Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Dariusz Zdybski

Nr albumu: 1077384

Badanie problemu kliki z językiem Python

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2016

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

Składam serdeczne podziękowania dla promotora tej pracy Pana dra hab. Andrzeja Kapanowskiego za wszelką pomoc, cierpliwość, zaangażowanie, poświęcony czas, oraz istotne uwagi merytoryczne, bez których niniejsza praca nie mogłaby powstać.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów dla problemu kliki. Powstało szereg narzędzi pomocniczych potrzebnych do wydajnego działania algorytmów wyszukujących kliki. Zaimplementowano generatory zbioru potęgowego, wszystkich klik, wszystkich klik z k wierzchołkami, wszystkich trójkątów w grafie. Są dostępne dwa sposoby wyznaczania współczynnika klastrowania sieci. Ulepszono cztery algorytmy wyznaczania maksymalnego zbioru niezależnego (można ustalać wierzchołek startowy).

Zaimplementowano algorytmy wyznaczania kliki maksymalnej lub największej: prosty algorytm sprawdzający wszystkie kliki, algorytm rosnącej kliki, algorytm zbiorów niezależnych, algorytm Boppany i Halldorssona.

Zaimplementowano cztery wersje algorytmu Brona-Kerboscha do znajdowania klik maksymalnych: wersję klasyczną, dwa warianty z punktem podziału (punkt przypadkowy, punkt o największym stopniu), oraz wersję z uporządkowaniem degeneracji.

Wszystkie algorytmy posiadają testy jednostkowe, a dla wybranych algorytmów sprawdzono ich praktyczną złożoność obliczeniową.

Słowa kluczowe: grafy, problem kliki, klika maksymalna, klika największa, zbiór niezależny, zbiór potęgowy

English title: Study of the clique problem with Python

Abstract

Python implementation of selected graph algorithms for the clique problem is presented. A few tools are created that are used in fast clique finding algorithms. Generators for a power set, all cliques, all cliques with k nodes, and all triangles are implemented. Two methods for finding a clustering coefficient are available. Four algorithms for finding a maximal independent set are improved (the starting node can be set).

Several algorithms for finding maximal or maximum cliques are implemented: a simple algorithm checking all cliques, the algorithm with a growing clique, the independent sets algorithm, and the Boppana and Halldorsson algorithm.

Four versions of the Bron-Kerbosh algorithm for finding maximal cliques are implemented: the classic version, two variants involving a pivot vertex (a random pivot, a maximum degree pivot), and the version with the degeneracy ordering.

All algorithms have own unit tests and for some algorithms the real computational complexity was checked.

Keywords: graphs, clique problem, maximal clique, maximum clique, independent set, power set

Spis treści

Spis tabel

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy jest problem kliki [1]. Jest to w istocie rodzina problemów związanych z wyszukiwaniem w danym grafie podgrafu będącego grafem pełnym. Do tej rodziny problemów należą między innymi:
— Znajdowanie największej kliki, czyli kliki o największej liczbie wierzchołków.
— Znajdowanie kliki o największej wadze w grafie ważonym Tutaj każdy wierzchołek grafu (rzadziej krawędź) ma przyporządkowaną wagę.
— Wypisywanie wszystkich klik maksymalnych, czyli klik nie zawierających się w większej klice.
— Wypisywanie jednej lub wszystkich klik o rozmiarze k.
Problem kliki znajduje swoje zastosowanie w badaniach sieci społecznościowych, w bioinformatyce, czy w chemii obliczeniowej. W samej teorii grafów rozmiar największej kliki jest dolnym ograniczeniem na liczbę chromatyczną grafu.

Problem kliki jest klasycznym problemem NP-zupełnym, czyli prawie nie ma szans na znalezienie dla niego algorytmu o złożoności wielomianowej. W ogólnym przypadku znane są różne algorytmy o złożoności wykładniczej, natomiast dla pewnych szczególnych rodzin grafów znaleziono algorytmy wielomianowe.

1.1. Cele pracy

Celem pracy jest implementacja różnych algorytmów związanych z problemem kliki, a także sprawdzenie ich wydajności dla różnego rodzaju grafów. Do zapisu algorytmów został użyty język Python [2]. Zaletą użycia tego języka jest bogata biblioteka standardowa, mnogość dostępnych dodatkowych modułów, czytelna i zwięzła składnia. Kod programu w języku Python ma formę bardzo zbliżoną do pseudokodu. Cecha ta pozwala skupić się na algorytmie oraz sposobie jego działania, zamiast na rozszyfrowywaniu składni oraz mechanizmów użytego języka programowania.

1.2. Organizacja pracy

Treść pracy jest zorganizowana w następujący sposób. Rozdział 1 zawiera krótkie wprowadzenie i cele pracy. Rozdział 2 wprowadza podstawowe pojęcia z teorii grafów, z których korzystano w dalszej części pracy. Rozdział 3 opisuje implementację w języku Python najważniejszych struktur grafowych. Największy w pracy rozdział 4 poświęcony jest algorytmom grafowym związanym z problemem kliki. Zaprezentowano również duży zestaw pomocni-

czych funkcji do wyznaczania różnych zestawów obiektów (zbiory potęgowe, kliki). Rozdział 5 zawiera podsumowanie pracy. W dodatku A zebrano wyniki testów zaimplementowanych algorytmów.

2. Teoria grafów

Teoria grafów to dział matematyki oraz informatyki zajmujący się badaniem właściwości grafów [3], [4], [5]. Informatyka w dużej mierze zajmuje się tworzeniem algorytmów pozwalających określić różne właściwości badanego grafu. Algorytmy te służą do rozwiązywania wielu praktycznych zadań, często w bardzo szerokim zakresie dziedzin, takich jak m.in.: ekonomia, socjologia, bankowość, czy problemów związanych z optymalizacją. Już w 1736 Leonhard Euler badał zagadnienia związane z grafami, było to zagadnienie mostów królewieckich.

2.1. Grafy skierowane i nieskierowane

Grafem prostym (ang. *simple graph*) nazywamy uporządkowaną parę wierzchołków $G = (V, E)$, w skład której wchodzi zbiór wierzchołków V, oraz zbiór krawędzi E wyznaczających połączenia pomiędzy wierzchołkami. *Multigrafy* mogą zawierać pętle przy wierzchołkach i krawędzie równoległe (krawędzie łączące te same dwa wierzchołki).

Graf skierowany (ang. *directed graph*) jest to rodzaj grafu, którego zbiór krawędzi zawiera krawędzie skierowane. Jest dobrze określony wierzchołek początkowy i wierzchołek końcowy. Krawędzie skierowane w grafie mogą służyć do ukazania kierunku poruszania się pomiędzy wierzchołkami, kierunku przepływu danych, itp.

Graf nieskierowany (ang. *undirected graph*) jest to graf, którego zbiór krawędzi zawiera krawędzie nieskierowane. Mówimy, że dwa wierzchołki należące do krawędzi są sąsiednie. Istnieje możliwość przechodzenia przez krawędź w obu kierunkach.

Stopień wierzchołka w grafie nieskierowanym jest to liczba krawędzi spotykających się w danym wierzchołku. Pętla własna przy wierzchołku liczy się podwójnie. W grafie skierowanym można wyróżnić stopień wychodzący i stopień przychodzący.

2.2. Kliki

Dla grafu nieskierowanego $G = (V, E)$ określamy *klikę* jako podzbiór zbioru wierzchołków V, który indukuje podgraf będący grafem pełnym. Klika maksymalna (ang. *maximal clique*) nie jest podzbiorem większej kliki. Klika największa (ang. *maximum clique*) jest to klika o największej liczności w grafie. Rozmiar największej kliki w grafie G (ang. *clique number*) oznacza się przez $\omega(G)$ [6]. Termin *klika* pochodzi z artykułu wykorzystującego grafy

pełne w sieciach społecznościowych do opisu klik ludzi, czyli grupy ludzi znających siebie nawzajem [7].

Problem znalezienia największej kliki jest w ogólności NP-zupełny. Jednak dla pewnych rodzin grafów istnieją rozwiązania wielomianowe. Są to przykładowo grafy dwudzielne, grafy regularne, grafy planarne [8]. Dla tych rodzin grafów istnieje ograniczenie na możliwy rozmiar największej kliki.

2.3. Zbiory niezależne

Dla grafu nieskierowanego $G = (V, E)$ określamy *zbiór niezależny* (ang. *independent set*) jako podzbiór zbioru wierzchołków V, w którym żadna para wierzchołków nie jest połączona krawędzią [9]. Maksymalny zbiór niezależny (ang. *maximal independent set*) nie jest podzbiorem innego zbioru niezależnego. Największy zbiór niezależny (ang. *maximum independent set*) jest zbiorem niezależnym o największej liczności w grafie.

Zbiór niezależny jest przeciwieństwem kliki. Wierzchołki zbioru niezależnego w grafie G tworzą klikę w dopełnieniu grafu (ang. *complement graph*), czyli w grafie $\bar{G}$. Stąd problem znalezienia największego zbioru niezależnego jest równoważny problemowi znalezienia największej kliki.

Warto zauważyć, że problem kliki i problem zbioru niezależnego mają inne właściwości, kiedy zawęzimy się do pewnych rodzin grafów. Przykładowo dla grafów planarnych problem kliki może być rozwiązany w czasie wielomianowym [10], podczas gdy problem zbioru niezależnego pozostaje NP-trudny.

2.4. Grafy planarne

Graf planarny (ang. *planar graph*) jest to graf, który może być narysowany na płaszczyźnie tak, że jego krawędzie nie będą się przecinać [11]. Twierdzenie Kuratowskiego (1930) mówi, że grafy planarne nie mogą zwierać podgrafu homeomorficznego z grafem pełnym K_5, ani z grafem dwudzielnym pełnym $K_{3,3}$. Stąd można wywnioskować, że największe możliwe kliki w grafach planarnych będą odpowiadały grafowi pełnemu K_4.

2.5. Grafy doskonałe

Graf doskonały (ang. *perfect graph*) jest to graf w którym liczba chromatyczna każdego podgrafu indukowanego jest równa rozmiarowi największej kliki tego podgrafu [12]. Dla grafów doskonałych problem kliki ma rozwiązanie działające w czasie wielomianowym. Przykładowe klasy grafów doskonałych:

— grafy dwudzielne (ang. *bipartite graphs*),
— grafy krawędziowe uzyskane z grafów dwudzielnych,
— grafy przedziałowe (ang. *interval graphs*) [13],
— grafy cięciwowe (and. *chordal graphs*).

3. Implementacja grafów

Kod w języku Python w naszej pracy powstał przy wykorzystaniu biblioteki grafowej rozwijanej w Instytucie Fizyki UJ [14]. Konieczne okazało się rozszerzenie interfejsu grafów o nową metodę complement(), która zwraca dopełnienie danego grafu. Metodę wykorzystano do wyznaczania maksymalnej kliki w grafie. Drugie rozszerzenie interfejsu grafów to metoda subgraph(nodes), która zwraca podgraf indukowany. Dalej przedstawimy najważniejsze cechy biblioteki.

3.1. Grafowe struktury danych

Poniżej pokażemy jak zaimplementowano wybrane obiekty grafowe w rozwijanej bibliotece grafowej.

Wierzchołek: Obiekt hashowalny.

Krawędź: Instancja klasy Edge.

Graf: Instancja klasy Graph.

Multigraf: Instancja klasy MultiGraph.

Algorytm: Klasa z typowymi metodami __init__ do inicjalizacji danych i run do właściwej pracy. Wyniki działania algorytmu są zapisane w odpowiednich atrybutach klasy.

Drzewo rozpinające: Instancja klasy Graph lub słownik parent z (node, node) lub (node, None) dla korzenia.

Klika: Zbiór wierzchołków clique lub słownik z (node, bool).

Zbiór niezależny: Zbiór wierzchołków independent_set lub słownik z (node, bool).

Pokrycie wierzchołkowe: Zbiór wierzchołków node_cover lub słownik z (node, bool).

Pokrycie krawędziowe: Zbiór krawędzi edge_cover, ale edge.source $<$ edge.target.

Skojarzenie: Zbiór krawędzi lub słownik mate z (node, node) lub (node, None). Inna możliwość to słownik mate z (node, edge) (krawędź do drugiego wierzchołka z pary) lub (node, None).

Kolorowanie wierzchołków: Słownik color z (node, int) lub (node, None) przy braku koloru. Kolory są numerowane od 0 w górę.

Kolorowanie krawędzi: Słownik color z (edge, int) lub (edge, None) przy braku koloru, ale edge.source $<$ edge.target. Kolory są numerowane od 0 w górę. Dla multigrafów krawędzie muszą być unikalne, tzn. przy krawędziach równoległych muszą być różnice w atrybucie edge.weight.

3.2. Przykładowe obliczenia

Przykładowa sesja interaktywna.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> graph_factory = GraphFactory(Graph)
>>> G = graph_factory.make_random(10, False, 0.5)

# Wyznaczanie kliki maksymalnej.
>>> from maximalclique import MaximalClique
>>> algorithm = MaximalClique(G)
>>> algoritm.run()
>>> algoritm.clique    # klika maksymalna

# Algorytm Boppany i Halldorssona.
>>> from isetramsey import MaximalCliqueRamsey
>>> algorithm = MaximalCliqueRamsey(self.G)
>>> algorithm.run()
>>> algoritm.clique    # klika maksymalna

# Algorytm Brona-Kerboscha.
>>> from bronkerbosch import ClassicBronKerbosch
>>> algorithm = ClassicBronKerbosch(G)
>>> algoritm.run()
>>> algoritm.cliques    # lista klik maksymalnych
```

4. Algorytmy

W pewnych algorytmach związanych z klikami zachodzi potrzeba generowania zbioru potęgowego dla danego zbioru wierzchołków grafu. Dlatego w ramach pracy stworzono funkcje realizujące to zadanie.

4.1. Wyznaczanie zbioru potęgowego

Zbiór potęgowy 2^X (ang. *power set*) jest to zbiór wszystkich podzbiorów danego zbioru X [15]. Jeżeli X jest zbiorem n-elementowym, to zbiór potęgowy ma 2^n elementów, stąd oznaczenie symboliczne 2^X.

Przy tworzeniu zbioru potęgowego wygodnie jest operować zbiorem w postaci listy Pythona. W zastosowaniach zwykle nie potrzebujemy od razu całego zbioru potęgowego, tylko chcemy przetwarzać po kolei wszystkie podzbiory. Optymalnym rozwiązaniem jest iterator po podzbiorach iter_power_set(). Warto zwrócić uwagę na jego strukturę rekurencyjną, która prawie odtwarza matematyczną definicję zbioru potęgowego. Przykłady innych implementacji w języku Python można znaleźć w serwisie *Rosetta Code* [16]. Istotnie różnym podejściem jest wykorzystanie reprezentacji binarnej różnych podzbiorów danego zbioru.

Listing 4.1. Moduł powersets.

```
#!/usr/bin/python

import sys

def iter_power_set(L):
    """Generuje wszystkie podzbiory danego zbioru."""
    # Glebokosc rekurencji rowna sie len(L).
    recursionlimit = sys.getrecursionlimit()
    sys.setrecursionlimit(max(len(L) * 2, recursionlimit))
    if len(L) == 0:
        yield []
    else:
        item = [L[0]]
        for subset in iter_power_set(L[1:]):
            yield subset
            yield subset + item
```

4.2. Wyznaczanie wszystkich klik

Sprawdzenie, czy dany zbiór k wierzchołków tworzy klikę jest proste, można to zrobić w czasie $O(k^2)$. Zadanie to realizuje funkcja is_clique().

Listing 4.2. Funkcja testująca klikę.

```
from edges import Edge

def is_clique(graph, nodes):
    """Sprawdzenie czy zbior wierzcholkow tworzy klike."""
    if len(nodes) == 0:   # zbior pusty to nie klika
        return False
    for node1 in nodes:
        for node2 in nodes:
            if node1 < node2:
                if not graph.has_edge(Edge(node1, node2)):
                    return False
    return True
```

Najprostszy algorytm siłowy wyznaczania wszystkich klik polega na testowaniu wszystkich elementów zbioru potęgowego zbudowanego na bazie zbioru wszystkich wierzchołków grafu. W naszej implementacji generator wszystkich klik korzysta z generatora zbioru potęgowego. Złożoność czasową szacujemy na $O(2^n n^2)$.

Listing 4.3. Generator wszystkich klik w grafie.

```
def iter_all_cliques(graph):
    """Generator wszystkich klik w grafie."""
    for nodes in iter_power_set(list(graph.iternodes())):
        if is_clique(graph, nodes):
            yield set(nodes)
```

4.3. Wyznaczanie wszystkich klik z k wierzchołkami

W pewnych sytuacjach potrzebna jest znajomość wszystkich klik zawierających dokładnie k wierzchołków. W naszej implementacji korzystamy z generatora kombinacji k-elementowych z biblioteki standardowej Pythona. Złożoność czasowa wynosi $O(n)$ dla $k = 1$, $O(m)$ dla $k = 2$, oraz $O(n^k k^2)$ dla $k > 2$.

Listing 4.4. Generator wszystkich klik z k wierzchołkami.

```
import itertools

def iter_k_cliques(graph, k):
    """Generator klik z k wierzcholkami."""
    if k == 1:
        for node in graph.iternodes():
            yield set([node])
    elif k == 2:
        for edge in graph.iteredges():
            yield set([edge.source, edge.target])
    else:
        for nodes in itertools.combinations(graph.iternodes(), k):
            if is_clique(graph, nodes):
                yield set(nodes)
```

4.4. Wyznaczanie wszystkich trójkątów w grafie

Przydatnym narzędziem może być generator trójkątów, czyli klik z trzema wierzchołkami, zawierających dany wierzchołek. W naszej implementacji korzystamy z generatora kombinacji 2-elementowych z biblioteki standardowej Pythona. Złożoność czasowa jest szacowana na $O(\Delta^2)$.

Warto zauważyć, że dla każdego grafu grafu planarnego można otrzymać graf topologiczny, czyli reprezentację grafu z wyznaczoną kolejnością sąsiadów każdego wierzchołka. Korzystając z grafu topologicznego można wyznaczyć wszystkie trójkąty zawierające dany wierzchołek w czasie $O(\Delta)$.

Listing 4.5. Generator wszystkich trójkątów.

```
import itertools

def iter_triangles(graph, node):
    """Generator trojkatow zawierajacych node."""
    for nodes in itertools.combinations(
        (edge.target for edge in graph.iteroutedges(node)), 2):
            triangle = set(nodes)
            triangle.add(node)
            if is_clique(graph, triangle):
                yield triangle
```

4.5. Wyznaczanie trójkątów w grafie planarnym

W grafie planarnym wszystkie trójkąty można znaleźć w czasie liniowym [18], [19]. Wykorzystuje się algorytm DFS.

4.6. Wyznaczanie liczby trójkątów z macierzy sąsiedztwa

Jednym jest sposobów zapisu struktury grafu jest kwadratowa macierz sąsiedztwa $A = [a_{ij}]$, gdzie indeksy i, j numerują wierzchołki, a_{ij} zawiera liczbę krawędzi łączących wierzchołki i-ty z j-tym [4]. Podana definicja stosuje się do grafów prostych i multigrafów, skierowanych i nieskierowanych. Wybrane informacje zawarte w macierzy sąsiedztwa [4]:

— Jeżeli macierz A jest symetryczna, to graf jest nieskierowany.
— Jeżeli $a_{ii} = 0$ dla każdego i, to graf nie ma pętli własnych.
— Jeżeli macierz A zawiera tylko liczby 0 i 1, to w grafie nie występują krawędzie równoległe.
— Dla grafu prostego nieskierowanego, stopień i-tego wierzchołka jest równy sumie wartości elementów i-tego wiersza lub i-tej kolumny.
— Dla grafu prostego skierowanego, stopień *wyjściowy* i-tego wierzchołka jest równy sumie wartości elementów i-tego wiersza.
— Dla grafu prostego skierowanego, stopień *wejściowy* i-tego wierzchołka jest równy sumie wartości elementów i-tej kolumny.

— Jeżeli $B = [b_{ij}] = A^2$ dla grafu prostego nieskierowanego, to (1) b_{ii} równa się stopniowi i-tego wierzchołka, (2) b_{ij} (i różne od j) równa się liczbie ścieżek o długości 2 od i do j.
— Jeżeli $B = [b_{ij}] = A^2$ dla grafu prostego skierowanego, to (1) b_{ii} równa się liczbie par krawędzi antyrównoległych incydentnych z i-tym wierzchołkiem, (2) b_{ij} (i różne od j) równa się liczbie ścieżek *skierowanych* o długości 2 od i do j.
— Jeżeli $C = [c_{ij}] = A^3$ dla grafu prostego, to c_{ii} równa się liczbie cykli *skierowanych* o długości 3 (trójkąty), przechodzących przez i-ty wierzchołek. W grafie nieskierowanym interesują nas cykle nieskierowane, czyli liczba trójkątów w c_{ii} będzie podwojona. Trzeba pamiętać, że każdy trójkąt przechodzi przez trzy wierzchołki, czyli jest wzięty pod uwagę w trzech różnych elementach c_{ii}.

Własności macierzy $C = A^3$ wykorzystamy do stworzenia przydatnego algorytmu.

Dane wejściowe: Graf prosty skierowany lub nieskierowany.

Problem: Wyznaczenie liczby trójkątów zawierających dowolny wierzchołek grafu i liczby wszystkich trójkątów w grafie.

Opis algorytmu: Algorytm rozpoczyna się od stworzenia macierzy A, ponieważ reprezentacja grafu jest ukryta za interfejsem. Jeżeli mamy na wejściu graf ważony podany w reprezentacji macierzy sąsiedztwa, to i tak musimy zbudować macierz zero-jedynkową. Następnie w potrójnej pętli po wierzchołkach wykonywane są działania odpowiadające mnożeniu macierzy.

Złożoność: Złożoność czasowa algorytmu jest $O(V^3)$, ze względu na potrójna pętlę po wierzchołkach grafu. Złożoność pamięciowa grafu jest szacowana na $O(V^2)$, ze względu na tworzenie macierzy A. W naszej implementacji słownikowej macierzy A przechowujemy tylko niezerowe elementy, co daje złożoność pamięciową $O(E)$.

Uwagi: Algorytm został wykorzystany przy tworzeniu jednej z wersji funkcji obliczającej współczynnik klastrowania.

Listing 4.6. Moduł triangles.

```
#!/usr/bin/python

class TriangleMatrix:
    """Finding triangles in O(V**3) time."""

    def __init__(self, graph):
        """The algorithm initialization."""
        # Graf moze byc skierowany lub nieskierowany.
        self.graph = graph
        # Przygotowanie macierzy sasiedztwa 0-1.
        # Przechowujemy tylko jedynki w slowniku a.
        self.a = dict()
```

```
        for source in self.graph.iternodes():
            self.a[source] = dict()
        for edge in self.graph.iteredges():    # O(E) time
            self.a[edge.source][edge.target] = 1
            if not self.graph.is_directed():
                self.a[edge.target][edge.source] = 1
        # Elementy diagonalne macierzy A**3.
        # self.triangle[node] zawiera liczbe trojkatow przy node.
        self.triangle = dict((node, 0)
            for node in self.graph.iternodes())

    def run(self, source=None):
        """Executable pseudocode."""
        for node in self.graph.iternodes():    # O(V**3) time
            for source in self.graph.iternodes():
                for target in self.graph.iternodes():
                    self.triangle[node] += (
                        self.a[node].get(source, 0) *
                        self.a[source].get(target, 0) *
                        self.a[target].get(node, 0))
        # W grafie nieskierowanym cykle sa liczone podwojnie,
        # w obie strony.
        if not self.graph.is_directed():
            for node in self.graph.iternodes():    # O(V) time
                self.triangle[node] /= 2

    def find_all_triangles(self):
        """Return the number of all triangles."""
        result = sum(self.triangle[node]
            for node in self.graph.iternodes())
        return result / 3
```

4.7. Wyznaczanie największej kliki w grafie

Najprostszy algorytm siłowy polega na sprawdzeniu wszystkich możliwych klik i zapisaniu największej z nich. Złożoność czasową szacujemy na $O(2^n n^2)$.

Listing 4.7. Wyznaczanie największej kliki w grafie.

```
def find_maximum_clique(graph):
    """Wyznaczanie kliki o najwiekszej licznosci w grafie."""
    return max(iter_all_cliques(graph), key=len)
```

4.8. Wyznaczanie maksymalnej kliki w grafie

Dane wejściowe: Graf prosty nieskierowany G, wierzchołek grafu v.

Problem: Wyznaczenie maksymalnej kliki zawierającej wierzchołek v.

Opis algorytmu: Algorytm rozpoczynamy od kliki zawierającej tylko podany wierzchołek v. Następnie do bieżącej kliki sukcesywnie dołączamy wierz-

chołki (jeden na raz), które są połączone z każdym wierzchołkiem należącym do bieżącej kliki. Inne wierzchołki grafu odrzucamy [1].

Złożoność: Złożoność czasowa algorytmu jest liniowa $O(V + E)$, ponieważ algorytm przebiega listy sąsiedztwa każdego wierzchołka grafu. Złożoność pamięciowa jest klasy $O(V)$, ponieważ zapamiętujemy jedynie wierzchołki należące do kliki.

Uwagi: Wyznaczona klika może być bardzo mała, np. dwuelementowa. Rozwiązanie może nie być największą kliką z wierzchołkiem v.

Listing 4.8. Moduł maximalclique.

```
#!/usr/bin/python

class MaximalClique:
    """Finding a maximal clique in O(V+E) time."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.clique = set()
        self.source = None
        self.cardinality = 0

    def run(self, source=None):
        """Executable pseudocode."""
        if source is None:   # get first random node
            source = self.graph.iternodes().next()
        self.source = source
        self.clique.add(self.source)
        for node in self.graph.iternodes():
            if node in self.clique:
                continue
            counter = 0
            for target in self.graph.iteradjacent(node):
                if target in self.clique:
                    counter += 1
            if counter == len(self.clique):
                self.clique.add(node)
        self.cardinality = len(self.clique)
```

4.9. Wyznaczanie maksymalnej kliki przez zbiór niezależny

Dane wejściowe: Graf prosty nieskierowany G, wierzchołek grafu v.

Problem: Wyznaczenie maksymalnej kliki zawierającej wierzchołek v przez znalezienie największego zbioru niezależnego w dopełnieniu grafu.

Opis algorytmu: Algorytm rozpoczynamy od wyznaczenia grafu $\bar{G}$, czyli dopełnienia grafu G. Następnie w grafie $\bar{G}$ wyznaczamy największy zbiór niezależny zawierający wierzchołek v.

Złożoność: Złożoność czasowa etapu wyznaczania dopełnienia grafu zajmuje czas $O(V^2)$. Czas wyznaczania największego zbioru niezależnego zależy od użytego algorytmu, a w najprostszym przypadku jest liniowy $O(V + E)$.

Uwagi: W algorytmie można wykorzystać różne algorytmy wyznaczania największego zbioru niezależnego. Dostępne są cztery moduły: isetus (ang. *unordered sequential independent set*), isetrs (ang. *random sequential independent set*), isetsf (ang. *smallest first independent set*), isetll (ang. *largest last independent set*). Rozwiązanie może nie być największą kliką z wierzchołkiem v.

Listing 4.9. Moduł isetclique.

```
#!/usr/bin/python

from isetus import UnorderedSequentialIndependentSet #as IndependentSet
from isetrs import RandomSequentialIndependentSet #as IndependentSet
from isetsf import SmallestFirstIndependentSet #as IndependentSet
from isetll import LargestLastIndependentSet #as IndependentSet


class MaximalCliqueFromIndependentSet:
    """Finding a maximal clique using the complement of a graph."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.clique = set()
        self.source = None
        self.cardinality = 0

    def run(self, source=None, ind_set_method='UnorderedSequentialIndependentSet'):
        """Executable pseudocode."""
        if source is None:   # get first random node
            source = self.graph.iternodes().next()
        self.source = source
        new_graph = self.graph.complement()

        if ind_set_method == 'UnorderedSequentialIndependentSet':
```

```
            algorithm = UnorderedSequentialIndependentSet(new_graph)
        elif ind_set_method == 'RandomSequentialIndependentSet':
            algorithm = RandomSequentialIndependentSet(new_graph)
        elif ind_set_method == 'SmallestFirstIndependentSet':
            algorithm = SmallestFirstIndependentSet(new_graph)
        elif ind_set_method == 'LargestLastIndependentSet':
            algorithm = LargestLastIndependentSet(new_graph)

        algorithm.run(source)
        self.clique = algorithm.independent_set
        self.cardinality = len(self.clique)
```

4.10. Algorytm Boppany i Halldorssona

W bibliotece NetworkX można znaleźć implementację algorytmu wyznaczania kliki maksymalnej [funkcja max_clique(G)] na bazie artykułu Boppany i Haldorssóna [20]. Algorytm wyznacza największy zbiór niezależny w dopełnieniu grafu, przy czym korzysta się z wyników teorii Ramseya.

Udało się przełożyć kod algorytmu do naszej biblioteki. Wymagało to rozszerzenia interfejsu grafów o metodę subgraph(), która zwraca podgraf indukowany z danego grafu na bazie dostarczonej kolekcji wierzchołków. Wstępne testy pokazały dłuższe czasy wyznaczania kliki maksymalnej, ale często była to klika większa niż te wyznaczone innymi metodami. Nie ma tu tylko możliwości zagwarantowania, że dany wierzchołek będzie należał do wyznaczonej kliki.

Listing 4.10. Moduł isetramsey.

```
#!/usr/bin/python

class MaximalCliqueRamsey:
    """Finding a maximal clique using Ramsey theory."""
    # Bazuje na kodzie z NetworkX, wykorzystane funkcje to:
    # max_clique(G), clique_removal(G), ramsey_R2(G).

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.clique = set()
        self.source = None
        self.cardinality = 0

    def run(self, source=None):
        """Executable pseudocode."""
        if source is None:   # get first random node
            source = self.graph.iternodes().next()
        self.source = source
        new_graph = self.graph.complement()
        self.clique, _ = self.clique_removal(new_graph)
        self.cardinality = len(self.clique)

    def clique_removal(self, graph):
```

```
        """ Repeatedly remove cliques from the graph."""
        graph_copy = graph.copy()
        clique_i, iset_i = self.ramsey(graph_copy)
        cliques = [clique_i]   # maximal cliques found
        isets = [iset_i]
        while graph_copy.v() > 0:
            for node in clique_i:
                graph_copy.del_node(node)
            clique_i, iset_i = self.ramsey(graph_copy)
            if clique_i:
                cliques.append(clique_i)
            if iset_i:
                isets.append(iset_i)
        max_iset = max(isets, key=len)
        return max_iset, cliques

    def ramsey(self, graph):
        """Approximately computes the Ramsey number R(2;s,t)."""
        if graph.v() == 0:
            return set([]), set([])
        pivot = graph.iternodes().next()
        neighbors = set(graph.iteradjacent(pivot))
        nonneighbors = set(node for node in graph.iternodes()
            if node not in neighbors and node != pivot)
        clique_1, iset_1 = self.ramsey(graph.subgraph(neighbors))
        clique_2, iset_2 = self.ramsey(graph.subgraph(nonneighbors))
        clique_1.add(pivot)
        iset_2.add(pivot)
        return (max([clique_1, clique_2], key=len),
                max([iset_1, iset_2], key=len))
```

4.11. Wyznaczanie współczynnika klastrowania

Wiele układów ma naturalną interpretację jako sieci złożone (ang. *complex networks*). Mogą to być sieci znajomych na Facebooku, połączenia między stronami WWW, oddziaływania białek w układach biologicznych. Istnieje wiele modeli sieci złożonych. Najprostsze modele to sieci przypadkowe (ang. *random networks*), w których lokalnie sieć jest drzewiasta (bez pętli). Jednak prawdziwe sieci często mają pewne podstruktury, których nie można zignorować, np. grupy przyjaciół znających się wzajemnie. Stąd powstały modele sieci oparte na klikach/klastrach (ang. *clustered networks*), wykorzystywane do symulacji epidemii [21]. Ilościowo klastrowanie w sieci opisuje współczynnik klastrowania (ang. *clustering coefficient*), zdefiniowany jako

$$C_{\triangle} = \frac{3 \times N_{\triangle}}{N_3}, \tag{4.1}$$

gdzie $N_{\triangle}$ jest liczbą wszystkich trójkątów w sieci, a N_3 liczbą połączonych trójek wierzchołków (połączona trójka to wierzchołek połączony bezpośrednio z nieuporządkowaną parą wierzchołków). Jeżeli $C_{\triangle} = 0$, to sieć lokalnie

jest drzewiasta (nie ma cykli długości 3). Przykładowo dla grafu koła mamy $C_{\triangle}(W_4) = C_{\triangle}(K_4) = 1$, a dla $n > 4$ dostajemy

$$C_{\triangle}(W_n) = \frac{3(n-1)}{3(n-1) + (n-1)(n-2)/2} = \frac{6}{n+4}. \tag{4.2}$$

Warto zauważyć, że dla grafów r-regularnych zachodzi $N_3 = nr(r-1)/2$. Listing przedstawia funkcję obliczającą współczynnik klastrowania. Wyznaczenie liczby wszystkich trójkątów zajmuje czas $O(n^3)$, wyznaczenie liczby połączonych trójek zajmuje czas $O(n)$, dlatego łączny czas szacujemy na $O(n^3)$.

Listing 4.11. Wyznaczanie współczynnika klastrowania.

```
def clustering_coefficient(graph, fraction_format=True):
    """Wyznaczanie wspolczynnika klastrowania."""
    n_t = sum(1 for triangle in iter_k_cliques(graph, 3))
    n_3 = sum(graph.degree(node) * (graph.degree(node)-1) / 2
        for node in graph.iternodes())
    if fraction_format:
        return Fraction(3 * n_t, n_3)
    else:
        return float(3 * n_t) / float(n_3)
```

Druga wersja współczynnika klastowania wykorzystuje algorytm wyznaczania liczby trójkątów w grafie oparty na własnościach macierzy sąsiedztwa.

Listing 4.12. Wyznaczanie współczynnika klastrowania z macierzy sąsiedztwa.

```
from triangles import TriangleMatrix

def clustering_coefficient(graph, fraction_format=True):
    """Wyznaczanie wspolczynnika klastrowania z macierzy sasiedztwa."""
    algorithm = TriangleMatrix(graph)
    algorithm.run()
    n_t = algorithm.find_all_triangles()
    n_3 = sum(graph.degree(node) * (graph.degree(node)-1) / 2
        for node in graph.iternodes())
    if fraction_format:
        return Fraction(3 * n_t, n_3)
    else:
        return float(3 * n_t) / float(n_3)
```

4.12. Klasyczny algorytm Brona-Kerboscha

Algorytm Brona-Kerboscha pozwala znaleźć wszystkie kliki maksymalne w grafie nieskierowanym [17]. Algorytm został opublikowany przez Brona i Kerboscha w roku 1973 [22]. Później powstało wiele ulepszeń podstawowego algorytmu [23].

Algorytm rekurencyjny Brona-Kerboscha wykorzystuje trzy zbiory wierzchołków: zbiór R wierzchołków stanowiących częściowe rozwiązanie (klikę), zbiór P kandydatów do rozważenia, zbiór X wierzchołków pominiętych. Zbiór

X zawiera wierzchołki, które były wcześniej w P, a przez to kliki maksymalne zawierające te wierzchołki były już raportowane. Tak więc celem przechowywania zbioru X jest uniknięcie ponownego raportowania klik.

Dane wejściowe: Graf prosty nieskierowany.

Problem: Wyznaczenie wszystkich klik maksymalnych w grafie.

Opis algorytmu: Przy pierwszym wywołaniu algorytm jest uruchamiany z pustymi zbiorami R i X, a zbiór P zawiera wszystkie wierzchołki grafu. W algorytmie na początku sprawdza się, czy zbiory P i X są puste. Jeżeli tak jest, to R jest kliką maksymalną. W przeciwnym razie ze zbioru P wybierane są kolejne wierzchołki v, a algorytm jest uruchamiany rekurencyjnie dla nowych argumentów R', P', X'. Oznaczmy przez $N(v)$ zbiór wszystkich sąsiadów wierzchołka v. Wtedy R' to zbiór R z dodanym wierzchołkiem v, $P' = P \cap N(v)$, $X' = X \cap N(v)$. Po wyjściu z rekurencyjnego wywołania algorytmu z argumentami R', P', X', wierzchołek v jest usuwany z P (lepiej jest to zrobić przed wyznaczeniem zbioru P'), następnie dodawany do X, a pętla po wierzchołkach jest kontynuowana.

Złożoność: Złożoność czasowa algorytmu Brona-Kerboscha jest szacowana na $O(3^{n/3})$ [23] i nie jest liniowa w liczbie generowanych klik.

Listing 4.13. Moduł bronkerbosch.

```
#!/usr/bin/python

class ClassicBronKerbosch:
    """Finding all maximal cliques using the Bron–Kerbosch algorithm."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.cliques = list()

    def run(self):
        """Executable pseudocode."""
        R = set()
        P = set(self.graph.iternodes())
        X = set()
        self.find_cliques(R, P, X)

    def find_cliques(self, R, P, X):
        """The basic form of the Bron–Kerbosch algorithm."""
        if len(P) == 0 and len(X) == 0:
            self.cliques.append(R)
        elif len(P) == 0:
            return
        else:
            for node in self.graph.iternodes():
                if node in P:
                    neighbors = set(self.graph.iteradjacent(node))
```

```
            P.remove(node)    # before new_P and new_X
            new_R = R.union([node])
            new_P = P.intersection(neighbors)
            new_X = X.intersection(neighbors)
            self.find_cliques(new_R, new_P, new_X)
            X.add(node)
```

4.13. Algorytm Brona-Kerboscha z przypadkowym punktem podziału

Algorytm Brona-Kerboscha doczekał się kilku ulepszeń, które opierały się na obserwacjach zachowania tego algorytmu dla różnych grafów [24], [25], [26], [27]. Klasyczny algorytm jest niewydajny w sytuacji, kiedy graf zawiera dużo klik niemaksymalnych. Można oszczędzić czas i pozwolić algorytmowi szybciej przetwarzać gałęzie drzewa poszukiwań, kiedy wprowadzi się wierzchołek podziału (ang. *pivot vertex*) wybrany z P lub X. Chodzi o to, że dla każdego wierzchołka v, na danym poziomie rekurencji albo wierzchołek v, albo inny wierzchołek nie będący sąsiadem znajdzie się w raportowanej klice, ale nigdy oba naraz. Wybór wierzchołka podziału ze zbioru $P \cup X$ zapobiega niekorzystnym sytuacjom, np. kiedy graf ma postać sumy grafu pełnego i grafu gwiazdy $K_{1,s} \cup K_s$ [27].

Możliwe są różne strategie wyboru wierzchołka podziału. Najprostszy jest losowy wybór wierzchołka ze zbioru $P \cup X$.

Listing 4.14. Moduł bronkerboschrp.

```
#!/usr/bin/python

import random
from edges import Edge

class RandomPivotBronKerbosch:
    """Bron–Kerbosch algorithm with a random pivot."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.cliques = list()

    def run(self):
        """Executable pseudocode."""
        R = set()
        P = set(self.graph.iternodes())
        X = set()
        self.find_cliques(R, P, X)

    def find_pivot(self, P, X):
        """Find a random pivot."""
        return random.choice(list(P.union(X)))

    def find_cliques(self, R, P, X):
```

```
        """The Bron–Kerbosch algorithm with pivoting."""
        if len(P) == 0 and len(X) == 0:
            self.cliques.append(R)
        elif len(P) == 0:
            return
        else:
            pivot = self.find_pivot(P, X)
            for node in self.graph.iternodes():
                if ((node in P) and
                    not self.graph.has_edge(Edge(node, pivot))):
                    neighbors = set(self.graph.iteradjacent(node))
                    P.remove(node)
                    new_R = R.union([node])
                    new_P = P.intersection(neighbors)
                    new_X = X.intersection(neighbors)
                    self.find_cliques(new_R, new_P, new_X)
                    X.add(node)
```

4.14. Algorytm Brona-Kerboscha z punktem podziału o największym stopniu

Testy komputerowe pokazują, że duże przyspieszenie działania algorytmu Brona-Kerboscha można osiągnąć wtedy, gdy za wierzchołek podziału przyjmnie się wierzchołek ze zbioru $P \cup X$ o największej liczbie sąsiadów w zbiorze P [27].

Listing 4.15. Moduł bronkerboschdp.

```
#!/usr/bin/python

import random
from edges import Edge

class DegreePivotBronKerbosch:
    """Bron–Kerbosch algorithm with a pivot with largest degree in P."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.cliques = list()

    def run(self):
        """Executable pseudocode."""
        R = set()
        P = set(self.graph.iternodes())
        X = set()
        self.find_cliques(R, P, X)

    def find_pivot(self, P, X):
        """Find a pivot with largest degree in P."""
        degree_dict = dict()
        nodes = P.union(X)
        for node in nodes:
```

```
            neighbors = set(self.graph.iteradjacent(node))
            degree_dict[node] = len(P.intersection(neighbors))
        return max(nodes, key=degree_dict.__getitem__)

    def find_cliques(self, R, P, X):
        """The Bron–Kerbosch algorithm with pivoting."""
        if len(P) == 0 and len(X) == 0:
            self.cliques.append(R)
        elif len(P) == 0:
            return
        else:
            pivot = self.find_pivot(P, X)
            for node in self.graph.iternodes():
                if ((node in P) and not
                    self.graph.has_edge(Edge(node, pivot))):
                    neighbors = set(self.graph.iteradjacent(node))
                    P.remove(node)
                    new_R = R.union([node])
                    new_P = P.intersection(neighbors)
                    new_X = X.intersection(neighbors)
                    self.find_cliques(new_R, new_P, new_X)
                    X.add(node)
```

4.15. Algorytm Brona-Kerboscha z uporządkowaniem degeneracji

Innym sposobem poprawienia klasycznego algorytmu Brona-Kerboscha jest wybranie szczególnego uporządkowania wierzchołków grafu na najwyższym poziomie rekurencji [23]. Chodzi o minimalizację rozmiaru zbioru P na kolejnych poziomach rekurencji.

Dla grafu G definiuje się *degenerację* d (ang. *degeneracy*), czyli najmniejszą liczbę d taką, że każdy podgraf grafu G ma wierzchołek stopnia d lub mniejszego. Każdy graf ma *uporządkowanie degeneracji* (ang. *degeneracy ordering*), czyli uporządkowanie wierzchołków w którym każdy wierzchołek ma d lub mniej sąsiadów przychodzących później w tym uporządkowaniu [28]. Przykłady: las ma degenerację jeden, graf planarny degenerację pięć lub mniej, sieć Apoloniusza degenerację trzy. Graf r-regularny ma degenerację r.

Jeżeli w pętli algorytmu Brona-Kerboscha zostanie wykorzystane uporządkowanie degeneracji, to rozmiar zbioru P w każdym wywołaniu będzie co najwyżej d. Istotne jest przy tym posiadanie szybkiego algorytmu wyznaczania uporządkowania degeneracji, najlepiej w czasie liniowym. Czasem uporządkowanie degeneracji stosuje się na pierwszym poziomie rekurencji, a głębiej jest wariant z wierzchołkiem podziału.

Listing 4.16. Moduł bronkerboschdeg.

```
#!/usr/bin/python

class DegeneracyBronKerbosch:
    """Bron–Kerbosch algorithm with a degeneracy ordering."""

    def __init__(self, graph):
```

```
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.cliques = list()
        self.order = None

    def run(self):
        """Executable pseudocode."""
        self.find_degeneracy_ordering()
        R = set()
        P = set(self.graph.iternodes())
        X = set()
        self.find_cliques(R, P, X)

    def find_degeneracy_ordering(self):
        """Find a degeneracy ordering in O(V**2) time."""
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())    # O(V) time
        self.order = list()
        nodes = set(self.graph.iternodes())
        for step in xrange(self.graph.v()):
            source = min(nodes, key=degree_dict.__getitem__)
            nodes.remove(source)
            self.order.append(source)
            for target in self.graph.iteradjacent(source):
                degree_dict[target] -= 1

    def find_cliques(self, R, P, X):
        """The Bron-Kerbosch algorithm with a degeneracy ordering."""
        if len(P) == 0 and len(X) == 0:
            self.cliques.append(R)
        elif len(P) == 0:
            return
        else:
            for node in self.order:
                if node in P:
                    neighbors = set(self.graph.iteradjacent(node))
                    P.remove(node)
                    new_R = R.union([node])
                    new_P = P.intersection(neighbors)
                    new_X = X.intersection(neighbors)
                    self.find_cliques(new_R, new_P, new_X)
                    X.add(node)
```

4.16. Algorytm Tsukiyamy

Maksymalne kliki grafu G odpowiadają maksymalnym zbiorom niezależnym w dopełnieniu grafu. Wykorzystuje to algorytm Tsukiyamy [29]. To podejście cechuje wrażliwość na wyjście (ang. *output sensitivity*), czyli zależność czasu działania od wyników pracy algorytmu (tu zależność czasu od liczby klik maksymalnych zawartych w grafie) [27]. [To podejście jest w pakiecie igraph.] Podobne podejście zaprezentowano w pracy [30], gdzie korzystano

z działań na macierzach $n \times n$. Rozważano również kliki maksymalne dwudzielne, które są grafami pełnymi dwudzielnymi.

5. Podsumowanie

W ramach pracy przygotowano implementacje wielu algorytmów wykorzystywanych do rozwiązywania problemu kliki, a także stworzono szereg narzędzi pomocniczych. Wszystkie klasy i funkcje mają kod testujący na bazie modułu unittest. Dla potrzeb testowania kodu stworzono generatory sieci kwadratowej, sieci trójkątnej, grafu drabina, wszystkie bez periodycznych warunków brzegowych.

W pracy przygotowano dwie wersje funkcji obliczających współczynnik klastrowania sieci. Pierwsza wersja generuje odpowiednie kliki trzyelementowe, a druga wersja bazuje na własnościach macierzy sąsiedztwa. Drugą wersję można prawdopodobnie przyspieszyć używając specjalizowanych bibliotek numerycznych do mnożenia macierzy, np. z pakietu NumPy.

W pracy rozszerzono interfejs grafów przez dodanie metody complement(), która zwraca dopełnienie danego grafu. Metodę wykorzystano do wyznaczania maksymalnej kliki w grafie metodą znajdowania największego zbioru niezależnego w dopełnieniu grafu. Przy okazji poprawiono algorytmy wyznaczania zbioru niezależnego przez dodanie możliwości wskazania pierwszego wierzchołka, który ma należeć do zbioru niezależnego. Powstało też kilka nowych wersji algorytmów zbiorów niezależych. Na bazie kodu z biblioteki NetworkX powstała implementacja algorytmu wyznaczania kliki maksymalnej, korzystająca z teorii Ramseya. Tutaj przydatne okazało się rozszerzenie interfejsu grafów o metodę subgraph(nodes), która zwraca podgraf indukowany przez podany zestaw wierzchołków.

W pracy zaimplementowano i zbadano cztery wersje algorytmu Brona-Kerboscha, który znajduje wszystkie kliki maksymalne w grafie nieskierowanym. Najbardziej uniwersalna okazała się wersja z punktem podziału o największym stopniu.

A. Testy algorytmów

W tym dodatku umieściliśmy testy wybranych algorytmów lub funkcji pomocniczych.

A.1. Testy dla zbioru potęgowego

W pracy przeprowadzono testy wydajności generowania elementów zbioru potęgowego [iter_power_set()]. Wyniki przedstawia rysunek A.1. Czas generowania wszystkich podzbiorów zbioru n-elementowego jest proporcjonalny do 2^n.

A.2. Testy dla klik z k wierzchołkami

W pracy przeprowadzono testy generacji klik z k wierzchołkami, które znajdują się w grafie pełnym. Jest to przypadek, w którym uzyskuje się największą możliwą liczbę klik. Wyniki testów są przedstawione na rysunkach A.4 i A.5. Testy pokazują zależność trochę przekraczającą $O(V^k)$. Sugeruje to istnienie czynnika nie uwzględnionego w naszych rozważaniach, który jest ukryty w funkcji bibliotecznej itertools .combinations().

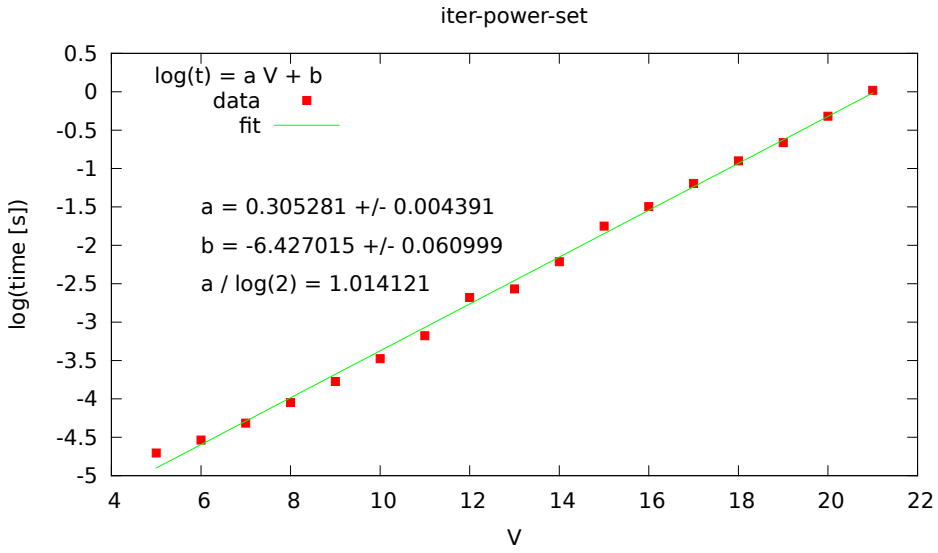
A.3. Testy dla kliki maksymalnej

Wyniki testów bezpośredniego znajdowania kliki maksymalnej są przedstawione na rysunkach A.4 i A.5. Algorytm działa w czasie liniowym w rozmiarze grafu, co potwierdzają wyniki dla grafów pełnych i dla drzew. Algorytmy oparte na zbiorach niezależnych działają znacznie wolniej, ponieważ budowa dopełnienia grafu jest kosztowna.

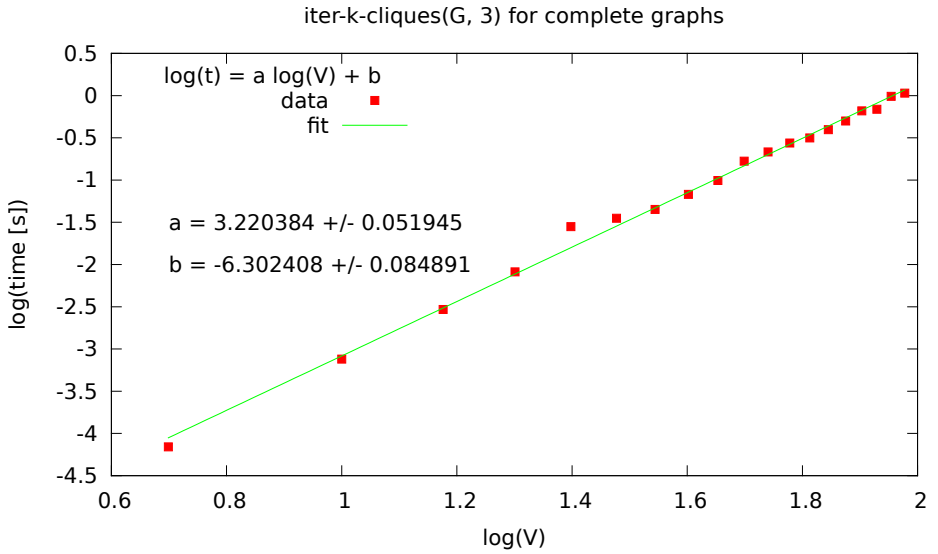
Sprawdzono wielkość kliki maksymalnej znajdowanej różnymi algorytmami, tzn. bezpośrednio (moduł maximalclique) i poprzez znajdowanie zbiorów niezależnych dla dopełnienia grafu (moduły isetclique i isetramsey). Dla ustalenia uwagi przyjmujemy, że klika ma zawierać wierzchołek grafu o największym stopniu. Wyniki znajdują się w tabeli A.1. W większości przypadków algorytm Boppany i Halldorssona znajduje klikę o największej liczności.

A.4. Testy dla sieci przypadkowych

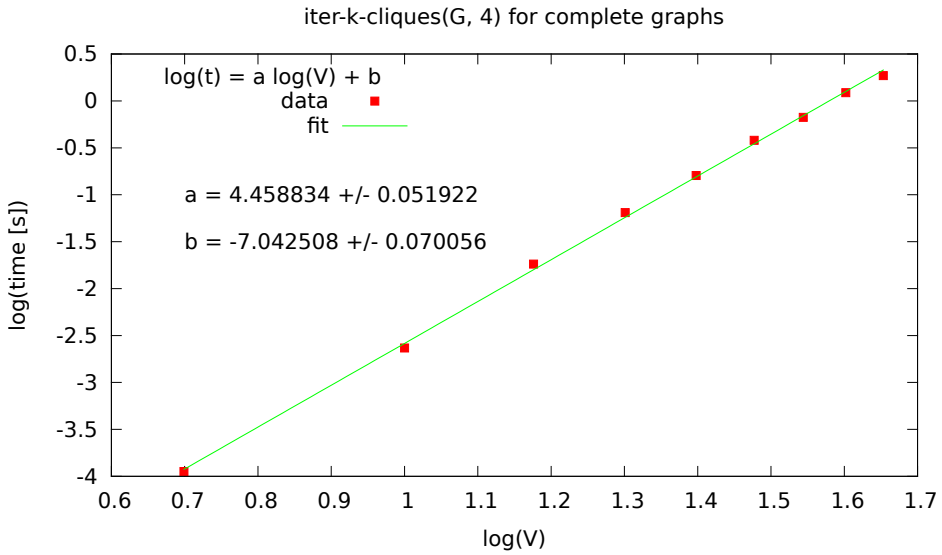
Testy dla współczynnika klastrowania. Można uzasadnić, że $C_\triangle \sim p$. Wyniki testu są przedstawione na rysunku A.6.



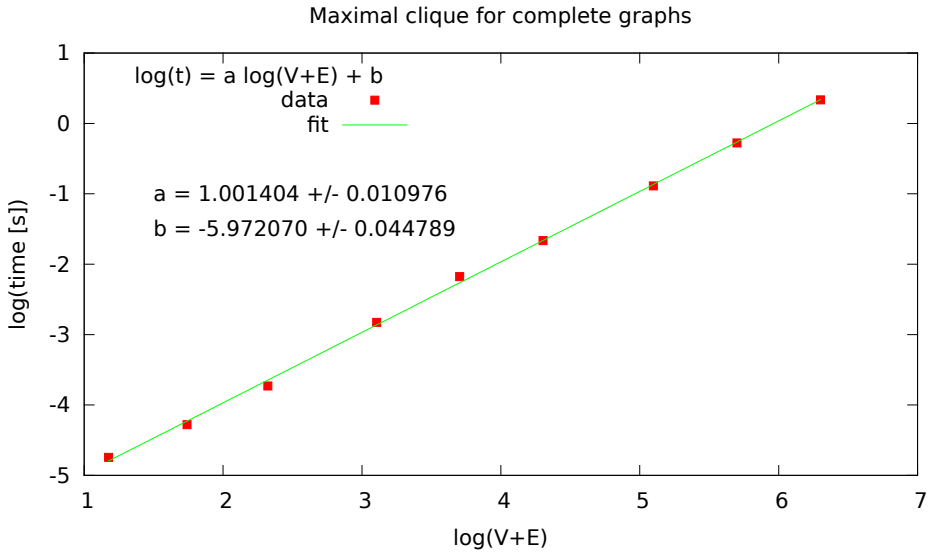
Rysunek A.1. Wykres wydajności algorytmu wyznaczania elementów zbioru potęgowego. Współczynnik $a/\log(2)$ bliski 1 potwierdza zależność wykładniczą.



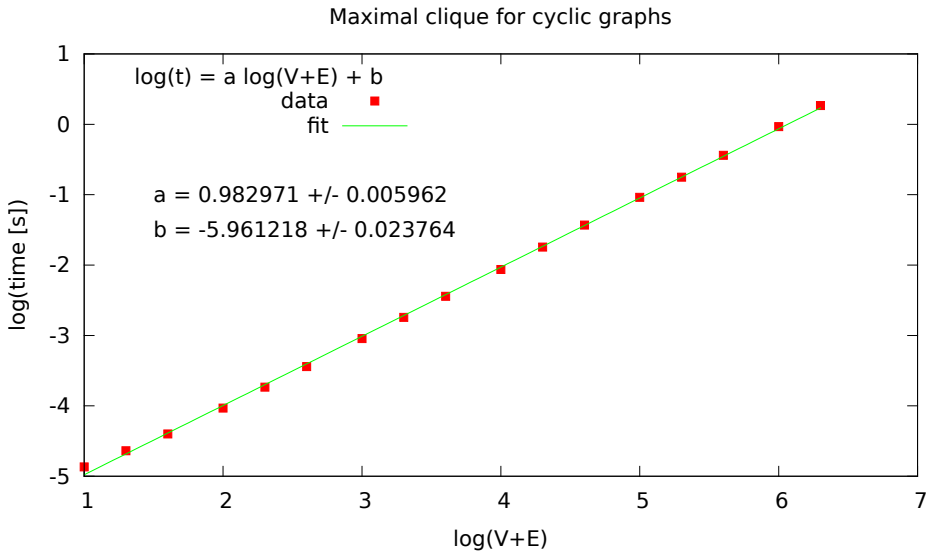
Rysunek A.2. Wykres wydajności algorytmu wyznaczania klik z trzema wierzchołkami dla grafu pełnego. Współczynnik a powyżej 3 sugeruje istnienie czynników, które pogarszają zależność teoretyczną $O(V^3)$.



Rysunek A.3. Wykres wydajności algorytmu wyznaczania klik z czterema wierzchołkami dla grafu pełnego. Współczynnik a powyżej 4 sugeruje istnienie czynników, które pogarszają zależność teoretyczną $O(V^4)$.



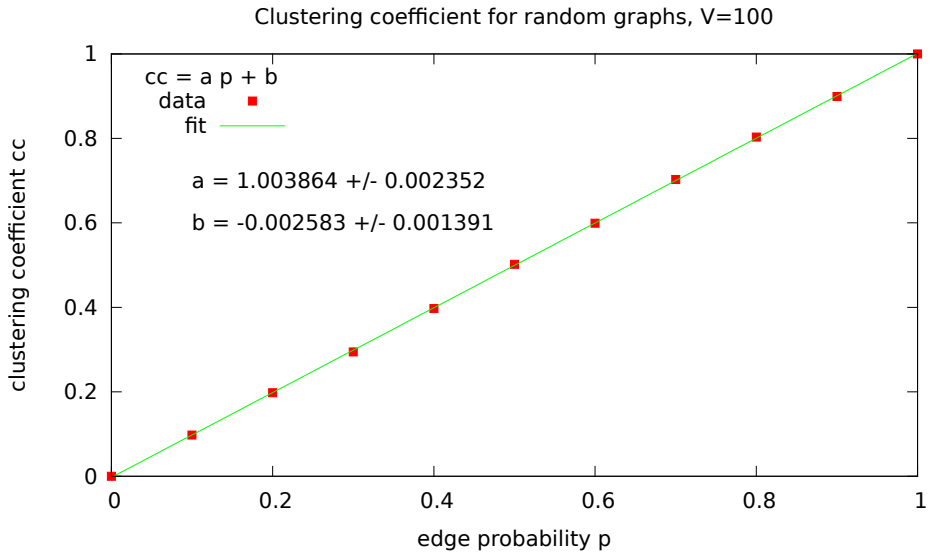
Rysunek A.4. Wykres wydajności algorytmu wyznaczania kliki maksymalnej dla grafu pełnego. Współczynnik a bliski 1 potwierdza zależność liniową.



Rysunek A.5. Wykres wydajności algorytmu wyznaczania kliki maksymalnej dla grafu cyklicznego. Współczynnik a bliski 1 potwierdza zależność liniową.

Tabela A.1. Średnia wielkość kliki maksymalnej wyznaczonej dla grafów przypadkowych z liczbą wierzchołków $n = 1000$. MC oznacza algorytm z modułu maximalclique, a US, RS, SF, LL to różne algorytmy zbiorów niezależnych zastosowane w module isetclique. BH oznacza algorytm z modułu isetramsey. Klika ma zawierać wierzchołek grafu o największym stopniu.

p	**MC**	**US**	**RS**	**SF**	**LL**	**BH**
0.1	3.8	3.8	3.7	4.0	2.7	5.0
0.2	5.2	5.0	5.1	5.6	3.8	6.1
0.3	6.4	6.1	6.5	6.6	5.2	7.7
0.4	7.7	7.7	7.7	8.3	6.5	9.1
0.5	9.5	9.7	9.9	10.9	9.7	11.3
0.6	12.8	12.4	12.4	12.9	12.0	14.6
0.7	17.4	16.5	16.3	18.3	16.0	19.0
0.8	24.1	24.5	25.7	26.3	25.5	27.4
0.9	45.5	44.9	44.9	49.0	48.7	48.8

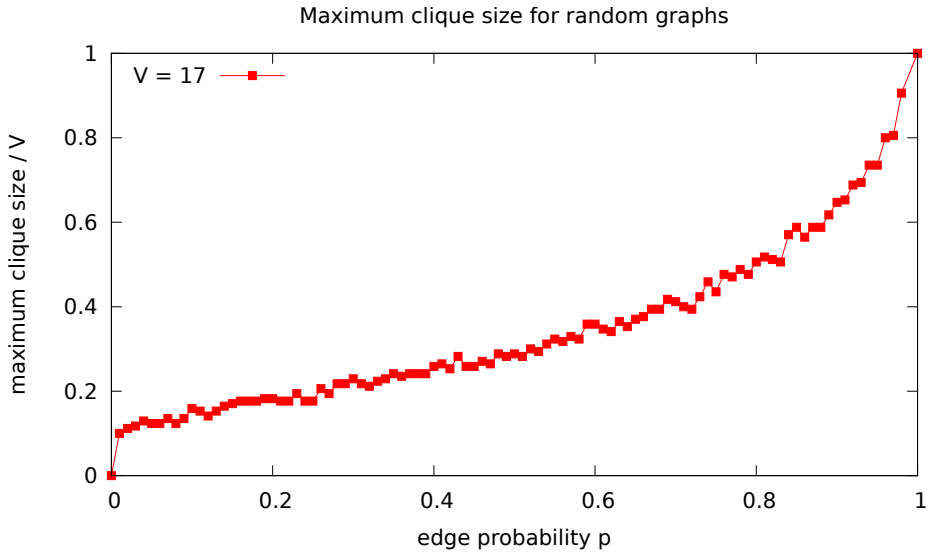


Rysunek A.6. Wykres zależności współczynnika klastrowania od prawdopodobieństwa p dla grafów przypadkowych z $|V| = 100$. Współczynnik a bliski 1 potwierdza zależność liniową.

Testy dla największej kliki. Dla p bliskiego zero największa klika to będzie krawędź (rozmiar kliki równy 2). Dla p zmierzającego do jedynki jest chyba szybki wzrost rozmiaru kliki do wielkości całego grafu (potrzebna jest większa rozdzielczość w pomiarach). Wyniki testu są przedstawione na rysunku A.7.

A.5. Testy algorytmu Brona-Kerboscha

Testy czterech wersji algorytmu Brona-Kerboscha dla różnych rodzajów grafów: grafy przypadkowe ($p = 0.5$), sieci trójkątne (kliki K_3), grafy pełne (kliki K_n), grafy cykliczne (kliki K_2). Wyniki testu są przedstawione w tabeli A.2. Dla grafów z małymi klikami K_2 lub K_3 czasy pracy różnych wersji są porównywalne, a wersja z uporządkowaniem degeneracji jest najwolniejsza. Dla grafów z dużymi klikami najszybsza jest wersja z punktem podziału o największym stopniu. Dla grafów pełnych wersje z punktem podziału zdecydowanie przewyższają inne wersje.



Rysunek A.7. Wykres zależności rozmiaru największej kliki od prawdopodobieństwa p dla grafów przypadkowych z $|V| = 17$. Rozmiar największej kliki rośnie monotonicznie od 2 do $|V|$.

Tabela A.2. Porównanie różnych wersji algorytmu Brona-Kerboscha: klasycznej (CBK), z przypadkowym punktem podziału (RPBK), z punktem podziału o największym stopniu (DPBK), z uporządkowaniem degeneracji (DEGBK). Czasy obliczeń są porównane z wersją klasyczną.

Graf	**CBK**	**RPBK**	**DPBK**	**DEGBK**
random $\|V\| = 100$	1.875s (100%)	1.206s (64.3%)	1.084s (57.8%)	1.850s (98.7%)
triangle $\|V\| = 25$	95.6ms (100%)	99.1ms (103.6%)	86.9ms (90.9%)	107.6ms (112.6%)
complete $\|V\| = 20$	5.295s (100%)	0.47ms (0.009%)	0.97ms (0.018%)	5.123s (96.8%)
cyclic $\|V\| = 10000$	7.312s (100%)	7.407s (101.3%)	7.440s (101.7%)	11.118s (152.0%)

Bibliografia

[1] Wikipedia, Clique problem, 2016, `https://en.wikipedia.org/wiki/Clique_problem`.

[2] Python Programming Language - Official Website, `https://www.python.org/`.

[3] Robin J. Wilson, *Wprowadzenie do teorii grafów*, Wydawnictwo Naukowe PWN, Warszawa 1998.

[4] Jacek Wojciechowski, Krzysztof Pieńkosz, *Grafy i sieci*, Wydawnictwo Naukowe PWN, Warszawa 2013.

[5] Narsingh Deo, *Teoria grafów i jej zastosowania w technice i informatyce*, PWN, Warszawa 1980.

[6] Wikipedia, Clique (graph theory), 2016, `https://en.wikipedia.org/wiki/Clique_(graph_theory)`.

[7] R. Duncan Luce, Albert D. Perry, *A method of matrix analysis of group structure*, Psychometrika 14, 95-116 (1949).

[8] Michael R. Garey, David S. Johnson, Larry Stockmeyer, *Some simplified NP-complete problems*, Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, 47-63 (1974).

[9] Wikipedia, Independent set (graph theory), 2016, `https://en.wikipedia.org/wiki/Independent_set_(graph_theory)`.

[10] N. Chiba, T. Nishizeki, *Arboricity and subgraph listing algorithms*, SIAM Journal on Computing 14, 210-223 (1985).

[11] Wikipedia, Planar graph, 2016, `https://en.wikipedia.org/wiki/Planar_graph`.

[12] Wikipedia, Perfect graph, 2016, `https://en.wikipedia.org/wiki/Perfect_graph`.

[13] Wikipedia, Interval graph, 2016, `https://en.wikipedia.org/wiki/Interval_graph`.

[14] Andrzej Kapanowski, graphs-dict, GitHub repository, 2015, `https://github.com/ufkapano/graphs-dict/`.

[15] Wikipedia, Power set, 2016, `https://en.wikipedia.org/wiki/Power_set`.

[16] Rosetta Code, Power set, 2016, `https://rosettacode.org/wiki/Power_set`.

[17] Jerzy Wałaszek, *Algorytmy, struktury danych. Znajdowanie klik w grafie*, 2016, `http://eduinf.waw.pl/inf/alg/001_search/0143.php`.

[18] Takao Nishizeki, *Planar Graph Problems*, Computing Supplementum 7, 53-68 (1990).

[19] Alon Itai, Michael Rodeh, *Finding a minimum circuit in a graph*, SIAM Journal on Computing 7, 413-423 (1978).

[20] Ravi Boppana, Magnus M. Halldórsson, *Approximating maximum independent sets by excluding subgraphs*, BIT Numerical Mathematics, 32(2), 180-196 (1992).

[21] David J. P. O'Sullivan, Gary J. O'Keeffe, Peter G. Fennell, James P. Gleeson,

Mathematical modeling of complex contagion on clustered networks, Frontiers in Physics 3, 71 (2015).
http://dx.doi.org/10.3389/fphy.2015.00071

[22] C. Bron, J. Kerbosch, *Algorithm 457: finding all cliques of an undirected graph*, Communications of the ACM 16 (9), 575-577 (1973).

[23] Wikipedia, Bron-Kerbosch algorithm, 2016, https://en.wikipedia.org/wiki/Bron-Kerbosch_algorithm.

[24] E. A. Akkoyunlu, *The enumeration of maximal cliques of large graphs*, SIAM Journal on Computing 2, 1-6 (1973).

[25] I. Koch, *Fundamental study: Enumerating all connected maximal common subgraphs in two graphs*, Theoretical Computer Science 250, 1-30 (2001).

[26] E. Tomita, A. Tanaka, H. Takahashi, *The worst-case time complexity for generating all maximal cliques and computational experiments*, Theoretical Computer Science 363, 28-42 (2006).

[27] F. Cazals, C. Karande, *A note on the problem of reporting maximal cliques*, Theoretical Computer Science 407, 564-568 (2008).

[28] Wikipedia, Degeneracy (graph theory), 2016, https://en.wikipedia.org/wiki/Degeneracy_(graph_theory).

[29] S. Tsukiyama, M. Ide, H. Ariyoshi, I. Shirawaka, *A new algorithm for generating all the maximal independent sets*, SIAM Journal on Computing 6, 505-517 (1977).

[30] K. Makino, T. Uno, *New Algorithms for Enumerating All Maximal Cliques*, 9th Scandinavian Workshop on Algorithm Theory (SWAT 2004), LNCS 3111, pp. 260-272, Springer-Verlag, 2004.