiór V grafu G = < V, E > zawiera 20 wierzchołków oznaczonych przez v(i), dla i = 1,..20; a zbiór E zawiera 19 krawędzi e(j,j+1), dla j = 1, ...,19, (przy czym zapis e(j,k) oznacza krawędź łączącą wierzchołki v(i) i v(k)). Zbiór H zawiera funkcję niezawodości 'h' przyporządkowującą każdej krawędzi e(j,k) ze zbioru E wartość 0.95 oznaczającą prawdopodobieństwo nieuszkodzenia (nierozerwania) tego kanału komunikacyjnego w dowolnym przedziale czasowym. (Zakładamy, że wierzchołki nie ulegaja uszkodzeniom).

- Napisz program szacujący niezawodność (rozumianą jako prawdopodobieństwo nierozspójnienia) takiej sieci w dowolnym interwale.
- Jak zmieni się niezawodność tej sieci po dodaniu krawędzi e(1,20) takiej, że h(e(1,20))=0.95
- A jak zmieni się niezawodność tej sieci gdy dodatkowo dodamy jeszcze krawędzie e(1,10) oraz e(5,15) takie, że: h(e(1,10))=0.8, a h(e(5,15))=0.7.
- A jak zmieni się niezawodność tej sieci gdy dodatkowo dodamy jeszcze 4 krawedzie pomiedzy losowymi wierzchołkami o h=0.4.

  Uwaga! Do szacowania niezawodności (spójności) najlepiej posłużyć się metodą Monte Carlo.

Zadanie realizuję w języku Java korzystając z frameworków JGraphT (framework pomagający w tworzeniu grafów i zapewniający podstawowe operacje na grafach) oraz JGraph (framework pomagający w wizualizacji grafów).

Graf będzie inicjowany na podstawie struktury:

```
private HashMap<String, HashSet<String>> graphStructure;
```

w której kluczami są nazwy wierzchołków (będą one nazywane odpowiednio v1, v2, …, v20) a wartościami jest zbiór wszystkich wierzchołków z którymi dany wierzchołek jest połączony krawędzią.

Drugą ważną strukturą jest struktura:

```
private HashMap<Edge, Double> edgeUnspoiltProbability;
```

której kluczami są krawędzie a wartościami prawdopodobieństwo nieuszkodzenia krawędzi (kanału komunikacyjnego). Struktura ta będzie używana podczas szacowania niezawodności sieci aby określić prawdopodobieństwo nieuszkodzenia dla każdej krawędzi.

Inicjacja struktury grafu polega na dodaniu pola z kluczem o nazwie wierzchołka i jego wartości, która jest zbiorem tych wierzchołków z którymi dany wierzchołek ma wspólną krawędź. Poniżej przedstawiona jest inicjacja grafu dla zadania 1 a).

```
private void initGraphStructure() {
    graphStructure = new HashMap<>();

    for(int i = 1; i <= vertexNumber; i++){
        HashSet<String> h = new HashSet<String>();
        if(i < vertexNumber){
            h.add("v" + (i + 1));
        }
        if(i > 1){
            h.add("v" + (i - 1));
        }
        graphStructure.put("v" + i, h);
    }
}
```

Inicjalizacja struktury odpowiedzialnej za przechowywanie prawdopodobieństwa dla danej krawędzi polega na wyodrębnieniu ze struktury grafu wszystkich krawędzi grafu i przypisanie im wartości 0.95, co jest częścią zadania 1.

```java
private void initEdgeUnspoiltProbability() {
    edgeUnspoiltProbability = new HashMap<>();
    for(String v1: graphStructure.keySet()){
        for(String v2: graphStructure.get(v1)){
            if(!edgeUnspoiltProbability.containsKey(new Edge(v1, v2))){
                edgeUnspoiltProbability.put(new Edge(v1, v2), 0.95);
            }
        }
    }
}
```

Mając te dwie struktury można przystąpić do wykorzystana frameworka JGraphT do którego można wprowadzić dane na podstawie wcześniej określonych struktur (w zasadzie tutaj potrzebna będzie tylko struktura grafu) co pokazuje poniższy wycinek kodu.

```java
public UndirectedGraph<String, DefaultEdge> createGraph(){
    UndirectedGraph<String, DefaultEdge> g =
            new SimpleGraph<String, DefaultEdge>(DefaultEdge.class);

    for(String v: graphStructure.keySet()){
        g.addVertex(v);
    }

    for (String v1 : graphStructure.keySet()) {
        for (String v : graphStructure.get(v1)) {
            g.addEdge(v1, v);
        }
    }
    return g;
}
```

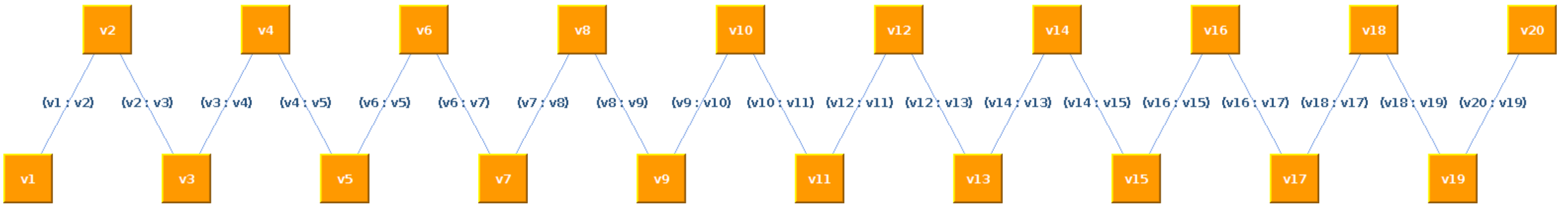
W zasadzie aby wykonać zadanie 1 a) nie potrzebne jest raczej wykonywanie testów, bo obliczenie niezawodności tej sieci jest bardzo proste: (p)^liczba krawędzi, dla zadania 1 a) będzie to (0.95)^19 = 0.3773536. Jednak testy na moim grafie również przeprowadziłem. Funkcja testująca działa na tej zasadzie, że zadaje się jej ilość testów do wykonania i ona uruchamia pętlę w której za każdym razem tworzy nowy graf następnie przechodzi po wszystkich krawędziach – dla każdej krawędzi losuje liczbę z przedziału 0 – 1 i sprawdza czy wylosowana wartość jest większa lub równa od prawdopodobieństwa nieuszkodzenia danej krawędzi. Jeśli wylosowana wartość jest większa, to dana krawędź zostaje usunięta, w przeciwnym razie nic nie robimy. Po przejściu po wszystkich krawędziach grafu sprawdzamy czy graf jest spójny, jeśli jest to inkrementujemy liczbę sukcesów i liczbę testów, jeśli nie jest to inkrementujemy tylko liczbę testów. W ten sposób dzieląc liczbę sukcesów przez liczbę testów otrzymujemy szacowaną niezawodność testów. Poniżej przedstawiam kod opisanej procedury:

```java
public TestNetworkResult testNetwork(int testSize){
    TestNetworkResult r = new TestNetworkResult();
    UndirectedGraph<String, DefaultEdge> g;
    for(int i = 0; i < testSize; i++){
        g = createGraph();
        runSingleNetworkTest(g);
        if(new ConnectivityInspector(g).isGraphConnected()){
            r.incrementSuccessNumber();
        }
        r.incrementTestNumber();
    }
    return r;
}


private void runSingleNetworkTest(UndirectedGraph<String, DefaultEdge> g){
    double randomNum;
    for(Edge e: edgeUnspoiltProbability.keySet()){
        randomNum = ThreadLocalRandom.current().nextDouble();
        if(randomNum >= edgeUnspoiltProbability.get(e)){
            g.removeEdge(e.getVertex1(), e.getVertex2());
        }
    }
}
```

Zgodnie z treścią zadania 1 a) wykonywałem testy uruchamiając poniższy kod (przeprowadzane testy bazują na próbie wykonywanej 100 000 razy) :

```java
public class Task1A {

    public static void main(String[] args){
        Graph graph = new Graph();
        System.out.println(graph.testNetwork(100000));

    }
}
```

Rozważany model sieci wygląda tak:



Oto przykładowe wyniki jakie otrzymywałem:

```
Test number: 100000 Success number: 37878
 as a percentage: 37.878 %

Test number: 100000 Success number: 37943
 as a percentage: 37.943 %

Test number: 100000 Success number: 38092
 as a percentage: 38.092 %

Test number: 100000 Success number: 37650
 as a percentage: 37.65 %

Test number: 100000 Success number: 37831
 as a percentage: 37.830999999999996 %

Test number: 100000 Success number: 37574
 as a percentage: 37.574000000000005 %

Test number: 100000 Success number: 37596
 as a percentage: 37.596000000000004 %

Test number: 100000 Success number: 37876
 as a percentage: 37.876 %
```

Wnioski:

- Wartości które otrzymywałem w testach są bardzo zbliżone do dokładnego wyniku obliczonego wcześniej

- Sieć cechuje się bardzo słabą niezawodnością, mimo dość wysokiego prawdopodobieństwa nieuszkodzenia pojedyńczego kanału komunikacyjnego, ma to niewątpliwie związek z tym, że wystarczy uszkodzenie jedynie jednego kanału do rozspójnienia sieci

W zadaniu 1 b) wykorzystywana będzie jedna funkcja nieopisana wcześniej:

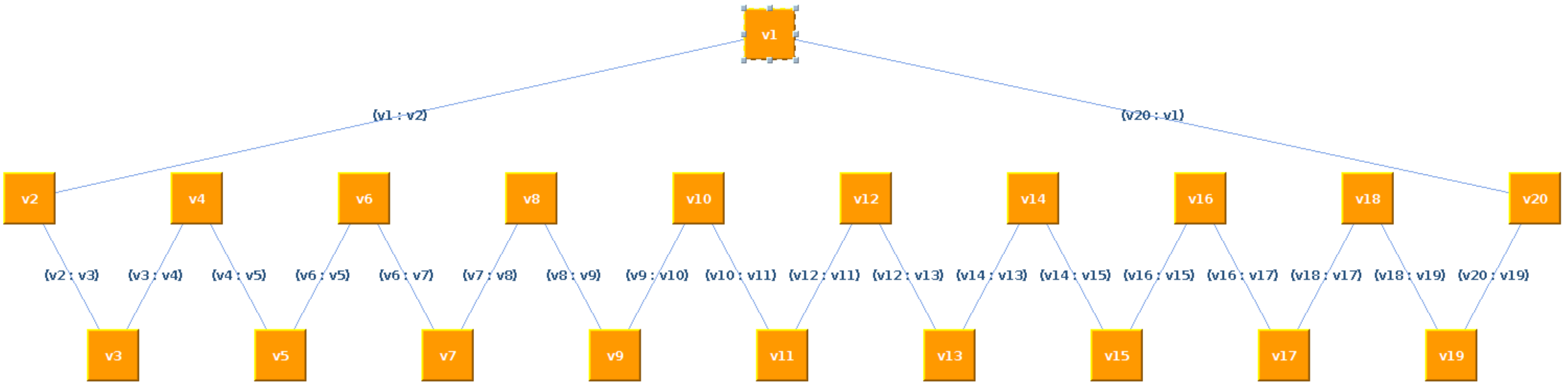
```java
public boolean addEdge(String v1, String v2, double unspoiltProbability){

    if(graphStructure.containsKey(v1) && graphStructure.containsKey(v2)){
        if(edgeExist(new Edge(v1, v2)) || v1.equals(v2)){
            return false;
        }
        graphStructure.get(v1).add(v2);
        graphStructure.get(v2).add(v1);
        initEdgeUnspoiltProbability();
        edgeUnspoiltProbability.put(new Edge(v1, v2), unspoiltProbability);
        return true;
    }

    return false;
}
```

Dodaje ona krawędź do grafu stworzonego na wzór grafu z zadania 1 a), oraz określa prawdopodobieństwo nieuszkodzenia tejże krawędzi.

Zadanie 1 b) różni się tym, że dodajemy krawędź łączącą wierzchołki v1 z v20. Kod którym wykonywne zostały testy do zadania 1 b) znajduje się poniżej (testy ponownie przeprowadzane są na próbie 100 000):

```java
public class Task1B {

    public static void main(String[] args){
        Graph graph = new Graph();

        UndirectedGraph<String, DefaultEdge> g = graph.createGraph();
        graph.addEdge("v1", "v20", 0.95);

        System.out.println(graph.testNetwork(100000).getPercentResult());
    }
}
```

Rozważany model sieci wygląda tak:



Oraz przykładowe wyniki:

```
Test number: 100000 Success number: 73592
 as a percentage: 73.592 %

Test number: 100000 Success number: 73470
 as a percentage: 73.47 %

Test number: 100000 Success number: 73769
 as a percentage: 73.76899999999999 %

Test number: 100000 Success number: 73668
 as a percentage: 73.668 %

Test number: 100000 Success number: 73585
 as a percentage: 73.585 %

Test number: 100000 Success number: 73628
 as a percentage: 73.628 %

Test number: 100000 Success number: 73445
 as a percentage: 73.44500000000001 %

Test number: 100000 Success number: 73407
 as a percentage: 73.407 %
```

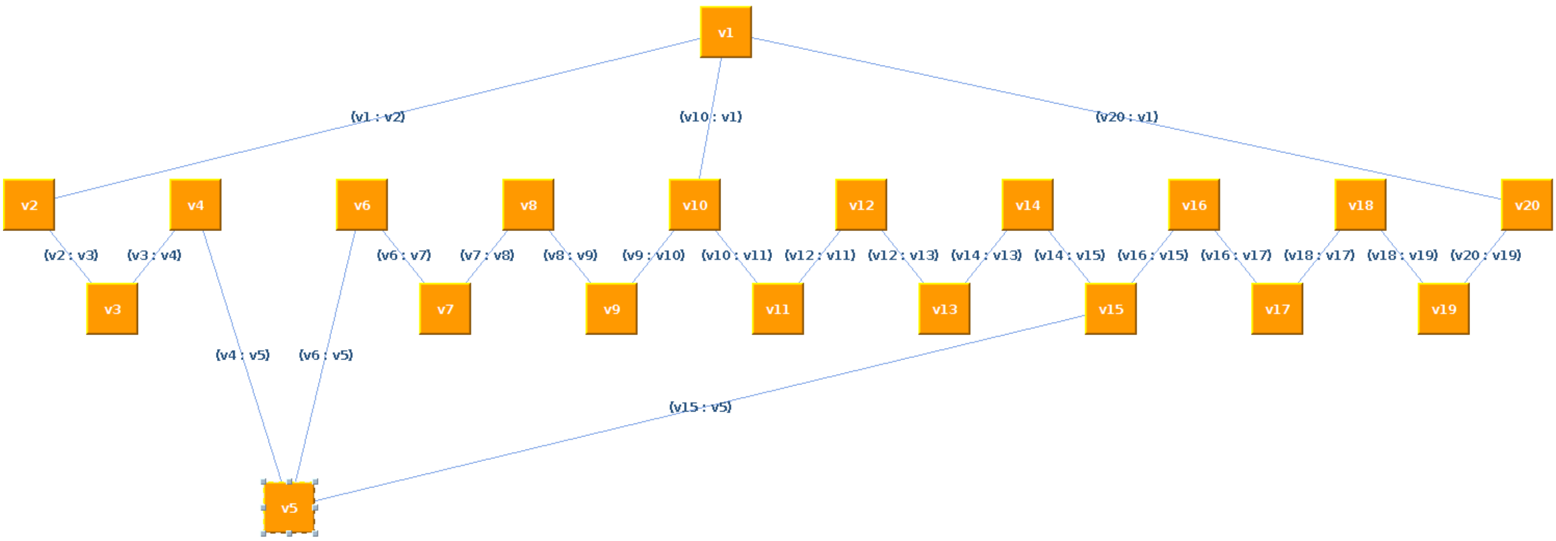
Wnioski:

- Wyraźnie widać, że po dodaniu jednej krawędzi z takim samym prawdopodobieństwem jak wszystkie pozostałe w taki sposób żeby rozważany graf przekształcić w jeden duży cykl, zwiększyło prawdopodobieństwo niezawodności niemalże dokładnie dwukrotnie
- Powodem takiego stanu rzeczy jest niewątpliwie to, że aby w tym momencie rozspójnić rozważany model trzeba uszkodzić dwie krawędzie

Podpunkt c) różni się od pozostałych tym, że dodajemy jeszcze krawędzie łączące v1 z v10 z prawdopodobieństwem niezawodności 0.8 oraz krawędź łączącą v5 z v15 z prawdopodobieństwem niezawodności 0.7. Kod którym wykonywne zostały testy do zadania 1 c) znajduje się poniżej (testy ponownie przeprowadzane są na próbie 100 000):

```java
public class Task1C {

    public static void main(String[] args){
        Graph graph = new Graph();

        UndirectedGraph<String, DefaultEdge> g = graph.createGraph();
        graph.addEdge("v1", "v20", 0.95);
        graph.addEdge("v1", "v10", 0.8);
        graph.addEdge("v5", "v15", 0.7);

        System.out.println(graph.testNetwork(100000));


    }
}
```

Rozważany model sieci wygląda tak:



Oraz przykładowe wyniki:

```
Test number: 100000 Success number: 88410
 as a percentage: 88.41 %

Test number: 100000 Success number: 88318
 as a percentage: 88.318 %

Test number: 100000 Success number: 88519
 as a percentage: 88.519 %

Test number: 100000 Success number: 88251
 as a percentage: 88.251 %

Test number: 100000 Success number: 88362
 as a percentage: 88.362 %

Test number: 100000 Success number: 88265
 as a percentage: 88.265 %

Test number: 100000 Success number: 88227
 as a percentage: 88.227 %

Test number: 100000 Success number: 88273
 as a percentage: 88.273 %
```

Wnioski:

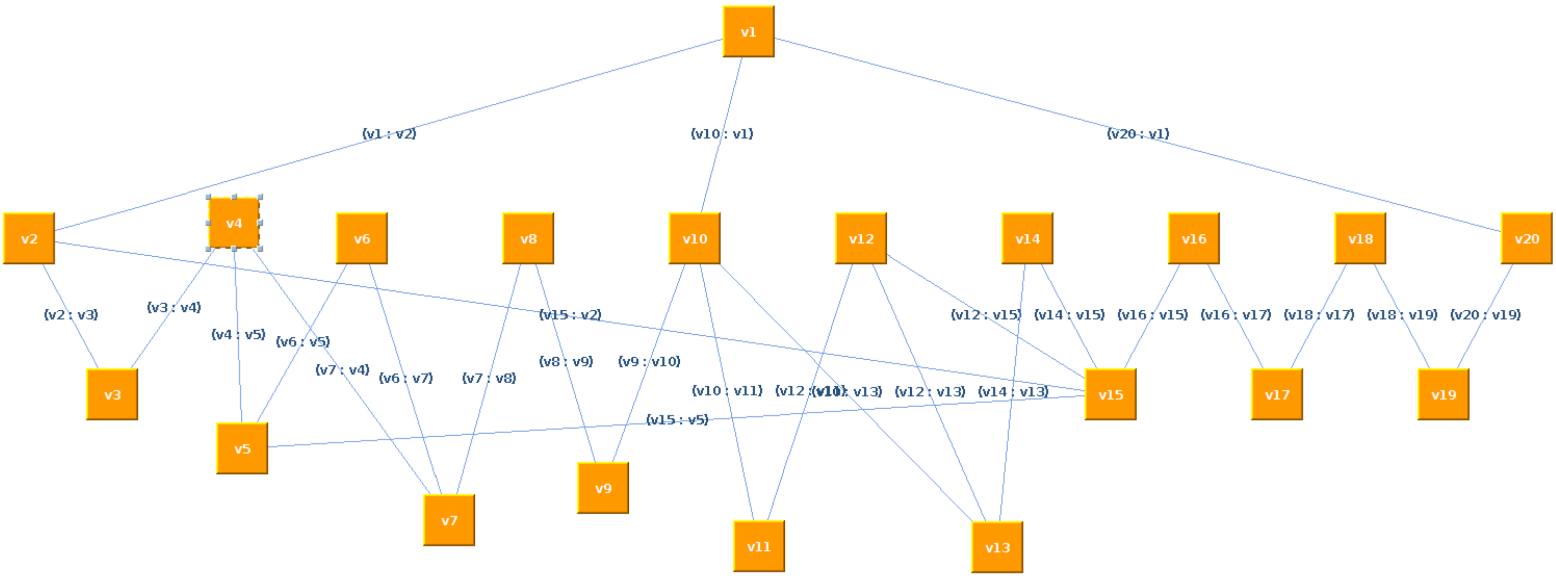
- Mimo że dodawane krawędzie mają mniejsze prawdopodobieństwo nieuszkodzenia to niezawodność sieci dość wyraźnie wzrasta
- Nowe krawędzie zabezpieczają graf w większym stopniu przed rozspójnieniem (np. krawędź od v5 do v15 zabezpiecza krawędzie e(v6, v7), e(v7, v8), …, e(v14, v15) w taki sposób że jedna z nich może uledz rozspójnieniu a mimo to graf nadal będzie nie dość że spójny to jeszcze będzie zawierał cykl, co daje mu dodatkowe możliwości uszkodzenia krawędzi bez rozspójnienia całej sieci)

Podpunkt d) różni się od pozostałych tym, że dodatkowo dodajemy 4 losowe krawędzie tym razem z prawdopodobieństwem nieuszkodzenia tylko 0.4 pomiędzy wierzchołkami naszego dotychczasowego grafu (zakładam że nie może być krawędzi wielokrotnych pomiędzy tymi samymi wierzchołkami). Kod którym zostały wykonane testy do zadania 1 d) znajduje się poniżej (testy ponownie przeprowadzone na próbie 100 000):

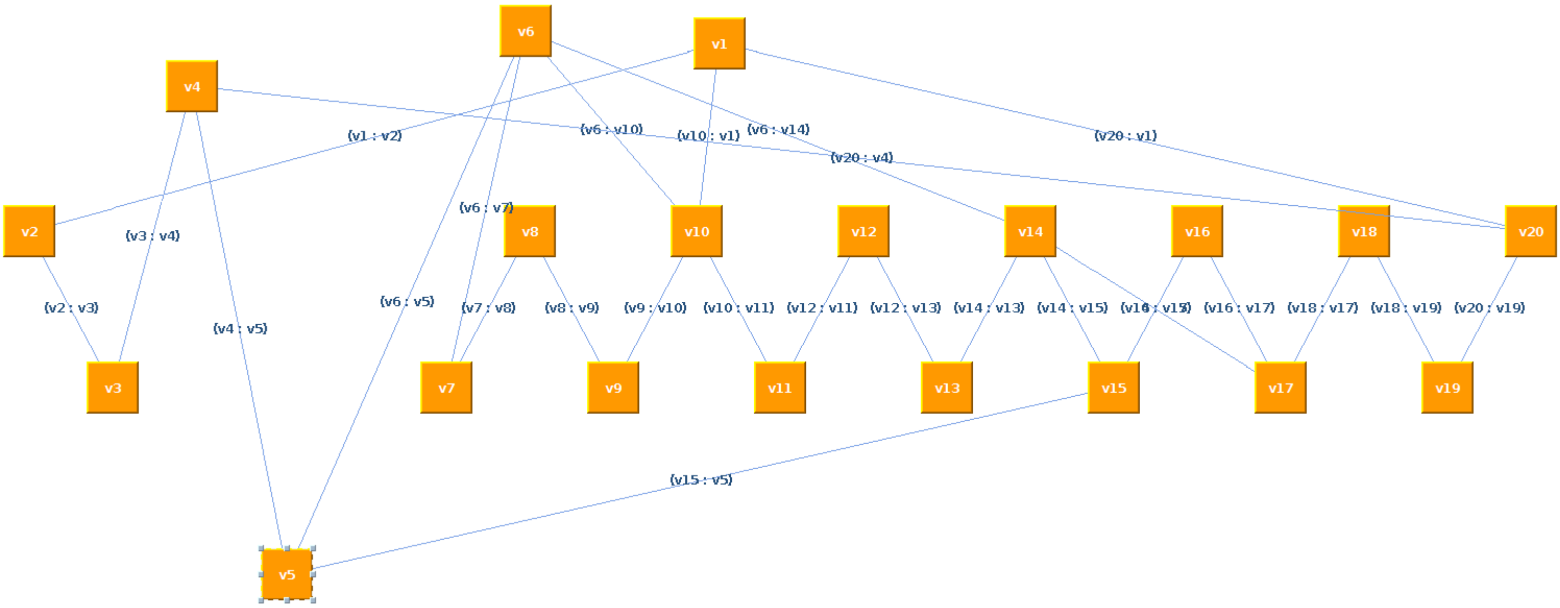
```java
public class Task1D {

    public static void main(String[] args){
        Graph graph = new Graph();
        int i, randomNum1, randomNum2;

        graph.addEdge("v1", "v20", 0.95);
        graph.addEdge("v1", "v10", 0.8);
        graph.addEdge("v5", "v15", 0.7);

        i = 0;
        randomNum1 = ThreadLocalRandom.current().nextInt(1, 20 + 1);
        randomNum2 = ThreadLocalRandom.current().nextInt(1, 20 + 1);
        while(i < 4){
            if(graph.addEdge("v" + randomNum1, "v" + randomNum2, 0.4)){
                i++;
                System.out.println("add edge: " + "e(" + "v" + randomNum1 + ", " + "v" + randomNum2 + ")");
            }
            randomNum1 = ThreadLocalRandom.current().nextInt(1, graph.getVertexNumber() + 1);
            randomNum2 = ThreadLocalRandom.current().nextInt(1, graph.getVertexNumber() + 1);
        }
        System.out.println(graph.testNetwork(100000));
    }
}
```

Zachowujemy tutaj strukturę grafu z poprzedniego podpunktu i dodajemy 4 losowe krawędzie.

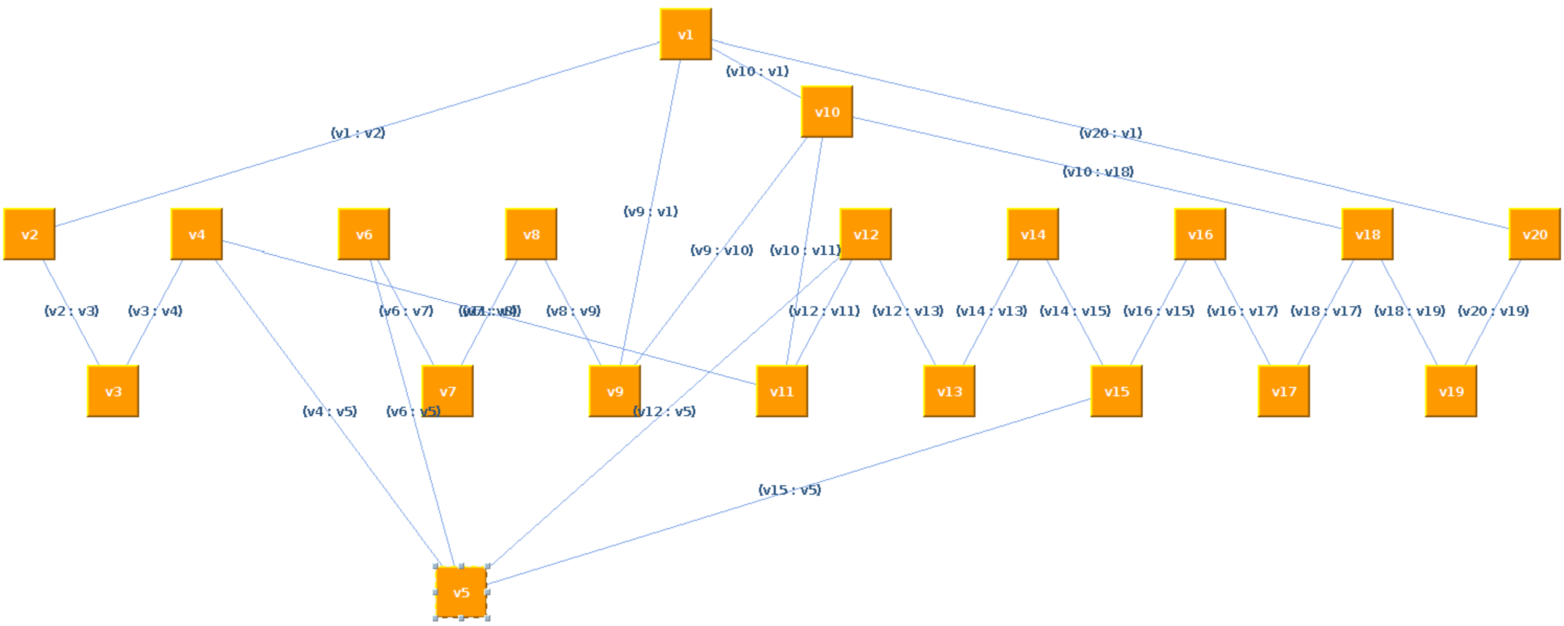
Oraz przykładowe wyniki:



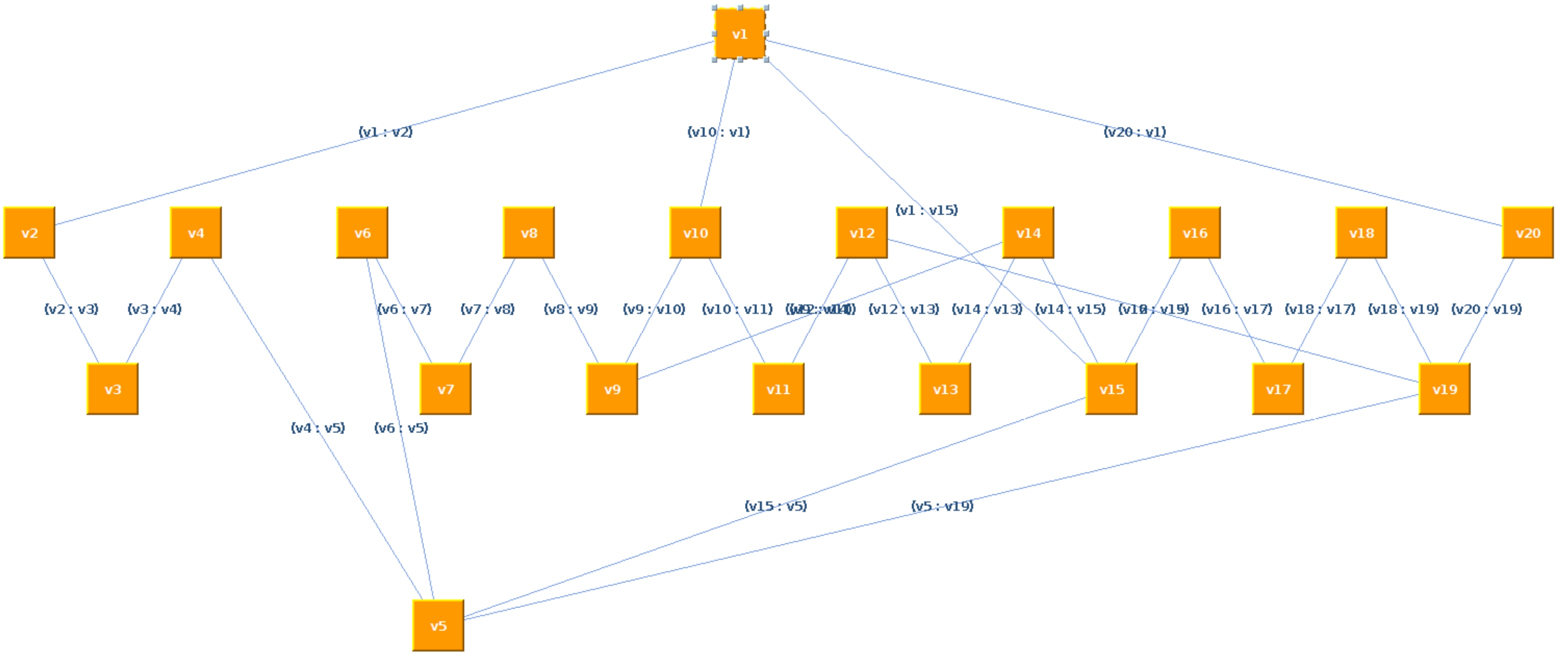

```
add edge: e(v10, v13)
add edge: e(v15, v2)
add edge: e(v12, v15)
add edge: e(v7, v4)
Test number: 100000 Success number: 93791
 as a percentage: 93.791 %
```




```
add edge: e(v17, v14)
add edge: e(v4, v20)
add edge: e(v6, v14)
add edge: e(v6, v10)
Test number: 100000 Success number: 95065
 as a percentage: 95.065 %
```




```
add edge: e(v11, v4)
add edge: e(v12, v5)
add edge: e(v9, v1)
add edge: e(v10, v18)
Test number: 100000 Success number: 94485
 as a percentage: 94.485 %
```




```
add edge: e(v19, v12)
add edge: e(v9, v14)
add edge: e(v19, v5)
add edge: e(v1, v15)
Test number: 100000 Success number: 95012
 as a percentage: 95.012 %
```

Wnioski:

- W porównaniu z wcześniejszym podpunktem notujemy w dalszym ciągu istotny wzrost niezawodności sieci (około 7%), mimo tego że tym razem dodawane krawędzie mają bardzo niskie prawdopodobieństwo niezawodności bo jest to wartość 0.4
- Da się zauważyć, że jeśli wylosowane zostały krawędzie stosunkowo blisko położonych siebie wierzchołków to niezawodność sieci jest mniejsza niż jeżeli te krawędzie zostały wylosowane dla wierzchołków oddalonych od siebie bardziej
- Widoczna jest dość duża różnica pomiędzy pojedynczymi testami, co do tej pory nie było widoczne. Ma to niewątpliwie związek z tym, że część krawędzi dodawanych do grafu ma duże przełożenie na jego niezawodności a inna część mniejsze.

Zadanie 2.

2.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).

•Zaproponuj topologię grafu G ale tak aby żaden wierzchołek nie był izolowany oraz aby: |V|=10, |E|<20. 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 komunikacyjnego w ciągu sekundy). Pamiętaj aby funkcja przeplywu realizowała macierz N oraz aby dla każdego kanału 'e' zachodziło: c(e) > a(e).
•Napisz program, w którym propozycje będzie można testować, tzn. który dla wybranych reprezentacji zadanych odpowiednimi macierzami, będzie obliczał średnie opóźnienie pakietu 'T' dane wzorem: T = 1/G * SUM_e(a(e)/(c(e)/m - a(e))), gdzie 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.
•Niech miarą niezawodności sieci jest prawdopodobieństwo tego, że w dowolnym przedziale czasowym, nierozspójniona sieć zachowuje T < T_max. 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. Napisz sprawozdanie!

W celu ułatwienia testów wszystkie dane odnośnie macierzy natężeń strumienia pakietów(N), funkcji przepustowości (c), faktyczną liczbę pakietów, które wprowadza się do kanału komunikacyjnego w ciągu sekundy (a), maksymalne opóźnienie pakietu (T_max) oraz prawdopodobieństwo nieuszkodzenia każdej krawędzi w dowolnym interwale (p) są umieszczone w zewnętrznym pliku o nazwie data.json. Poniższy screen przedstawia zmienne które są uzupełniane danymi z zewnętrznego pliku bądź obliczane na ich podstawie.

```
private ArrayList<String> vertexes;
private HashMap<Edge, EdgeProperty> edges;
private String dataFilePath;
private double T_max;
private HashMap<String, HashMap<String, Integer>> packagesIntensityMap;
private int packagesIntensitySum;
```

Struktura vertixes zawiera wszystkie wierzcholki modelu sieci.

Struktura edges zawiera wszystkie krawedzie wraz z wymaganymi informacjami:

```
public class EdgeProperty{
    private int capacity;
    private int currentPackages;
    private double unspoiltProbability;
    private int averagePackageBitNumber = 1480;
```

T_max to oczywiście maksymalne opóźnienie pakietu. Struktura packagesIntensityMap zawiera dane odnośnie natężeń strumienia pakietów, natomiast packegesIntensitySum jest sumą wszystkich elementów macierzy natężeń. Poniżej ilustruje metodę która jest odpowiedzialna za inicjacje danych dla wszystkich tych struktur:

```
public void initGraphStructure() throws IOException {
    initVertexes();
    initEdges();
    initPackagesIntensityMap();
    packagesIntensitySum = getSumPackagesIntensityMap();
}
```

Omawiając po krotce podstawowe struktury danych pozwolę sobie przejść do sedna programu czyli funkcji testującej model sieci, oto ona:

```
private NetworkTestResult networkSingleTest(){
    initEdges();
    g = createGraph();
    for(String v: packagesIntensityMap.keySet()){
        for(String vv: packagesIntensityMap.get(v).keySet()){
            String currentVertex = v;
            String nextVertex = "";
            while(!nextVertex.equals(vv)){
                nextVertex = getNextShortestEdge(currentVertex, g, vv).getVertex2();
                currentVertex = getNextShortestEdge(currentVertex, g, vv).getVertex1();

                if(!edges.get(new Edge(currentVertex, nextVertex)).addCurrentPackages(packagesIntensityMap.get(v).get(vv))){
                    return NetworkTestResult.PACKAGES_INTENSITY_FAIL;
                }
                if(sum_e() > T_max){
                    return NetworkTestResult.TIME_FAIL;
                }
                if(edges.get(new Edge(currentVertex, nextVertex)).getUnspoiltProbability() < ThreadLocalRandom.current().nextDouble()){
                    g.removeEdge(currentVertex, nextVertex);
                    edges.remove(new Edge(currentVertex, nextVertex));
                    if(!new ConnectivityInspector(g).isGraphConnected()){
                        return NetworkTestResult.GRAPH_INCONNECTION_FAIL;
                    }
                }
                currentVertex = nextVertex;
            }
        }
    }
    return NetworkTestResult.SUCCESSFUL;
}
```

Funkcja zaczyna się od wywołania initEdges(); - jest to wywołanie funkcji pobierającej na nowo dane z pliku zewnętrznego o krawędziach, jest to wymagane ponieważ funkcja będzie wykorzystywana aby testować model sieci wielokrotnie, a chcielibyśmy aby testy były przeprowadzane za każdym razem na pełnym modelu. W kolejnej linijce wywołujemy funkcje createGraph(); robimy to w celu takim samym jak przy poprzedniej funkcji (zabezpieczany się przed tym, ze w poprzednim teście niektóre krawędzie mogły zostać usunięte). Następnie w celu dostania się do informacji o przesyłaniu pakietów miedzy węzłami robimy 2 fory które. Pierwszy for iteruje po węzłach z których wysyłamy pakiet, a drugi po węzłach do których ten pakiet ma dotrzeć. Najkrótsza ścieżkę obliczamy za pomocą funkcji getNextShostestEdge:

```java
public Edge getNextShortestEdge(String v, UndirectedGraph<String, DefaultEdge> g, String v3){
    GraphPath a = (GraphPath) new KShortestPaths(g, v, 1).getPaths(v3).get(0);
    String currentVertex = getSecondVertexFromEdgeSyntax(a.getEdgeList().get(0).toString());
    String nextVertex = getFirtsVertexFromEdgeSyntax(a.getEdgeList().get(0).toString());
    if(currentVertex.equals(v3)){
        currentVertex = nextVertex;
        nextVertex = v3;
    }
    if(!nextVertex.equals(v3)){
        nextVertex = ((GraphPath) new KShortestPaths(g, nextVertex, 1).getPaths(v3).get(0)).getEdgeList().size() <
                ((GraphPath) new KShortestPaths(g, currentVertex, 1).getPaths(v3).get(0)).getEdgeList().size()
                ? nextVertex : currentVertex;
    }
   if(nextVertex.equals(currentVertex)){
        currentVertex = getFirtsVertexFromEdgeSyntax(a.getEdgeList().get(0).toString());
    }
    return new Edge(currentVertex, nextVertex);
}
```

Która się nie co skomplikowała w związku z tym, ze funkcja z frameworka KshortestPaths (funkcja zwraca listę krawędzi mówiąca jaka jest najkrótsza droga od wierzchołka vy do vx) nie zawsze zwraca wierzchołki w odpowiedniej kolejności. Następnie wpadamy w pętlę tym razem while która ma za zadanie przetransportować nasz pakiet od jednego węzła do następnego, po najkrótszej ścieżce jednocześnie sprawdzając czy test nie powinien się zakończyć niepowodzeniem, a niepowodzeniem może zakończyć się w 3 przypadkach:
- jeśli c(e) / m < a(e) czyli jeśli zostanie przekroczona pojemność danej krawędzi

```java
    private double sum_e(){
        double s = 0.0;
        for(Edge e: edges.keySet()){
            s += (double)edges.get(e).getCurrentPackages()/((double)edges.get(e).getCapacity()
                    - (double)edges.get(e).getCurrentPackages());
        }
        return (double)s / (double)packagesIntensitySum;
    }
```

- jesli średnie opóźnienie pakietu zostalo przekroczone (T > T_max)

dzieje sie tu dokladnie to co w tresci zadania “T = 1/G * SUM_e(a(e)/(c(e)/m - a(e))), gdzie SUM_e oznacza sumowanie po wszystkich krawędziach 'e' ze zbioru E”

- jesli nastapi uszkodzenie krawedzi ktore spowoduje ze graf przestaje byc spojny

Jesli przelecimy przez wszystkie wartosci macierzy natezen strumienia pakietow bez wystapienia zadnego z powyzszych bledow, to znaczy ze test zakonczyl sie sukcesem.

Została do omówienia tylko jedna funkcja która ma za zadanie wykonanie określonej liczby testów oraz przeprowadzenie statystyki odnośnie tego co było powodem niepowodzenia.

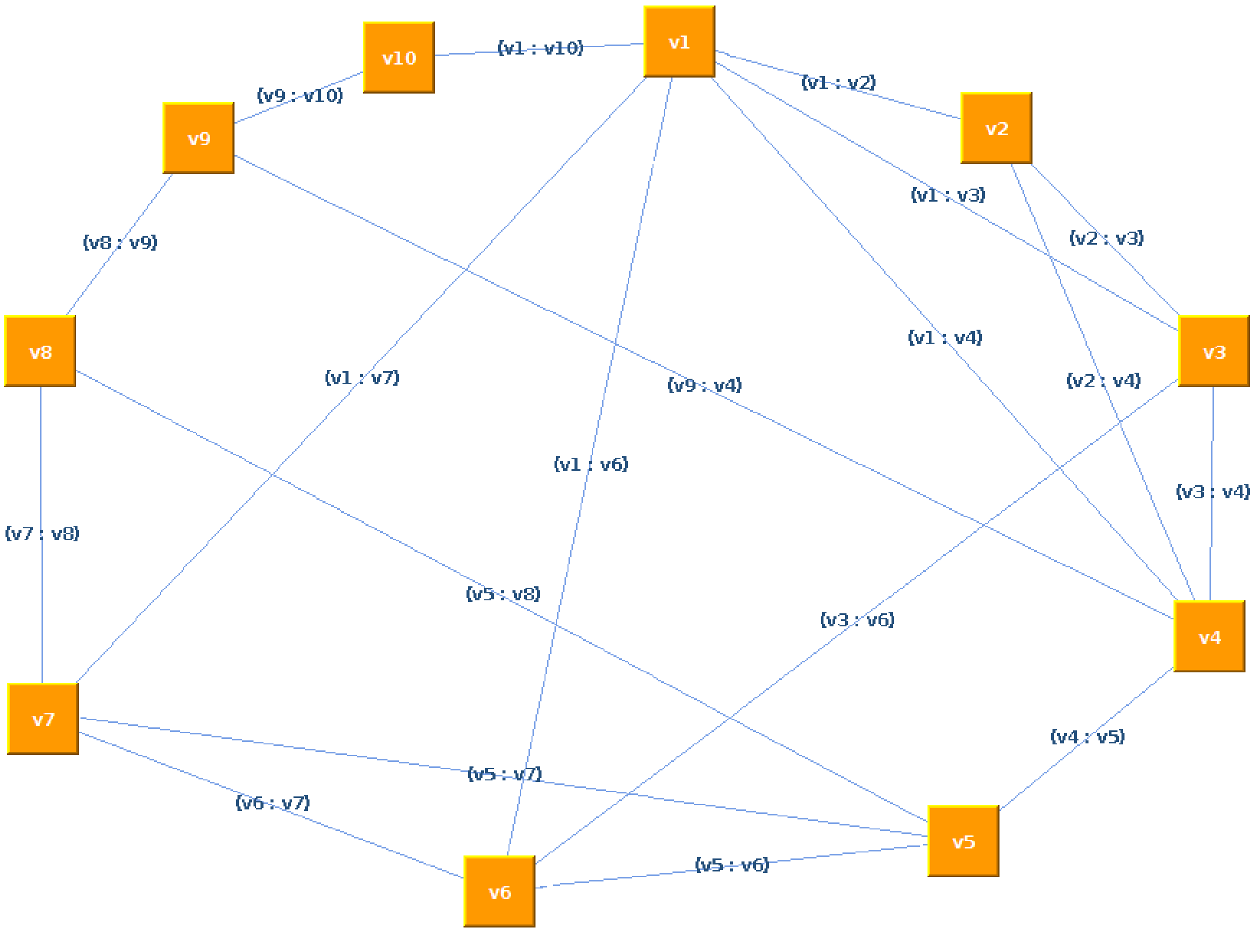
```java
public void networkTest(int n){
    int testSuccessful = 0;
    int packagesIntensityFail = 0;
    int graphInconnectionFail = 0;
    int timeFail = 0;
    for(int i = 0; i < n; i++){
        NetworkTestResult t= networkSingleTest();
        if(t == NetworkTestResult.PACKAGES_INTENSITY_FAIL){
            packagesIntensityFail++;
        }
        else if(t == NetworkTestResult.GRAPH_INCONNECTION_FAIL){
            graphInconnectionFail++;
        }
        else if(t == NetworkTestResult.TIME_FAIL){
            timeFail++;
        }
        else if(t == NetworkTestResult.SUCCESSFUL){
            testSuccessful++;
        }
    }
    System.out.println("test number: " + n + " successful tests: " +
            testSuccessful + " reliability: " +
            (double)testSuccessful/(double)n * 100  + " %  packages intensity fail: "
            + packagesIntensityFail + " graph inconnection fail: " +
            graphInconnectionFail + " time fail: "
            + timeFail);
}
```

Wykonujemy tutaj testy w petli od 1 do n oraz zbieramy informacje na temat rodzaju bledu ktory wystapil. Na koniec wyswietlamy komunikat. Ponizej przedstawiam program za pomoca ktorego uruchamiam testy:

```java
public class Task2 {
    public static void main(String[] args) throws IOException {
        Graph2 g = new Graph2();
        g.initGraphStructure();
        g.networkTest(1000);
    }
}
```

Jak widac wiekszosc testow bede przeprowadzal dla 100 prob.

Pierwsze testy beda przeprowadzane dla modelu sieci o 10 wezlach oraz 19 krawedziach o ponizszej strukturze:



Test1

Natomiast plik zawierajacy szczegolowe informacje wyglada tak:


```
{
  "T_max": 0.25,
  "edge-unspoilt-probability": 0.99,
  "edge-capacity": [
    { "vertex1": "v1", "vertex2": "v2", "capacity": 55000 },
    { "vertex1": "v1", "vertex2": "v3", "capacity": 32000 },
    { "vertex1": "v1", "vertex2": "v4", "capacity": 67500 },
    { "vertex1": "v1", "vertex2": "v6", "capacity": 95000 },
    { "vertex1": "v1", "vertex2": "v7", "capacity": 80000 },
    { "vertex1": "v1", "vertex2": "v10", "capacity": 65000 },
    { "vertex1": "v2", "vertex2": "v3", "capacity": 35000 },
    { "vertex1": "v2", "vertex2": "v4", "capacity": 47000 },
    { "vertex1": "v3", "vertex2": "v4", "capacity": 32000 },
    { "vertex1": "v3", "vertex2": "v6", "capacity": 55000 },
    { "vertex1": "v4", "vertex2": "v5", "capacity": 48500 },
    { "vertex1": "v5", "vertex2": "v6", "capacity": 57000 },
    { "vertex1": "v5", "vertex2": "v7", "capacity": 45500 },
    { "vertex1": "v5", "vertex2": "v8", "capacity": 45500 },
    { "vertex1": "v6", "vertex2": "v7", "capacity": 70000 },
    { "vertex1": "v7", "vertex2": "v8", "capacity": 75500 },
    { "vertex1": "v8", "vertex2": "v9", "capacity": 57500 },
    { "vertex1": "v9", "vertex2": "v10","capacity": 50500 },
    { "vertex1": "v9", "vertex2": "v4","capacity": 50500 }],
  "packages-intensity-matrix":[
    {"v1": {"v2": 2, "v3":2,"v4":8, "v5":1, "v6":9, "v7":5, "v8":3, "v9":1, "v10":3}},
    {"v2": {"v1": 4, "v3":2,"v4":5, "v5":1, "v6":4, "v7":2, "v8":2, "v9":1, "v10":1}},
    {"v3": {"v1": 6, "v2": 3, "v4":3, "v5":2, "v6":7, "v7":1, "v8":2, "v9":2, "v10":1}},
    {"v4": {"v1": 8, "v2": 6, "v3":3, "v5":4, "v6":7, "v7":3, "v8":2, "v9":1, "v10":1}},
    {"v5": {"v1": 6, "v2": 3, "v3":2,"v4":5, "v6":6, "v7":4, "v8":3, "v9":1, "v10":2}},
    {"v6": {"v1": 9, "v2": 2, "v3":6,"v4":3, "v5":5, "v7":4, "v8":3, "v9":1, "v10":2}},
    {"v7": {"v1": 6, "v2": 1, "v3":2,"v4":5, "v5":2, "v6":6, "v8":3, "v9":2, "v10":1}},
    {"v8": {"v1": 5, "v2": 2, "v3":1,"v4":4, "v5":4, "v6":5, "v7":2, "v9":1, "v10":1}},
    {"v9": {"v1": 4, "v2": 1, "v3":1,"v4":3, "v5":1, "v6":4, "v7":1, "v8":4, "v10":3}},
    {"v10": {"v1": 5, "v2": 2, "v3":1,"v4":4, "v5":2, "v6":4, "v7":2, "v8":1, "v9":2}}

  ]
}
```


Oto przykladowe wywolania:

```
test number: 1000 successful tests: 695 reliability: 69.5 %  packages intensity fail: 213 graph inconnection fail: 21 time fail: 71

test number: 1000 successful tests: 698 reliability: 69.8 %  packages intensity fail: 226 graph inconnection fail: 12 time fail: 64

test number: 1000 successful tests: 708 reliability: 70.8 %  packages intensity fail: 208 graph inconnection fail: 22 time fail: 62

test number: 1000 successful tests: 690 reliability: 69.0 %  packages intensity fail: 215 graph inconnection fail: 20 time fail: 75

test number: 1000 successful tests: 717 reliability: 71.7 %  packages intensity fail: 204 graph inconnection fail: 19 time fail: 60

test number: 1000 successful tests: 678 reliability: 67.80000000000001 %  packages intensity fail: 236 graph inconnection fail: 14 time fail: 72

test number: 1000 successful tests: 713 reliability: 71.3 %  packages intensity fail: 216 graph inconnection fail: 18 time fail: 53

test number: 1000 successful tests: 693 reliability: 69.3 %  packages intensity fail: 216 graph inconnection fail: 24 time fail: 67
```

Wnioski:

- Główną przyczyna występowania błędów jest przeciążenie pakietami na krawędziach, w celu poprawienia statystyki oczywistym wydaje się powiększenie capacity dla poszczególnych krawędzi Mimo ze maksymalne opóźnienie nie gra tutaj kluczowej roli jeśli chodzi o niezawodność, to jest jednak zauważalne
- Problem rozspójnienia sieci wydaje się całkiem mały. Może to sugerować, ze struktura modelu sieci jest dobrze przemyślana lub to ze prawdopodobieństwo nieuszkodzenia krawędzi jest wysokie.

Test2

W kolejnym teście postanowiłem spróbować podnieść niezawodność sieci I pomnożyłem pojemność każdej krawędzi przez 10. Plik z danymi wyglądał wówczas tak:


```
{
  "T_max": 0.25,
  "edge-unspoilt-probability": 0.99,
  "edge-capacity": [
    { "vertex1": "v1", "vertex2": "v2", "capacity": 550000 },
    { "vertex1": "v1", "vertex2": "v3", "capacity": 320000 },
    { "vertex1": "v1", "vertex2": "v4", "capacity": 675000 },
    { "vertex1": "v1", "vertex2": "v6", "capacity": 950000 },
    { "vertex1": "v1", "vertex2": "v7", "capacity": 800000 },
    { "vertex1": "v1", "vertex2": "v10", "capacity": 650000 },
    { "vertex1": "v2", "vertex2": "v3", "capacity": 350000 },
    { "vertex1": "v2", "vertex2": "v4", "capacity": 470000 },
    { "vertex1": "v3", "vertex2": "v4", "capacity": 320000 },
    { "vertex1": "v3", "vertex2": "v6", "capacity": 550000 },
    { "vertex1": "v4", "vertex2": "v5", "capacity": 485000 },
    { "vertex1": "v5", "vertex2": "v6", "capacity": 570000 },
    { "vertex1": "v5", "vertex2": "v7", "capacity": 455000 },
    { "vertex1": "v5", "vertex2": "v8", "capacity": 455000 },
    { "vertex1": "v6", "vertex2": "v7", "capacity": 700000 },
    { "vertex1": "v7", "vertex2": "v8", "capacity": 755000 },
    { "vertex1": "v8", "vertex2": "v9", "capacity": 575000 },
    { "vertex1": "v9", "vertex2": "v10","capacity": 505000 },
    { "vertex1": "v9", "vertex2": "v4","capacity": 505000 }],
  "packages-intensity-matrix":[
    {"v1": {"v2": 2, "v3":2,"v4":8, "v5":1, "v6":9, "v7":5, "v8":3, "v9":1, "v10":3}},
    {"v2": {"v1": 4, "v3":2,"v4":5, "v5":1, "v6":4, "v7":2, "v8":2, "v9":1, "v10":1}},
    {"v3": {"v1": 6, "v2": 3, "v4":3, "v5":2, "v6":7, "v7":1, "v8":2, "v9":2, "v10":1}},
    {"v4": {"v1": 8, "v2": 6, "v3":3, "v5":4, "v6":7, "v7":3, "v8":2, "v9":1, "v10":1}},
    {"v5": {"v1": 6, "v2": 3, "v3":2,"v4":5, "v6":6, "v7":4, "v8":3, "v9":1, "v10":2}},
    {"v6": {"v1": 9, "v2": 2, "v3":6,"v4":3, "v5":5, "v7":4, "v8":3, "v9":1, "v10":2}},
    {"v7": {"v1": 6, "v2": 1, "v3":2,"v4":5, "v5":2, "v6":6, "v8":3, "v9":2, "v10":1}},
    {"v8": {"v1": 5, "v2": 2, "v3":1,"v4":4, "v5":4, "v6":5, "v7":2, "v9":1, "v10":1}},
    {"v9": {"v1": 4, "v2": 1, "v3":1,"v4":3, "v5":1, "v6":4, "v7":1, "v8":4, "v10":3}},
    {"v10": {"v1": 5, "v2": 2, "v3":1,"v4":4, "v5":2, "v6":4, "v7":2, "v8":1, "v9":2}}

  ]
}
```


A oto przykladowe wywolania:

```
test number: 1000 successful tests: 974 reliability: 97.39999999999999 %  packages intensity fail: 0 graph inconnection fail: 26 time fail: 0

test number: 1000 successful tests: 971 reliability: 97.1 %  packages intensity fail: 0 graph inconnection fail: 29 time fail: 0

test number: 1000 successful tests: 974 reliability: 97.39999999999999 %  packages intensity fail: 0 graph inconnection fail: 26 time fail: 0

test number: 1000 successful tests: 965 reliability: 96.5 %  packages intensity fail: 0 graph inconnection fail: 35 time fail: 0

test number: 1000 successful tests: 974 reliability: 97.39999999999999 %  packages intensity fail: 0 graph inconnection fail: 26 time fail: 0

test number: 1000 successful tests: 979 reliability: 97.89999999999999 %  packages intensity fail: 0 graph inconnection fail: 21 time fail: 0

test number: 1000 successful tests: 973 reliability: 97.3 %  packages intensity fail: 0 graph inconnection fail: 27 time fail: 0

test number: 1000 successful tests: 980 reliability: 98.0 %  packages intensity fail: 0 graph inconnection fail: 20 time fail: 0
```

Wnioski:

- Niezawodnosc sieci w znaczacy sposob sie zwiekszyla
- Bledy spowodowane przepelnieniem danych na krawedziach zmniejszylo sie do 0
- Bledy spowodowane przekroczeniem maksymalnego czasu opoznienia rowniez spadly do zera, co mozna wytlumaczyc wzorem T = 1/G * SUM_e(a(e)/(c(e)/m - a(e)))
- Bledy spowodowane rozspojnieniem zachowaly wczesniejsze wartosci, p zostalo bez zmian wiec taki wynik nie powinien zaskakiwac

Test3

W kolejnym tescie postanowilem przetestowac czulosc wskaznika p (prawdopodobienstwo nieuszkoczenia kazdej krawedzi), wskaznik ten smniejszylem do wartosci:

```
"edge-unspoilt-probability": 0.95,
```

Oto przykladowe wyniki:

```
test number: 1000 successful tests: 287 reliability: 28.7 %  packages intensity fail: 0 graph inconnection fail: 713 time fail: 0
test number: 1000 successful tests: 307 reliability: 30.7 %  packages intensity fail: 0 graph inconnection fail: 693 time fail: 0
test number: 1000 successful tests: 274 reliability: 27.400000000000002 %  packages intensity fail: 0 graph inconnection fail: 726 time fail: 0
test number: 1000 successful tests: 267 reliability: 26.700000000000003 %  packages intensity fail: 0 graph inconnection fail: 733 time fail: 0
test number: 1000 successful tests: 291 reliability: 29.099999999999998 %  packages intensity fail: 0 graph inconnection fail: 709 time fail: 0
test number: 1000 successful tests: 288 reliability: 28.799999999999997 %  packages intensity fail: 0 graph inconnection fail: 712 time fail: 0
test number: 1000 successful tests: 309 reliability: 30.9 %  packages intensity fail: 0 graph inconnection fail: 691 time fail: 0
test number: 1000 successful tests: 297 reliability: 29.7 %  packages intensity fail: 0 graph inconnection fail: 703 time fail: 0
```

Wnioski:

- Nagle zmniejszajac wartosc p tylko o 0.04 notujemy spadek niezawodnosci z ~97% do ~30%
- Z wynikow testu wynika, ze w modelach sieci w ktorych wystepuje spora liczba przesylania pakietow pomiedzy wezlami, parametr p jest bardzo czuly

Test4

Kolejny test bedzie podobny do testu nr 1 z ta roznica, ze dwukrotnie zmniejszymy maksymalne opoznienie pakietu:

```
"T_max": 0.125,
```

Oto przykladowe wywolania:

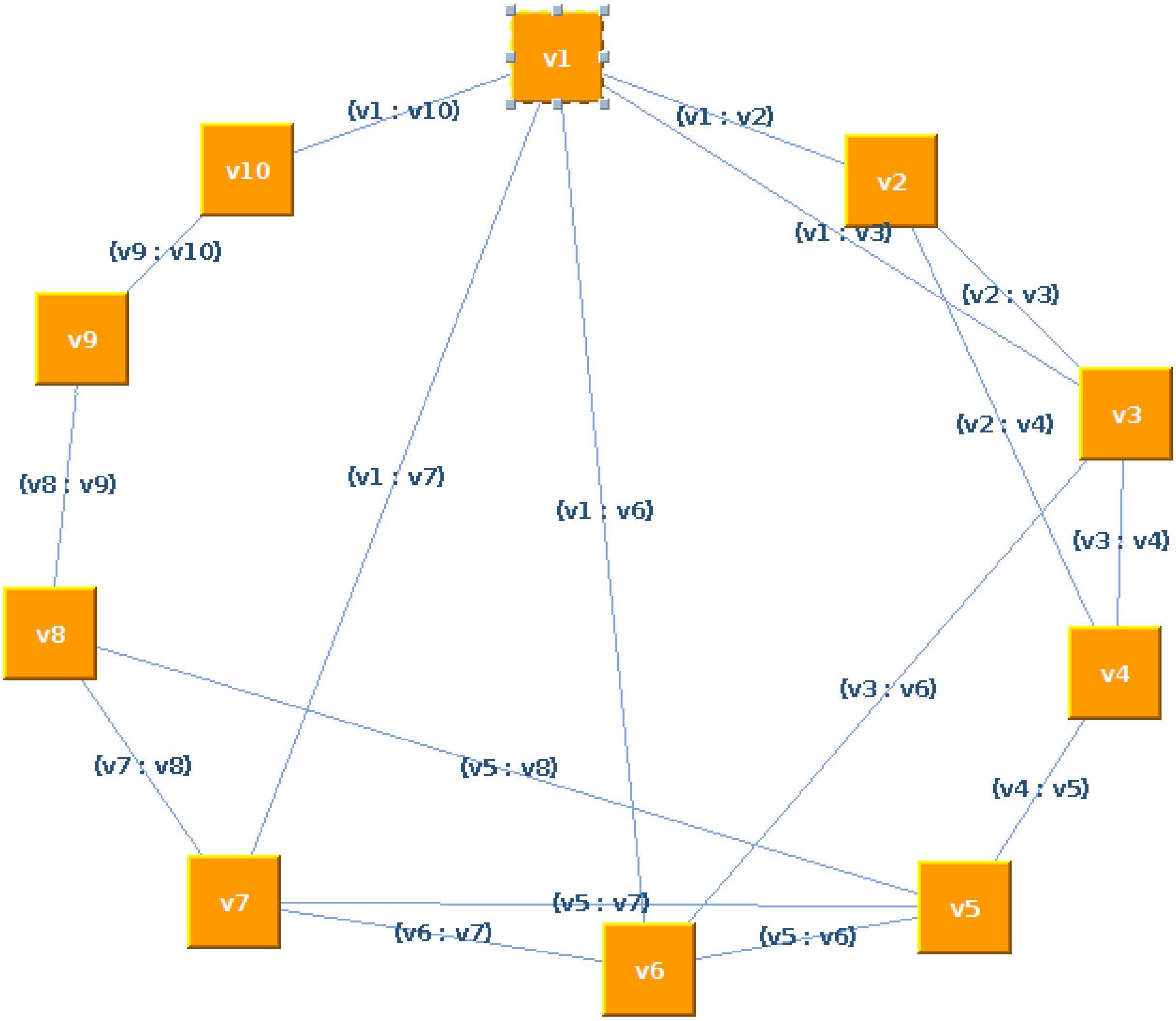
```
test number: 1000 successful tests: 547 reliability: 54.7 %  packages intensity fail: 111 graph inconnection fail: 16 time fail: 326

test number: 1000 successful tests: 529 reliability: 52.900000000000006 %  packages intensity fail: 97 graph inconnection fail: 23 time fail: 351

test number: 1000 successful tests: 553 reliability: 55.300000000000004 %  packages intensity fail: 102 graph inconnection fail: 12 time fail: 333

test number: 1000 successful tests: 525 reliability: 52.5 %  packages intensity fail: 102 graph inconnection fail: 28 time fail: 345

test number: 1000 successful tests: 530 reliability: 53.0 %  packages intensity fail: 99 graph inconnection fail: 21 time fail: 350

test number: 1000 successful tests: 516 reliability: 51.6 %  packages intensity fail: 112 graph inconnection fail: 25 time fail: 347

test number: 1000 successful tests: 546 reliability: 54.6 %  packages intensity fail: 107 graph inconnection fail: 15 time fail: 332

test number: 1000 successful tests: 548 reliability: 54.800000000000004 %  packages intensity fail: 122 graph inconnection fail: 15 time fail: 315
```

Wnioski:

- Bledy powodowane przekroczeniem maksymalnego opoznienia pakietu zwiekszyly sie z wartosci na poziomie ~60 do ~340
- Bledy powodowane przepelnieniem pakietow zmalaly z wartosci ~210 do ~100 co mozna tlumaczyc tym, ze jesli krawedzie byly blisko kranca swojej pojemnosci, to test nie zostawal przerywany ze wzgledu na przepelnienie, tylko ze wzgledu na przekroczenie opoznienia co mozna uargumentowac wzorem $T = 1/G * SUM\_e(a(e)/(c(e)/m - a(e)))$
- Patrzac ma roznice w bledach powodowanych przekroczeniem maksymalnego opoznienia mozna bylo sie spodziewac duzego spadku w niezawodnosci tego modelu, jednak nastapil spadek z ~70% do ~53% co jest spowodowane obnizeniem sie bledow powodowanych przez przekraczanie pojemnosci krawedzi

Test 5

Test bedzie taki sam jak Test1 z ta roznica, ze model sieci bedzie zawieral o jedna krawedz mniej (usuwamy krawedz v9 – v4).



```
{
  "T_max": 0.25,
  "edge-unspoilt-probability": 0.99,
  "edge-capacity": [
    { "vertex1": "v1", "vertex2": "v2", "capacity": 55000 },
    { "vertex1": "v1", "vertex2": "v3", "capacity": 32000 },
    { "vertex1": "v1", "vertex2": "v4", "capacity": 67500 },
    { "vertex1": "v1", "vertex2": "v6", "capacity": 95000 },
    { "vertex1": "v1", "vertex2": "v7", "capacity": 80000 },
    { "vertex1": "v1", "vertex2": "v10", "capacity": 65000 },
    { "vertex1": "v2", "vertex2": "v3", "capacity": 35000 },
    { "vertex1": "v2", "vertex2": "v4", "capacity": 47000 },
    { "vertex1": "v3", "vertex2": "v4", "capacity": 32000 },
    { "vertex1": "v3", "vertex2": "v6", "capacity": 55000 },
    { "vertex1": "v4", "vertex2": "v5", "capacity": 48500 },
    { "vertex1": "v5", "vertex2": "v6", "capacity": 57000 },
    { "vertex1": "v5", "vertex2": "v7", "capacity": 45500 },
    { "vertex1": "v5", "vertex2": "v8", "capacity": 45500 },
    { "vertex1": "v6", "vertex2": "v7", "capacity": 70000 },
    { "vertex1": "v7", "vertex2": "v8", "capacity": 75500 },
    { "vertex1": "v8", "vertex2": "v9", "capacity": 57500 },
    { "vertex1": "v9", "vertex2": "v10","capacity": 50500 }],
  "packages-intensity-matrix":[
    {"v1": {"v2": 2, "v3":2,"v4":8, "v5":1, "v6":9, "v7":5, "v8":3, "v9":1, "v10":3}},
    {"v2": {"v1": 4, "v3":2,"v4":5, "v5":1, "v6":4, "v7":2, "v8":2, "v9":1, "v10":1}},
    {"v3": {"v1": 6, "v2": 3, "v4":3, "v5":2, "v6":7, "v7":1, "v8":2, "v9":2, "v10":1}},
    {"v4": {"v1": 8, "v2": 6, "v3":3, "v5":4, "v6":7, "v7":3, "v8":2, "v9":1, "v10":1}},
    {"v5": {"v1": 6, "v2": 3, "v3":2,"v4":5, "v6":6, "v7":4, "v8":3, "v9":1, "v10":2}},
    {"v6": {"v1": 9, "v2": 2, "v3":6,"v4":3, "v5":5, "v7":4, "v8":3, "v9":1, "v10":2}},
    {"v7": {"v1": 6, "v2": 1, "v3":2,"v4":5, "v5":2, "v6":6, "v8":3, "v9":2, "v10":1}},
    {"v8": {"v1": 5, "v2": 2, "v3":1,"v4":4, "v5":4, "v6":5, "v7":2, "v9":1, "v10":1}},
    {"v9": {"v1": 4, "v2": 1, "v3":1,"v4":3, "v5":1, "v6":4, "v7":1, "v8":4, "v10":3}},
    {"v10": {"v1": 5, "v2": 2, "v3":1,"v4":4, "v5":2, "v6":4, "v7":2, "v8":1, "v9":2}}

  ]
}
```


Oto przykladowe wyniki:

```
test number: 1000 successful tests: 63 reliability: 6.3 %  packages intensity fail: 391 graph inconnection fail: 54 time fail: 492

test number: 1000 successful tests: 78 reliability: 7.8 %  packages intensity fail: 367 graph inconnection fail: 62 time fail: 493

test number: 1000 successful tests: 72 reliability: 7.199999999999999 %  packages intensity fail: 387 graph inconnection fail: 51 time fail: 490

test number: 1000 successful tests: 81 reliability: 8.1 %  packages intensity fail: 381 graph inconnection fail: 52 time fail: 486

test number: 1000 successful tests: 88 reliability: 8.799999999999999 %  packages intensity fail: 377 graph inconnection fail: 58 time fail: 477

test number: 1000 successful tests: 69 reliability: 6.9 %  packages intensity fail: 388 graph inconnection fail: 56 time fail: 487

test number: 1000 successful tests: 80 reliability: 8.0 %  packages intensity fail: 402 graph inconnection fail: 53 time fail: 465

test number: 1000 successful tests: 85 reliability: 8.5 %  packages intensity fail: 390 graph inconnection fail: 41 time fail: 484
```

Wnioski:

- Roznica w niezawodnosci modelu jest ogoromna (z ~70 % do ~7 %)
- Zdecydowana wiekszosc bledow jest spowodowana przekroczeniem maksymalego opoznnienia pakietow (wzrost z ~60 do ~480 bledow). Mozna to tlumaczyc wzrostem

przepelnienia na innych krawedziach oraz wydluzenia najkrotrzej sciezki w niektorych przypadkach

- Mozna zanotowac rozniez wzrost bledow spowodowanych przepelnieniem pakietow na krawedziach