Sprawozdanie – TS. Lista 2

1. Środowisko symulacyjne:

Java, bibliotek JGraphT

Dane do testów są zawarte w klasie „DataForTests”

`intensityMatrix` – macierz(20x20) natężeń – liczba pakietów które chcemy przesłać z i do j wierzchołka

metoda **`calculateAForEdge`**

```
private int calculateAForEdge(int i, int j) {

    // Get all packets that will go through that edge in this
graph
    int packetsInEdge = 0;

    for (int a = 0; a < n; a++) {
        for (int b = 0; b < n; b++) {
            GraphPath<Integer, DefaultEdge> shortest_path =
DijkstraShortestPath.findPathBetween(g, a, b);
           try {
                if
(shortest_path.getEdgeList().contains(g.getEdge(i, j))) {
                    packetsInEdge = packetsInEdge +
intensityMatrix[a][b];
                }
            } catch (NullPointerException e) {
                System.out.println("NO PATH");
            }

        }
    }
    return packetsInEdge;
}
```

Oblicza ile pakietów faktycznie może przepłynąć w krawędzi (i,j). Działa na zasadzie „przechodzenia” każdej możliwej najkrótszej ścieżki w grafie (reprezentowaną jako shortest_path od a do b), jeśli nasza krawędź (i,j) zawiera się w tej najkrótszej ścieżce, to dodajemy liczbę pakietów która płynie z a do b do faktycznej liczby pakietów mogących przepłynąć przez (i,j) (packetsInEdge).

Metoda **initFunctionCMatrix**

```
private void initFunctionCMatrix(){
    functionCMatrix = new int[n][n];
    generator = new Random();
    for(int i=0; i<n; i++){
        for(int j=0; j<n;j++){
            if(g.containsEdge(i,j))

functionCMatrix[i][j]=m*(calculateAForEdge(i,j)+3000);
            else
              functionCMatrix[i][j]=0;
        }
    }
}
```

Uzupełnia macierz przepustowości, wykorzystuje do tego funkcjęA, mnoży podana wartość razy średnia wielkość pakietu w bitach (m=8).

Metoda **incrementIntensityMatrix –** zwiększa wartości w macierzy przepustowości o 4 pakiety

Metoda incrementCMatrix – zwiększa przepustowości o 40 pakietów (*8 bitów czyli 320bitów)

Metoda **averageTime**

```
public boolean averageTime(){

    double matrixSum = 0;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            matrixSum += intensityMatrix[i][j];
        }
    }

    double sumEdgesEquation = 0;
    for (DefaultEdge e : g.edgeSet()) {

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

        double c = functionCMatrix[i][j];
        double a = calculateAForEdge(i, j);
```

```
            if(c<=a*m){
                return false;
            }
            sumEdgesEquation += (a/(c/m-a));
        }

        this.averageTime = (1/matrixSum) * sumEdgesEquation;
        return true;
    }
```

Oblicza średni czas opóźnienia według podanego wzoru w poleceniu do zadania**. T = 1/G * SUM_e(a(e)/(c(e)/m - a(e)))**
Z tym, gdy nasza przepustowość krawędzi podzielona przez m będzie mniejsza od faktycznej liczby pakietów przepływających przez krawędź zwraca false**.**

2. Zestawy testowe:

```
double T_Max = 0.03

int numbOfTests = 5000;
int numbOfTestTypes = 7;
boolean connectedGraph = true;
int p = 0.95; //niezawodność łącza
```

indeksy w tablicy data to numer typu testu

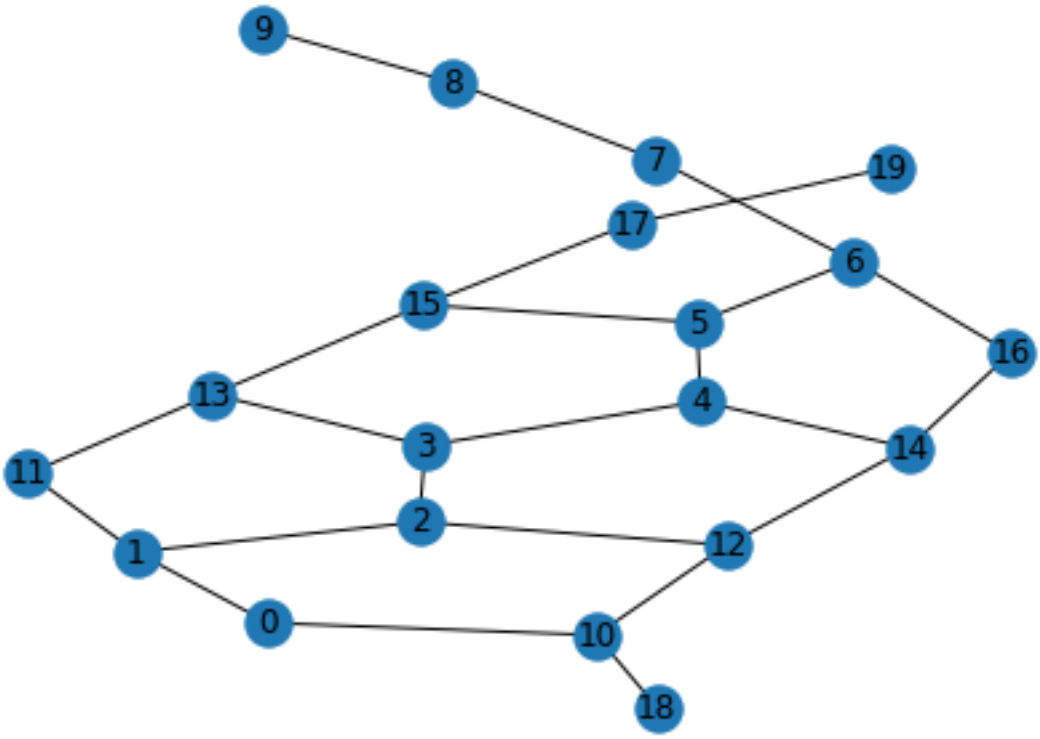
```
DataForTests[] data = new
DataForTests[numbOfTestTypes];

int m = 8;
functionCMatrix
```

= (wyliczane na podstawie funkcjiA + 3000)*m

Test type 0:

Topologia nr 1



failures: 2423, averageTime: 0.0013187009400390025

Prawdopodobieństwo, że

1. Graf pozostanie spójny
2. Przepustowość będzie wystarczająca żeby przesłać faktyczną liczbę bitów na krawędzi
3. Czas maksymalny nie zostanie przekroczony

Wyniosło: **2577/5000 = 0.5154**

Test type 1:

Jak poprzednio, ale zwiększamy wartości w macierzy natężeń o 4.

failures: 2539, averageTime: 0.0014477640557951223

Prawdopodobieństwo (Pr) określone jak wyżej wyniosło:

2461/5000= 0.4922

Test type 2:

Jak w pkt. 1, ale ponownie zwiększamy wartości w macierzy natężeń (w sumie o 8 pakietów).

failures: 2723, averageTime: 0.0015757225518903044

Pr **= 0.4554**

Test type 3:

jak w pkt 0, ale zwiększamy wartości w macierzy przepustowości

failures: 2428, averageTime: 0.001297802173685219

Pr = **0.5144**

Test type 4:

Jak w pkt 3, ale znowu zwiększamy wartości w macierzy przepustowości.

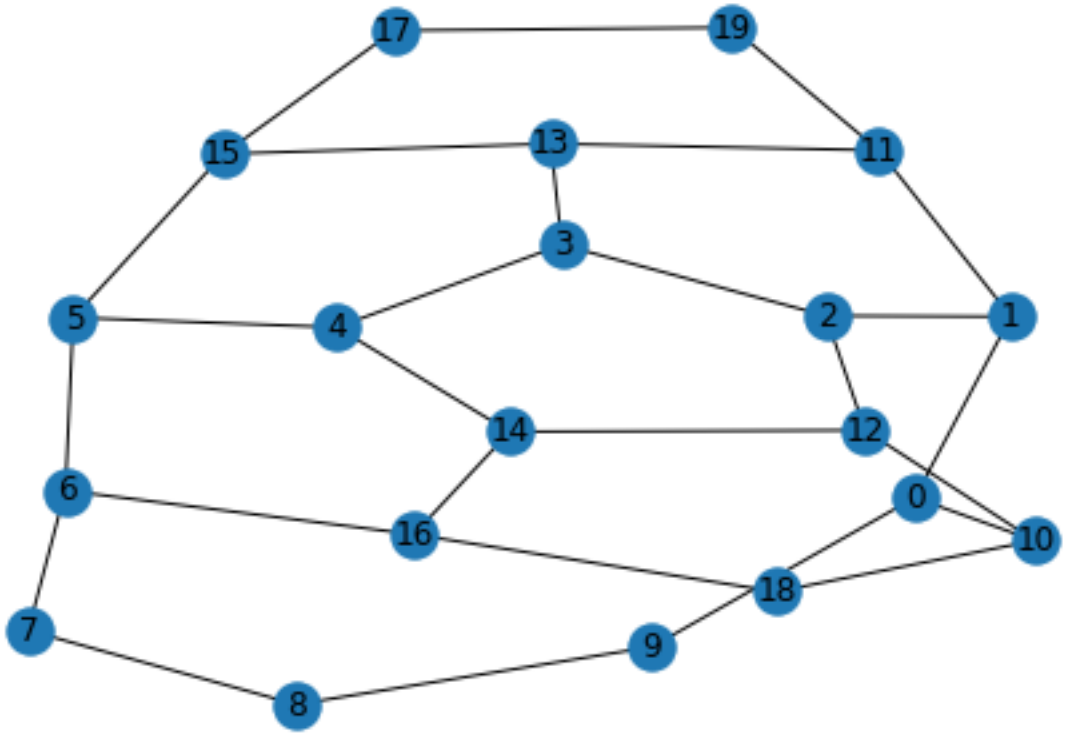
failures: 2350, averageTime: 0.0012860703968252485

Pr = **0.53**

Test type 5

jak w pkt 1, ale dodajemy 3 krawędzie do grafu

Topologia nr 2:



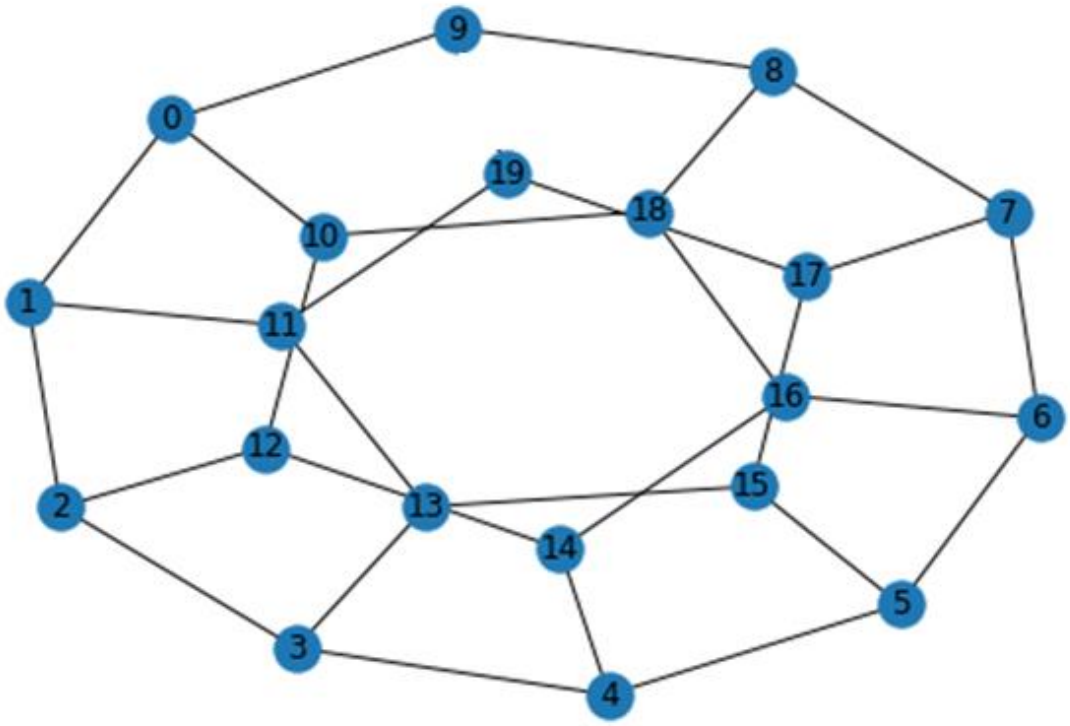
failures: 682, averageTime: 0.0011272898005776763

Pr = **4318/5000 = 0.8636**

Test type 6

Dodane dwie krawędzie pomiędzy cyklem wewnętrznym
a zewnętrznym.

Topologia nr 3:



failures: 130, averageTime: 9.012866245727595E-4

Pr = **0.974**

3. Wnioski:

Pakiety: gdy zwiększymy liczbę przesyłanych pakietów, dostajemy więcej niepowodzeń, wyższe czasy. Jest to spowodowane tym, że zadana przepustowość jest za mała, żeby przesłać tyle pakietów na raz, czy to w normalnym wypadku, albo gdy węzły aktywne muszą „wytrzymać” przesyłanie pakietów, które normalnie zostałyby przesłane przez usuniętą w symulacji krawędź.

Przepustowość: gdy zwiększamy przepustowość dostajemy mniej niepowodzeń, niższe czasy. W tym wypadku możemy przesłać pakiety w tych próbach, w których w poprzednim teście (nr 0) dochodziło do zapchania kanałów komunikacyjnych (krawędź nie zdała testu c(e)/m<a(e)) albo po prostu potrafimy obsłużyć więcej pakietów na raz.

Zmiana topologii: Po dodaniu krawędzi niezawodność sieci wzrasta. Stało się tak, ponieważ, mając więcej krawędzi, jest mniejsza szansa na to, że graf się rozspójni po usunięciu krawędzi z prawdopodobieństwem p. Ścieżki mogą być krótsze i nie tak przeciążone, stąd też występują niższe opóźnienia.

Przemyślenia: najlepiej, żeby projektowana sieć miała dużą przepustowość każdego kanału komunikacyjnego, nie posiadała wąskich gardeł w postaci węzłów stopnia 1 (najlepiej, żeby posiadała min. Stopnia 3).