Szymon Głąb 244755 05.03.2019, Wrocław

Sprawozdanie 2

Zadanie 1

Opis doświadczenia

Celem doświadczenia było oszacowanie niezawodności sieci w zależności w zależności od ilość przesyłanych danych, maksymalnego czasu oczekiwania, prawdopodobieństwa awarii sprzętowych oraz topologii. Wybrane z możliwych parametrów (dla systemu Windows):

Metoda Monte Carlo

Metoda stosowana do modelowania procesów zbyt złożonych, aby można było przewidzieć ich wyniki za pomocą podejścia analitycznego. Istotną rolę w tej metodzie odgrywa losowanie wielkości charakteryzujących proces, przy czym losowanie dokonywane jest zgodnie z rozkładem, który musi być znany.2 Wszystkie moje szacowania są oparte na tej metodzie.

Polecenie

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 niezawodności „h”** przyporządkowującą każdej krawędzi **e(j, k)** ze zbioru **E** wartość **0.95** oznaczającą prawdopodobieństwo nie uszkodzenia (nierozerwania) tego kanału komunikacyjnego w dowolnym przedziale czasowym. (Zakładamy, że wierzchołki nie ulegają uszkodzeniom).

1.1

Napisz program szacujący niezawodność wyżej wymienionej sieci w dowolnym interwale.

Podany poniżej fragment został uruchomiony odpowiednio 100, 1000, 10 000 i 100 000 razy.

```
SimpleWeightedGraph<Integer, DefaultWeightedEdge> aGraph = new A().createGraph();

aGraph.edgeSet().forEach(edge -> {

  if (r.nextInt( bound: 100) > aGraph.getEdgeWeight(edge)) {
    edgesToRemove.add(edge);
  }
});

edgesToRemove.forEach(aGraph::removeEdge);      //removes bad edges
if (isCohesive(aGraph)) {
  passed++;
}
counter++;
result = (double) passed / counter;
totalSum += result;
```

Gdzie graf A generuje graf o 20 wierzchołkach [1…20] i krawędziach [(1,2)…(19,20)], a każda krawędź ma wagę 0.95.

Metoda tworząca graf A

```
SimpleWeightedGraph<Integer, DefaultWeightedEdge> createGraph() {

  SimpleWeightedGraph<Integer, DefaultWeightedEdge> g = new SimpleWeightedGraph<>(DefaultWeightedEdge.class);

  for (int i = 1; i <= 20; i++) {
    g.addVertex(i);
  }

  for (int i = 1; i < 20; i++) {
    g.setEdgeWeight(g.addEdge(i,  targetVertex: i + 1),  weight: 95);
  }

  return g;
}
```

Tak prezentują się wyniki które otrzymałem:

Liczba prób	**100**	**1000**	**10 000**	**100 000**
Niezawodność sieci	0.46	0.45	0.46	0.46

Zrzut ekranu potwierdzający otrzymane wyniki

```
1.1 average result = 0,46 (with 100 tries)
1.1 average result = 0,45 (with 1000 tries)
1.1 average result = 0,46 (with 10000 tries)
1.1 average result = 0,46 (with 100000 tries)
```

1.2

Jak zmieni się niezawodność tej sieci po dodaniu krawędzi **e(1,20)** takiej, że **h(e(1,20))=0.95**

W podanym wcześniej kodzie zmieniłem tylko rodzaj tworzonego grafu, dodałem do niego wymienioną w poleceniu krawędź.

Metoda tworząca graf B

```
SimpleWeightedGraph<Integer, DefaultWeightedEdge> createGraph() {
  SimpleWeightedGraph<Integer, DefaultWeightedEdge> g = new SimpleWeightedGraph<>(DefaultWeightedEdge.class);

  for (int i = 1; i <= 20; i++) {
    g.addVertex(i);
  }

  for (int i = 1; i < 20; i++) {
    g.setEdgeWeight(g.addEdge(i,  targetVertex: i + 1),  weight: 95);
  }

  g.setEdgeWeight(g.addEdge( sourceVertex: 1,  targetVertex: 20),  weight: 95);

  return g;
}
```

Jak widać graf rozbudowaliśmy o dodatkową krawędź [(1,20)] o wadzę 0.95.

Otrzymałem następujące wyniki

Liczba prób	**100**	**1000**	**10 000**	**100 000**
Niezawodność sieci	0.81	0.82	0.81	0.81

Zrzut ekranu potwierdzający otrzymane wyniki

```
1.2 average result = 0,81 (with 100 tries)
1.2 average result = 0,82 (with 1000 tries)
1.2 average result = 0,81 (with 10000 tries)
1.2 average result = 0,81 (with 100000 tries)
```

1.3

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.

W podanym wcześniej kodzie zmieniłem tylko rodzaj tworzonego grafu, dodałem do niego wymienione w poleceniu krawędzie.

Metoda tworząca graf C

```
SimpleWeightedGraph<Integer, DefaultWeightedEdge> createGraph() {

  SimpleWeightedGraph<Integer, DefaultWeightedEdge> graph = new SimpleWeightedGraph<>(DefaultWeightedEdge.class);

  for (int i = 1; i <= 20; i++) {
    graph.addVertex(i);
  }

  for (int i = 1; i < 20; i++) {
    graph.setEdgeWeight(graph.addEdge(i, targetVertex: i + 1), weight: 95);
  }

  graph.setEdgeWeight(graph.addEdge( sourceVertex: 1, targetVertex: 10), weight: 80);
  graph.setEdgeWeight(graph.addEdge( sourceVertex: 5, targetVertex: 15), weight: 70);
  graph.setEdgeWeight(graph.addEdge( sourceVertex: 1, targetVertex: 20), weight: 95);

  return graph;
}
```

Jak widać graf rozbudowaliśmy o dodatkową krawędź [(1,10)] o wadzę 0.8 oraz krawędź [(5,15)] o wadzę 0.7.

Otrzymałem następujące wyniki

Liczba prób	100	1000	10 000	100 000
Niezawodność sieci	0.88	0.91	0.91	0.91

Zrzut ekranu potwierdzający wyniki

```
1.3 average result = 0,88 (with 100 tries)
1.3 average result = 0,91 (with 1000 tries)
1.3 average result = 0,91 (with 10000 tries)
1.3 average result = 0,91 (with 100000 tries)
```

1.4

A jak zmieni się niezawodność tej sieci gdy dodatkowo dodamy jeszcze 4 krawędzie pomiędzy losowymi wierzchołkami o h=0.4.

W podanym wcześniej kodzie zmieniłem tylko rodzaj tworzonego grafu, dodałem do niego wymienioną w poleceniu 4 losową krawędzie.

Metoda tworząca graf D

```
public SimpleWeightedGraph<Integer, DefaultWeightedEdge> createGraph() {

  SimpleWeightedGraph<Integer, DefaultWeightedEdge> g =
          new SimpleWeightedGraph<>(DefaultWeightedEdge.class);

  for (int i = 1; i <= 20; i++) {
    g.addVertex(i);
  }

  for (int i = 1; i < 20; i++) {
    g.setEdgeWeight(g.addEdge(i,  targetVertex: i + 1),  weight: 95);
  }

  g.setEdgeWeight(g.addEdge( sourceVertex: 1,  targetVertex: 20),  weight: 95);
  g.setEdgeWeight(g.addEdge( sourceVertex: 1,  targetVertex: 10),  weight: 80);
  g.setEdgeWeight(g.addEdge( sourceVertex: 5,  targetVertex: 15),  weight: 70);

  for (int i = 0; i < 4; i++) {
    int i1 = 0;
    int i2 = 0;

    while (i1 == i2 || g.containsEdge(i1, i2) || g.containsEdge(i2, i1)) {
      i1 = randInt(1, 20);
      i2 = randInt(1, 20);
    }

    g.setEdgeWeight(g.addEdge(i1, i2),  weight: 70);
  }

  return g;

}
```

Jak widać graf rozbudowaliśmy o kolejną krawędź, tym razem losową o wadze 0.7.

Otrzymałem następujące wyniki

Liczba prób	100	1000	10 000	100 000
Niezawodność sieci	0.97	0.95	0.96	0.96

Zrzut ekranu potwierdzający wyniki

```
1.4 average result = 0,97 (with 100 tries)
1.4 average result = 0,95 (with 1000 tries)
1.4 average result = 0,96 (with 10000 tries)
1.4 average result = 0,96 (with 100000 tries)
```

Zadanie 2

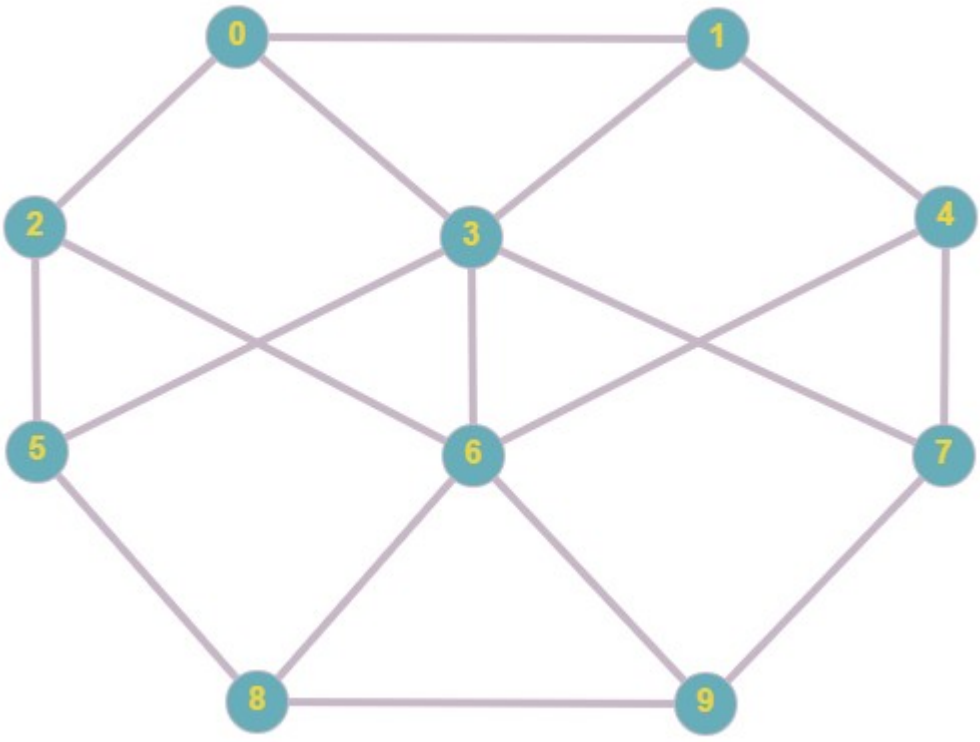
Polecenie

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).**

2.1

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 przepływu realizowała macierz N oraz aby dla każdego kanału **„e”** zachodziło: **c(e) > a(e)**.

Moja propozycja topologii grafu



Moja propozycja macierzy N

00	01	02	03	04	05	06	07	08	09
01	00	02	03	04	05	06	07	08	09
02	02	00	03	04	05	06	07	08	09
03	03	03	00	04	05	06	07	08	09
04	04	04	04	00	05	06	07	08	09
05	05	05	05	05	00	06	07	08	09
06	06	06	06	06	06	00	07	08	09
07	07	07	07	07	07	07	00	08	09
08	08	08	08	08	08	08	08	00	09
09	09	09	09	09	09	09	09	09	00

Funkcję „c" (przepustowości) reprezentuję za pomocą macierzy postaci:

150000	150000	150000	150000	150000	150000	150000	150000	150000	150000
150000	150000	150000	150000	150000	150000	150000	150000	150000	150000
150000	150000	150000	150000	150000	150000	150000	150000	150000	150000
150000	150000	150000	150000	150000	150000	150000	150000	150000	150000
150000	150000	150000	150000	150000	150000	150000	150000	150000	150000
150000	150000	150000	150000	150000	150000	150000	150000	150000	150000
150000	150000	150000	150000	150000	150000	150000	150000	150000	150000
150000	150000	150000	150000	150000	150000	150000	150000	150000	150000
150000	150000	150000	150000	150000	150000	150000	150000	150000	150000
150000	150000	150000	150000	150000	150000	150000	150000	150000	150000

Natomiast funkcja „a" (przepływu) za pomocą takiej macierzy:

00	06	29	23	00	00	00	00	00	00
06	00	00	39	00	00	00	00	00	00
29	00	00	00	00	22	18	00	00	00
23	39	00	00	00	23	25	09	00	00
00	00	00	00	00	00	22	00	00	00
00	00	22	23	00	00	00	00	09	00
00	00	18	25	22	00	00	00	00	00
00	00	00	09	00	00	00	00	00	08
00	00	00	00	00	09	00	00	00	00
00	00	00	00	00	00	00	08	00	00

2.2

Napisz program, w którym propozycje będzie można testować, tzn. który dla wybranych reprezentacji zadanych odpowiednimi macierzami, będzie obliczał średnie opóźnienie pakietu 'T' dane wzorem:

$$T = \frac{1}{\sum_{i=0} \sum_{j=0} N[i][j]} \cdot \sum_{e \in E} \frac{a(e)}{\frac{c(e)}{M} - a(e)}$$

M – średnia wielkość pakietu (przyjąłem 1500B)

E – zbiór wszystkich krawędzi grafu

Program sprawdza czy któraś z krawędzi została uszkodzona, a następnie sprawdza spójność grafu oraz podlicza średnią arytmetyczną czasu opóźnienia.

Otrzymałem następujące wyniki:

Liczba prób	10 000	100 000	1 000 000
Średnie opóźnienie	0.00443 s	0.00442 s	0.0044203 s

Dla 10 000

```
Average delay 0.004438887432115287 (with 10000 tries)
```

Dla 100 000

```
Average delay 0.004420048780340882 (with 100000 tries)
```

Dla 1 000 000

```
Average delay 0.00442032291349371 (with 1000000 tries)
```

A to kod mojego programu, który użyłem do tych obliczeń

```
for (int i = 0; i < relabilityTestAmount; i++) {
  tempGraph = utils.createGraph();
  matrixA = utils.createMatrixA(matrixN, tempGraph);

  tempGraph.edgeSet().forEach(edge -> {
    if (r.nextDouble() > failProbability) {
      edgesToRemove.add((DefaultWeightedEdge) edge);
    }
  });

  edgesToRemove.forEach(tempGraph::removeEdge);
  edgesToRemove.clear();

  if (utils.isCohesive(tempGraph)) {
    passed++;
    delay += utils.calcDelay(matrixN, matrixC, matrixA, tempGraph);
  }
}

  double result = (delay / passed);
  System.out.println("Average delay " + result + " (with " + relabilityTestAmount + " tries)");
```

2.3

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 nie uszkodzenia każdej krawędzi w dowolnym interwale jest równe „p”.

Program rozbudowałem o dodatkową funkcję **graphReliability(),** która sprawdza czy opóźnienie dla danego grafu nie przekracza T_{max}

Kod tej metody:

```
Boolean graphReliability(SimpleWeightedGraph graph){

    int[][] matrixA = createMatrixA(matrixN,graph);
    double delay = calcDelay(matrixN,matrixC,matrixA,graph);
    if(delay>0 && delay<maxDelay){
      return true;
    }
  return false;
}
```

Program testowałem dla różnych wartości T_{max} oraz różnej ilości prób. Poniżej znajdują się wyniki tego eksperymentu. Wszystkie próby wykonałem przy prawdopodobieństwie p = 0.8

	1000	10 000	100 000	1 000 000
T_{max} = 0.1	0.881	0.8844	0.8821	0.8820
T_{max} = 0.01	0.579	0.5975	0.5941	0.5925
T_{max} = 0.001	0.001	0.0021	0.0014	0.0016

W tabeli wpisany jest stosunek prób udanych do prób nieudanych

Po stosunkowo dobrych wynikach postanowiłem zmniejszyć maksymalne przepływy o połowę, a więc moja macierz C wyglądała tak:

```
75000 75000 75000 75000 75000 75000 75000 75000 75000 75000
75000 75000 75000 75000 75000 75000 75000 75000 75000 75000
75000 75000 75000 75000 75000 75000 75000 75000 75000 75000
75000 75000 75000 75000 75000 75000 75000 75000 75000 75000
75000 75000 75000 75000 75000 75000 75000 75000 75000 75000
75000 75000 75000 75000 75000 75000 75000 75000 75000 75000
75000 75000 75000 75000 75000 75000 75000 75000 75000 75000
75000 75000 75000 75000 75000 75000 75000 75000 75000 75000
75000 75000 75000 75000 75000 75000 75000 75000 75000 75000
75000 75000 75000 75000 75000 75000 75000 75000 75000 75000
```

I otrzymałem takie wyniki:

	1000	10 000	100 000	1 000 000
T_{max} = 0.1	0.631	0.6185	0.6171	0.6186
T_{max} = 0.01	0.076	0.0742	0.0734	0.0723
T_{max} = 0.001	0.007	0.0071	0.057	0.0071

W tabeli wpisany jest stosunek prób udanych do prób nieudanych

Zaciekawiony znaczną zmianą wyników zmniejszyłem macierz C do:

```
30000 30000 30000 30000 30000 30000 30000 30000 30000 30000
30000 30000 30000 30000 30000 30000 30000 30000 30000 30000
30000 30000 30000 30000 30000 30000 30000 30000 30000 30000
30000 30000 30000 30000 30000 30000 30000 30000 30000 30000
30000 30000 30000 30000 30000 30000 30000 30000 30000 30000
30000 30000 30000 30000 30000 30000 30000 30000 30000 30000
30000 30000 30000 30000 30000 30000 30000 30000 30000 30000
30000 30000 30000 30000 30000 30000 30000 30000 30000 30000
30000 30000 30000 30000 30000 30000 30000 30000 30000 30000
30000 30000 30000 30000 30000 30000 30000 30000 30000 30000
```

Otrzymując następujące wyniki:

	1000	10 000	100 000	1 000 000
$T_{max} = 0.1$	0.091	0.079	0.076	0.078
$T_{max} = 0.01$	0.039	0.037	0.040	0.041
$T_{max} = 0.001$	0.015	0.011	0.011	0.010

Kod mojego programu, który wykorzystałem do tych obliczeń

```
// RELIABILITY CALCULATOR
for (int i = 0; i < relabilityTestAmount; i++) {

  tempGraph = utils.createGraph();

    tempGraph.edgeSet().forEach(edge ->{
      if (r.nextDouble() > failProbability) {
        edgesToRemove.add((DefaultWeightedEdge)edge);
      }
    });

    edgesToRemove.forEach(tempGraph::removeEdge);
    edgesToRemove.clear();


    if(utils.isCohesive(tempGraph)){
      if(utils.graphReliability(tempGraph)){
        passed++;
      }
    }
  }
double reliability = (double)passed/relabilityTestAmount;
System.out.println("Network reliability = " + reliability + "(with " + relabilityTestAmount + " tries)");
```

Wnioski

Po przeprowadzeniu powyższych eksperymentów, możemy wyciągnąć wnioski, że podczas projektowania sieci należy przede wszystkim na przepustowość kanałów, którą powinno się dobrać na podstawie przewidywanej faktycznej liczby pakietów. Kolejnym ważnym czynnikiem jest niezawodność kanałów, bo to jej zawdzięczamy potem niezawodność całej sieci. Należy również zwrócić uwagę na odpowiednią topologię.