Sprawozdanie lista 1

Paweł Krzyszczak

October 2024

1 Zadanie 1

W ramach niniejszego zadania zaimplementowano algorytmy przeszukiwania grafów: wgłąb (DFS) oraz wszerz (BFS). Program obsługuje zarówno grafy skierowane, jak i nieskierowane, zwracając kolejność odwiedzania wierzchołków oraz drzewo przeszukiwania na życzenie.

Algorytm przeszukiwania wgłąb (DFS) działa na zasadzie eksploracji jak najgłębiej w obrębie grafu, zanim wróci do wierzchołków z poprzedniego poziomu. Implementacja algorytmu DFS może być opisana w następujących krokach:

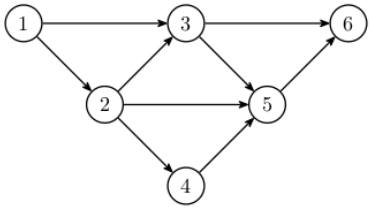
1. Rozpocznij od wierzchołka startowego.
2. Oznacz wierzchołek jako odwiedzony.
3. Dla każdego nieodwiedzonego sąsiada wierzchołka wykonaj rekurencyjnie DFS.

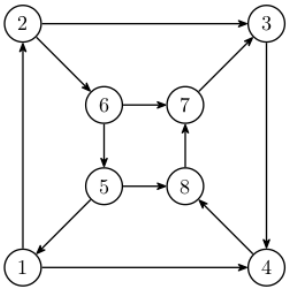
Algorytm przeszukiwania wszerz (BFS) eksploruje wszystkie sąsiednie wierzchołki przed przejściem do wierzchołków na niższym poziomie. Proces ten może być opisany w kilku krokach:

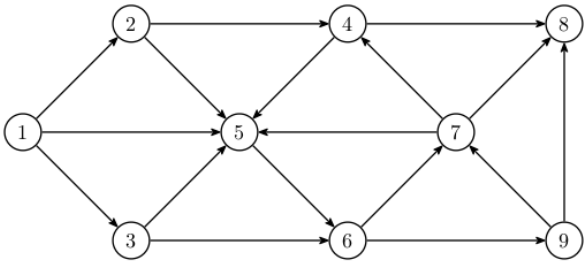
1. Rozpocznij od wierzchołka startowego i umieść go w kolejce.
2. Oznacz wierzchołek jako odwiedzony.
3. Dopóki kolejka nie jest pusta, wykonuj następujące kroki:
    - Usuń wierzchołek z kolejki.
    - Dla każdego nieodwiedzonego sąsiada dodaj go do kolejki i oznacz jako odwiedzonego.

W celu przetestowania algorytmów wykorzystano następujące grafy:

- Graf skierowany (rysunek 1).
- Graf nieskierowany (rysunek 1).
- Własne przykłady grafów z 10 do 100 wierzchołkami.

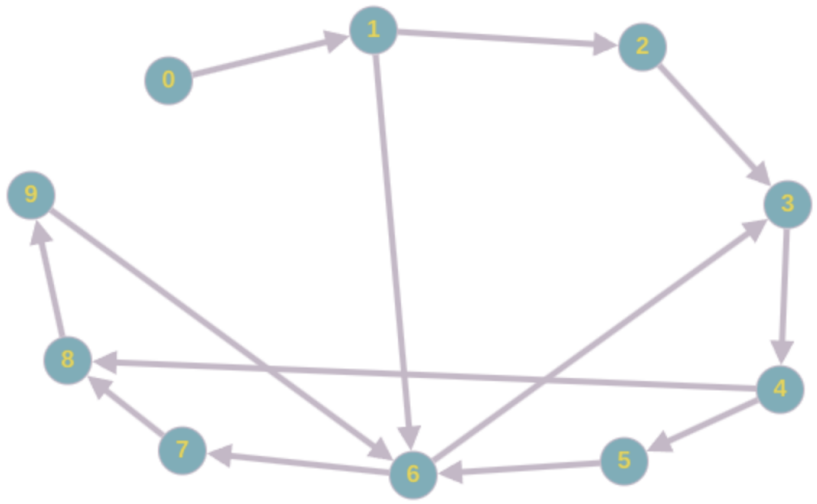







```
>java Main < g1_BFS.txt
BFS Order: [1, 2, 3, 4, 5, 6]
BFS Tree:
+- 1
|  +- 2
|  |  +- 4
|     +- 5
   +- 3
      +- 6
Czas wykonania: 122627489 nanosekund
>java Main < g1_DFS.txt
DFS Order: [1, 2, 3, 5, 6, 4]
DFS Tree:
+- 1
   +- 2
   |  +- 3
   |     +- 5
   |        +- 6
      +- 4
Czas wykonania: 26953194 nanosekund

>java Main < g2_BFS.txt
BFS Order: [1, 2, 3, 4, 5, 6]
```

```
BFS Tree:
+- 1
|  +- 2
|  |  +- 4
|     +- 5
   +- 3
      +- 6
Czas wykonania: 23458737 nanosekund
>java Main < g2_DFS.txt
DFS Order: [1, 2, 3, 5, 4, 6]
DFS Tree:
+- 1
   +- 2
      +- 3
         +- 5
         |  +- 4
            +- 6
Czas wykonania: 130484552 nanosekund
```




```
>java Main < g3_BFS.txt
BFS Order: [1, 2, 6, 3, 7, 4, 8, 5, 9]
BFS Tree:
+- 1
|  +- 2
```

```
|     +- 3
|        +- 4
|           +- 5
   +- 6
      +- 7
         +- 8
            +- 9
Czas wykonania: 23581370 nanosekund
>java Main < g3_DFS.txt
DFS Order: [1, 2, 3, 4, 5, 6, 7, 8, 9]
DFS Tree:
+- 1
   +- 2
      +- 3
         +- 4
            +- 5
               +- 6
                  +- 7
                     +- 8
                        +- 9
Czas wykonania: 23584881 nanosekund
```

2 Zadanie 2

Celem niniejszego zadania było zaimplementowanie algorytmu sortowania topologicznego dla grafów skierowanych oraz wykrywanie istnienia cykli skierowanych. Program powinien informować, czy dany graf zawiera cykl, a jeśli nie, wypisywać wierzchołki w kolejności topologicznej, jeśli liczba wierzchołków wynosi do 200.

Sortowanie topologiczne można zrealizować za pomocą algorytmu DFS (przeszukiwanie wgłąb) lub metody Kahn'a. W przypadku wykrywania cyklu w grafie, użyjemy algorytmu DFS, który będzie oznaczał wierzchołki jako odwiedzone i śledził ich stan.

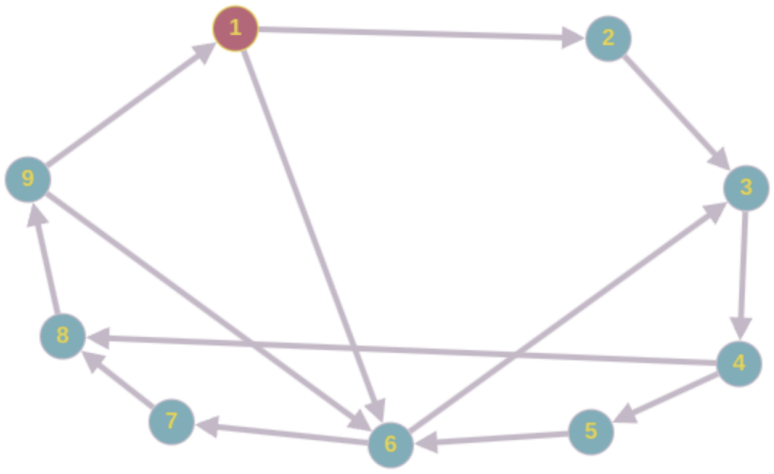
W celu przetestowania algorytmu wykorzystano następujące dane:

- Własne przykłady grafów acyklicznych i z cyklem, z 10 do 100 wierzchołkami.
- Grafy z folderu 2 z paczki "aod testy1.zip" (12 plików).

```
>java Main < g2a-1.txt
Graf jest acykliczny.
Kolejność topologiczna:
1 5 9 13 2 6 10 14 3 7 11 15 4 8 12 16
Czas wykonania: 21206013 nanosekund
```

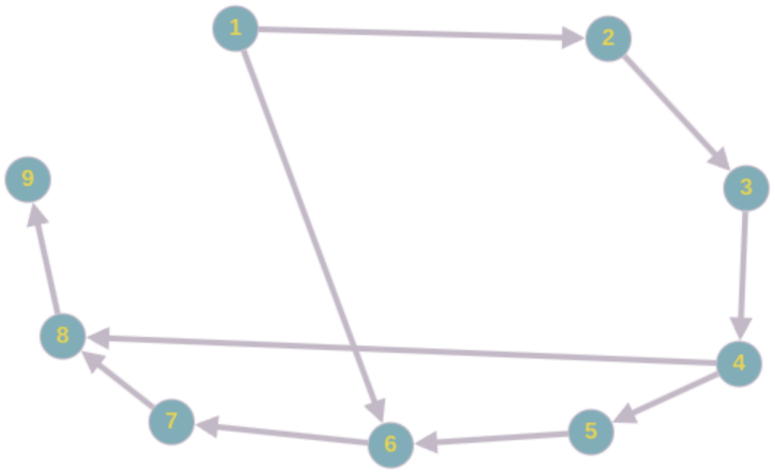
```
>java Main < g2a-2.txt
Graf jest acykliczny.
Kolejność topologiczna:
1 11 21 31 41 51 61 71 81 91 2 12 22 32 42 52 62 72 82 92 3 13 23 33 43 53 63 73 83 93 4 14
Czas wykonania: 25544273 nanosekund

>java Main < g2a-3.txt
Graf jest acykliczny.
Czas wykonania: 39120124 nanosekund

>java Main < g2a-4.txt
Graf jest acykliczny.
Czas wykonania: 76709902 nanosekund

>java Main < g2a-5.txt
Graf jest acykliczny.
Czas wykonania: 465039040 nanosekund

>java Main < g2a-6.txt
Graf jest acykliczny.
Czas wykonania: 2252711534 nanosekund

>java Main < g2b-1.txt
Graf zawiera skierowany cykl.
Czas wykonania: 127854822 nanosekund

>java Main < g2b-2.txt
Graf zawiera skierowany cykl.
Czas wykonania: 26773016 nanosekund

>java Main < g2b-3.txt
Graf zawiera skierowany cykl.
Czas wykonania: 42692830 nanosekund

>java Main < g2b-4.txt
Graf zawiera skierowany cykl.
Czas wykonania: 79188208 nanosekund

>java Main < g2b-5.txt
Graf zawiera skierowany cykl.
Czas wykonania: 521590117 nanosekund

>java Main < g2b-6.txt
Graf zawiera skierowany cykl.
Czas wykonania: 2243380517 nanosekund
```

Acykliczny




```
>java Main < gm1a-1.txt
Graf jest acykliczny.
Kolejność topologiczna:
1 2 3 4 5 6 7 8 9
Czas wykonania: 19869479 nanosekund
```

Cykliczny




```
>java Main < gm1b-1.txt
Graf zawiera skierowany cykl.
Czas wykonania: 18759544 nanosekund
```

3 Zadanie 3

Celem zadania było zaimplementowanie algorytmu, który dla podanego na wejściu grafu skierowanego $G = (V, E)$ zwróci jego rozkład na silnie spójne składowe. Silnie spójna składowa grafu to maksymalny podgraf, w którym istnieje ścieżka między każdymi dwoma wierzchołkami.

W zadaniu zaimplementowano algorytm oparty na metodzie DFS (Depth-First Search) oraz algorytmie Kosaraju. Algorytm działa w czasie $O(|V| + |E|)$, co czyni go efektywnym w przypadku dużych grafów.

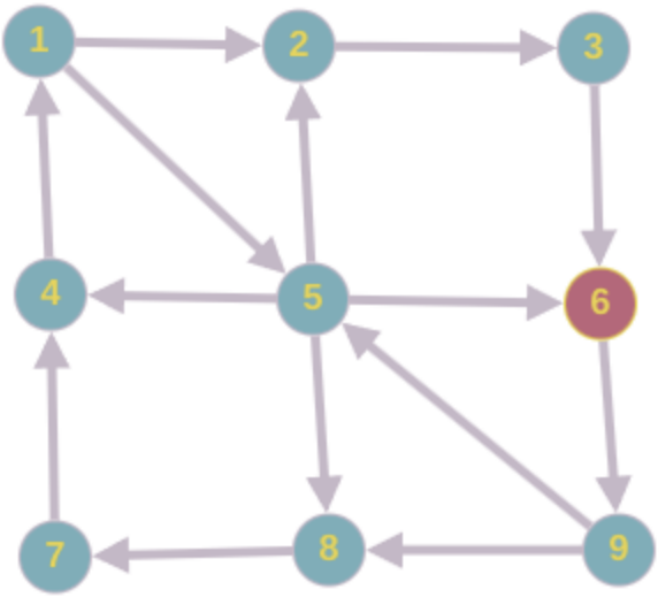
3.1 Kroki algorytmu

1. Wykonaj DFS na oryginalnym grafie G, aby obliczyć porządek odwiedzin wierzchołków.
2. Odwróć krawędzie w grafie G.
3. Wykonaj DFS na odwróconym grafie, wykorzystując porządek odwiedzin z kroku 1, aby zidentyfikować silnie spójne składowe.

W celu przetestowania algorytmu wykorzystano następujące dane:

- Własny przykład grafu silnie spójnego.
- Grafy z folderu 3 z paczki `aod_testy1.zip` (6 plików).

```
>java Main3 g3-1.txt
Number of Strongly Connected Components: 5
Sizes of Strongly Connected Components: [5, 4, 4, 2, 1]
Strongly Connected Components:
[1, 5, 4, 3, 2]
[12, 15, 14, 13]
[6, 8, 9, 7]
[10, 11]
[16]
Duration: 550710 ns

>java Main3 g3-1.txt
Number of Strongly Connected Components: 5
Sizes of Strongly Connected Components: [5, 4, 4, 2, 1]
Strongly Connected Components:
[1, 5, 4, 3, 2]
[12, 15, 14, 13]
[6, 8, 9, 7]
[10, 11]
[16]
Duration: 550710 ns

>java Main3 g3-3.txt
Number of Strongly Connected Components: 5
Sizes of Strongly Connected Components: [7, 400, 400, 200, 1]
Duration: 2384359 ns

>java Main3 g3-4.txt
Number of Strongly Connected Components: 5
Sizes of Strongly Connected Components: [8, 4000, 3600, 2400, 1]
Duration: 15146673 ns

>java Main3 g3-5.txt
Number of Strongly Connected Components: 5
Sizes of Strongly Connected Components: [9, 40000, 40000, 20000, 1]
Duration: 87055377 ns

java Main3 g3-6.txt
Number of Strongly Connected Components: 5
Sizes of Strongly Connected Components: [10, 400000, 360000, 240000, 1]
Duration: 411149336 ns
```




```
>java Main3 gm3-1.txt
Number of Strongly Connected Components: 1
Sizes of Strongly Connected Components: [9]
Strongly Connected Components:
[1, 4, 7, 8, 9, 6, 5, 3, 2]
Duration: 492888 ns
```

4 Zadanie 4

Celem zadania było zaimplementowanie algorytmu, który dla podanego grafu $G = (V, E)$, gdzie V to zbiór wierzchołków, a E to zbiór krawędzi, zwraca informację o tym, czy graf jest dwudzielny. Dodatkowo, w przypadku grafu dwudzielnego, algorytm powinien dla $n < 200$ wypisać rozbicie zbioru V na dwa podzbiory V_0 i V_1.

Zaimplementowany algorytm wykorzystuje metodę przeszukiwania w szerz (BFS) do sprawdzenia dwudzielności grafu. Klasa 'BipartiteChecker' zawiera metody do budowy listy sąsiedztwa, sprawdzania dwudzielności oraz wypisywania podzbiorów.

Przeprowadzono testy na własnych przykładach grafów dwudzielnych oraz niedwudzielnych (skierowanych i nieskierowanych) z 10 do 100 wierzchołków.

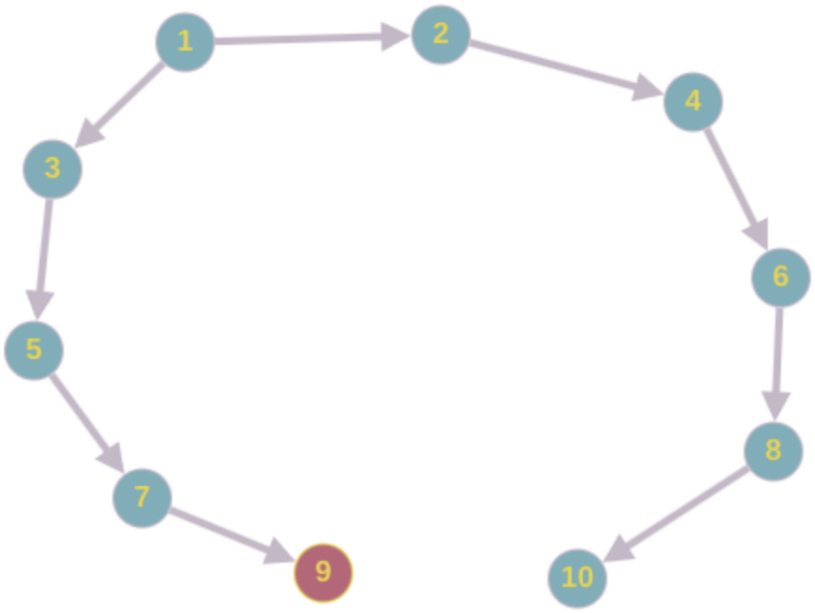
Dodatkowo wykorzystano grafy z folderu 4 z paczki `aod_testy1.zip`, składającej się z 24 plików.

```
>java Main4 d4a-1.txt
The graph is bipartite.
[1, 3, 6, 8, 9, 11, 14, 16]
[2, 4, 5, 7, 10, 12, 13, 15]
Duration: 252779 ns

>java Main4 d4a-2.txt
The graph is bipartite.
[1, 3, 5, 7, 9, 12, 14, 16, 18, 20, 21, 23, 25, 27, 29, 32, 34, 36, 38, 40,
41, 43, 45, 47, 49, 52, 54, 56, 58, 60, 61, 63, 65, 67, 69, 72, 74, 76, 78,
80, 81, 83, 85, 87, 89, 92, 94, 96, 98, 100]
[2, 4, 6, 8, 10, 11, 13, 15, 17, 19, 22, 24, 26, 28, 30, 31, 33, 35, 37, 39,
42, 44, 46, 48, 50, 51, 53, 55, 57, 59, 62, 64, 66, 68, 70, 71, 73, 75, 77,
79, 82, 84, 86, 88, 90, 91, 93, 95, 97, 99]
Duration: 344214 ns

>java Main4 d4a-3.txt
The graph is bipartite.
Duration: 1190674 ns

>java Main4 d4a-4.txt
The graph is bipartite.
Duration: 5926964 ns

>java Main4 d4a-5.txt
The graph is bipartite.
Duration: 30383830 ns

>java Main4 d4a-6.txt
The graph is bipartite.
Duration: 129412352 ns

>java Main4 d4b-1.txt
The graph is not bipartite.
Duration: 249571 ns

>java Main4 d4b-2.txt
The graph is not bipartite.
Duration: 279662 ns

>java Main4 d4b-3.txt
The graph is not bipartite.
Duration: 820281 ns
```

```
>java Main4 d4b-4.txt
The graph is not bipartite.
Duration: 3093981 ns

>java Main4 d4b-5.txt
The graph is not bipartite.
Duration: 18839450 ns

>java Main4 d4b-6.txt
The graph is not bipartite.
Duration: 87310428 ns

>java Main4 u4a-1.txt
The graph is bipartite.
[1, 4, 5, 6, 7]
[2, 3, 8, 9, 10, 11, 12, 13, 14, 15]
Duration: 264305 ns

>java Main4 u4a-2.txt
The graph is bipartite.
[1, 4, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
31, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115,
116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127]
[2, 3, 8, 9, 10, 11, 12, 13, 14, 15, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63]
Duration: 398568 ns

>java Main4 u4a-3.txt
The graph is bipartite.
Duration: 854941 ns

>java Main4 u4a-4.txt
The graph is bipartite.
Duration: 7887710 ns

>java Main4 u4a-5.txt
The graph is bipartite.
Duration: 16759725 ns

>java Main4 u4a-6.txt
The graph is bipartite.
Duration: 60936731 ns
```

```
>java Main4 u4b-1.txt
The graph is not bipartite.
Duration: 992113 ns

>java Main4 u4b-2.txt
The graph is not bipartite.
Duration: 347077 ns

>java Main4 u4b-3.txt
The graph is not bipartite.
Duration: 601548 ns

>java Main4 u4b-4.txt
The graph is not bipartite.
Duration: 3783088 ns

>java Main4 u4b-5.txt
The graph is not bipartite.
Duration: 12028569 ns

>java Main4 u4b-6.txt
The graph is not bipartite.
Duration: 92950090 ns
```

Graf dwudzielny nieskierowany

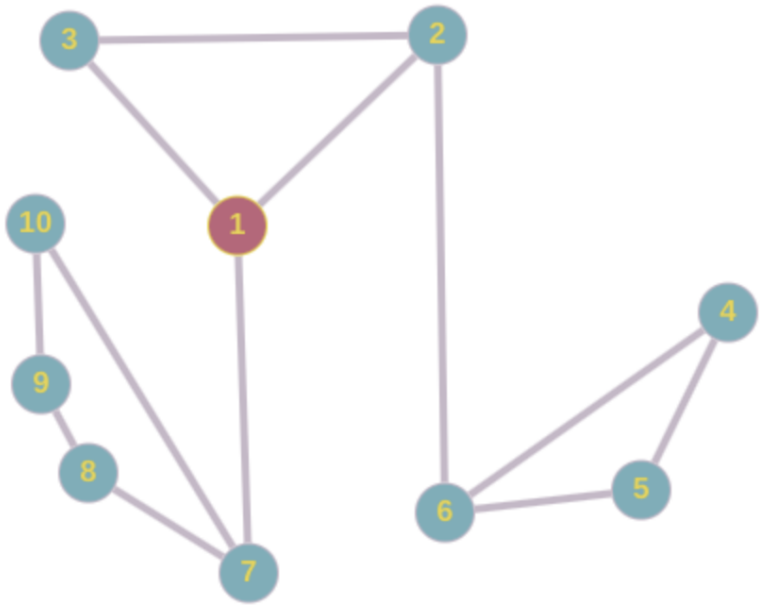



```
>java Main4 gm4-1.txt
The graph is bipartite.
[1, 4, 5, 8, 9]
[2, 3, 6, 7, 10]
Duration: 234260 ns
```

Graf dwudzielny nieskierowany

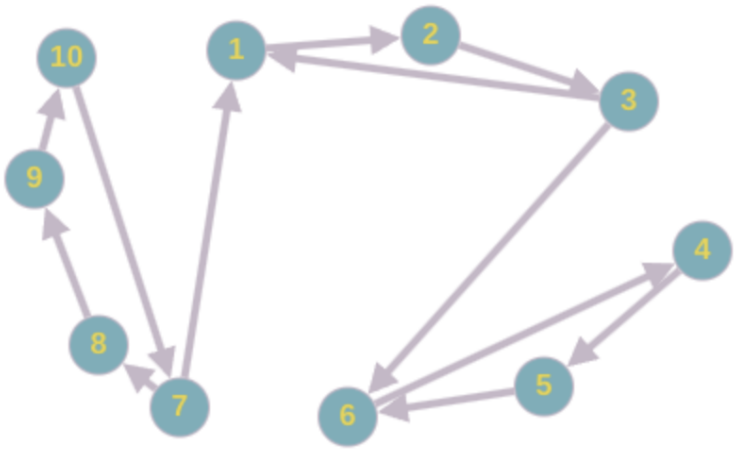

```
>java Main4 gm4-2.txt
The graph is bipartite.
[1, 2, 3, 4, 5, 9, 10]
[6, 7, 8]
Duration: 256842 ns
```

Graf niedwudzielny nieskierowany




```
>java Main4 gm4-3.txt
The graph is not bipartite.
Duration: 243897 ns
```

Graf niedwudzielny skierowany




```
>java Main4 gm4-4.txt
The graph is not bipartite.
Duration: 236624 ns
```