Sprawozdanie

Technologie sieciowe – lista 2

Aleksandra Malinowska, WPPT INF, 4 semestr, kwiecień 2019

1. Badanie niezawodności sieci

1.1. Model sieci

Niech modelem sieci będzie S = <G,H>, który posiada nieskierowany graf G z wierzchołkami v i krawędziami e oraz funkcję niezawodności h(e), która przypisuje krawędziom prawdopodobieństwo nierozerwania kanału komunikacyjnego.

1.2. Program testujący

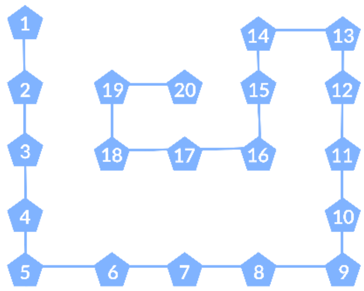
Aby oszacować niezawodność sieci posłużymy się metodą Monte Carlo. W ten sposób otrzymamy algorytm, który dla każdej krawędzi losuje wartość z przedziału (0,1) i sprawdza, czy jest ona większa od wartości funkcji h dla tej krawędzi i usuwa krawędź z grafu, jeśli warunek zostaje spełniony. Następnie algorytm sprawdza spójność grafu (tym samym spójność sieci) i zwiększa licznik sukcesów w przypadku pozytywnego wyniku. Wartością otrzymywaną w wyniku działania programu jest liczba sukcesów podzielona przez liczbę wszystkich prób.

```
private static double reliabilityEstimator(Graph graph, int interval, int testTime) {
    int time = 0;
    double success = 0, attempts = 0;
    Random rand = new Random();
    ArrayList<WeightedEdge> edges = new ArrayList<>();
    ArrayList<WeightedEdge> removedEdges = new ArrayList<>();
    graph.edgeSet().forEach(e -> edges.add(((WeightedEdge) e)));
    while (time < testTime) {
        attempts++;
        for (WeightedEdge edge : edges) {
            if (rand.nextDouble() > edge.getWeight()) {
                removedEdges.add(edge);
                graph.removeEdge(edge);
            }
        }
        if (GraphTests.isConnected(((UndirectedGraph) graph))) {
            success++;
        }
        removedEdges.forEach(edge -> graph.addEdge(edge.getSource(),
edge.getTarget(), edge););
        time += interval;
    }
    return success/attempts;
}
```

1.3. Próby dla różnych wariantów modelu

1.3.1. Model podstawowy

Podstawowy model sieci zakłada $v = 1, \dots, 20$, $e = (1,2), \dots, (19,20)$ oraz $\forall e\ h(e) = 0.95$. Testy przeprowadzamy dla różnych interwałów czasowych.

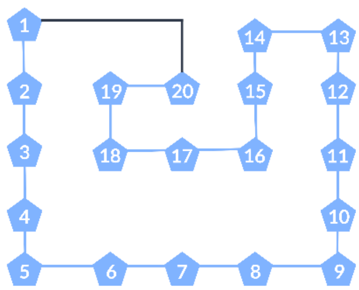


Interwał (ilość prób)	Prawdopodobieństwo uszkodzenia sieci
100	38.0%
1000	35.8%
10000	37.84%
100000	37.632%

Analizując wyniki wyraźnie widać, że zwiększony interwał powoduje większą dokładność wyników pomiarów. Widać również, że sieć, w której wystarczy usunięcie jednej krawędzi prowadzi do rozspójnienia grafu, jest bardzo podatna na uszkodzenia.

1.3.2. Dodatkowa krawędź

W tym teście rozszerzymy model sieci o dodatkową krawędź e = (1,20), która powoduje stworzenie grafu cyklicznego. Teraz do uszkodzenia sieci potrzebne jest usunięcie z grafu dwóch krawędzi.

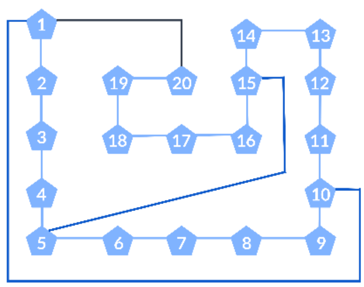


Interwał (ilość prób)	Prawdopodobieństwo uszkodzenia sieci
100	72.0%
1000	73.4%
10000	74.02%
100000	73.5701%

Wyniki pokazują, że graf cykliczny wykazuje dwukrotnie większą odporność na uszkodzenia.

1.3.3. Trzy dodatkowe krawędzie

W następnym teście dodajemy kolejne dwie krawędzie e=(1,10) oraz e=(5,15).



Interwał (ilość prób)	Prawdopodobieństwo uszkodzenia sieci
100	89.0%
1000	86.9%
10000	87.25%
100000	87.13%

Teraz widać, że dodatnie kolejnych krawędzi ponownie polepszyło niezawodność sieci. Tym razem jednak różnica nie jest aż tak znaczna, jak w poprzednim przypadku.

1.3.4. Siedem dodatkowych krawędzi

W ostatnim teście sprawdzamy niezawodność sieci dla grafu z rodziału 1.3.3. rozszerzonego o 4 losowe wierzchołki wygenerowane przez program.

Interwał (ilość prób)	Prawdopodobieństwo uszkodzenia sieci
100	84.0%
1000	92.601%
10000	91.38%
100000	91.362%

Tu wyraźnie widać, że dodawanie kolejnych krawędzi nie wypływa już tak wyraźnie na niezawodność sieci, jak w rozdziale 1.3.2. Wyniki są nieznacznie lepsze.

2. Rozszerzone badanie niezawodności sieci

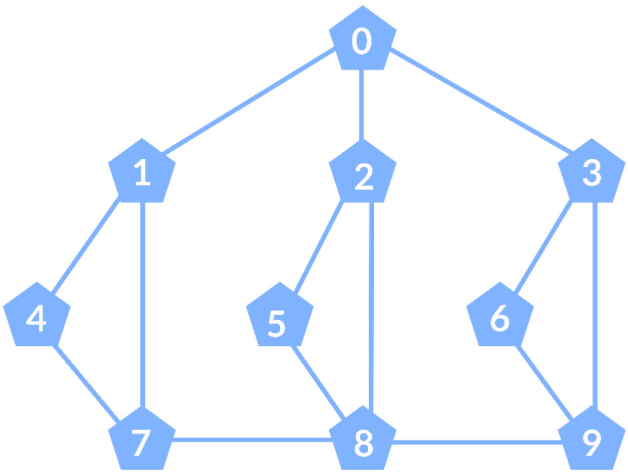
2.1. Model sieci

2.1.1. Oznaczenia

Do modelu utworzonego w rozdziale 1.1. dodajmy macierz N wymiaru n×n, gdzie n to liczba wierzchołków w grafie G, funkcję c(e), której wartością jest maksymalna liczba bitów, którą można wprowadzić do kanału, oraz funkcję a(e) przyporządkowującą krawędziom faktyczną liczbę pakietów wprowadzanych do kanału. Dodatkowo przez m oznaczymy średnią wielkość pakietu w bitach, przez T średnie opóźnienie pakietu, a przez TMax maksymalne średnie opóźnienie pakietu.

2.1.2. Propozycja modelu

Do testów przyjmijmy poniższy graf.



Niech macierz N będzie określona następująco:

```
0 1 2 3 4 5 1 2 3 4
1 0 3 4 5 1 2 3 4 5
2 3 0 5 1 2 3 4 5 1
3 4 5 0 2 3 4 5 1 2
4 5 1 2 0 4 5 1 2 3
5 1 2 3 4 0 1 2 3 4
1 2 3 4 5 1 0 3 4 5
2 3 4 5 1 2 3 0 5 1
3 4 5 1 2 3 4 5 0 2
4 5 1 2 3 4 5 1 2 0
```

Stąd wynikowa macierz A powstała z obliczenia wartości funkcji a(e) dla wszystkich możliwych połączeń w grafie wygląda następująco:

```
 0 50 40 52  0  0  0  0  0  0
50  0  0  0 24  0  0 38  0  0
40  0  0  0  0 16  0  0 32  0
52  0  0  0  0  0 14  0  0 20
 0 24  0  0  0  0  0 30  0  0
 0  0 16  0  0  0  0  0 34  0
 0  0  0 14  0  0  0  0  0 42
 0 38  0  0 30  0  0  0 76  0
 0  0 32  0  0 34  0 76  0 72
 0  0  0 20  0  0 42  0 72  0
```

Wartości w tej macierzy zostały wygenerowane przez funkcję obliczającą przepływność kanału. Algorytm uzyskiwania tej wartości polega na szukaniu najkrótszej ścieżki między dwoma wierzchołkami, a następnie powiększa dla każdej krawędzi na ścieżce jej wartość o natężenie na tym kanale.

```
private void setBitRateMatrix() {
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            bitRate[i][j] = 0;
        }
    }
    DijkstraShortestPath<Integer, WeightedEdge> path = new
DijkstraShortestPath<>(graph);
    for (int i = 0; i < 10; i++) {
        for (int j = i+1; j < 10; j++) {
            List<WeightedEdge> edges = path.getPath(i,j).getEdgeList();
            for (WeightedEdge e :
                    edges) {
                bitRate[e.getSource()][e.getTarget()] += intensity[i][j] +
intensity[j][i];
                bitRate[e.getTarget()][e.getSource()] =
bitRate[e.getSource()][e.getTarget()];
            }
        }
    }
}
```

Określmy też macierz C, która zawiera wartości funkcji c dla poszczególnych połączeń.

```
0 100000 100000 100000 100000 100000 100000 100000 100000 100000
100000 0 100000 100000 100000 100000 100000 100000 100000 100000
100000 100000 0 100000 100000 100000 100000 100000 100000 100000
100000 100000 100000 0 100000 100000 100000 100000 100000 100000
100000 100000 100000 100000 0 100000 100000 100000 100000 100000
100000 100000 100000 100000 100000 0 100000 100000 100000 100000
100000 100000 100000 100000 100000 100000 0 100000 100000 100000
100000 100000 100000 100000 100000 100000 100000 0 100000 100000
100000 100000 100000 100000 100000 100000 100000 100000 0 100000
100000 100000 100000 100000 100000 100000 100000 100000 100000 0
```

Ustalając również m = 1024 otrzymujemy model, w którym spełniony jest warunek c(e) > a(e) * m, zatem model ten jest poprawny.

2.2. Badanie średniego opóźnienia pakietu

2.2.1. Wzór

Do obliczenia średniego opóźnienia pakietu użyjemy poniższego wzoru.

$$T = \frac{\sum_{e \in E} \frac{a(e)}{\frac{c(e)}{m} - a(e)}}{\sum_{i=0}^{9} \sum_{j=0}^{9} N[i][j]}$$

2.2.2. Program

Po zaimplementowaniu powyższego modelu sieci, możemy wywołać funkcję obliczającą średnie opóźnienie pakietu.

```
double getPacketDelay(int packageSize) {
    double sumN = 0;
    for (int[] i : intensity) {
        for (int j : i) {
            sumN += j;
        }
    }
    ArrayList<WeightedEdge> edges = new ArrayList<>(graph.edgeSet());
    double sumE = 0, bitRate;
    for (WeightedEdge edge:
            edges) {
        bitRate = getEdgeBitRate(edge);
        sumE += (bitRate/((getEdgeCapacity(edge)/packageSize) - bitRate));
    }
    return sumE/sumN;
}
```

Wynikiem działania tego programu jest T = 48.151197159531776ms. Wartości T dla innych wartości m (zgodnych z modelem) znajdują się w tabeli poniżej.

Wielkość pakietu	Średnie opóźnienie pakietu
512 b	13.739472934025645 ms
128 b	2.726405637044796 ms
256 b	5.841708956282627 ms

2.3. Niezawodność sieci w zależności od opóźnienia pakietu

2.3.1. Program

Aby doprecyzować prawdopodobieństwo uszkodzenia sieci, możemy wzbogacić algorytm o dodatkowy warunek: T < TMax. W tym przypadku po każdorazowym usunięciu losowych krawędzi graf wciąż jest spójny, ustalamy nową macierz A, sprawdzamy czy w każdym przypadku przepływność jest mniejsza od przepustowości, a następnie sprawdzamy warunek T < TMax.

```
double reliabilityEstimator(int interval, int testTime, double probability,
double TMax, int packageSize) {
    int time = 0;
    double success = 0, attempts = 0;
    ArrayList<WeightedEdge> edges = new ArrayList<>();
    graph.edgeSet().forEach(e -> {
        WeightedEdge edge = ((WeightedEdge) e);
        edge.setWeight(probability);
        edges.add(edge);

    });
    ArrayList<WeightedEdge> removedEdges = new ArrayList<>();
    while (time < testTime) {
        attempts++;
        for (WeightedEdge edge :
                edges) {
            Random rand = new Random();
            double p = rand.nextDouble();
            if (p > edge.getWeight()) {
                removedEdges.add(edge);
                graph.removeEdge(edge);
            }
        }
        if (GraphTests.isConnected(((UndirectedGraph) graph))) {
            setBitRateMatrix();
            if (isBitRateMatrixFine(packageSize)) {
                if (getPacketDelay(packageSize) < TMax) {
                    success++;
                }
            }
        }
        for (WeightedEdge edge : removedEdges) {
            graph.addEdge(edge.getSource(), edge.getTarget(), edge);
        }
        time += interval;
    }
    return success/attempts;
}
```

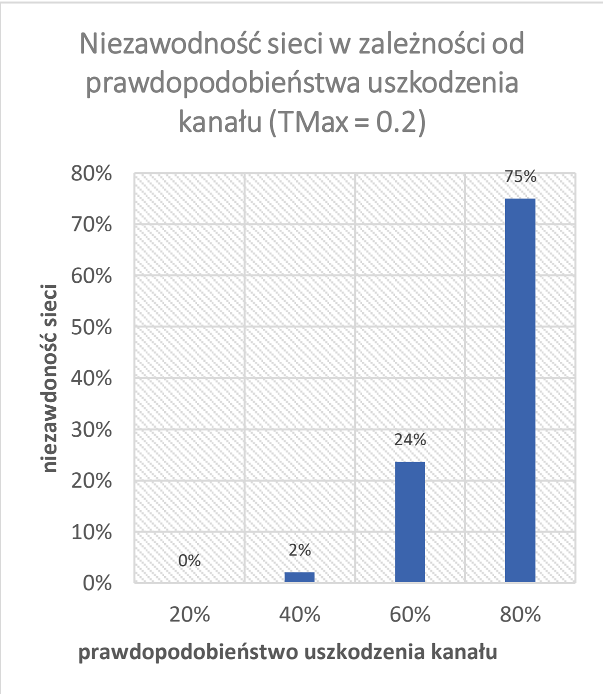
Algorytm z rozdziału 1.2. rozszerzamy również o możliwość ustalenia wartości funkcji h dla wszystkich krawędzi.

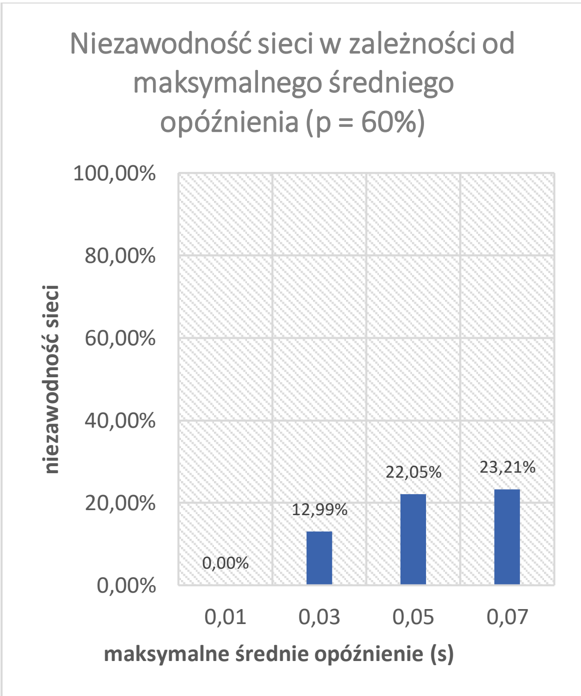
2.3.2. Przykładowe testy niezawodności sieci

Program uruchomiany dla różnych wartości funkcji h oraz TMax daje różne wyniki.

Wielkość pakietu (b)	Prawdopodobieństwo nieuszkodzenia kanału	Maksymalne średnie opóźnienie pakietu (s)	Prawdopodobieństwo nierozspójnienia sieci
1024	0.9292640283074929	0.27168999357198487	70.38%
1024	0.5500598355486743	0.26398623739463056	1.6%
1024	0.609013949408805	0.2922793083750094	3.5700003%
1024	0.4900252360404622	0.34780808366987337	0.62%
1024	0.7479866831681379	0.46747555498197035	18.56%
512	0.5869432714232597	0.8810639595181041	20.76%
512	0.5209448501975861	0.951891892336308	10.94%
512	0.22280120353045207	0.08238605581869185	0.01%
512	0.6957280101241328	0.8976484146819043	46.22%
512	0.9408818218942732	0.8498080362966527	97.83%

Z powyższych danych wynika, że niezawodność sieci jest silniej związana z wartością funkcji h niż z TMax. Widać, że dla niskich wartości h i TMax niezawodność wyraźnie spada, natomiast zmniejszenie wielkości pakietu wpływa na nią korzystnie.





Testy przeprowadzone dla stałej średniej wielkości pakietu równej 512 b pokazują, że prawdopodobieństwo uszkodzenia kanału bardziej wpływa na niezawodność sieci niż maksymalne średnie opóźnienie.

3. Wnioski

Z powyższych testów wynika, że niezawodność sieci jest ściśle związana z niezawodnością kanałów komunikacyjnych oraz ich przepustowością. Wpływa na nią również ilość połączeń między wierzchołkami grafu modelu sieci.