Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych

Grafy i Sieci

Wyznaczanie klik w grafie skierowanym

Sprawozdanie 3

Autorzy:
Maciej Suchecki
Jacek Witkowski

Prowadzący:
doc. dr inż. Dariusz Bursztynowski

14 stycznia 2015

1 Część 1

1.1 Treść zadania

Tytuł Wyznaczanie klik w grafie skierowanym

Opis Wyznacz wszystkie kliki grafu skierowanego G. Wyjaśnienie: Kliką w grafie nieskierowanym G nazywamy każdy największy pełny podgraf G (w grafie pełnym każda para węzłów jest połączona krawędzią). Dla potrzeb projektu kliką grafu skierowanego G nazwiemy każdy największy pełny podgraf grafu G' powstałego z G przez zastąpienie par krawędzi skierowanych (i→j), (j→i) krawędziami nieskierowanymi (i,j) i usunięcie krawędzi pozostałych. Problem znajdowania klik występuje m.in. w optymalizacji szeregowania transmisji w radiowych sieciach wieloskokowych (ang. multihop).

1.2 Implementacja

Język programowania Wybranym językiem programowania jest Java.

Założenia Zakładamy, że graf wejściowy będzie skierowanym grafem spójnym.

Dane wejściowe Dane wejściowe grafu będą w postaci listy krawędzi. Każda krawędź będzie reprezentowana w nowej linii jako para uporządkowana (A, B), przy czym A będzie wierzchołkiem początkowym, a B wierzchołkiem docelowym.

Dane wyjściowe Wynikiem działania programu będzie lista klik, przy czym każda klika będzie reprezentowana jako osobna linia składająca się z wierzchołków oddzielonych spacjami.

2 Część 2

2.1 Algorytm

2.1.1 Wybór

Do rozwiązania problemu wybrany został algorytm rekurencyjny Brona-Kerboscha w wersji z piwotingiem oraz sortowaniem wierzchołków. Wybór jest uzasadniony wydajnością rozwiązania. Wersja z piwotingiem charakteryzuje się zmniejszoną liczbą wywołań rekurencyjnych w stosunku do wersji podstawowej algorytmu. Dodatkowo zastosowane zostało sortowanie wierzchołków, aby uzyskać lepszą klasę złożoności algorytmu.

2.1.2 Pseudokod

```
BronKerbosch(G):
  P = V(G)
  R = X = empty
  sort vertices in G
  for each vertex v in G:
    BronKerboschRecursive(R ∪ {v}, P ∩ N(v), X ∩ N(v))
    P := P \ {v}
    X := X ∪ {v}

BronKerboschRecursive(R, P, X):
  if P is empty and X is empty:
    report R as a maximal clique
  choose a pivot vertex u in P ∪ X
  for each vertex v in P \ N(u):
    BronKerbosch2(R ∪ {v}, P ∩ N(v), X ∩ N(v))
    P := P \ {v}
    X := X ∪ {v}
```

Listing 1: Pseudokod algorytmu Brona-Kerboscha

2.1.3 Opis

Algorytm otrzymuje na wejściu graf G. Na początku tworzone są trzy zbiory P, R i X, przy czym: P to zbiór wierzchołków, które są kandydatami do rozważenia, R to zbiór wierzchołków będących częściowym wynikiem szukania kliki, a X to zbiór wierzchołków pominiętych. Przy pierwszym wywołaniu algorytmu zbiory R i X są puste, a zbiór P zawiera wszystkie wierzchołki grafu. Po zainicjalizowaniu zmiennych, wierzchołki grafu G są sortowane względem degeneracji, aby przyśpieszyć działanie algorytmu. Następnie dla każdego wierzchołka grafu wywoływana jest procedura rekurencyjna algorytmu, której zasada działania jest następująca:

1. Najpierw sprawdzamy, czy zbiory P i X są puste. Jeśli tak, to zbiór R zawiera maksymalną klikę. Wypisujemy zawartość zbioru R i kończymy.
2. Następnie wybieramy ze zbioru $P \cup X$ wierzchołek u, który posłuży nam za *pivot*.
3. Ze zbioru P odejmujemy sąsiadów wierzchołka u, tworząc nowy zbiór do badania w pętli.
4. Jeśli nowopowstały zbiór nie jest pusty, to wybieramy z niego kolejne wierzchołki v i dla każdego z nich:
   (a) Wywołujemy rekurencyjnie algorytm ze zbiorami R ∪ v, P ∩ N(v), X ∩ N(v).
   (b) Ze zbioru P usuwamy wierzchołek v, dodając go jednocześnie do zbioru R.

Opis zmiennych

- G – badany graf
- $V(G)$ – zbiór wierzchołków badanego grafu
- P – zbiór wierzchołków, które są kandydatami do rozważenia
- R – zbiór wierzchołków będących częściowym wynikiem szukania kliki
- X — zbiór wierzchołków pominiętych
- v, u — zmienne pomocnicze przechowujące wierzchołki
- $N(v)$ – zbiór sąsiadów wierzchołka v

2.2 Opis struktur danych

W programie graf jest reprezentowany jako zbiór krawędzi oraz zbiór węzłów (wierzchołków). Każda krawędź jest powiązana z dwoma węzłami. Każdy węzeł zawiera informację o etykiecie (unikalnej dla każdego węzła) oraz o swoich sąsiadach.

2.3 Projekt testów

Testy będą się odbywać w dwóch etapach. Na początku zostanie przeprowadzona ręczna weryfikacja poprawności działania programu dla kilku prostych grafów wejściowych.

Ponadto, na potrzeby testów zakładamy napisanie skryptu w języku Python, który będzie generował losowe grafy spełniające warunki zadania, uruchamiał program zbierając wyniki jego działania, a następnie wyświetlał wykresy obrazujące uzyskane wyniki. W ramach testów automatycznych zamierzamy otrzymać wykresy zależności czasu wykonania programu od liczby wierzchołków grafu oraz od jego gęstości.

Testy zakładają uśrednianie wyników dla kilku grafów o tych samych właściwościach (odpowiednio liczba wierzchołków oraz gęstość) dla zapewnienia wiarygodności wyników.

2.4 Założenia programu

2.4.1 Złożoność obliczeniowa

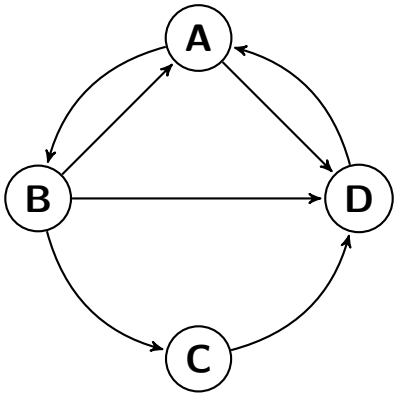
Pesymistyczna złożoność algorytmu Brona-Kerboscha (z piwotingiem, który minimalizuje liczbę rekurencyjnych wywołań wykonanych w każdym kroku) wynosi $O(3^{n/3})$.

Natomiast wersja algorytmu z sortowaniem wierzchołków – wykorzystana w programie – osiąga złożoność $O(dn3^{d/3})$, gdzie d oznacza degenerację grafu, miarę jego rozproszenia. Zatem złożoność programu będzie liniowa względem liczby wierzchołków.

2.4.2 Wejście

Opis Dane wejściowe do programu będą przekazywane w postaci listy krawędzi grafu. Każda krawędź będzie reprezentowana w nowej linii jako para uporządkowana (A, B), przy czym A będzie wierzchołkiem początkowym, a B wierzchołkiem docelowym.

Przykład Dla grafu zdefiniowanego poniżej:



Następujący plik wejściowy będzie uznany za poprawny:

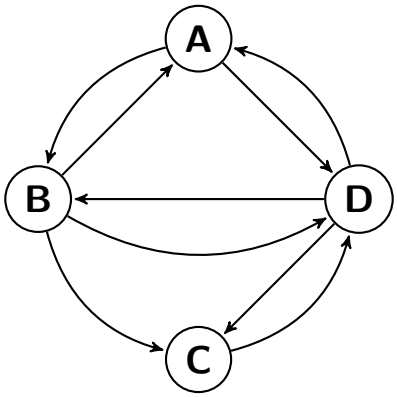
```
A B
A D
B A
B C
B D
C D
D A
```

Listing 2: Przykładowy plik wejściowy

2.4.3 Wyjście

Opis Program będzie wypisywał w kolejnych liniach listy wierzchołków stanowiące znalezione maksymalne kliki.

Przykład Dla grafu zdefiniowanego poniżej:



Program wygeneruje następujące wyjście:

```
A B D
D C
```

Listing 3: Wynik działania programu

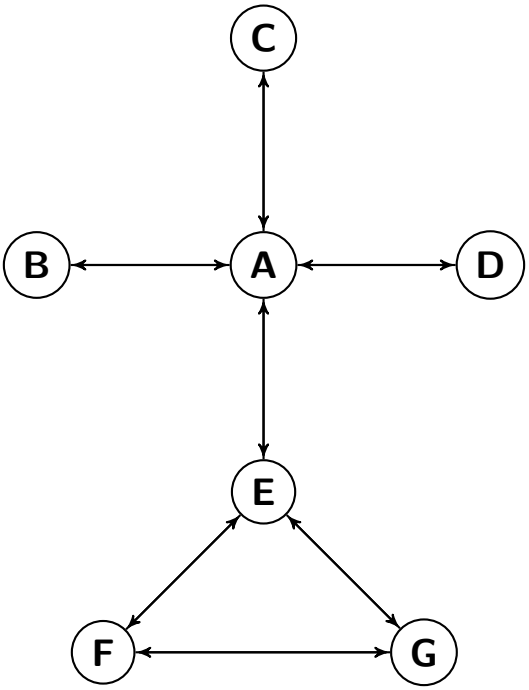
3 Część 3

W ramach tej części projektu zostały stworzone testy algorytmu, sprawdzające zarówno jego poprawność, jak i wydajność w zależności od różnych czynników. Testy poprawności zostały wykonane w języku Java, z użyciem bibliotek: *JUnit* oraz *Mockito*. Z kolei testy wydajnościowe zostały napisane w języku skryptowym Python, z wykorzystaniem bibliotek takich, jak: *matplotlib* oraz *numpy*. Generowanie grafów testowych zostało wykonane własnoręcznie, również w języku Python. Testy te miały na celu wyznaczenie wydajności algorytmu w zależności od trzech czynników:

- rozmiaru grafu,
- gęstości grafu,
- degeneracji grafu.

3.1 Testy poprawności

W celu sprawdzenia poprawności działania algorytmu wykonano dwa testy. Pierwszy z nich miał na celu sprawdzić czy w zadanym grafie:



Rysunek 1: Graf utworzony na potrzeby testów poprawności

zostaną odnalezione odpowiednie kliki: (E, F, G), (A, B), (A, C), (A, D) oraz (A, E).

Drugi z testów miał za zadanie sprawdzić, czy algorytm sortowania węzłów względem ich degeneracji działa zgodnie z oczekiwaniami. W tym celu użyto ponownie grafu znajdującego się na Rysunku 1 i spodziewano się uzyskać listę wierzchołków taką, że na początku tej listy znajdą się wierzchołki o etykietach B, C oraz D (w dowolnej kolejności), po nich wystąpi wierzchołek A, natomiast po nim wystąpią wierzchołki o etykietach E, F oraz G.

Oba testy zakończyły się powodzeniem.

3.2 Testy wydajnościowe

W ramach testów wydajnościowych zdecydowaliśmy się wykonać wykresy czasu działania algorytmu w zależności od różnych czynników. W tym celu używane są następujące funkcje napisane w języku Python:

```
# runs the app with desired input graph
def runSolver(graphFilename):
  command = ["java", "-jar", "./GIS.jar", "-i", str(graphFilename)]
  result = subprocess.check_output(command)
  return result

# runs solver and records run time
def runSolverAndMeasureTime(graphFilename):
  start = timeit.default_timer()
  runSolver(graphFilename)
  stop = timeit.default_timer()
  time = stop - start
  return time

# runs solver multiple times with different graphs and collects average time
def collectAverageTime(graphFilenames):
  averageTime = 0
  for graph in graphFilenames:
    time = runSolverAndMeasureTime(graph)
    averageTime += time
  averageTime /= repetitions
  return averageTime

# plots a simple graph
def plotGraph(data, xlabel, ylabel):
  plot.plot(list(data.keys()), list(data.values()))
  plot.ylabel(ylabel)
  plot.xlabel(xlabel)
  plot.show()
```

Listing 4: Funkcje pomocnicze dla testów wydajnoćiowych

Służą one odpowiednio do: uruchomienia algorytmu z zadanym grafem, zmierzenia czasu jego wykonywania, uśredniania otrzymanych wyników oraz do rysowania wykresu na podstawie uzyskanych danych. Test dla każdego czynnika (rozmiaru, gęstości oraz degeneracji grafu) zakładają wykonanie następujących kroków:

1. wyczyszczenie pamięci przechowującej rezultaty,
2. dla każdej testowanej wartości parametru:
   (a) wygenerowanie 20 grafów o zadanych parametrach,
   (b) uruchomienie algorytmu dla każdego z wygenerowanych grafów,
   (c) obliczenie średniej z uzyskanych czasów wykonania,
   (d) zapisanie średniej wraz z testowaną wartością do pamięci.
3. wyświetlenie wykresu.

3.2.1 Generowanie grafów

W celu wygenerowania grafów testowych został napisany mały moduł w języku Python, którego fragment zaprezentowany jest poniżej:

```
# generates random directed graph with desired number of vertices and density
def generateAndSaveGraphWithDesiredDensity(file, verticesCount, density):
  graphFile = open(file, "w")
  vertices = list(range(verticesCount))

  # calculate desired number of edges from desired graph density
  maxPossibleEdgeCount = verticesCount * (verticesCount - 1)
  edgeCount = int(density * maxPossibleEdgeCount)

  # generate random degrees in range [0, verticesCount-1] that sums to edgeCount
  degrees = generateDegrees(verticesCount, edgeCount, 0, verticesCount - 1)

  # generate dictionary containing vertex/degree pairs, then iterate over it,
  # drawing <degree> vertices from all possible ones and save edges to file
  for vertice, degree in dict(zip(vertices, degrees)).items():
    possibleSucc = list(vertices)
    possibleSucc.remove(vertice)     # avoid loops
    for successor in random.sample(possibleSucc, degree):
      graphFile.write(str(vertice) + " " + str(successor) + "\n")

  graphFile.close()

# generates random directed graph with desired number of vertices and degeneracy
def generateAndSaveGraphWithDesiredDegeneracy(file, verticesCount, degeneracy):
  graphFile = open(file, "w")
  vertices = list(range(verticesCount))
  possibleSucc = vertices

  # generate random degree in range [0, degeneracy] for every vertex to satisfy
  # degeneracy criterion (there should be at least 1 vertex with degree = deg.)
  degrees = [degeneracy]
  for _ in range(verticesCount - 1):
    degrees.append(random.randint(0, degeneracy))
  degrees.sort(reverse=True)

  # for every vertice, generate random number of its successors (less than
  # degeneracy), then take that number of vertices from successors set and save
  # resulting pairs to file with desired filename as two-way edges;
  # from the rest, draw a random number to generate one-way edges
  for vertice, degree in dict(zip(vertices, degrees)).items():
    possibleSucc.remove(vertice)     # prevent loops and duplicate edges
    twoWaySucc = random.sample(possibleSucc, min(degree, len(possibleSucc)))
    oneWaySucc = [item for item in possibleSucc if item not in twoWaySucc]

    # generate two way edges
    for successor in twoWaySucc:
      graphFile.write(str(vertice) + " " + str(successor) + "\n")
      graphFile.write(str(successor) + " " + str(vertice) + "\n")

    # generate couple random one-way edges
    for successor in random.sample(oneWaySucc, random.randint(0, len(oneWaySucc))):
      graphFile.write(str(vertice) + " " + str(successor) + "\n")

  graphFile.close()
```

Listing 5: Funkcje generujące grafy dla testów wydajnościowych

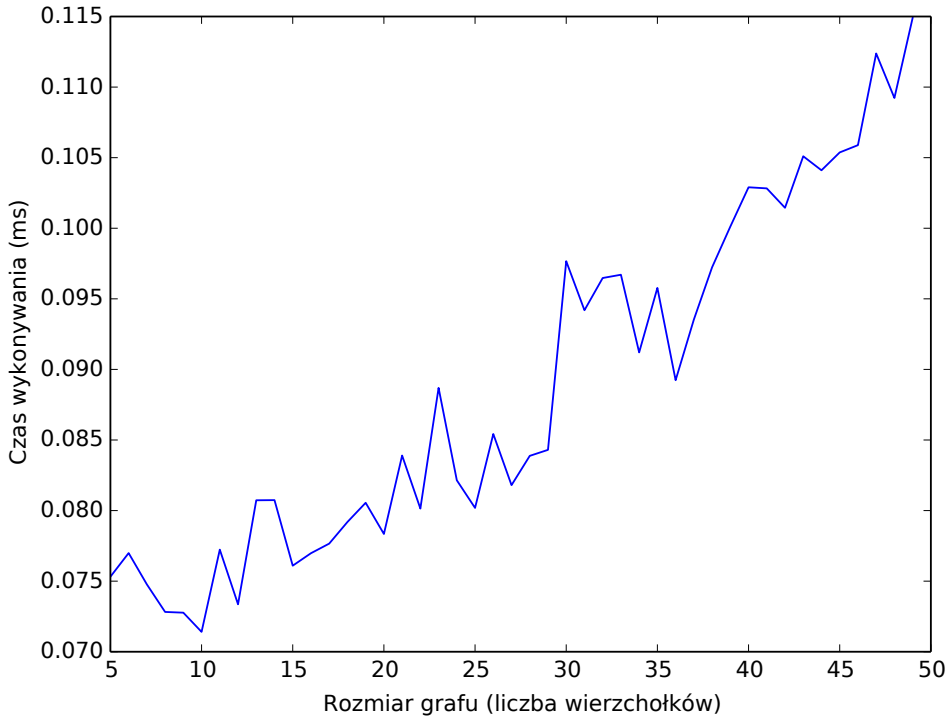
3.2.2 Wydajność w zależności od rozmiaru grafu

Dla wykonania testów wydajności w funkcji rozmiaru grafu, uruchomiono następujący kod:

```
results = {}
for size in sizes:
  graphFilenames = graph.generateRandomGraphs(count, size)
  results[size] = collectAverageTime(graphFilenames)
plotGraph(results, "Rozmiar grafu (liczba wierzcholkow)", "Czas wykonywania (ms)")
```

Listing 6: Testy wydajności w zależności od rozmiaru grafu

W wyniku jego działania uzyskano następujący wykres:



Rysunek 2: Wpływ rozmiaru grafu na wydajność algorytmu.

Po przeanalizowaniu wykresu widoczny jest oczywiście wyraźny wzrost czasu wykonywania algorytmu podczas wzrostu rozmiaru grafu. Po dokładnym przyjrzeniu się kształtowi otrzymanego wykresu, można zaryzykować stwierdzenie, że złożoność algorytmu jest liniowa w zależności od rozmiaru grafu.

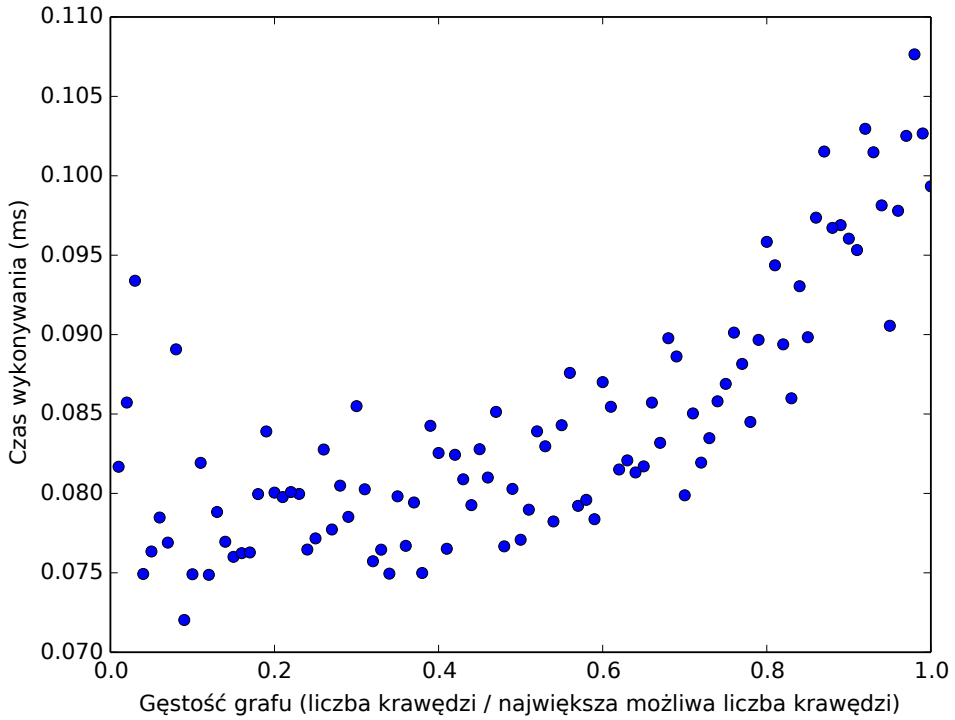
3.2.3 Wydajność w zależności od gęstości grafu

Dla wykonania testów wydajności w funkcji gęstości grafu, uruchomiono następujący kod:

```
results = {}
for density in densities:
  graphFilenames = graph.generateRandomGraphsWithDensity(count, size, density)
  results[size] = collectAverageTime(graphFilenames)
plotGraph(results,
  "Gestosc grafu (liczba krawedzi / najwieksza mozliwa liczba krawedzi)",
  "Czas wykonywania (ms)")
```

Listing 7: Testy wydajności w zależności od gęstości grafu

W wyniku jego działania uzyskano następujący wykres:



Rysunek 3: Wpływ gęstości grafu na wydajność algorytmu.

Na wykresie nie zauważyliśmy wyraźnej i jednocześnie jednoznacznej zależności wydajności algorytmu w funkcji gęstości testowanego grafu. Można oczywiście zauważyć wyraźny wzrost czasu wykonywania algorytmu podczas wzrostu gęstości grafu, jednakże trudno tu o jednoznaczne stwierdzenie złożoności wykładniczej z racji na dość duży rozrzut uzyskiwanych wyników (pomimo ich uśredniania).

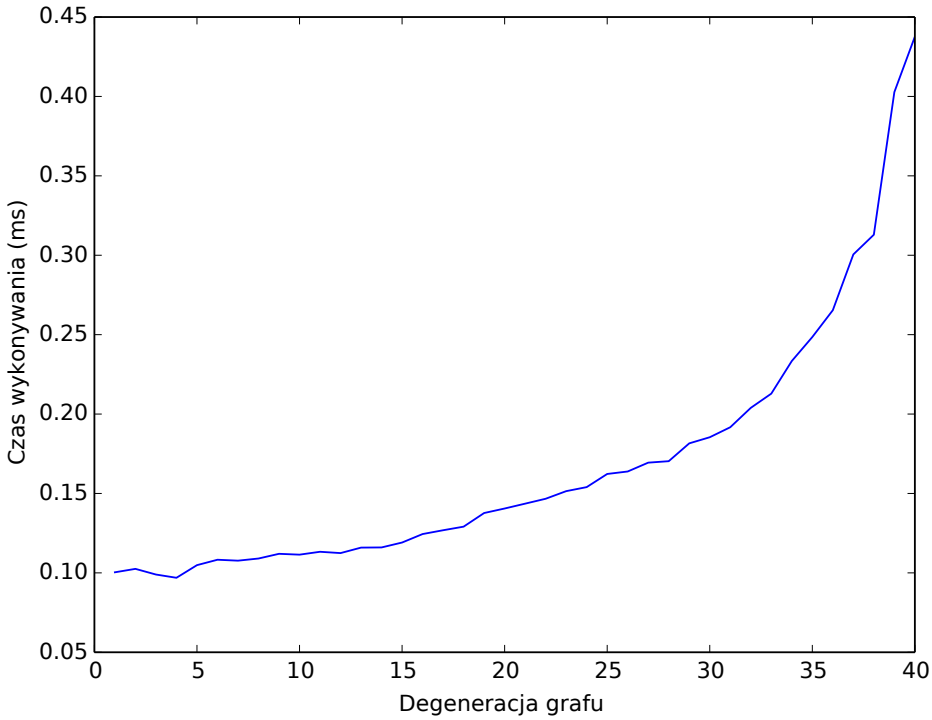
3.2.4 Wydajność w zależności od degeneracji grafu

Z powodu niezauważenia przez nas wyraźnej i jednoznacznej zależności wydajności algorytmu od gęstości grafu, zdecydowaliśmy się na wykonanie wykresu w zależności od degeneracji grafu. Dla wykonania testów wydajności w funkcji rozmiaru grafu, uruchomiono następujący kod:

```
results = {}
for degeneracy in degeneracies:
  graphFilenames =
    graph.generateRandomGraphsWithDegeneracy(count, size, degeneracy)
  results[degeneracy] = collectAverageTime(graphFilenames)
plotGraph(results, "Degeneracja grafu", "Czas wykonywania (ms)")
```

Listing 8: Testy wydajności w zależności od degeneracji grafu

W wyniku jego działania uzyskano następujący wykres:



Rysunek 4: Wpływ degeneracji grafu na wydajność algorytmu.

Jak widać, wyniki są zgodne z założeniami teoretycznymi nt. algorytmu – złożoność algorytmu w zależności od degeneracji grafu jest wykładnicza.