Karol Szymończyk 243434 30.03.2019r. Wrocław

Lista 2

♦ Zadanie 1

Celem zadania pierwszego było napisanie programu, który szacuje niezawodność (prawdopodobieństwo nierozspójnienia) poszczególnych sieci w dowolnym przedziale czasowym. Aby tego dokonać, przedstawiłem dane sieci w postaci grafów ważonych oraz posłużyłem się zalecaną metodą Monte Carlo w celu szacowania ich niezawodności.

- **Graf ważony –** struktura matematyczna służąca do przedstawiania i badania relacji między obiektami. Jest to zbiór wierzchołków, które mogą być połączone krawędziami, które posiadają wagę. Waga krawędzi to przypisana jej liczba, która określa jej wartość zależnie od użycia (np. może to być odległość w przypadku, gdy graf reprezentuje połączenia między miastami).

- **Metoda Monte Carlo –** jest to metoda stosowania do modelowania matematycznego procesów zbyt złożonych, aby móc przewidzieć ich wyniki za pomocą podejścia analitycznego. Istotną rolę w tej metodzie odgrywa wybór przypadkowy, przy czym losowanie dokonywane jest zgodnie z rozkładem, który musi być znany.

Początkowy model sieci reprezentowany był przez graf G = <V, E>, który składał się z dwudziestu wierzchołków v(i), odpowiednio dla i = 1,…,20 oraz dziewiętnastu krawędzi e(j, j+1), dla j = 1,…,19, gdzie e(j, k) oznacza krawędź łączącą wierzchołki v(j) oraz v(k). Dodatkowo każdej krawędzi e przyporządkowana była waga 0,95 oznaczająca prawdopodobieństwo nierozerwania w dowolnym przedziale czasowym.

Do implementacji podanych sieci w postaci grafów posłużyłem się biblioteką Javy o nazwie JGraph.

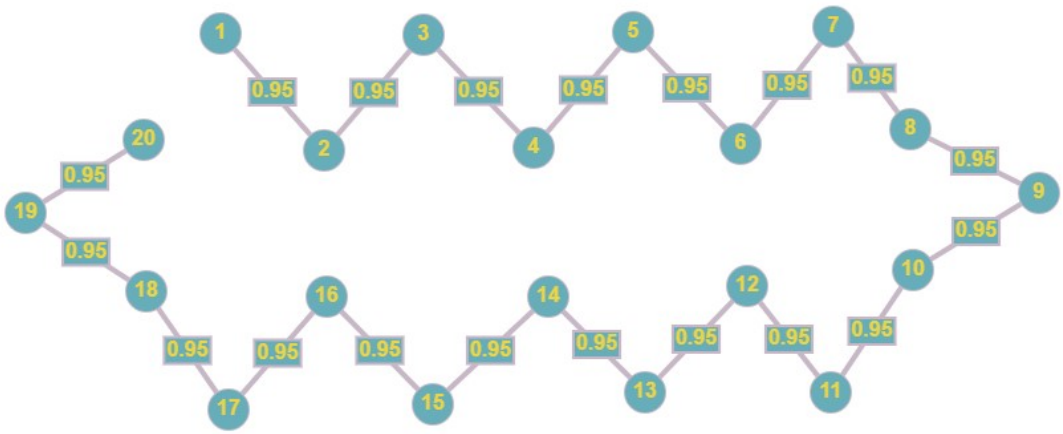
W każdym podpunkcie dla danego grafu przeprowadzanych jest 100 000 symulacji uszkodzenia jego krawędzi metodą Monte Carlo przy podanych prawdopodobieństwach nierozerwania, a następnie sprawdzana jest spójność powstałego w ten sposób grafu. Następnie wyliczane jest prawdopodobieństwo jego nierozspójnienia. Wynik podawany jest w procentach.

```java
private static void checkGraph(GraphType graphType) {
  SimpleWeightedGraph<Integer, DefaultWeightedEdge> graph;
  ArrayList<DefaultWeightedEdge> edges = new ArrayList<>();
  String name = graphType.getClass().getName();
  double weight;
  double random;
  int all = 0;
  int cohesive = 0;
  while (all != 100000) {
    graph = graphType.getGraph();
    edges.addAll(graph.edgeSet());
    for (DefaultWeightedEdge edge : edges) {
      weight = graph.getEdgeWeight(edge);
      random = generator.nextDouble();
      if (random > weight) {
        graph.removeEdge(edge);
      }
    }
    if(isCohesive(graph)) {
      cohesive++;
    }
    all++;

    calculateReliability(name, cohesive, all);
  }
  System.out.println();
}

private static boolean isCohesive(SimpleWeightedGraph<Integer, DefaultWeightedEdge> graph) {
  return  new ConnectivityInspector<>(graph).isGraphConnected();
}
```

- **Graf 1 –** składał się z wierzchołków oraz krawędzi przedstawionych poniżej.



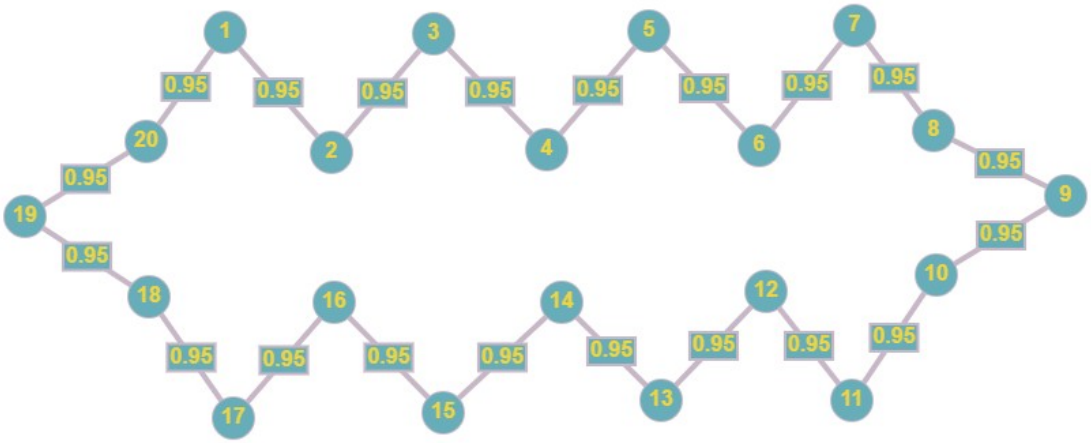

```
private void createGraph() {
  for(int i = 1; i <= 20; i++) {
    graph.addVertex(i);
    if (i > 1) {
      graph.setEdgeWeight(graph.addEdge( sourceVertex: i-1, i), weight: 0.95);
    }
  }
}
```

Wynik :

```
Graph1: 37%  37888/100000
```

Wynik wydaje się być wiarygodny, ponieważ jego poprawność możemy sprawdzić w łatwy sposób. Aby graf ten przestał być spójny wystarczy uszkodzenie co najmniej jednej jego krawędzi. Zatem wszystkie jego krawędzie muszą pozostać nieuszkodzone. Możemy wykonać obliczenia :
$0{,}95^{19} = 0{,}37735\ldots$ co w zaokrągleniu daje nam około 37%.

- **Graf 2 –** składał się z wierzchołków oraz krawędzi, które posiadał Graf 1 z dodatkową krawędzią e(1, 20), której prawdopodobieństwo również nierozerwania wynosiło 0,95.




```
private void createGraph() {
  for(int i = 1; i <= 20; i++) {
    graph.addVertex(i);
    if (i > 1) {
      graph.setEdgeWeight(graph.addEdge( sourceVertex: i-1, i), weight: 0.95);
    }
  }
  graph.setEdgeWeight(graph.addEdge( sourceVertex: 1, targetVertex: 20), weight: 0.95);
}
```

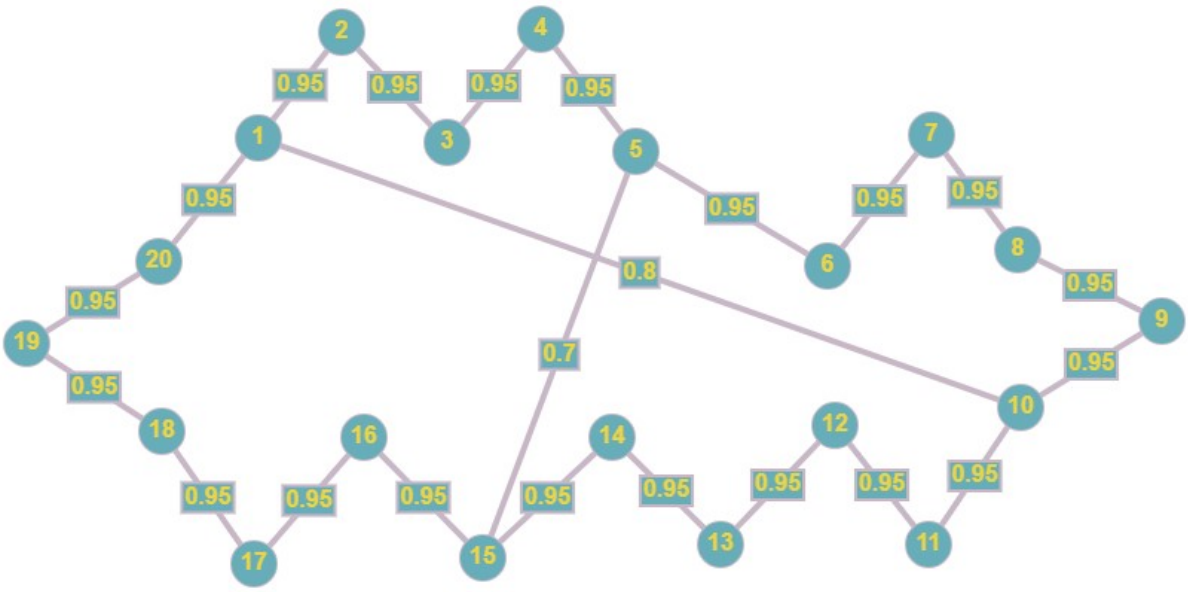
Wynik :

```
Graph2: 73%  73506/100000
```

W tym przypadku wynik również wydaje się być poprawny, ponieważ można zauważyć, że dopiero w przypadku uszkodzenia dwóch lub więcej krawędzi graf przestaje być spójny. Zatem uszkodzeniu może ulec maksymalnie jedna krawędź. Możemy wykonać obliczenia :

$\binom{20}{0} \cdot 0{,}95^{20} + \binom{20}{01} \cdot 0{,}95^{19} = 0.73583\ldots$ co w zaokrągleniu daje około 73%

- **Graf 3 -** składał się z wierzchołków oraz krawędzi, które posiadał Graf 2 z dodatkowymi krawędziami e(1, 10) oraz e(5, 15), które miały prawdopodobieństwo nierozerwania kolejno 0,8 oraz 0,7.




```
private void createGraph() {
  for(int i = 1; i <= 20; i++) {
    graph.addVertex(i);
    if (i > 1) {
      graph.setEdgeWeight(graph.addEdge( sourceVertex: i-1, i), weight: 0.95);
    }
  }
  graph.setEdgeWeight(graph.addEdge( sourceVertex: 1, targetVertex: 20), weight: 0.95);
  graph.setEdgeWeight(graph.addEdge( sourceVertex: 1, targetVertex: 10), weight: 0.8);
  graph.setEdgeWeight(graph.addEdge( sourceVertex: 5, targetVertex: 15), weight: 0.7);
}
```

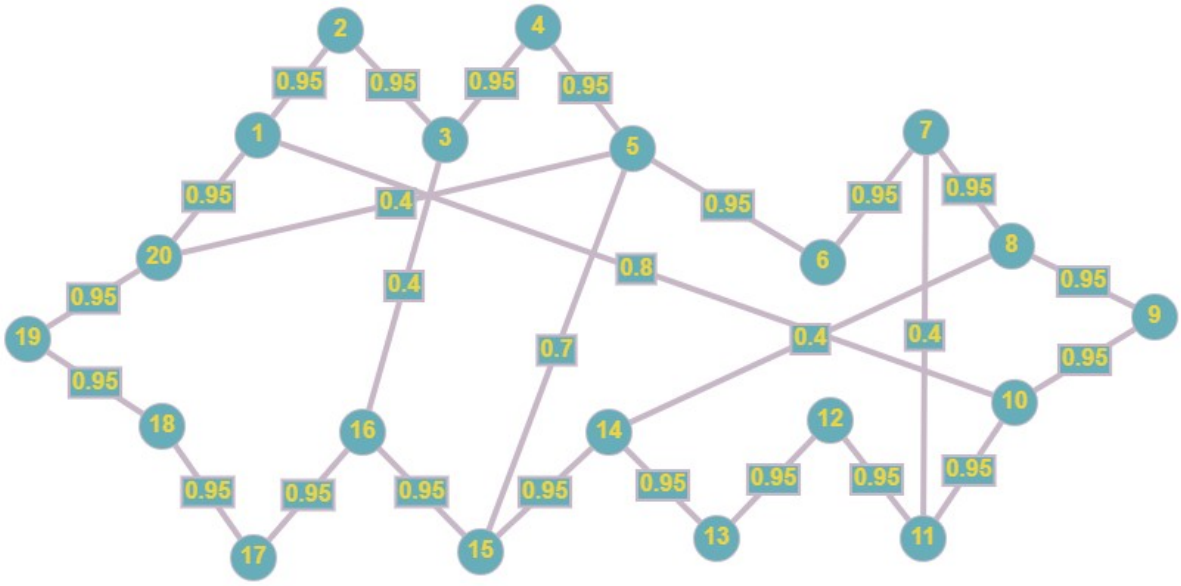
Wynik :

```
Graph3: 87%  87118/100000
```

W tym przypadku ciężko już jest oszacować oczekiwany wynik, jednak można założyć, że jest on wiarygodny na podstawie wcześniejszych rezultatów.

- **Graf 4 –** składał się z wierzchołków oraz krawędzi, które posiadał Graf 3 z dodatkowymi czterema krawędziami generowanymi pomiędzy losowymi wierzchołkami, których prawdopodobieństwo nierozerwania wynosiło 0,4.

Przykładowy, możliwy graf



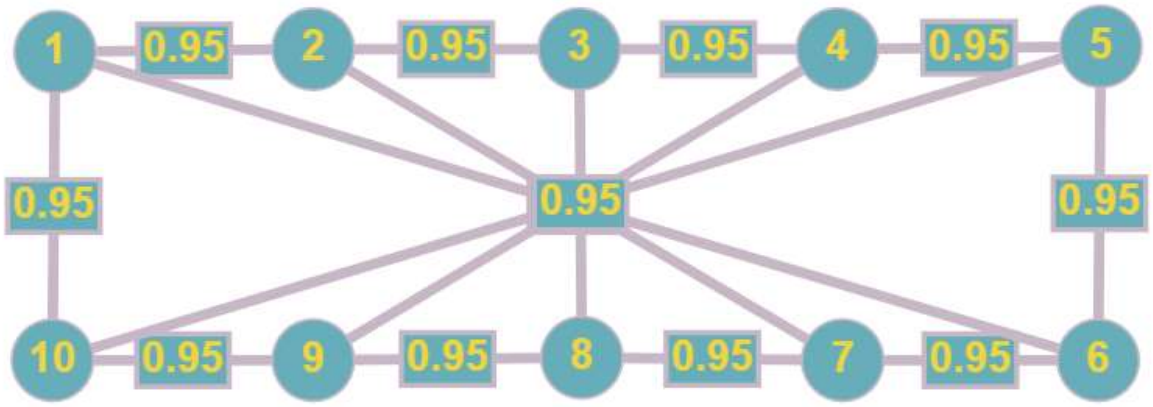

```
private void createGraph() {
  for(int i = 1; i <= 20; i++) {
    graph.addVertex(i);
    if (i > 1) {
      graph.setEdgeWeight(graph.addEdge( sourceVertex: i-1, i), weight: 0.95);
    }
  }
  graph.setEdgeWeight(graph.addEdge( sourceVertex: 1, targetVertex: 20), weight: 0.95);
  graph.setEdgeWeight(graph.addEdge( sourceVertex: 1, targetVertex: 10), weight: 0.8);
  graph.setEdgeWeight(graph.addEdge( sourceVertex: 5, targetVertex: 15), weight: 0.7);
  int counter = 0;
  while (counter != 4) {
    int n1 = generator.nextInt( bound: 19)+1;
    int n2 = generator.nextInt( bound: 19)+1;
    if(n1 == n2 || graph.containsEdge(n1, n2) || graph.containsEdge(n2, n1)) {
      continue;
    }
    graph.setEdgeWeight(graph.addEdge(n1, n2), weight: 0.4);
    counter++;
  }
}
```

Wynik :

```
Graph4: 91%  91198/100000
```

♦ Zadanie 2

W tym zadaniu należało zaproponować topologię grafu, tak aby żaden wierzchołek nie był izolowany, ilość wierzchołków była równa dziesięć oraz liczba krawędzi nie przekraczała dziewiętnastu. W moim przypadku liczba krawędzi jest równa 15.



Następnie należało zaproponować macierz natężeń strumienia pakietów N, gdzie każdy element n(i, j) jest liczbą pakietów przesyłanych w ciągu sekundy od źródła v(i) do ujścia v(j). Dodatkowo należało zaproponować funkcję przepustowości ‘c’, która określała maksymalną liczbę bitów, którą można wprowadzić do kanału w ciągu sekundy oraz funkcję ‘a’, jako faktyczną liczbę pakietów, które wprowadza się do kanału w ciągu sekundy. Funkcja przepływu miała realizować macierz N, tak aby dla każdego kanału ‘e’ zachodziło: c(e) > a(e).

Za jednostkę wielkości pakietu przyjąłem 1500 bitów.

Proponowana macierz N

0	9	8	7	6	5	4	3	2	1
9	0	9	8	7	6	5	4	3	2
8	9	0	9	8	7	6	5	4	3
7	8	9	0	9	8	7	6	5	4
6	7	8	9	0	9	8	7	6	5
5	6	7	8	9	0	9	8	7	6
4	5	6	7	8	9	0	9	8	7
3	4	5	6	7	8	9	0	9	8
2	3	4	5	6	7	8	9	0	9
1	2	3	4	5	6	7	8	9	0

Wyliczona macierz A

000	085	000	000	000	042	000	000	000	035
085	000	101	000	000	000	040	000	000	000
000	101	000	102	000	000	000	043	000	000
000	000	102	000	095	000	000	000	037	000
000	000	000	095	000	097	000	000	000	046
042	000	000	000	097	000	091	000	000	000
000	040	000	000	000	091	000	087	000	000
000	000	043	000	000	000	087	000	080	000
000	000	000	037	000	000	000	080	000	061
035	000	000	000	046	000	000	000	061	000

Macierz A została wygenerowana na podstawie najkrótszych ścieżek od każdego wierzchołka do każdego innego. Reprezentuje ona faktyczną liczbę pakietów, które wprowadza się do kanału w ciągu sekundy.

Macierz C

```
000 170 100 100 100 084 100 100 100 070
170 000 202 100 100 100 080 100 100 100
100 202 000 204 100 100 100 086 100 100
100 100 204 000 190 100 100 100 074 100
100 100 100 190 000 194 100 100 100 092
084 100 100 100 194 000 182 100 100 100
100 080 100 100 100 182 000 174 100 100
100 100 086 100 100 100 174 000 160 100
100 100 100 074 100 100 100 160 000 122
070 100 100 100 092 100 100 100 122 000
```

Macierz C powstała przez przemnożenie odpowiednich komórek macierzy A razy dwa. Miejsca, które w macierzy A były równe zero zostały zastąpione wartością 100. Reprezentuje ona maksymalną przepustowość zaproponowanej przeze mnie sieci.

- **Obliczanie średniego opóźnienia pakietu**

Aby obliczyć średnie opóźnienie pakietu posłużyłem się wzorem :

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

m – średnia wielkość pakietu (1500 bitów)

E – zbiór wszystkich krawędzi grafu

Następnie napisałem program, który na początku, podobnie jak w zadaniu pierwszym sprawdza każdą krawędź czy została uszkodzona, a następnie przeprowadza test czy powstały w ten sposób graf jest spójny. Później sprawdza, czy czas oczekiwania jest skończony oraz liczy średnią arytmetyczną ze wszystkich prób zakończonych powodzeniem. Graf został sprawdzony 100 000 razy.

Funkcja obliczająca opóźnienie pojedynczego grafu.

```
private static double calculateDelay(SimpleWeightedGraph graph) {
  double sum = 0;
  double downSum;
  int es;
  int et;

  for (Object e : graph.edgeSet()) {
    es = (int) graph.getEdgeSource(e);
    et = (int) graph.getEdgeTarget(e);
    downSum = C[es - 1][et - 1] - A[es - 1][et - 1];
    if (downSum == 0) {
      return 0;
    }
    sum += A[es - 1][et - 1] / downSum;
  }
  return sum / sumN;
}
```

Wynik :

```
MyNet average delay: 0.03817462285023714   3456.902972203224/90555
```

- **Szacowanie niezawodności sieci przy zachowaniu T < T_max**

Do programu opisanego wyżej dodałem opcję sprawdzania czy czas oczekiwania pojedynczego pakietu nie jest większy lub równy wcześniej ustalonemu maksymalnemu czasu oczekiwania T_max.

Za T_max przyjąłem 0,05 sekundy.

```
  if (isCohesive(graph)) {
    generateA(graph);
    double t = calculateDelay(graph);
    if (t > 0 && t < T_max) {
      sum += t;
      counter++;
    }
  }
  return sum / counter;
}
```

Wynik :

```
MyNet reliability: 0.72639   72639.0/100000
```

Co daje nam 72% niezawodności sieci.

Poniżej wyniki prób dla różnych T_MAX oraz prawdopodobieństwa pojawienia się krawędzi H.

T_MAX [s]	0,02	0,03	0,04	0,05	0,06	0,07	0,1
Niezawodność [%] Przy H = 0,95	2%	52%	66%	72%	81%	84%	85%
Niezawodność [%] Przy H = 0,85	2%	24%	35%	41%	52%	56%	59%
Niezawodność [%] Przy H = 0,75	1%	15%	21%	25%	29%	33%	37%

♦ Wnioski

Z doświadczeń przeprowadzonych w zadaniu pierwszym można wyciągnąć wnioski, że niezawodność sieci rośnie wraz ze wzrostem liczby jej kanałów. Jest to logiczne ponieważ im więcej połączeń ma dana sieć tym mniejsze prawdopodobieństwo, że po uszkodzeniu pewnej liczby kanałów sieć stanie się niespójna. Ważna jest także niezawodność poszczególnych kanałów rozumiana jako prawdopodobieństwo nieuszkodzenia danego połączenia.

Natomiast z zadania drugiego możemy wnioskować, że równie ważnym elementem planowania sieci jest odpowiednio dobrana przepustowość kanałów. Nawet w przypadku, gdy nie zawiodą kanały, a przepustowość okaże się niewystarczająco duża, niezawodność sieci znacząco spada.