Sprawozdanie do listy 2.

1. Testowanie sieci metodą Monte Carlo

Celem pierwszego zadania było szacowanie niezawodności sieci, która jest przedstawiona jako graf. Pierwszym modelem do zbadania była sieć w której każdy z dwudziestu wierzchołków V_1, V_2, …, V_{20} był połączony krawędziami $E_{1\text{-}2}$, $E_{2\text{-}3}$, …, $E_{19\text{-}20}$, gdzie każda krawędź posiadała parametr niezawodności równy 0.95. Następnie dodaliśmy połączenie pomiędzy wierzchołkami V_1, V_{20} o takim samym parametrze. Na koniec połączyliśmy wierzchołki V_1 i V_{10} oraz V_5 i V_{15} krawędziami z parametrami odpowiednio 0.8 i 0.7 oraz wygenerowaliśmy 4 losowe krawędzie o parametrze 0.4. Graf w poszczególnych krokach wyglądał następująco:

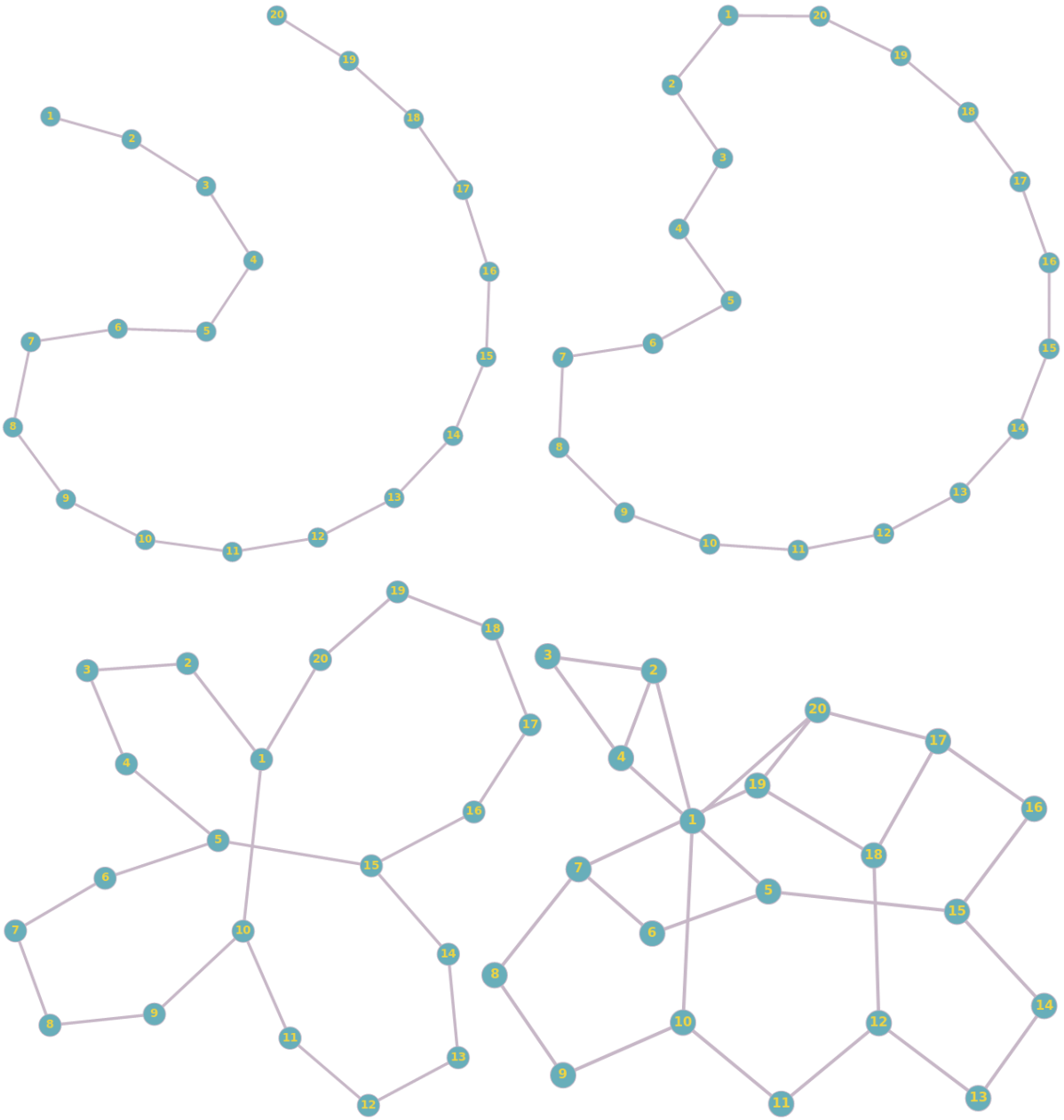
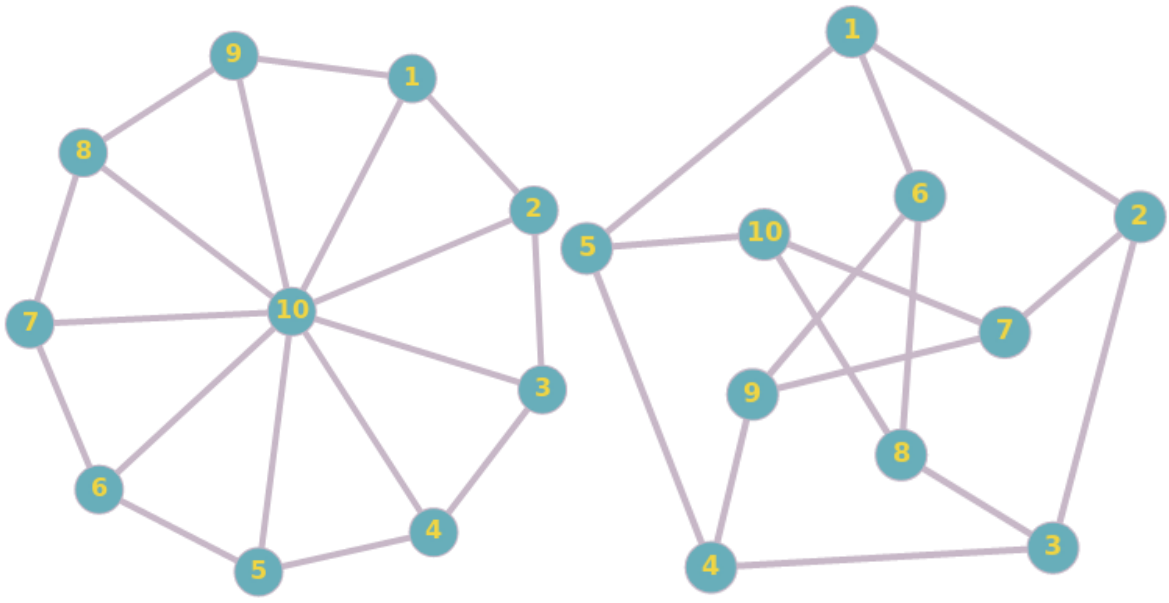


Tabela niezawodności grafu				
Typ grafu	Ilość prób	Niezawodność	Wierzchołki	Krawędzie
Bez Cyklu	1 000 000	37,66%	20	19
Z Cyklem	1 000 000	73,67%	20	20
Dwa Łączenia	1 000 000	86,06%	20	22
Cztery Losowe	3 000 000	91,14%	20	26

Tabela powstała w oparciu o wyniki algorytmów zapisanych w języku programowania java. Kod źródłowy będzie pokazany w drugim zadaniu. Można zauważyć, że stopniowe dodawanie wierzchołków wpłynęło na zwiększenie niezawodności sieci, co więcej dodanie dwudziestej krawędzi, aby otrzymać cykl praktycznie podwoiło jej niezawodność, dlatego, że aby sieć się „rozerwała" musiały zawieść dwa połączenia, a nie jedno. Na koniec pozwoliłem sobie przetestować daną metodą grafy wykorzystane w zadaniu drugim, co pomogło mi od razu wykluczyć jeden z nich. Testowałem 3 możliwości grafu: cykliczny, kołowy oraz Petersena. Wyniki pozwoliły mi od razu wykluczyć graf cykliczny, jako, że wynik ok. 90% niezawodności nie był zadowalającym.

Typ grafu	Ilość prób	Niezawodność	Wierzchołki	Krawędzie
Cykliczny	1 000 000	91,39%	10	10
Kołowy	1 000 000	99,88%	10	18
Petersen	1 000 000	99,87%	10	15

Otrzymane wyniki dla parametru 0.95 dla każdej krawędzi skłoniły mnie do testowania w następnym zadaniu grafy z następującymi topologiami:



2. Badanie natężenia i opóźnienia sieci

W poprzednim zadaniu doszliśmy do wniosku, że najsensowniejszymi grafami będą kolisty i Petersena. Do testowania stabilności sieci wykorzystałem zmodyfikowany kod z zadania pierwszego.

```
private static void calculate(int minPackage, int maxPackage, int c_e, double T_Max, int m){
        int failures1 = 0;
        int overload1 = 0;
        int connection1 = 0;
        int timeout1 = 0;
        int failures2 = 0;
        int overload2 = 0;
        int connection2 = 0;
        int timeout2 = 0;
        int g = 0;
        double T;
        double T2;
        double max1=0;
        double max2=0;
```

Na początku stworzyłem funkcję „calculate” przyjmującą za parametry minimalną oraz maksymalną liczbę pakietów wysłanych od jednego wierzchołka do drugiego, maksymalną przepustowość pojedynczego połączenia, maksymalne opóźnienie wysłanych pakietów oraz ilość bitów zawierającą się w pojedynczym pakiecie.

```
for(int k=0; k<V; k++){
    for (int j=0 ; j<V; j++){
        N[k][j] = r.nextInt(maxPackage - minPackage + 1) + minPackage;
        g += N[k][j];
    }
}
```

Kiedy test się rozpoczyna zostaje wylosowana macierz natężeń z wartościami zawartych w parametrach funkcji oraz każda wartość jest dodawana do zmiennej „g”, która trzyma informację o sumie natężeń w macierzy.

```
Graph G = new Graph(V);
            G.addEdge(1,2,c_e/m,0.95);
            G.addEdge(2,3,c_e/m,0.95);
            G.addEdge(3,4,c_e/m,0.95);
            G.addEdge(4,5,c_e/m,0.95);
            G.addEdge(5,1,c_e/m,0.95);
            G.addEdge(1,6,c_e/m,0.95);
            G.addEdge(2,7,c_e/m,0.95);
            G.addEdge(3,8,c_e/m,0.95);
            G.addEdge(4,9,c_e/m,0.95);
            G.addEdge(5,10,c_e/m,0.95);
            G.addEdge(6,8,c_e/m,0.95);
            G.addEdge(8,10,c_e/m,0.95);
            G.addEdge(10,7,c_e/m,0.95);
            G.addEdge(7,9,c_e/m,0.95);
            G.addEdge(9,6,c_e/m,0.95);
            G.testReliability();
            G.createPaths();
            G.setIntensity(N);
```

```
T = 1.0/(double) g * G.calculateSum(m);
    if(G.getIntensity() == -1){
        failures1++;
        overload1++;
    } else if (!G.checkConnection()){
        failures1++;
        connection1++;
    } else if (T > T_Max){
        failures1++;
        timeout1++;
    }
```

Następnie tworzymy graf o zadanej (V) liczbie wierzchołków i dodajemy interesujące nas krawędzie. Funkcja testReliability dla każdego wierzchołka losuje liczbę z zakresu (0;1) i gdy zostanie wylosowana liczba większa od 0.95 krawędź w danej iteracji pętli zostaje usunięta, z powodu „awarii". Następnie za pomocą createPaths każda krawędź dostaje informacje dla jakich połączeń wierzchołków zawiera się ona w najkrótszej dla nich drodze a setIntensity wykorzystuje informacje przekazaną przez createPaths do obliczenia liczby bitów przekazywanych do krawędzi w ciągu sekundy. Podobnie dzieje się z drugim grafem. Na koniec obliczamy z podanego wzoru średni czas dostarczenia pakietu i sprawdzamy czy połączenie jest stabilne. Jeżeli na którymś z kanału wartość a(e) jest większa niż wcześniej określona przepustowość c(e) wtedy inkrementujemy zmienne failures<nr grafu> i overload<nr grafu> i ogłaszamy niepowodzenie w działaniu sieci. Kiedy żadna z nich nie jest przeciążona wtedy sprawdzamy czy test niezawodności wywołany funkcją testReliability nie spowodował rozerwania naszego połączenia, jeżeli tak się stało to również dodajemy 1 do wartości niepowodzeń. Na samym końcu sprawdzamy czy czas przesyłu pakietu nie był większy od wcześniej zadeklarowanej wartości maksymalnej, tutaj też stwierdzamy niepowodzenie. Funkcję calculate następnie wywołujemy w funkcji main.

```
public static void main(String[] args) {

System.out.format("%3s%4s%5s%7s%3s%7s%7s%7s%7s%7s%7s
        calculate(10,20,3200, 0.02, 32);
        calculate(10,15,3200,0.02,32);
        calculate(15,20,3200,0.02,32);
        calculate(10,20,4800, 0.02, 32);
        calculate(10,15,4800,0.02,32);
        calculate(15,20,4800,0.02,32);
        calculate(5,10,1600,0.02,32);
        calculate(5,7,1600,0.02,32);
        calculate(8,10,1600,0.02,32);
        calculate(10,20,3200, 0.03, 32);
        calculate(10,15,3200,0.03,32);
        calculate(15,20,3200,0.03,32);
        calculate(10,20,4800, 0.03, 32);
        calculate(10,15,4800,0.03,32);
        calculate(15,20,4800,0.03,32);
        calculate(5,10,1600,0.03,32);
        calculate(5,7,1600,0.03,32);
        calculate(8,10,1600,0.03,32);;
        calculate(10,20,3200, 0.01, 32);
        calculate(10,15,3200,0.01,32);
        calculate(15,20,3200,0.01,32);
        calculate(10,20,4800, 0.01, 32);
        calculate(10,15,4800,0.01,32);
        calculate(15,20,4800,0.01,32);
        calculate(5,10,1600,0.01,32);
        calculate(5,7,1600,0.01,32);
        calculate(8,10,1600,0.01,32);
        calculate(40,40,8960,0.02,32);
        calculate(20,20,4480,0.02,32);
        calculate(10,10,2240,0.02,32);
        calculate(5,5,1120,0.02,32);
        //calculate();
    }
```

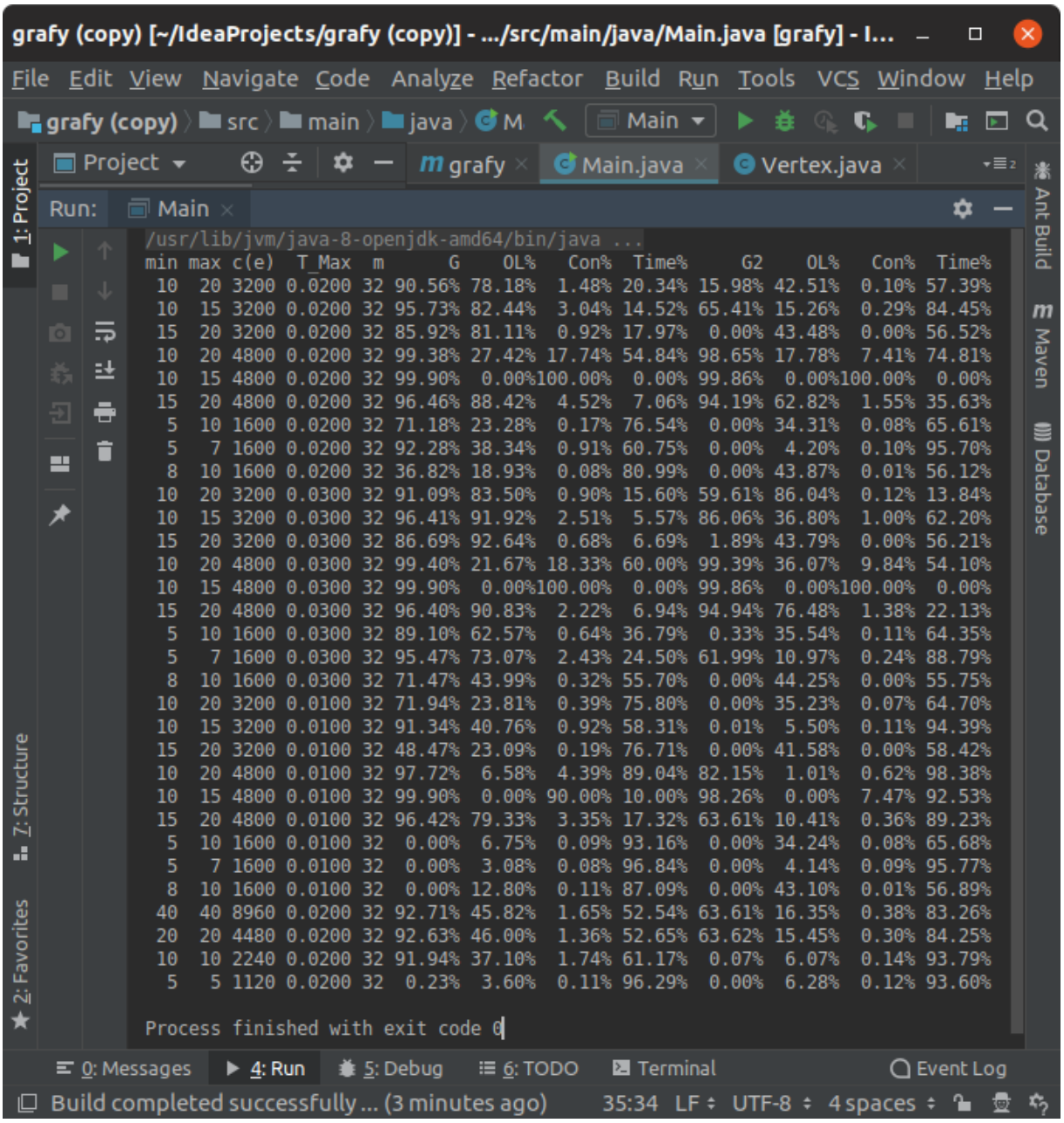
```
void createPaths(){
    for(Vertex v: vertices){
        BFS(v);
    }
}

void BFS(Vertex vertex)
{
    checkedVertices.add(vertex);
    vertex.setChecked(true);
    int counter=0;
    while(counter<checkedVertices.size()){
        Vertex[] out = checkedVertices.get(counter).linkedVertices();
        for(Vertex v: out){
            if(!v.isChecked()){
                v.setPathParent(checkedVertices.get(counter));
                checkedVertices.add(v);
                v.setChecked(true);
            }
        }
        counter++;
    }
    edgePaths();
    checkedVertices.clear();
    for (Vertex v:vertices){
        v.setPathParent(null);
        v.setChecked(false);
    }
}

void edgePaths(){
    Vertex tmp;
    for(int i=1; i<checkedVertices.size();i++){
        tmp = checkedVertices.get(i);
        while(tmp.getPathParent()!=null){
            for (Edge e:edges) {
                if(e.exists(tmp,tmp.getPathParent())){
                    e.addPath(checkedVertices.get(0),tmp);
                }
            }
            tmp = tmp.getPathParent();
        }
    }
}
```

```
boolean checkConnection(){
    checkedVertices.add(vertices[0]);
    vertices[0].setChecked(true);
    int counter=0;
    while(counter<checkedVertices.size()){
        Vertex[] out = checkedVertices.get(counter).linkedVertices();
        for(Vertex v: out){
            if(!v.isChecked()){
                checkedVertices.add(v);
                v.setChecked(true);
            }
        }
        counter++;
    }
    for (Vertex v:vertices
         ) {
        v.setChecked(false);
    }
    if(vertices.length==checkedVertices.size()){
        //System.out.println("This graph is connected");
        return true;
    } else {
        //System.out.println("This graph is not connected");
        return false;
    }
```

```
public void testReliability(){
    Random r = new Random();
    int counter = 0;
    while (counter<edges.size()){
        if(r.nextDouble()>edges.get(counter).getWeight()){
            //System.out.println("Connection between " + edges.get(counter).getSrc(
edges.get(counter).getDest().getIndex() + " lost!");
            removeEdge(edges.get(counter).getSrc(),edges.get(counter).getDest());
        } else {
            counter++;
        }
    }
}
```

Na samym końcu program drukuje nam tabelę z wszystkimi danymi, która przestawia nam:

- minimalną i maksymalną ilość pakietów wysyłanych między wierzchołkami
- przepustowość pojedynczej krawędzi oraz maksymalny czas przesyłu
- niezawodność danej sieci z procentowym rozkładem powodu awarii:
    - awaria przez niespójność sieci
    - awaria przez przeciążenie kanału
    - awaria przez przekroczenie czasu



Z powyższej tabeli można wywnioskować ciekawą rzecz:

Graf Petersena ma mniej połączeń pomiędzy wierzchołkami a mimo to jest mniej awaryjny, w grafie kołowym widać jak często on psuje się przez przekroczenie czasu oraz przeciążenia węzłów nawet przy wysokich ustawieniach maksymalnej przepustowości oraz czasu pakietu. Graf Petersena reprezentuje lepszy model sieci, który zawodził tylko w momentach kiedy został ustawiony zbyt mały czas oczekiwania albo zbyt mała przepustowość sieci. Jest to w pewnej mierze spowodowane tym, że w grafie Petersena przez jeden kanał przechodzi maksymalnie 3 lub 4 (w zależności od implementacji algorytmu badającego najkrótsze ścieżki) najkrótszych ścieżek dla połączeń, a w przypadku grafu kołowego takich ścieżek może przejść nawet 7 co

zmierzyłem za pomocą funkcji pathAmountMax. W dodatku warto zauważyć (4 ostatnie przykłady), że pomimo zmniejszania przepustowości i natężenia proporcjonalnie to sieć dla małej ilości pakietów i małej przepustowości częściej była timeoutowana niż ta, która miała dużą ilość pakietów i dużą przepustowość

```
int pathAmountMax(){
        int max = 0;
        for (Edge e: edges){
            if(e.pathsAmount()>max){
                max = e.pathsAmount();
            }
        }
        return max;
    }
```

Z eksperymentu można wywnioskować:

- Sieć powinna być budowana w ten sposób, aby zerwanie pojedynczego połączenia nie powodowało dużego wpływu na nią.
- Najistotniejszymi parametrami jest topologia sieci, jej przepustowość oraz wpływ na nią może mieć opóźnienie akceptowane przez użytkowników.
- Lepszą topologię sieci komputerowych reprezentują grafy regularne i k-spójne.
- Trudno jest określić, który z parametrów ma największy wpływ na wydajność sieci, lecz na pewno większa przepustowość pozwala na obniżenie średniego czasu oczekiwania na pakiet.
-