ZADANIE 1

1.1

W pierwszym podpunkcie graf reprezentujący badaną sieć zawiera 20 wierzchołków i 19 krawędzi, jest to graf spójny w kształcie przypominający nitkę. Warto zauważyć, że po zerwaniu dowolnej krawędzi grafu ulega on rozspójnieniu.

Aby doświadczalnie wyznaczyć prawdopodobieństwo rozspójnienia tego grafu, napisałem program w Javie. Funkcja niezawodności przyporządkuje każdej krawędzi tą samą wartość wynoszącą 0.95, a test został wykonany dla miliona prób.

```java
static void zad1() {
    Random gen = new Random(System.currentTimeMillis());

    Graph<Integer, DefaultEdge> graph = new DefaultUndirectedWeightedGraph<>(DefaultEdge.class);
    int vertices = 20;
    int edges = 19;

    //wierzcholki
    for(int i = 1 ; i <= vertices ; i++) {
        graph.addVertex(i);
    }

    //krawedzie
    for(int i = 1 ; i <= edges ; i++) {
        graph.addEdge(i,  targetVertex: i+1);
        graph.setEdgeWeight(i,  targetVertex: i+1,  weight: 0.95);
    }

    //test, ilosc prob okreslona w zmiennej TEST_AMOUNT
    int TEST_AMOUNT = 1000000;
    int fail = 0;

    for(int i = 0 ; i < TEST_AMOUNT ; i++) {
        for(DefaultEdge edge : graph.edgeSet()) {
            double rand = (double)gen.nextInt( bound: 100)*0.01;

            if((gen.nextDouble() > graph.getEdgeWeight(edge))) {
                fail++;
                break;
            }
        }
    }

    System.out.println("Ilość prób: "+TEST_AMOUNT+", ilość rozerwań: "+fail+", niezawodność: "+(double)(TEST_AMOUNT-fail)/TEST_AMOUNT);
}
```

A oto wynik doświadczenia :

```
Ilość prób: 1000000, ilość rozerwań: 622624, niezawodność: 0.377376
```

Prawdopodobieństwo rozspójnienia tego grafu możemy obliczyć również prostym wzorem, działa on tylko wówczas gdy krawędzie mają jednakową szanse zerwania.

Oznaczmy :

n – ilość krawędzi

h – prawdopodobieństwo nieuszkodzenia krawędzi

P – prawdopodobieństwo rozspójnienia grafu

$P = h^n = 0.95^{19} \approx 0.37735$

Widzimy, że podczas symulacji komputerowej otrzymaliśmy wynik bardzo zbliżony do wzoru.

1.2

W drugim podpunkcie dodajemy krawędź e(20,1) i otrzymujemy graf spójny 2 stopnia, to znaczy, że z każdego wierzchołka wychodzą 2 krawędzie, wygląda to jakbyśmy złączyli końce naszej nitki. Tym razem aby rozspójnić graf trzeba rozerwać 2 krawędzie.

```java
static void zad1() {
    Random gen = new Random(System.currentTimeMillis());

    Graph<Integer, DefaultEdge> graph = new DefaultUndirectedWeightedGraph<>(DefaultEdge.class);
    int vertices = 20;
    int edges = 19;

    //wierzcholki
    for(int i = 1 ; i <= vertices ; i++) {
        graph.addVertex(i);
    }

    //krawedzie
    for(int i = 1 ; i <= edges ; i++) {
        graph.addEdge(i, targetVertex: i+1);
        graph.setEdgeWeight(i, targetVertex: i+1, weight: 0.95);
    }

    graph.addEdge( sourceVertex: 20, targetVertex: 1);
    graph.setEdgeWeight( sourceVertex: 20, targetVertex: 1, weight: 0.95);

    //test, ilosc prob okreslona w zmiennej TEST_AMOUNT
    int TEST_AMOUNT = 1000000;
    int fail = 0;
    int edgeRupture;

    for(int i = 0 ; i < TEST_AMOUNT ; i++) {
        edgeRupture = 0;
        for(DefaultEdge edge : graph.edgeSet()) {
            double rand = (double)gen.nextInt( bound: 100)*0.01;

            if((gen.nextDouble() > graph.getEdgeWeight(edge))) {
                edgeRupture++;
                if(edgeRupture == 2) {
                    fail++;
                    break;
                }
            }
        }
    }

    System.out.println("Ilość prób: "+TEST_AMOUNT+", ilość rozerwań: "+fail+", niezawodność: "+(double)(TEST_AMOUNT-fail)/TEST_AMOUNT);
}
```

A oto wynik doświadczenia:

```
Ilość prób: 1000000, ilość rozerwań: 264673, niezawodność: 0.735327
```

Tym razem widzimy znaczny wzrost niezawodności, około dwukrotny.

1.3

W tym kroku dodajemy kolejne dwie krawędzie e(1,10) i e(5,15) tym razem nasz algorytm potrzebuje nieco większych zmian, idea będzie taka, że będziemy dodawać krawędzie z prawdopodobieństwem o wartości równej prawdopodobieństwu nierozerwania krawędzi, po czym będziemy sprawdzać czy graf jest spójny.

```java
static void zad1() {
    Random gen = new Random(System.currentTimeMillis());
    //test, ilosc prob okreslona w zmiennej TEST_AMOUNT
    int TEST_AMOUNT = 1000000;
    int fail = 0;
    int edgeRupture;

    for(int i = 0 ; i < TEST_AMOUNT ; i++) {

        Graph<Integer, DefaultEdge> graph = new DefaultUndirectedGraph<>(DefaultEdge.class);
        int vertices = 20;
        int edges = 19;

        //wierzcholki
        for(int k = 1 ; k <= vertices ; k++) {
            graph.addVertex(k);
        }

        //dodajemy krawedzie z pewnym prawdopodobienstwem
        for(int j = 1 ; j <= edges ; j++) {
            if(gen.nextDouble() < 0.95) {
                graph.addEdge(j, targetVertex: j+1);
            }
        }

        if((double)gen.nextInt( bound: 100)*0.01 < 0.95) {
            graph.addEdge( sourceVertex: 20, targetVertex: 1);
        }

        if((double)gen.nextInt( bound: 100)*0.01 < 0.8) {
            graph.addEdge( sourceVertex: 1, targetVertex: 10);
        }

        if((double)gen.nextInt( bound: 100)*0.01 < 0.7) {
            graph.addEdge( sourceVertex: 5, targetVertex: 15);
        }

        //sprawdzamy czy graf jest spójny
        if(!GraphTests.isConnected(graph))
            fail++;
    }

    System.out.println("Ilość prób: "+TEST_AMOUNT+", ilość rozerwań: "+fail+", niezawodność: "+(double)(TEST_AMOUNT-fail)/TEST_AMOUNT);
}
```

A oto wynik doświadczenia:

```
Ilość prób: 1000000, ilość rozerwań: 129312, niezawodność: 0.870688
```

Widzimy, że prawdopodobieństwo nierozerwania sieci również się zwiększa, choć nie tak znacząco jak w poprzednim podpunkcie.

1.4

W tym podpunkcie dodajemy 4 krawędzie pomiędzy losowo wybranymi wierzchołkami. Idea działania algorytmu będzie taka sama jak w podpunkcie poprzednim.

```java
    static void zad1() {
        Random gen = new Random(System.currentTimeMillis());
        //test, ilosc prob okreslona w zmiennej TEST_AMOUNT
        int TEST_AMOUNT = 1000000;
        int fail = 0;
        int edgeRupture;

        for(int i = 0 ; i < TEST_AMOUNT ; i++) {

            Graph<Integer, DefaultEdge> graph = new DefaultUndirectedGraph<>(DefaultEdge.class);
            int vertices = 20;
            int edges = 19;

            //wierzcholki
            for(int k = 1 ; k <= vertices ; k++) {
                graph.addVertex(k);
            }

            //dodajemy krawedzie z pewnym prawdopodobienstwem
            for(int j = 1 ; j <= edges ; j++) {
                if(gen.nextDouble() < 0.95) {
                    graph.addEdge(j, targetVertex: j+1);
                }
            }

            if((double)gen.nextInt( bound: 100)*0.01 < 0.95) {
                graph.addEdge( sourceVertex: 20, targetVertex: 1);
            }

            if((double)gen.nextInt( bound: 100)*0.01 < 0.8) {
                graph.addEdge( sourceVertex: 1, targetVertex: 10);
            }

            if((double)gen.nextInt( bound: 100)*0.01 < 0.7) {
                graph.addEdge( sourceVertex: 5, targetVertex: 15);
            }

            //4 losowo wybrane krawedzie
            for(int l = 0 ; l < 4 ; l++) {
                if((double)gen.nextInt( bound: 100)*0.01 < 0.4)
                    if(graph.addEdge( sourceVertex: gen.nextInt( bound: 20)+1, targetVertex: gen.nextInt( bound: 20)+1) == null) {
                        l--;
                    }
            }

            //sprawdzamy czy graf jest spójny
            if(!GraphTests.isConnected(graph))
                fail++;
        }

        System.out.println("Ilość prób: "+TEST_AMOUNT+", ilość rozerwań: "+fail+", niezawodność: "+(double)(TEST_AMOUNT-fail)/TEST_AMOUNT);
    }
```

A oto wynik doświadczenia:

```
Ilość prób: 1000000, ilość rozerwań: 91584, niezawodność: 0.908416
```

WNIOSKI

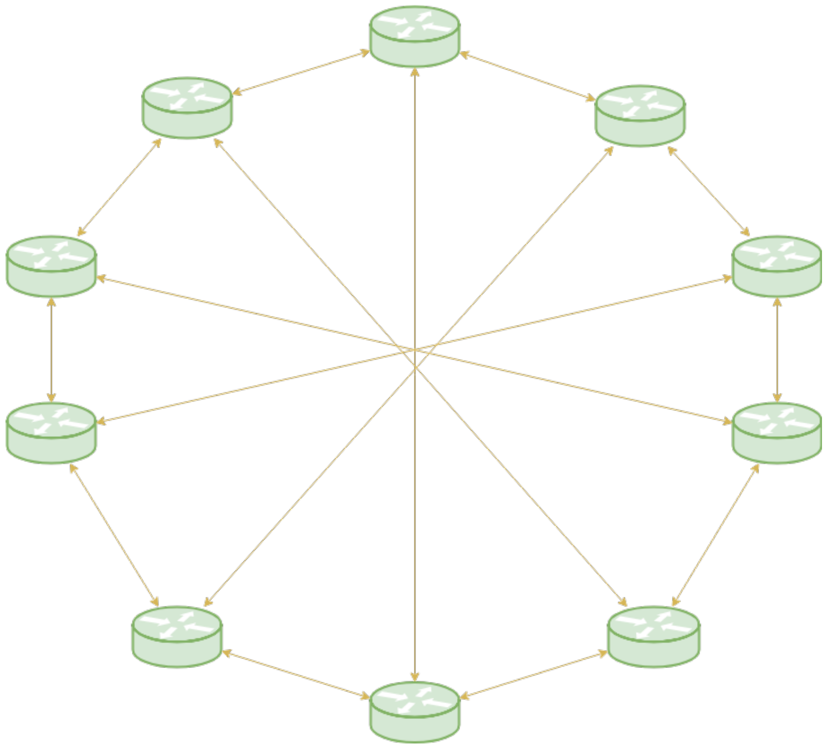
Model sieci z punkcie pierwszym jest bardzo zawodny, w punkcie drugim po dodaniu jedne krawędzi uzyskujemy poprawę o połowę, w pozostałych podpunktach również notujemy poprawę, widzimy, że im więcej krawędzi tym bardziej nasza siec jest niezawodna. Oczywiście nie chodzi tylko o ilość krawędzi ale również o odpowiednie ich rozmieszczenie.

ZADANIE 2

1.1

Moja propozycja topologi grafu, do każdego wierzchołka dochodzą trzy krawędzie więc powinniśmy osiągnąć wysoką spojność.

|V| = 10
|E| = 15



Niech N będzie macierzą natężeń strumienia pakietów, gdzie element n(i,j) to liczba przesyłanych pakietów w ciągu sekundy od v(i) do v(j). Zdefiniujmy m jako średnia ilość bitów przypadająca na jeden pakiet oraz funkcję przepustowości c (maksymalna ilość bitów przechodzących przez krawędź) jako stałą równą $m * \sum_{\forall n \in N} n$ czyli sumę bitów przesyłanych we wszystkich wysłanych pakietach w sieci w ciągu sekundy, tak dobrane c zawsze spełni nierówność c(v) > a(v). Funkcja a – funkcja przepływu, czyli faktyczna liczba pakietów jaką wprowadza się do kanału w ciągu sekundy.

Tworzenie grafu

```java
//tworzenie grafu
Graph<Integer, DefaultEdge> graph = new DefaultUndirectedGraph<>(DefaultEdge.class);

//wierzcholki
for(int i = 1 ; i <= 10 ; i++) {
    graph.addVertex(k);
}

//krawedzie
for(int i = 1 ; i <= 9 ; i++) {
    graph.addEdge(j, targetVertex: j+1);
}
graph.addEdge( sourceVertex: 10, targetVertex: 1);

for(int i = 1 ; i <= 5 ; i++) {
    graph.addEdge(i, targetVertex: i+5);
}
```

Generowanie macierzy N, załóżmy że maksymalna liczba pakietów to 4

```java
//generowanie macierzy N
int [][]N = new int[10][10];

for(int i = 0 ; i < 10 ; i++) {
    for(int j = 0 ; j < 10 ; j++) {
        if(i == j)
            N[i][j] = 0;
        else
            N[i][j] = gen.nextInt( bound: 5);
    }
}
```

2.2

Teraz obliczymy średnie opóźnienie pakietu zadane wzorem:

$$T=\frac{1}{G}*\sum_{e\in E}\frac{a(e)}{\frac{c(e)}{m}-a(e)}$$

G – suma elementów macierzy natężeń

Weźmy m = 1000 bitów

Teraz obliczymy potrzebne nam dane G, c i A – macierz funkcji przepływu

```
        //suma elementów macierzy G
        double G = 0;
        for(int i = 0 ; i < 10 ; i++) {
            for (int j = 0; j < 10; j++) {
                G += N[i][j];
            }
        }


        //stala c jako funkcja przepustowości
        double c = m * G;


        //generowanie macierzy funkcji przeplywu a
        int [][]A = new int[10][10];
        for(int i = 0 ; i < 10 ; i++) {
            for(int j = 0 ; j < 10 ; j++) {
                if(graph.containsEdge( sourceVertex: i+1,  targetVertex: j+1))
                    A[i][j] = N[i][j];
                else
                    N[i][j] = 0;
            }
        }

```

Czas na obliczenie wartości opóźnienia

```

        //obliczanie opoznienia T
        double sum = 0;
        for(int i = 0 ; i < 10 ; i++) {
            for(int j = 0 ; j < 10 ; j++) {
                sum += (double) A[i][j]/((c/m) - A[i][j]);
            }
        }

        double T = sum/G;

```

A oto wynik doświadczenia

```
Srednie opoznienie pakietu: 0.0020872320753910897
```

2.3

Test niezawodności ze względu na ograniczenie T_max i spójność wykonałem dla miliona prób. Wartości które ustawiłem:

T_max = 0.002

p = 0.9

m = 1000

N = generowana losowo, watrości modulo 5

c = m*G

```java
static void zad2_3() {

    //parametry
    double p = 0.9;
    double T_max = 0.002;
    double m = 1000 ;

    int TEST_AMOUNT = 1000000;
    int fail = 0;
    double reliability = 0;

    for(int i = 0 ; i < TEST_AMOUNT ; i++) {

        //tworzymy graf
        Graph graph = zad2graph();

        //sprawdzamy spojność
        for(int j = 1 ; j <= 10 ; j++ ) {
            for(int l = 1 ; l <= 10 ; l++) {
                if(graph.containsEdge(j, l) && (double)gen.nextInt( bound: 100)/100 > p) {
                    graph.removeEdge(j, l);
                }
            }
        }
        if(!GraphTests.isConnected(graph)) {
            fail++;
            continue;
        }
        else {
            //liczymy T
            int N[][] = zad2N();

            double G = zad2G(N);

            double c = m * G;

            int [][]A = zad2A(graph, N);

            if(zad2T(G, m, c, A) < T_max) {

            }
            else {
                fail++;
                continue;
            }
        }
    }

    reliability = (double)(TEST_AMOUNT - fail)/TEST_AMOUNT;

    System.out.println("Ilosc prob: " + TEST_AMOUNT + " Ilosc niepowodzen: " + fail + " Niezawodnosc: " + reliability);
}
```

A oto wynik doświadczenia

```
Ilosc prob: 1000000 Ilosc niepowodzen: 138734 Niezawodnosc: 0.861266
```

WNIOSKI

Nauczyłem się, jak duży wpływ na niezawodność sieci ma topologia grafu sieci oraz ilość połączeń między wierzchołkami tej siec. Ważnym składnikiem charakteryzującym sieć jest również średnie opóźnienie przesyłania pakietu, nauczyłem się je obliczać i wiem co ma wpływ na jego wartość.