Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Bartłomiej Stachów

Nr albumu: 1177017

Problem kliki w teorii grafów

Praca magisterska na kierunku Informatyka stosowana

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Informatyki Stosowanej

Kraków 2026

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 promotorowi tej pracy, Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu, za poświęcony czas, cenne wskazówki oraz zaangażowanie w jej przygotowanie.

Streszczenie

W pracy przedstawiono analizę oraz implementację w języku Python wybranych algorytmów związanych z problemem kliki, a także omówiono rodziny grafów o właściwościach istotnych z punktu widzenia tego problemu.

Kliką w grafie nazywamy taki podzbiór wierzchołków grafu, w którym każda para wierzchołków jest połączona krawędzią. Problem kliki jest problemem NP-zupełnym dla grafów ogólnych, można go podzielić na kilka podproblemów, takich jak znajdowanie największej kliki w grafie, znajdowanie kliki o największej wadze w grafach z wagami wierzchołków, wyszukiwanie wszystkich klik maksymalnych oraz wyszukiwanie jednej lub wszystkich klik o zadanym rozmiarze.

Dla grafów przedziałowych i cięciwowych przygotowano iteratory po klikach maksymalnych oraz podano sposób wyznaczania kliki o największej wadze. Dla grafów permutacji i grafów kołowych zaimplementowano algorytmy wyznaczania kliki o największej wadze. Zaimplementowano dwa algorytmy Szwarcfitera-Barroso do wyznaczania wszystkich klik maksymalnych i osobno zliczania tych klik w grafach kołowych. Dla grafów łuków na okręgu przygotowano algorytm wielomianowy znajdujący największą klikę. Wszystkie algorytmy zostały także przetestowane pod kątem ich poprawności oraz wydajności.

Słowa kluczowe: grafy, problem kliki, grafy przedziałowe, grafy cięciwowe, grafy permutacji, grafy kołowe, grafy łuków na okręgu

English title: Clique problem in graph theory

Abstract

This paper presents an analysis and Python implementation of selected algorithms related to the clique problem. It also discusses families of graphs whose properties are relevant to this problem.

A clique in a graph is a subset of vertices in which every pair of vertices is connected by an edge. The clique problem is NP-complete for general graphs. The broader clique problem can be divided into several variants, including finding a maximum clique, finding a maximum-weight clique in vertex-weighted graphs, enumerating all maximal cliques, and finding one or all cliques of a given size.

For interval graphs and chordal graphs two iterators over all maximal cliques were prepared and the procedure for finding a maximum-weight clique is given. For permutation graphs and circle graphs two algorithms finding a maximum-weight clique were implemented. Additionally, two versions of the Szwarcfiter-Barroso algorithm were implemented, for counting cliques and for generating all maximal cliques. For circular-arc graphs a polynomial time algorithm finding a maximum clique is prepared. All algorithms were also tested with respect to their correctness and performance.

Keywords: graphs, clique problem, interval graphs, chordal graphs, permutation graphs, circle graphs, circular-arc graphs

Spis treści

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy jest problem kliki w teorii grafów [1]. **Kliką** nazywamy podzbiór C zbioru wierzchołków danego grafu taki, że każda para różnych wierzchołków ze zbioru C jest połączona krawędzią. Czasem przez klikę rozumie się cały podgraf pełny indukowany zbiorem C. W sieciach społecznościowych klika oznacza grupę osób znających się nawzajem. Kliki mają wiele innych zastosowań, które przedstawimy w dalszej części pracy. Klika **największa** to klika z największą liczbą wierzchołków, przy czym graf może mieć kilka klik największych. Klika **maksymalna** to klika, która nie jest podzbiorem innej kliki z większą liczbą wierzchołków.

Problem kliki to właściwie rodzina problemów, do której należą następujące zagadnienia:

— znajdowanie największej kliki w danym grafie (jednej z kilku możliwych),
— znajdowanie kliki o największej wadze w grafach z wagami wierzchołków, przy czym waga kliki to suma wag jej wierzchołków,
— wypisywanie wszystkich klik maksymalnych,
— wypisywanie jednej lub wszystkich klik o danym rozmiarze k.

Problem kliki jest problemem NP-zupełnym, czyli nie spodziewamy się znalezienia optymalnego algorytmu działającego w czasie wielomianowym dla ogólnych grafów. Stąd wynika między innymi zainteresowanie szczególnymi rodzinami grafów, dla których można znaleźć szybkie algorytmy działające w czasie wielomianowym. W literaturze polskiej jest sporo książek poświęconych częściowo lub w całości grafom. W celu ustalenia definicji i podstawowych faktów korzystano z kilku z nich: [2], [3], [4], [5]. Pozostałe informacje zaczerpnięto z literatury zagranicznej i artykułów źródłowych.

1.1. Cele pracy

Celem pracy jest opis i implementacja w języku Python wybranych algorytmów związanych z problemem kliki. Jest to kontynuacja badań rozpoczętych w pracy magisterskiej Dariusza Zdybskiego (2016) [6]. Badania zostały rozszerzone na szczególne rodziny grafów oraz na znajdowanie klik o największej wadze w grafach z wagami wierzchołków.

Do implementacji algorytmów wybrano język Python [7] ze względu na czytelną składnię i bogatą bibliotekę standardową. Implementacje powstałe w ramach niniejszej pracy uzupełnią bibliotekę algorytmów grafowych graphtheory [8], rozwijaną na Wydziale Fizyki, Astronomii i Informatyki Stosowanej Uniwersytetu Jagiellońskiego w Krakowie.

1.2. Zastosowania problemu kliki

Problem kliki znajduje zastosowanie w wielu dziedzinach nauki i techniki. Klikę można interpretować jako grupę elementów silnie ze sobą powiązanych, ponieważ każdy element należący do tej grupy pozostaje w określonej relacji z każdym innym jej elementem. W zależności od rozpatrywanego zagadnienia wierzchołki grafu mogą reprezentować między innymi osoby, obiekty biologiczne, fragmenty obrazów lub cząsteczki chemiczne, natomiast krawędzie opisują zachodzące między nimi relacje.

Jednym z najbardziej intuicyjnych obszarów zastosowań jest analiza sieci społecznych. W tym przypadku wierzchołki grafu reprezentują osoby lub użytkowników, a krawędzie oznaczają istniejące między nimi relacje. Klika odpowiada grupie, w której każda para osób jest ze sobą bezpośrednio powiązana na przykład przez znajomość lub współpracę. Kliki mogą być wykorzystywane do identyfikowania silnie połączonych grup. Ponadto społeczności mogą być wyznaczane jako zbiory nakładających się klik, dzięki czemu możliwe jest również uwzględnienie przynależności poszczególnych osób do wielu społeczności [9].

Innym zastosowaniem problemu kliki jest bioinformatyka i badania sieci białek. Gęsto połączone podgrafy w sieciach białek mogą odpowiadać kompleksom białkowym pełniącym wspólną funkcję. Program CFinder wykorzystuje metodę opartą na klikach do lokalizowania nakładających się modułów w sieciach biologicznych. Autorzy pokazują również możliwość wykorzystania takich modułów do przewidywania funkcji białek [10].

Problem kliki znajduje zastosowanie także w chemoinformatyce. Cząsteczki chemiczne można przedstawiać jako grafy, w których wierzchołki reprezentują atomy lub określone cechy cząsteczek. Podczas porównywania dwóch cząsteczek tworzy się graf potencjalnych dopasowań ich elementów. Krawędź oznacza, że dwa dopasowania są ze sobą zgodne, na przykład odpowiadające im odległości przestrzenne są dostatecznie podobne. Klika odpowiada wówczas zbiorowi wzajemnie zgodnych dopasowań. W programie CLIP algorytm Brona–Kerboscha został wykorzystany do przeszukiwania baz trójwymiarowych struktur chemicznych i identyfikowania cząsteczek mających duże wspólne fragmenty z cząsteczką wzorcową [11].

Problem kliki wykorzystywany jest również w rozpoznawaniu obrazów i dopasowywaniu struktur. W dwóch porównywanych obrazach wyznacza się charakterystyczne elementy, na przykład punkty, krawędzie lub fragmenty obiektów, a następnie tworzy się graf potencjalnych dopasowań. Każdy jego wierzchołek reprezentuje możliwe przyporządkowanie elementu pierwszego obrazu elementowi drugiego obrazu. Dwa wierzchołki są połączone krawędzią, jeżeli odpowiadające im dopasowania są wzajemnie zgodne, na przykład zachowują określone relacje przestrzenne lub geometryczne. Klika odpowiada zatem zbiorowi dopasowań, które mogą występować jednocześnie. Klika maksymalna reprezentuje dopasowanie, którego nie można rozszerzyć o kolejną zgodną parę elementów, natomiast największa klika odpowiada dopasowaniu obejmującemu największą liczbę elementów [12].

1.3. Organizacja pracy

Praca została podzielona na cztery rozdziały. Rozdział 1 zawiera wprowadzenie do tematu, opis celu pracy oraz przykładowe zastosowania problemu kliki. Rozdział 2 zawiera podstawowe definicje i twierdzenia z teorii grafów, problem liczności klik maksymalnych oraz przegląd rodzin grafowych pod kątem problem kliki. Rozdział 3 zawiera opis algorytmów zaimplementowanych w ramach tej pracy. W rozdziale 4 przedstawiono podsumowanie wyników pracy. Dodatek A zawiera opisy testów badanych algorytmów.

2. Teoria grafów

W tym rozdziale zostały przedstawione podstawowe definicje związane z grafami, które będą przydatne w dalszej części pracy.

2.1. Podstawowe definicje

Definicja: Grafem prostym nieskierowanym nazywamy uporządkowaną parę $G = (V(G), E(G))$, gdzie $V(G)$ jest niepustym zbiorem wierzchołków, natomiast $E(G)$ jest zbiorem krawędzi postaci $\{u, v\}$, gdzie $u, v \in V$ oraz $u \neq v$. Graf prosty nie zawiera pętli ani krawędzi wielokrotnych. Liczbę wierzchołków grafu oznaczamy przez $n = |V(G)|$, natomiast liczbę jego krawędzi przez $m = |E(G)|$ [3, 13].

Definicja: Dwa różne wierzchołki $u, v \in V(G)$ nazywamy sąsiednimi, jeżeli $\{u, v\} \in E(G)$. Sąsiedztwem otwartym wierzchołka v nazywamy zbiór $N(v) = \{u \in V(G) : \{u, v\} \in E(G)\}$, natomiast sąsiedztwem domkniętym wierzchołka v nazywamy zbiór $N[v] = N(v) \cup \{v\}$. Stopniem wierzchołka v, oznaczanym przez $\deg(v)$, nazywamy liczbę jego sąsiadów, czyli $\deg(v) = |N(v)|$. Wierzchołek stopnia zero nazywamy wierzchołkiem izolowanym [3, 4, 13].

Definicja: Graf H nazywamy podgrafem grafu G, jeżeli $V(H) \subseteq V(G)$ oraz $E(H) \subseteq E(G)$. Dla ustalonego zbioru $X \subseteq V(G)$ podgrafem indukowanym przez zbiór X nazywamy graf $G[X] = (X, E(X))$, gdzie $E(X) = \{\{u, v\} \in E(G) : u, v \in X\}$. Podgraf indukowany zawiera zatem wszystkie krawędzie grafu G, których oba końce należą do zbioru X [3, 13].

Definicja: Ścieżką nazywamy ciąg różnych wierzchołków $(v_0, v_1, \ldots, v_k)$, w którym każde dwa kolejne wierzchołki są połączone krawędzią. Długością ścieżki nazywamy liczbę jej krawędzi. [13]

Definicja: Cyklem długości $k \geqslant 3$ nazywamy ciąg $(v_0, v_1, \ldots, v_{k-1}, v_0)$, w którym wierzchołki $v_0, \ldots, v_{k-1}$ są różne, a każde dwa kolejne wierzchołki są połączone krawędzią [13].

Definicja: Graf nazywamy spójnym, jeżeli dla każdej pary jego wierzchołków istnieje łącząca je ścieżka.

Definicja: Grafem pełnym nazywamy graf, w którym każda para różnych wierzchołków jest połączona krawędzią.

Definicja: Dopełnieniem grafu prostego $G = (V, E)$ nazywamy graf G^c, który ma ten sam zbiór wierzchołków co graf G, a dwa różne wierzchołki są połączone krawędzią w G^c wtedy i tylko wtedy, gdy nie są połączone krawędzią w G [13].

Definicja: Grafem z wagami wierzchołków nazywamy graf, w którym każdemu wierzchołkowi przypisano dodatnią liczbę określającą jego wagę. Wagą kliki nazywamy sumę wag wszystkich należących do niej wierzchołków. Kliką o największej wadze nazywamy taką klikę, dla której suma ta jest największa spośród wszystkich klik występujących w grafie [14].

Definicja: Zbiorem niezależnym w grafie G nazywamy taki zbiór $I \subseteq V(G)$, że żadne dwa różne wierzchołki należące do I nie są połączone krawędzią. Zbiór niezależny nazywamy maksymalnym, jeżeli nie jest właściwym podzbiorem innego zbioru niezależnego. Zbiór niezależny nazywamy największym, jeżeli ma największą liczność spośród wszystkich zbiorów niezależnych w grafie [13].

2.2. Problem liczności klik maksymalnych

W pracy [15] zaproponowano ciekawy problem utworzenia grafu, w którym będą występowały kliki maksymalne o rozmiarach od 1 do k, gdzie izolowany wierzchołek tworzy klikę rozmiaru 1. W celu utworzenia takiego grafu z najmniejszą liczbą wierzchołków możemy zastosować metodę znaną z grafów cięciwowych. Zaczynamy od kliki K_k. Wybieramy klikę K_{k-2} i łączymy jej wierzchołki z nowym wierzchołkiem, aby powstała klika K_{k-1}. Wybieramy klikę K_{k-3} i łączymy jej wierzchołki z nowym wierzchołkiem, aby powstała klika K_{k-2}. Czynności powtarzamy aż do utworzenia kliki K_2. Na końcu dodajemy izolowany wierzchołek. Liczba wierzchołków końcowego grafu wynosi $n = 2k - 1$. Uogólniając możemy zapisać, że w grafie z n wierzchołkami mogą występować kliki maksymalne o kolejnych licznościach od 1 do $\lfloor (n+1)/2 \rfloor$.

2.3. Problem kliki

W tym rozdziale przedstawimy różne algorytmy dla problemu kliki obecne w bibliotece graphtheory oraz zaimplementowane w ramach tej pracy. W typowym przypadku algorytm jest reprezentowany przez klasę, której konstruktor otrzymuje graf i ewentualnie słownik z wagami wierzchołków. Metoda run() uruchamia obliczenia, a rozwiązanie jest zapisane w atrybutach takich jak cliques. W innych przypadkach metoda run() zwraca iterator po klikach maksymalnych.

2.3.1. Ogólne grafy

Do znajdowania wszystkich klik maksymalnych w ogólnych grafach wykorzystuje się algorytm Brona-Kerboscha, który w klasycznej postaci ma

złożoność obliczeniową $O(3^{n/3})$. W pracy magisterskiej Dariusza Zdybskiego przygotowano cztery wersje tego algorytmu, w których kliki maksymalne były zapisywane na liście w atrybucie cliques. Zwykle jednak nie jest potrzebna cała lista klik maksymalnych, bo zajmuje dużo pamięci komputera, tylko chcemy mieć możliwość przejścia kolejno po wszystkich klikach. Dlatego przygotowano drugie rozwiązanie oparte na iteratorach. Korzystanie z iteratorów przedstawia listing.

Listing 2.1. Algorytm Brona-Kerboscha.

```
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.cliques.bronkerbosch2 import BronKerboschClassicIterator
from graphtheory.cliques.bronkerboschrp2 import BronKerboschRandomPivotIterator
from graphtheory.cliques.bronkerboschdp2 import BronKerboschDegreePivotIterator
from graphtheory.cliques.bronkerboschdeg2 import BronKerboschDegeneracyIterator

G = Graph()
# Add nodes and edges here.
algorithm = BronKerboschClassicIterator(G)
#algorithm = BronKerboschRandomPivotIterator(G)
#algorithm = BronKerboschDegreePivotIterator(G)
#algorithm = BronKerboschDegeneracyIterator(G)
clique_iterator = algoritm.run()
for clique in clique_iterator:    # loop over maximal cliques
    print(clique)
```

W przypadku grafów z wagami wierzchołków można podać siłowe rozwiązanie problemu znajdowania kliki o największej wadze. Wystarczy przejść po wszystkich klikach maksymalnych za pomocą iteratora i wybrać klikę o największej wadze. Takie podejście może być sensowne jeżeli wiemy, że dla danej rodziny grafów liczba klik maksymalnych zależy liniowo lub wielomianowo od rozmiaru grafu. Przykładowy kod zamieszczono na listingu, przy czym wagi wierzchołków są przechowywane w słowniku weight_dict.

Listing 2.2. Korzystanie z ogólnego iteratora klik maksymalnych.

```
max_clique = max(clique_iterator, key=len)    # maximum clique
max_weight_clique = max(clique_iterator,
    key=lambda clique: sum(weight_dict[v] for v in clique))
```

2.3.2. Grafy dwudzielne i drzewa

W grafach dwudzielnych największa możliwa klika to K_2, czyli pojedyncza krawędź. Klikę o największej wadze możemy łatwo znaleźć w czasie $O(m)$ stosując kod podany na listingu.

Listing 2.3. Kliki maksymalne w grafach dwudzielnych.

```
# G is a bipartite graph.
max_weight_edge = max(G.iteredges(), key=lambda edge:
    weight_dict[edge.source] + weight_dict[edge.target]) # O(m) time
max_weight_clique = {max_weight_edge.source, max_weight_edge.target}
```

2.3.3. Grafy przedziałowe

Grafy przedziałowe wygodnie jest zapisywać w reprezentacji permutacyjnej. Każdy wierzchołek grafu (przedział na osi liczbowej) jest reprezentowany przez etykietę, zwykle liczbę całkowitą lub string. Graf ma postać permutacji etykiet wierzchołków. Etykieta wierzchołka pojawia się dwa razy w permutacji, pierwszy raz gdy napotykamy początek przedziału, drugi raz gdy napotykamy koniec przedziału. Reprezentacja permutacyjna automatycznie zapisuje przedziały tak, że są one posortowane wg początków i wg końców, co jest wykorzystywane w wielu algorytmach.

W bibliotece graphtheory były dostępne dwie przydatne funkcje związane z klikami. Funkcja find_peo_cliques() pozwala na wyznaczenie PEO i listy klik maksymalnych w czasie $O(n+m)$. Funkcja find_max_clique_size() pozwala na wyznaczenie rozmiaru największej kliki w czasie $O(n)$. W ramach niniejszej pracy przygotowano nową funkcję iter_cliques_interval(), która zwraca iterator po klikach maksymalnych działający w czasie $O(n+m)$. Jest to rozwiązanie oszczędzające pamięć. Co ciekawe, iterator można wykorzystać do znalezienia kliki o największej wadze w grafach przedziałowych z wagami wierzchołków. Złożoność obliczeniowa pozostaje rzędu $O(n+m)$. Dokładny opis algorytmu użytego w funkcji iter_cliques_interval() znajduje się w rozdziale 3.1, a testy wydajnościowe opisano w dodatku A.1. Na bazie funkcji wyznaczającej rozmiar największej kliki przygotowano szybką funkcję find_clique_number_interval() do znajdowania liczby klik maksymalnych w grafie przedziałowym. Funkcja działa w czasie $O(n)$.

Listing 2.4. Kliki maksymalne w grafach przedziałowych.

```
# 'perm' is an interval graph in permutation representation.
from graphtheory.chordality.intervaltools import find_peo_cliques
from graphtheory.chordality.intervaltools import find_max_clique_size
from graphtheory.chordality.intervaltools import find_clique_number_interval
from graphtheory.chordality.intervaltools import iter_cliques_interval

peo, clique_list = find_peo_cliques(perm)    # O(n+m) time
max_size = find_max_clique_size(perm)    # O(n) time
# New results.
max_clique = max(iter_cliques_interval(perm), key=len) # O(n+m) time
n_cliques = find_clique_number_interval(perm)    # O(n) time
clique_list = list(iter_cliques_interval(perm))
max_weight_clique = max(iter_cliques_interval(perm),
    key=lambda clique: sum(weight_dict[v] for v in clique)) # O(n+m) time
```

2.3.4. Grafy cięciwowe

Grafy cięciwowe były badane w pracy magisterskiej Małgorzaty Olak [16]. W grafach cięciwowych struktura maksymalnych klik jest wyjątkowo uporządkowana. Brak indukowanych cykli długości co najmniej cztery sprawia, że kliki nie pojawiają się „chaotycznie", lecz wynikają z lokalnych zależności, które można uchwycić jednym porządkiem wierzchołków. Graf cięciwowy posiada porządek doskonałej eliminacji PEO (ang. *perfect elimination ordering*), w którym dla każdego wierzchołka zbiór jego późniejszych sąsiadów

tworzy klikę. Zbiory te wyznaczają naturalne kliki-kandydatów, dzięki czemu największa klika może zostać znaleziona przez analizę tej rodziny klik generowanych przez porządek eliminacji, a nie przez globalne rozważanie wszystkich podzbiorów wierzchołków. Liczba klik maksymalnych w grafie cięciwowym jest rzędu $O(n)$.

Uporządkowanie PEO dla grafów cięciwowych można znaleźć w czasie liniowym $O(n+m)$ za pomocą leksykograficznego BFS lub algorytmem MCS (ang. *maximum cardinality search*). Na bazie PEO można znaleźć klikę największą w czasie $O(n+m)$. Na bazie PEO można również wyznaczyć wszystkie kliki maksymalne w czasie $O(n+m)$, ponieważ suma liczności klik maksymalnych w grafie cięciwowym jest ograniczona przez $n+m$. W ramach tej pracy został przygotowany iterator po klikach maksymalnych w grafie cięciwowym iter_cliques_chordal(), który pozwala wyznaczyć w czasie liniowym $O(n+m)$ klikę o największej wadze. Dokładny opis algorytmu użytego w funkcji iter_cliques_chordal() znajduje się w rozdziale 3.2, a testy wydajnościowe opisano w dodatku A.2.

Listing 2.5. Kliki maksymalne w grafach cięciwowych.

```
from graphtheory.chordality.peotools import find_peo_lex_bfs
from graphtheory.chordality.peotools import find_peo_mcs
from graphtheory.chordality.peotools import find_maximum_clique_peo
from graphtheory.chordality.peotools import find_all_maximal_cliques
from graphtheory.chordality.peotools import is_peo1, is_peo2
from graphtheory.chordality.peotools import iter_cliques_chordal

# G is a chordal graph, PEO can be found.
#peo = find_peo_lex_bfs(G)    # O(n+m) time
peo = find_peo_mcs(G)    # O(n+m) time
assert is_peo1(G, peo)    # testing PEO, O(n+m) time
assert is_peo2(G, peo)    # testing PEO, O(n+m) time

max_clique = find_maximum_clique_peo(G, peo)    # O(n+m) time
clique_list = find_all_maximal_cliques(G, peo)    # O(n+m) time
# New results.
max_clique = max(iter_cliques_chordal(G, peo), key=len) # O(n+m) time
clique_list = list(iter_cliques_chordal(G, peo))
max_weight_clique = max(iter_cliques_chordal(G, peo),
    key=lambda clique: sum(weight_dict[v] for v in clique)) # O(n+m) time
```

2.3.5. Grafy planarne

Graf planarny to graf, który może być narysowany na płaszczyźnie w taki sposób, że jego krawędzie nie przecinają się. Zgodnie z twierdzeniem Kuratowskiego w grafach planarnych nie mogą pojawiać się kliki K_5 i większe, więc największa możliwa klika to K_4.

W problemie z wagami wierzchołków należy zbadać wagi wszystkich klik K_2, K_3, K_4. Algorytm siłowy będzie miał złożoność $O(n^4)$.

2.3.6. Grafy permutacji

Graf permutacji jest to graf nieskierowany, którego wierzchołki reprezentują elementy permutacji, a krawędzie reprezentują pary elementów permu-

tacji tworzących inwersję [17]. Z definicji grafy te mają zwartą reprezentację jako ciąg liczb całkowitych od 1 do n odpowiednio przestawionych, ewentualnie w programach komputerowych jako ciąg liczb od 0 do $n-1$. Grafy permutacji były badane w pracy licencjackiej Alberta Surmacza [18]. Podano tam algorytm wyznaczania największej kliki, który szuka największej malejącej podsekwencji w permutacji. Rozwiązanie można znaleźć w czasie $O(n^2)$ (programowanie dynamiczne) lub $O(n \log n)$ (metoda dziel i zwyciężaj).

Nowy algorytm wyznaczania kliki o największej wadze w grafie permutacji zostanie pokazany w rozdziale 3.3.

2.3.7. Grafy kołowe

Graf kołowy (ang. *circle graph*) jest to graf nieskierowany, którego wierzchołki reprezentują cięciwy okręgu. Dwa wierzchołki są połączone krawędzią, jeżeli odpowiednie cięciwy przecinają się [19]. Każdy graf permutacji jest grafem kołowym, ale nie na odwrót. Grafy kołowe były badane w pracy magisterskiej Alberta Surmacza [20]. Graf kołowy można reprezentować przez podwójną permutację etykiet cięciw. Jeżeli będziemy obiegać okrąg od ustalonego punktu, to podwójna permutacja będzie zawierać etykiety napotykanych cięciw. Zakłada się, że dwie cięciwy nie mogą mieć wspólnego końca. Dla n cięciw podwójna permutacja zawiera $2n$ elementów.

W pracy Surmacza podano algorytm wyznaczania największej kliki w grafie kołowym. Dla każdego wierzchołka grafu indukuje się graf permutacji, dla którego następnie wyznacza się największą klikę. Możliwe są dwie implementacje o złożonościach $O(n^3)$ lub $O(n^2 \log n)$. Wykorzystując algorytm wyznaczania kliki o największej wadze w grafie permutacji możemy otrzymać algorytm wyznaczający klikę o największej wadze w grafie kołowym. Dokładny opis tego algorytmu znajduje się w rozdziale 3.4.

2.3.8. Grafy łuków na okręgu

Grafy łuków na okręgu były badane w pracy magisterskiej Oliwii Gil [21]. W takich grafach każdemu wierzchołkowi przypisany jest łuk leżący na okręgu, a dwa wierzchołki są połączone krawędzią wtedy i tylko wtedy, gdy odpowiadające im łuki mają punkt wspólny [22]. Grafy łuków na okręgu są uogólnieniem grafów przedziałowych. Każdy graf przedziałowy można przedstawić jako graf łuków na okręgu, ale nie każdy graf łuków na okręgu jest grafem przedziałowym. Klika w takim grafie odpowiada zbiorowi łuków, które parami się przecinają. Ta geometryczna interpretacja stanowi podstawę wielu algorytmów wyznaczania klik w grafach łuków na okręgu, w tym algorytmu Gavrila służącego do wyznaczania kliki największej [23]. W pracy Gil pojawiła się implementacja tego algorytmu, ale kluczowe obliczenia nie były wydajne. W rozdziale 3.6 przedstawiono optymalną wersję algorytmu, korzystającą z twierdzenia Königa.

W roku 1985 Hsu [14] podał algorytm działający w czasie $O(nm)$ do wyznaczania kliki o największej wadze. Z kolei w roku 1992 Kashiwabara ze współautorami podali algorytm do generowania klik maksymalnych w czasie $O(n^{3.5} + a)$, gdzie a to rozmiar wyjścia, czyli suma liczności klik maksymalnych [24].

3. Algorytmy

W tym rozdziale zostaną szczegółowo opisane nowe algorytmy zaimplementowane w ramach badania klik w różnych rodzinach grafów.

3.1. Wyznaczanie klik maksymalnych w grafach przedziałowych

W tym algorytmie skorzystaliśmy ze specyficznej struktury klik w grafach przedziałowych. Zgodnie z geometryczną interpretacją tej rodziny grafów, wierzchołki reprezentują przedziały na osi liczbowej, a krawędź występuje wtedy, gdy dwa przedziały mają niepuste przecięcie. Wobec tego każda klika odpowiada rodzinie przedziałów mających wspólny punkt przecięcia.

Posiadając reprezentację permutacyjną grafu przedziałowego, można efektywnie wyznaczać takie rodziny przez przejście po uporządkowanych końcach przedziałów i utrzymywanie zbioru aktualnie aktywnych wierzchołków. Zmiana z fazy powiększania tego zbioru na fazę jego zmniejszania odpowiada znalezieniu kolejnej kliki maksymalnej.

Dane wejściowe: Graf przedziałowy w reprezentacji permutacyjnej.

Problem: Wyznaczanie wszystkich klik maksymalnych w grafie przedziałowym.

Dane wyjściowe: Iterator po klikach maksymalnych.

Opis algorytmu: Algorytm rozpoczynamy od stworzenia zmiennej logicznej growing oraz zbioru used. Zmienna growing będzie przechowywać wartość True, gdy aktualna klika jest w fazie wzrostu, oraz False, gdy maksymalny rozmiar kliki został już osiągnięty. Zbiór used będzie przechowywać wierzchołki obecne w aktualnie budowanej klice. Ciałem algorytmu jest pętla **for** po wierzchołkach grafu w reprezentacji permutacyjnej. W każdej iteracji rozpatrujemy, czy dany wierzchołek v należy już do zbioru used:

— jeśli należy i flaga growing jest ustawiona na True, to zwracamy aktualną klikę maksymalną, usuwamy wierzchołek v ze zbioru used i ustawiamy flagę growing na False (klika maleje),
— jeśli nie należy, to go dodajemy wierzchołek v do zbioru used i ustawiamy flagę growing na True (rośnie nowa klika).

Algorytm jest wykonywany aż wszystkie wierzchołki z reprezentacji permutacyjnej zostaną rozpatrzone.

Złożoność: Prawie wszystkie operacje w pętli **for** zajmują stały czas, co prowadziłoby do złożoności $O(n)$. Jedynie raportowanie znalezionej kliki zajmuje czas proporcjonalny do liczności danej kliki. Okazuje się, że suma liczności klik maksymalnych w grafie przedziałowym nie przekracza $n+m$, co prowadzi do całkowitej złożoność czasowej algorytmu postaci $O(n+m)$.

Uwagi: Iterator po klikach maksymalnych w grafie przedziałowym można wykorzystać do znalezienia kliki o największej wadze w czasie $O(n+m)$.

Listing 3.1. Funkcja iter_cliques_interval().

```
def iter_cliques_interval(perm):    # O(n+m) time
    """Generate all maximal cliques on demand for the interval graph."""
    growing = True
    used = set()    # current clique
    for node in perm:
        if node in used:
            if growing:    # new maximal clique
                yield set(used)    # clique copy
            used.remove(node)
            growing = False
        else:    # clique is growing
            used.add(node)
            growing = True
```

3.2. Wyznaczanie klik maksymalnych w grafach cięciwowych

Grafy cięciwowe są klasą grafów, której struktura klik jest ściśle związana z porządkiem doskonałej eliminacji. W takim porządku późniejsi sąsiedzi każdego wierzchołka tworzą klikę, dlatego kliki maksymalne można wyznaczać przez rozważanie zbiorów złożonych z kolejnych wierzchołków oraz ich późniejszych sąsiadów. Przedstawiony algorytm wykorzystuje właśnie tę własność.

Dane wejściowe: Graf cięciwowy i jego PEO.

Problem: Wyznaczanie wszystkich klik maksymalnych w grafie cięciwowym.

Dane wyjściowe: Iterator po klikach maksymalnych.

Opis algorytmu: Na początku tworzony jest słownik M przypisujący każdemu wierzchołkowi jego pozycję w porządku PEO oraz słownik S, służący do eliminowania klik niemalejących maksymalnych.

Następnie algorytm przechodzi po kolejnych wierzchołkach source w porządku PEO. Dla każdego z nich wyznaczany jest zbiór X zawierający sąsiadów leżących na prawo od source w tym porządku. Z własności PEO wynika, że zbiór X jest kliką, zatem X $\cup$ {source} jest kandydatem na klikę maksymalną.

Jeżeli wierzchołek jest izolowany, algorytm zwraca jednoelementową klikę. Jeżeli zbiór X jest pusty, przechodzi do następnego wierzchołka. W pozostałych przypadkach wybierany jest najwcześniejszy wierzchołek ze zbioru X, a następnie aktualizowana jest wartość pomocnicza S. Jeśli warunek S[source] < **len**(X) jest spełniony, kandydat X ∪ {source} zostaje zwrócony jako kolejna maksymalna klika.

Złożoność: Suma liczności klik maksymalnych w grafie cięciwowym nie przekracza $n + m$, co prowadzi do całkowitej złożoności czasowej algorytmu postaci $O(n + m)$.

Uwagi: Iterator po klikach maksymalnych w grafie cięciwowym można wykorzystać do znalezienia kliki o największej wadze w czasie $O(n + m)$.

Listing 3.2. Funkcja iter_cliques_chordal().

```
def iter_cliques_chordal(graph, order):
    """Generate all maximal cliques on demand for a chordal graph."""
    # position of each node in the PEO
    M = dict((node, i) for (i, node) in enumerate(order))
    # additional values used to skip non-maximal cliques
    S = dict((node, 0) for node in order)

    for source in order:
        # later neighbors of source in the PEO
        X = set()
        # collect neighbors placed after source in the PEO
        for target in graph.iteradjacent(source):   # total O(E) time
            if M[source] < M[target]:
                X.add(target)
        # isolated vertex is a maximal clique
        if graph.degree(source) == 0:   # isolated node
            yield {source}
        # no later neighbors, so there is no clique to generate
        if not X:
            continue
        # first vertex from X in the PEO
        node = min(X, key=M.__getitem__)
        # remember the largest clique already seen for this node
        S[node] = max(S[node], len(X) - 1)
        # generate the clique only if it is maximal
        if S[source] < len(X):
            yield X | {source}
```

3.3. Wyznaczanie kliki o największej wadze w grafach permutacji

Korzystając z reprezentacji grafu permutacji jako grafu inwersji, problem kliki o największej wadze można sprowadzić do problemu znalezienia malejącego podciągu o największej sumie wag [25]. Jeżeli elementy permutacji tworzą podciąg malejący, to każda para tych elementów tworzy inwersję, a

więc odpowiadające im wierzchołki są parami połączone krawędziami. Taki podciąg odpowiada zatem klice w grafie permutacji.

Dane wejściowe: Graf permutacji reprezentowany przez permutację wierzchołków oraz słownik przypisujący wierzchołkom ich wagi.

Problem: Wyznaczanie kliki o największej wadze w grafie permutacji.

Dane wyjściowe: Lista wierzchołków tworzących klikę o największej wadze oraz suma wag tych wierzchołków.

Opis algorytmu: Algorytm wykorzystuje programowanie dynamiczne. Nie wyznacza wszystkich klik maksymalnych, lecz dla każdego elementu permutacji zapamiętuje najlepsze rozwiązanie kończące się na tym elemencie.

Niech dp[i] oznacza największą sumę wag malejącego podciągu kończącego się na elemencie perm[i]. Odpowiada to największej wadze kliki, której ostatnim elementem w reprezentacji permutacyjnej jest perm[i]. Dodatkowo tablica parent przechowuje poprzednika danego elementu w najlepszym rozwiązaniu, co pozwala później odtworzyć znalezioną klikę.

Na początku dla każdego indeksu i przyjmujemy, że najlepsza klika kończąca się na perm[i] składa się tylko z tego jednego wierzchołka. Wtedy wartość dp[i] jest równa wadze wierzchołka perm[i], a parent[i] ma wartość -1.

Następnie algorytm przechodzi po kolejnych elementach permutacji. Dla ustalonego elementu v = perm[i] sprawdzane są wszystkie wcześniejsze elementy u = perm[j], gdzie j < i. Jeżeli zachodzi warunek u > v, to elementy u oraz v tworzą inwersję, a więc odpowiadające im wierzchołki są połączone krawędzią. Oznacza to, że malejący podciąg kończący się na u można przedłużyć o element v.

Jeżeli suma dp[j] + weight[v] jest większa od aktualnej wartości dp[i], algorytm aktualizuje dp[i] oraz zapisuje j jako poprzednika elementu i w tablicy parent. Po przetworzeniu całej permutacji największa wartość w tablicy dp jest wagą szukanej kliki. Sama klika jest odtwarzana przez cofanie się po tablicy parent, zaczynając od indeksu o największej wartości dp.

Złożoność: Dla każdego elementu permutacji sprawdzane są wszystkie wcześniejsze elementy, dlatego złożoność czasowa algorytmu wynosi $O(n^2)$.

Uwagi: Przedstawiony algorytm nie enumeruje wszystkich klik maksymalnych. Wykorzystuje specjalną strukturę grafów permutacji, w których kliki odpowiadają malejącym podciągom permutacji. Dzięki temu możliwe jest bezpośrednie znalezienie kliki o największej wadze za pomocą programowania dynamicznego.

Listing 3.3. Funkcja find_maximum_weight_clique_perm().

```
def find_maximum_weight_clique_perm(perm, weight):
    """Finding a maximum weight clique in a perm graph in O(n^2) time."""
    n = len(perm)
    # preparing the dynamic programming table and parent pointers
    dp = [0] * n
```

```
    parent = [-1] * n

    for i in range(n):
        v = perm[i]
        dp[i] = weight[v]   # initial weight of the clique with only vertex v
        for j in range(i):
            u = perm[j]
            if u > v:    # creates inversion, so they can be in the same clique
                # update dp[i] if we can get a heavier clique
                if dp[j] + weight[v] > dp[i]:
                    dp[i] = dp[j] + weight[v]
                    parent[i] = j

    # finding the index of last element in the maximum weight clique
    best = max(range(n), key=lambda i: dp[i])
    clique = []
    i = best
    # reconstructing the clique from the parent pointers
    while i != -1:
        clique.append(perm[i])
        i = parent[i]

    clique.reverse()
    return clique, dp[best]
```

3.4. Wyznaczanie kliki o największej wadze w grafach kołowych

Na bazie algorytmu dla grafów permutacji przygotowano nowy algorytm wyznaczający klikę o największej wadze w grafie kołowym.

Dane wejściowe: Graf wejściowy jest zadany podobnie jak w przypadku grafów przedziałowych w postaci listy zawierającej etykiety wierzchołków - pierwsze wystąpienie każdej etykiety oznacza początek cięciwy, a drugie wystąpienie oznacza jej koniec.

Problem: Wyznaczanie kliki o największej wadze w grafie kołowym.

Dane wyjściowe: Lista wierzchołków tworzących klikę o największej wadze oraz suma wag tych wierzchołków.

Opis algorytmu: Na początku algorytm wyznacza pozycje obu wystąpień każdej etykiety w reprezentacji grafu. Następnie rozważa kolejno każdy wierzchołek grafu. Dla ustalonego wierzchołka wyznaczane jest jego domknięte sąsiedztwo, czyli zbiór złożony z tego wierzchołka oraz wszystkich jego sąsiadów. Z oryginalnej reprezentacji grafu kołowego wybierane są następnie tylko te wystąpienia etykiet, które należą do wyznaczonego zbioru. W ten sposób otrzymywany jest podgraf indukowany przez domknięte sąsiedztwo danego wierzchołka.

Otrzymany podgraf jest przekształcany do reprezentacji grafu permutacji. Po przepisaniu wag na nowe etykiety uruchamiany jest algorytm wyznaczający klikę o największej wadze w grafie permutacji. Znaleziona klika jest następnie dekodowana z powrotem do oryginalnych etykiet grafu kołowego. Jeżeli jej waga jest większa od najlepszego dotychczasowego wyniku, zostaje ona zapamiętana.

Poprawność algorytmu wynika z faktu, że każda klika zawierająca wierzchołek v jest zawarta w domkniętym sąsiedztwie tego wierzchołka. Algorytm rozważa domknięte sąsiedztwo każdego wierzchołka grafu, dlatego w jednym z kroków analizowany jest podgraf zawierający klikę optymalną. Ponieważ dla każdego takiego podgrafu rozwiązywany jest problem kliki o największej wadze, najlepszy wynik spośród wszystkich kroków jest rozwiązaniem dla całego grafu kołowego.

Złożoność: Dla każdego wierzchołka algorytm wyznacza jego domknięte sąsiedztwo, buduje odpowiedni podgraf, przekształca go do grafu permutacji oraz wywołuje algorytm dla ważonej kliki w grafie permutacji. W przypadku pesymistycznym rozważany podgraf może mieć rozmiar rzędu n, a algorytm dla grafów permutacji działa w czasie $O(n^2)$. Ponieważ procedura wykonywana jest dla każdego z n wierzchołków, całkowita złożoność czasowa algorytmu wynosi $O(n \cdot n^2) = O(n^3)$.

Listing 3.4. Funkcja find_maximum_weight_clique_circle().

```
#!/usr/bin/env python3

from circle4 import circle2perm
from lds1 import find_maximum_clique_perm
from lis3 import find_maximum_iset_perm
from perm1 import find_maximum_weight_clique_perm

def find_maximum_weight_clique_circle(double_perm, weight_dict):
    """Finding a maximum weight clique in a circle graph in O(n^3) time."""
    # Musze miec info o sasiadach.
    nodes = set(double_perm)
    pairs = dict((node, []) for node in nodes)    # O(n) time
    for idx, node in enumerate(double_perm):    # O(n) time
        pairs[node].append(idx)
    # Dla kazdego v budujemy indukowany perm graph G_v.
    best_clique = set()    # max clique
    best_weight = 0    # max weight
    for source in nodes:    # O(n) time
        neighbors = set([source])
        for target in nodes:    # O(n) time
            s1, s2 = pairs[source]
            t1, t2 = pairs[target]
            if (s1 < t1 < s2 < t2) or (t1 < s1 < t2 < s2):
                neighbors.add(target)
        # Buduje graf indukowany jako double perm.
        subgraph = [node for node in double_perm if node in neighbors]
        perm, n2l, l2n = circle2perm(subgraph)    # O(n) time
        new_dict = dict((i, weight_dict[n2l[i]]) for i in perm) # O(n) time
        clique, max_weight = find_maximum_weight_clique_perm(perm, new_dict)
        # Dekodowanie kliki w czasie O(n) - generator.
```

```
        if best_weight < max_weight:
            best_weight = max_weight
            # przejscie do etykiet cieciw
            best_clique = set(n2l[i] for i in clique)
    return best_clique, best_weight
```

3.5. Algorytm Szwarcfitera-Barroso dla grafów kołowych - wyznaczanie klik maksymalnych

W oparciu o algorytm Szwarcfitera-Barroso [26] przygotowano algorytm wyznaczający wszystkie kliki maksymalne w grafie kołowym. W tym algorytmie również wykorzystano podwójną permutację jako reprezentację wejściowego grafu.

Dane wejściowe: Graf kołowy w reprezentacji podwójnej permutacji.

Problem: Wyznaczanie klik maksymalnych w grafie kołowym.

Dane wyjściowe: Iterator po klikach maksymalnych grafu kołowego.

Opis algorytmu: Zasadniczo algorytm można podzielić na cztery etapy:
1. Zbudowanie orientacji S_1.
2. Wyznaczenie redukcji tranzytywnej.
3. Znalezienie par krańców klik maksymalnych.
4. Wykonanie nieograniczonego przeszukiwania DFS w redukcji tranzytywnej między parami krańcowymi.

W pierwszym etapie algorytmu dla każdej etykiety wierzchołka zapisujemy pozycje jej pierwszego i drugiego wystąpienia w podwójnej permutacji. W oparciu o te pozycje budujemy orientację S_1, czyli skierowaną wersję grafu kołowego. Dla każdej pary wierzchołków sprawdzane jest, czy ich wystąpienia w podwójnej permutacji się przeplatają. Jeżeli dla wierzchołków s i t zachodzi układ $s_1 < t_1 < s_2 < t_2$, to do grafu skierowanego dodawana jest krawędź (s, t). Natomiast jeżeli zachodzi $t_1 < s_1 < t_2 < s_2$, dodawana jest krawędź (t, s). Oznacza to, że dla przecinających się cięciw kierunek krawędzi jest ustalany od wierzchołka, którego pierwsze wystąpienie pojawia się wcześniej, do wierzchołka, którego pierwsze wystąpienie pojawia się później.

Drugi etap algorytmu polega na wyznaczeniu redukcji tranzytywnej. Redukcja tranzytywna jest to graf skierowany, który zawiera krawędzie z orientacji S_1, ale bez krawędzi, które można wywnioskować z innych krawędzi z relacji tranzytywności. Dla każdej krawędzi (u, v) sprawdzane jest, czy istnieje wierzchołek x, taki że istnieją krawędzie (u, x) oraz (x, v). W tym celu wykorzystywane są zbiory następników $A_{out}(u)$ oraz poprzedników $A_{in}(v)$. Jeżeli przecięcie $A_{out}(u) \cap A_{in}(v)$ jest puste, krawędź (u, v) zostaje zachowana. W przeciwnym przypadku istnieje wierzchołek pośredni x, dlatego krawędź (u, v) jest pomijana.

W trzecim etapie algorytmu, bazując na orientacji S_1, wyznaczane są pary krańców klik maksymalnych. Należy podkreślić, że w tym kroku algorytm

operuje na pełnej orientacji S_1, a nie na grafie po redukcji tranzytywnej uzyskanym w poprzednim etapie. Dla każdej krawędzi (u, v) sprawdzane jest, czy wierzchołki u i v mają wspólnego następnika lub wspólnego poprzednika.

Jeżeli istnieje wspólny następnik x, to klikę zawierającą u i v można rozszerzyć w prawo, dodając wierzchołek x. Jeżeli istnieje wspólny poprzednik x, to klikę można rozszerzyć w lewo. W obu przypadkach krawędź (u, v) nie może wyznaczać krańców kliki maksymalnej.

Z tego powodu para (u, v) jest uznawana za parę krańcową tylko wtedy, gdy spełnione są oba warunki:

$$A_{out}(u) \cap A_{out}(v) = \emptyset \quad \text{oraz} \quad A_{in}(u) \cap A_{in}(v) = \emptyset.$$

Oznacza to, że nie istnieje żaden wierzchołek, który pozwalałby rozszerzyć rozważaną parę ani przed u, ani za v. Takie pary są zapisywane jako możliwe początki i końce klik maksymalnych.

W ostatnim etapie wykonywane jest zmodyfikowane przeszukiwanie DFS w grafie będącym redukcją tranzytywną orientacji S_1. W odróżnieniu od klasycznego DFS celem tego przeszukiwania nie jest jedynie odwiedzenie wierzchołków grafu, lecz wygenerowanie wszystkich ścieżek prowadzących od danego wierzchołka początkowego do jednego z odpowiadających mu końców krańcowych. Przeszukiwanie uruchamiane jest dla każdego wierzchołka grafu, jednak dla wierzchołków, które nie są początkiem żadnej pary krańcowej, kończy się natychmiast. Dla wierzchołka s, który jest początkiem co najmniej jednej pary krańcowej, algorytm wykonuje DFS od s, przyjmując jako zbiór celów wszystkie wierzchołki zapisane w ends[s]. Każda ścieżka w redukcji tranzytywnej prowadząca od s do jednego z tych celów odpowiada jednej klice maksymalnej i jest zwracana jako zbiór wierzchołków.

Złożoność: Niech n oznacza liczbę wierzchołków grafu, m liczbę krawędzi w orientacji S_1, natomiast a liczbę klik maksymalnych zwracanych przez algorytm. Wyznaczenie pozycji wystąpień etykiet w podwójnej permutacji wymaga czasu $O(n)$, a zbudowanie orientacji S_1 czasu $O(n^2)$.

Wyznaczenie redukcji tranzytywnej wymaga przejrzenia wszystkich krawędzi orientacji S_1. Dla każdej krawędzi sprawdzane jest przecięcie odpowiednich zbiorów poprzedników i następników. Każdy z tych zbiorów może w najgorszym przypadku zawierać $O(n)$ wierzchołków, dlatego sprawdzenie jednej krawędzi może wymagać czasu $O(n)$. Ponieważ takich krawędzi jest m, cały etap ma złożoność $O(nm)$. Analogicznie, wyznaczenie par krańcowych klik maksymalnych również wymaga czasu $O(nm)$.

W ostatnim etapie algorytm wykonuje zmodyfikowane przeszukiwanie DFS w redukcji tranzytywnej i zwraca znalezione ścieżki jako kliki maksymalne. Ponieważ każda zwracana klika może zawierać co najwyżej n wierzchołków, koszt wypisania wszystkich klik wynosi $O(na)$.

Całkowita złożoność czasowa przedstawionej implementacji wynosi zatem $O(n^2 + nm + na)$, czyli $O(n^2 + n(m + a))$. W klasycznej analizie algorytmu Szwarcfitera-Barroso złożoność zapisuje się krócej jako $O(n(m+a))$, ponieważ składnik $O(nm)$ oraz $O(na)$ obejmują zasadniczy koszt działania algorytmu. W szczególności, gdy dla grafu spójnego $m + a \geqslant n$, składnik $O(n^2)$ jest zdominowany przez $O(n(m + a))$.

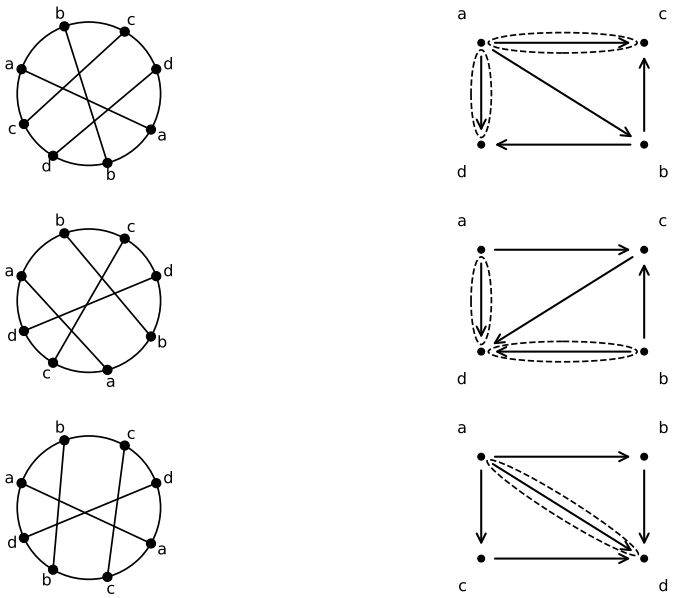
Uwagi: W pewnych sytuacjach może nas interesować tylko liczba klik maksymalnych, a nie dokładna postać tych klik. Okazuje się, że zmodyfikowany algorytm Szwarcfitera-Barroso pozwala wyznaczyć liczbę klik maksymalnych w czasie $O(nm)$. W ostatnim kroku pierwotnego algorytmu nie możemy przechodzić z użyciem zmodyfikowanego DFS po wszystkich ścieżkach w redukcji tranzytywnej, bo złożoność obliczeniowa pozostanie $O(n(m+a))$. Zamiast tego należy przejść przez wszystkie wierzchołki w kolejności odwrotnej do sortowania topologicznego redukcji tranzytywnej w czasie $O(n)$. Podczas tego przejścia oblicza się rekurencyjną funkcję $\alpha(u)$ dla każdego wierzchołka u. Funkcja zlicza rozgałęzienia ścieżek w danym wierzchołku. Suma wartości tej funkcji dla wszystkich wierzchołków daje liczbę klik maksymalnych.

Biorąc pod uwagę fakt, że każdy graf permutacji jest jednocześnie grafem kołowym, algorytm Szwarcfitera-Barroso może być użyty do wyznaczania klik maksymalnych także w tej rodzinie grafów. Należy jedynie dokonać odpowiedniego przekształcenia reprezentacji grafu permutacji do podwójnej permutacji grafu kołowego:

Listing 3.5. Przekształcenie grafu permutacji do grafu kołowego.

```
double_perm = perm2circle(single_perm)
```

Różne reprezentacje tego samego grafu kołowego mogą prowadzić do różnych orientacji S_1 oraz różnych redukcji tranzytywnych. Przykładowe przypadki przedstawiono na rysunku poniżej. Model geometryczny z cięciwami jest taki sam, ale różne są punkty na okręgu, od których rozpoczyna się zapis etykiet cięciw w podwójnej permutacji.



Rysunek 3.1. Różne reprezentacje tego samego grafu kołowego oraz odpowiadające im orientacje S_1. Krawędzie wyróżnione przerywanymi elipsami są usuwane w wyniku redukcji tranzytywnej.

Listing 3.6. Klasa SzwarcfiterBarroso tworząca iterator po klikach maksymalnych grafu kołowego.

```
class SzwarcfiterBarroso:
    """Finding maximal cliques of a circle graph."""

    def __init__(self, perm):
        self.perm = perm
        self.nodes = set(self.perm)    # O(n) time, usuwam duplikaty
        self.dag = Graph(directed=True)    # S1-orientation
        self.transitive_reduction = Graph(directed=True)
        for node in self.nodes:
            self.dag.add_node(node)
            self.transitive_reduction.add_node(node)
        # Grupowanie koncow maximal edges
        self.ends = dict((node, set()) for node in self.nodes)    # O(n) time
        # Zwiekszam limit rekurencji.
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(len(self.nodes) * 2, recursionlimit))

    def run(self):
        self.find_S1_orientation()
        self.find_transitive_reduction()
        self.find_maximal_edges()
        yield from self.find_maximal_cliques()

    def find_S1_orientation(self):    # O(n^2) time
        # dict do zapisu pierwszego i drugiego wystapienia wierzcholka w perm.
        pairs = dict((node, []) for node in self.nodes)    # O(n) time
        for idx, node in enumerate(self.perm):    # O(n) time
            pairs[node].append(idx)
        assert all(len(pairs[node]) == 2 for node in pairs)

        # Budowanie grafu skierowanego (S1-orientation).
        for (s,t) in itertools.combinations(self.nodes, 2):    # O(n^2) time
            s1, s2 = pairs[s]
            t1, t2 = pairs[t]
            if (s1 < t1 < s2 < t2):
                self.dag.add_edge(Edge(s,t))
            if (t1 < s1 < t2 < s2):
                self.dag.add_edge(Edge(t,s))
        #self.dag.show()

    def find_transitive_reduction(self):    # podobno O(n m) time
        # poczatki krawedzi wchodzacych do node
        self.Ain = dict((node, set()) for node in self.nodes)
        # konce krawedzi wychodzacych z node
        self.Aout = dict((node, set()) for node in self.nodes)

        for edge in self.dag.iteredges():    # O(m) time
            self.Ain[edge.target].add(edge.source)
            self.Aout[edge.source].add(edge.target)

        for edge in self.dag.iteredges():    # total O(n m) time
            if len(self.Aout[edge.source] & self.Ain[edge.target]) == 0:
                self.transitive_reduction.add_edge(edge)
        #print("transitive reduction ...")
```

```
        #self.transitive_reduction.show()

    def find_maximal_edges(self):    # total O(n m) time
        for edge in self.dag.iteredges():    # chyba O(n m) time przy hashowaniu
            if (len(self.Aout[edge.source] & self.Aout[edge.target]) == 0 and
                len(self.Ain[edge.source] & self.Ain[edge.target]) == 0):
                # Tu zapisuje maximal edges.
                self.ends[edge.source].add(edge.target)

    def find_maximal_cliques(self):
        # Dla kazdego wierzcholka s robie DFS.
        # Dla jednego wierzcholka moze byc kilka startujacych maximal edges.
        # Sciezki koncza sie na koncach maximal edges.
        # Zlozonosc bedzie O(n m) plus wypisywanie klik O(n a).

        # Teraz moge robic DFS jeden raz z jednego wierzcholka.
        self.order = []    # s-t path items
        for node in self.nodes:
            yield from self.dfs_visit(node, self.ends[node])
            assert self.order == []    # na koncu wyczyszczone

    def dfs_visit(self, source, target_set):
        """Modified DFS from source in transitive reduction."""
        if len(target_set) == 0:
            #print("empty target set for", source)
            return
        self.order.append(source)
        for node in self.transitive_reduction.iteradjacent(source):
```

3.6. Wyznaczanie kliki największej w grafach łuków na okręgu

W ramach pracy udało się przeprowadzić modyfikację implementacji algorytmu Gavrila przygotowanej przez Oliwię Gil [21]. Zachowano ogólny schemat działania pierwotnej implementacji, natomiast zmieniono sposób wyznaczania największego zbioru niezależnego w dopełnieniu rozpatrywanego podgrafu. Zastosowany wcześniej algorytm z nawrotami, przeznaczony dla grafów ogólnych, zastąpiono algorytmem wykorzystującym dwudzielność tego dopełnienia.

Dane wejściowe: Dowolny graf łuków na okręgu w reprezentacji w postaci listy par zdarzeń.

Problem: Wyznaczanie kliki największej w grafie kołowym.

Dane wyjściowe: Lista zawierająca wszystkie łuki grafu należące do jego największej kliki.

Opis algorytmu: W pierwotnej implementacji, aby wyznaczyć największy zbiór niezależny w dopełnieniu grafu M_i, wykorzystywany był algorytm z nawrotami przeznaczony dla grafów ogólnych. Nie uwzględniał on faktu, że

dopełnienie grafu M_i jest grafem dwudzielnym. W zmodyfikowanej wersji zastosowano algorytm przeznaczony specjalnie dla tej klasy grafów.

Listing 3.7. Wyznaczanie największego zbioru niezależnego - algorytm z nawrotami.

```
algorithm = BacktrackingIndependentSet(M_i_complement)
algorithm.run()
clique_i = algorithm.independent_set
```

Listing 3.8. Wyznaczanie największego zbioru niezależnego - algorytm dla grafów dwudzielnych.

```
clique_i = max_independent_set_bipartite(M_i_complement)
```

W nowej implentacji wyznaczania zbioru niezależnego najpierw za pomocą algorytmu Hopcrofta–Karpa wyznaczane jest maksymalne skojarzenie w grafie M_i^c. Następnie wykonywane jest przejście po ścieżkach alternujących, rozpoczynających się w nieskojarzonych wierzchołkach lewej części grafu. W ten sposób wyznaczane są zbiory osiągalnych wierzchołków Z_L i Z_R należących odpowiednio do lewej i prawej części grafu.

Na ich podstawie konstruowane jest minimalne pokrycie wierzchołkowe:

$$C = (L \setminus Z_L) \cup Z_R,$$

gdzie L i R oznaczają części grafu dwudzielnego. Zgodnie z twierdzeniem Kőniga otrzymane pokrycie jest minimalne [13, s. 37, tw. 2.1.1]. Jego dopełnienie w zbiorze wierzchołków

$$V(M_i^c) \setminus C$$

jest największym zbiorem niezależnym w grafie M_i^c, a więc odpowiada największej klice w grafie M_i.

Pozostałe etapy implementacji, obejmujące konstruowanie grafów M_i, przywracanie pierwotnej numeracji wierzchołków oraz wybór największej spośród otrzymanych klik, nie zostały zmienione.

Złożoność: W zmodyfikowanej wersji etap wyznaczania zbioru niezależnego ma złożoność wielomianową. Dla grafu o n wierzchołkach i m krawędziach algorytm Hopcrofta–Karpa działa w czasie $O(m\sqrt{n})$, natomiast wyznaczenie minimalnego pokrycia wierzchołkowego i jego dopełnienia wymaga czasu $O(n+m)$. Złożoność zmienionego etapu jest więc równa $O(m\sqrt{n})$. Pełna złożoność obliczeniowa całego algorytmu wyznaczania największej kliki szacowana jest na $O(mn^{1.5}) = O(n^{3.5})$. Co ciekawe, sam Gavril postulował złożoność $O(n^3)$ i to się potwierdziło w testach praktycznych.

Uwagi: Ciekawym elementem algorytmu jest wyznaczanie najmniejszego pokrycia wierzchołkowego na bazie największego skojarzenia w grafie dwudzielnym. Ścieżki alternujące są wyznaczane równolegle przy użyciu kolejki wierzchołków, podobnie do przeszukiwania BFS.

Listing 3.9. Funkcja max_independent_set_bipartite() wyznaczająca największy zbiór niezależny w grafie dwudzielnym, wywoływana w funkcji znajdującej największą klikę w grafie łuków max_clique_gavril().

```
from collections import deque
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.bipartiteness.hopcroftkarp import HopcroftKarpSet

def max_clique_gavril(graph):
    size = len(graph)
    max_clique = []

    for i in range(size):
        X_i, Y_i = get_X_Y(graph, size, i)
        M_i_arcs = X_i + Y_i
        M_i_model = [graph[node] for node in M_i_arcs]
        M_i = make_abstract_arc_graph_efficient(M_i_model, size)
        M_i_complement = M_i.complement()
        clique_i = max_independent_set_bipartite( M_i_complement)
        clique_i_old_numbering = [M_i_arcs[node] for node in clique_i]

        if len(clique_i_old_numbering) > len(max_clique):
            max_clique = clique_i_old_numbering
    return max_clique

def max_independent_set_bipartite(graph):
    """Find a maximum independent set in a bipartite graph."""
    matching = HopcroftKarpSet(graph)
    matching.run()
    left_part = matching.v1

    reachable_left = set(node for node in left_part
        if matching.mate[node] is None)
    reachable_right = set()

    queue = deque(reachable_left)

    while queue:
        left = queue.popleft()
        for right in graph.iteradjacent(left):
            if matching.mate[left] == right:
                continue
            if right in reachable_right:
                continue

            reachable_right.add(right)
            matched_left = matching.mate[right]

            if (matched_left is not None
                and matched_left not in reachable_left):
                reachable_left.add(matched_left)
                queue.append(matched_left)

    minimum_vertex_cover = (
        (left_part - reachable_left) | reachable_right)
    return set(graph.iternodes()) - minimum_vertex_cover
```

4. Podsumowanie

Badania w niniejszej pracy koncentrowały się na trzech zagadnieniach związanych z klikami. Pierwsze zagadnienie to znajdowanie kliki o największej liczności. To było na ogół badane w innych pracach dyplomowych, więc dokonaliśmy tylko przeglądu istniejących narzędzi i wyników.

Drugie zagadnienie to znajdowanie wszystkich klik maksymalnych w wybranych rodzinach grafów, takich jak grafy przedziałowe, grafy cięciwowe, grafy permutacji. Na etapie implementacji w języku Python odpowiednim narzędziem jest iterator po klikach maksymalnych, który na żądanie zwraca kolejne kliki maksymalne. Zwykle przetwarza się kliki sekwencyjnie i nie należy marnować pamięci komputera na przechowywanie całej listy klik. Przygotowano szereg iteratorów po klikach maksymalnych i przetestowano ich praktyczną złożoność obliczeniową.

Trzecim zagadnieniem było znajdowanie kliki o największej wadze w grafach z wagami wierzchołków. W szczególnym przypadku równych wag wierzchołków problem redukuje się do znajdowania kliki o największej liczności, jednak zwykle jest to trudniejsze zadanie. W przypadku grafów dwudzielnych, grafów przedziałowych i grafów cięciwowych pokazano, że można w czasie liniowym $O(n+m)$ znaleźć klikę o największej wadze przez prosty przegląd klik maksymalnych. W przypadku grafów permutacji potrzebny był osobny algorytm wykorzystujący programowanie dynamiczne. Ten algorytm stał się podstawą algorytmu wyznaczającego klikę o największej wadze w grafach kołowych.

Podczas badania problemu kliki można poczynić ciekawą obserwację. W pewnych przypadkach znalezienie liczby klik maksymalnych może być szybsze niż faktyczne jawne wygenerowanie tych klik jako podzbiorów zbioru wierzchołków danego grafu. Dla grafów przedziałowych wystarczy powstrzymać się przed wypisaniem klik i już złożoność spadnie z $O(n+m)$ do $O(n)$. Dla grafów kołowych generowanie klik zajmuje czas $O(n(m+a))$, co może dać czas rosnący wykładniczo z rozmiarem grafu, bo liczba klik maksymalnych a może rosnąć wykładniczo. Jednak można użyć innego algorytmu, aby dostać liczbę klik maksymalnych w czasie wielomianowym $O(nm)$. Podane przykłady są prawdopodobnie jednymi z powodów, dla których teoria grafów od lat cieszy się ogromnym zainteresowaniem badaczy.

Inna ciekawa obserwacja dotyczy niejednoznaczności wyboru modelu geometrycznego dla danego grafu abstrakcyjnego w różnych rodzinach grafów. Dla grafów przedziałowych przykładem może być graf pełny K_n modelowany przez n przedziałów na osi liczbowej mających niepuste przecięcie. Mamy dowolność w wyborze kolejności początków przedziałów, a następnie dowolność w wyborze kolejności końców przedziałów. Analogiczna sytuacja zachodzi dla grafów łuków na okręgu, chociaż tam problem jest głębszy. Przykładowo

graf pełny K_3 można narysować jako zbiór trzech łuków, gdzie na okręgu nie będzie punktu pokrytego przez trzy łuki. Dla grafów permutacji wiadomo, że pewne grafy abstrakcyjne można reprezentować przez kilka różnych permutacji. Wreszcie dla grafów kołowych wiadomo, że pewne grafy abstrakcyjne mają kilka różnych modeli geometrycznych z cięciwami na okręgu. Z kolei dany model geometryczny można zapisać na kilka sposobów jako podwójną permutację etykiet cięciw. Oczywiście końcowe wyniki nie zależą od wyboru konkretnego modelu.

A. Testy algorytmów

A.1. Testy wyznaczania klik maksymalnych w grafach przedziałowych

Algorytm iter_cliques_interval() został przetestowany na różnych rodzajach grafów przedziałowych o różnych rozmiarach. Testy polegały na przeprowadzeniu pięciu pomiarów czasu dziesięciu kolejnych uruchomień algorytmu dla każdego rozmiaru grafu, a następnie obliczeniu mediany z tych pięciu pomiarów. Grafy zostały wygenerowane za pomocą następujących funkcji należących do pakietu graphtheory:

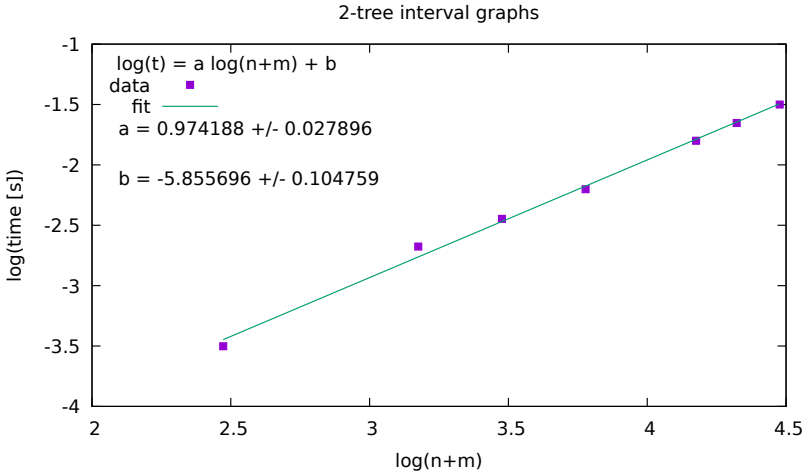
— random_interval_graph(),
— complete_interval_graph(),
— path_interval_graph(),
— 2tree_interval_graph(),
— star_interval_graph(),
— ktree_interval_graph().

Wyniki testów przedstawione na wykresach A.1, A.2, A.3, A.4, A.5 pokazują, że zarówno w grafach rzadkich jak i gęstych wartość współczynnika dopasowania a jest bliska 1, co wskazuje na liniową zależność czasu działania algorytmu $O(n+m)$. Wyjątkiem są grafy pełne, dla których uzyskano współczynnik $a = 0.542(18)$. W tym przypadku niska wartość współczynnika wynika z faktu, że w grafie pełnym będziemy mieli jedną klikę maksymalną złożoną ze wszystkich wierzchołków, co powoduje uproszczenie algorytmu poprzez ograniczenie liczby operacji generowania klik.

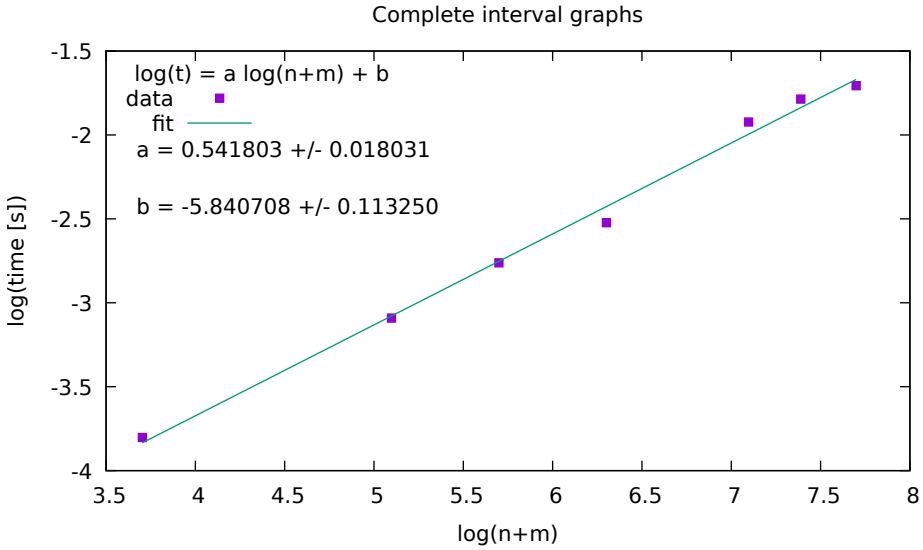
A.2. Testy wyznaczania klik maksymalnych w grafach cięciwowych

Algorytm iter_cliques_chordal() został przetestowany na różnych rodzajach grafów cięciwowych o różnych rozmiarach. Porządek eliminacji doskonałej (PEO) był wyznaczany przed rozpoczęciem pomiaru, dlatego jego czas nie został uwzględniony w wynikach testów. Grafy zostały wygenerowane za pomocą funkcji make_random_chordal() oraz make_random_ktree() należących do pakietu graphtheory.

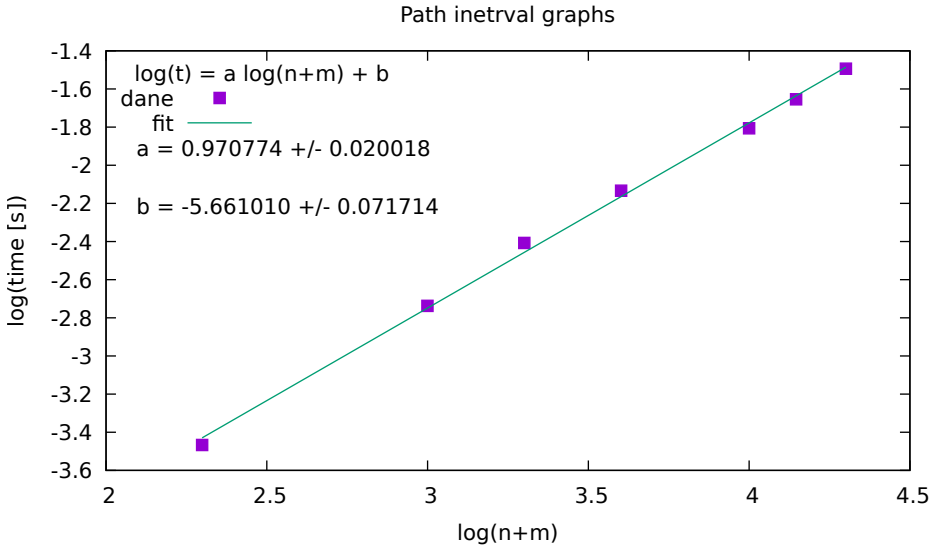
Wyniki testów z rysunków A.6, A.7, A.8 pokazują, że zarówno dla grafów rzadkich jak i gęstych (również z ważonymi wierzchołkami) wartość współczynnika dopasowania a pozostaje bliska 1, co potwierdza liniową zależność czasu działania algorytmu od rozmiaru wejścia $O(n+m)$.



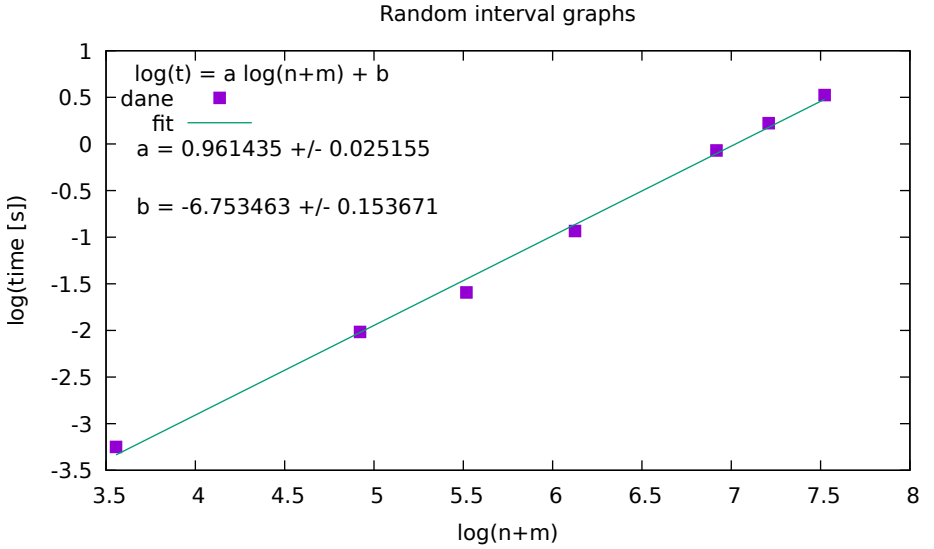
Rysunek A.1. Generowanie klik maksymalnych dla 2-drzew przedziałowych. Wartość współczynnika dopasowania $a = 0.974(28)$ potwierdza liniową złożoność obliczeniową algorytmu $O(n + m)$.



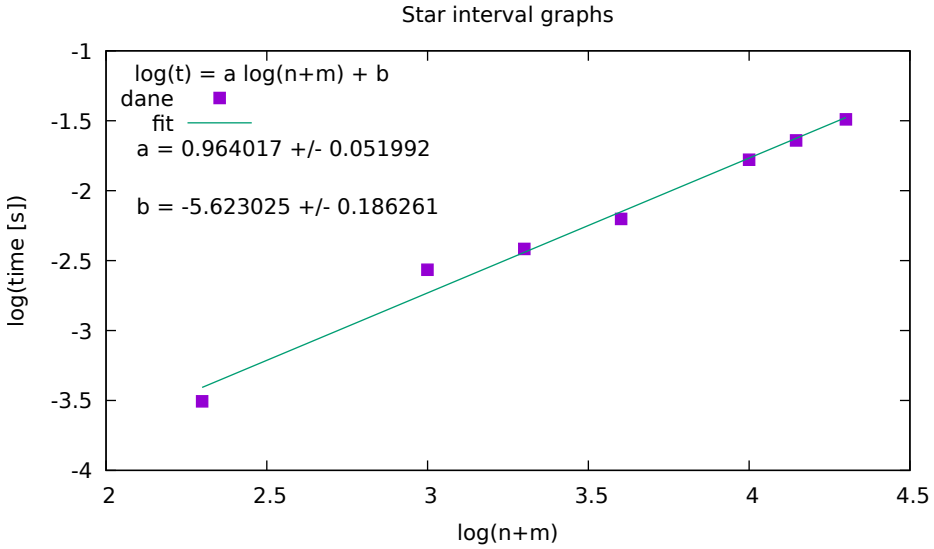
Rysunek A.2. Generowanie klik maksymalnych dla grafów pełnych K_n. Wartość współczynnika dopasowania $a = 0.542(18)$ potwierdza liniową złożoność obliczeniową algorytmu $O(n + m)$.



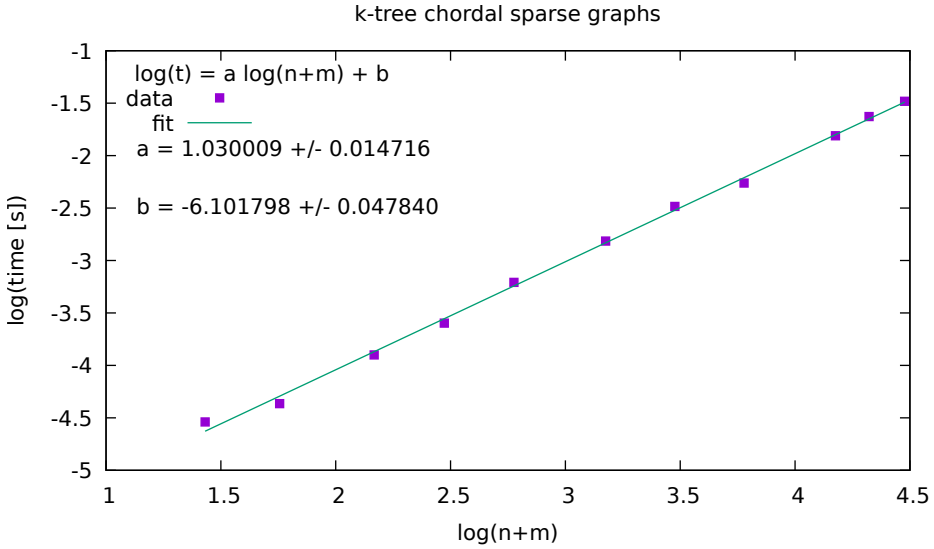
Rysunek A.3. Generowanie klik maksymalnych dla grafu ścieżki P_n. Wartość współczynnika dopasowania $a = 0.97(2)$ potwierdza liniową złożoność obliczeniową algorytmu $O(n + m)$.



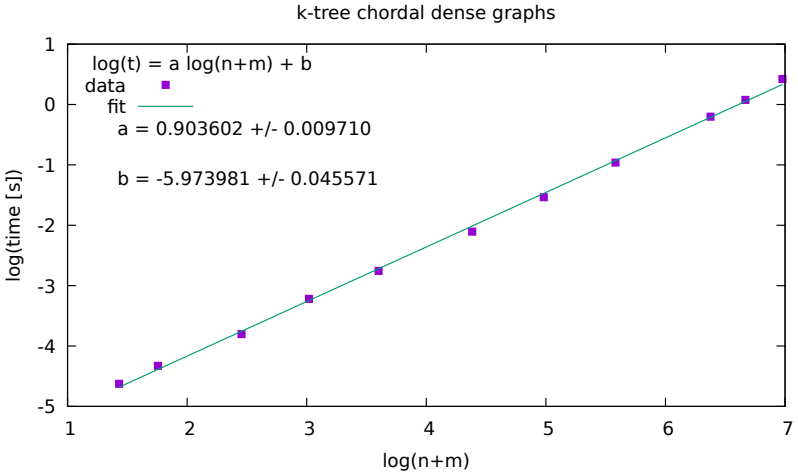
Rysunek A.4. Generowanie klik maksymalnych dla przypadkowych grafów przedziałowych. Wartość współczynnika dopasowania $a = 0.961(25)$ potwierdza liniową złożoność obliczeniową algorytmu $O(n + m)$.



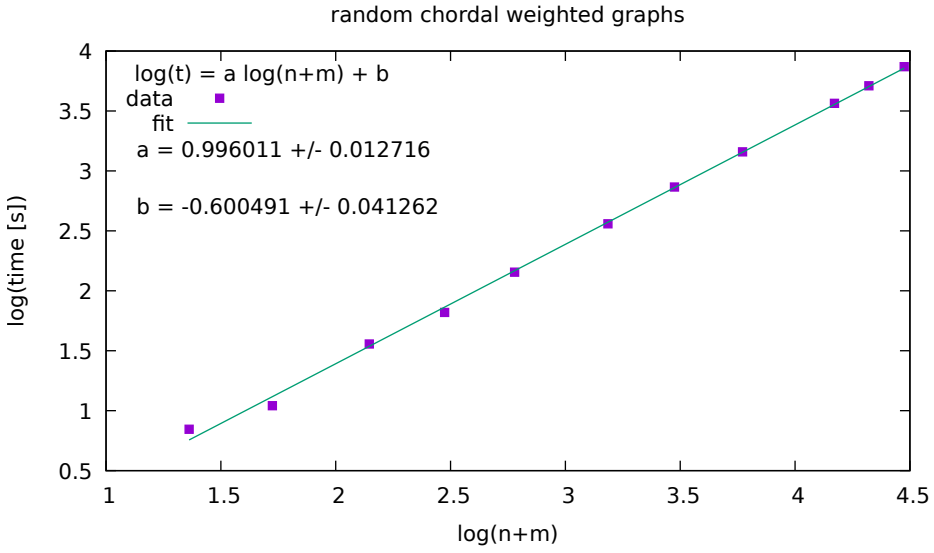
Rysunek A.5. Generowanie klik maksymalnych dla grafów typu gwiazda $K_{1,n-1}$. Wartość współczynnika dopasowania $a = 0.964(52)$ potwierdza liniową złożoność obliczeniową algorytmu $O(n+m)$.



Rysunek A.6. Generowanie klik maksymalnych dla grafów cięciwowych rzadkich. Wartość współczynnika dopasowania $a = 1.030(15)$ potwierdza złożoność obliczeniową algorytmu $O(n+m)$.



Rysunek A.7. Generowanie klik maksymalnych dla grafów cięciwowych gęstych. Wartość współczynnika dopasowania $a = 0.904(10)$ potwierdza złożoność obliczeniową algorytmu $O(n + m)$.



Rysunek A.8. Generowanie klik maksymalnych dla grafów cięciwowych losowych z ważonymi wierzchołkami. Wartość współczynnika dopasowania $a = 0.996(13)$ potwierdza złożoność obliczeniową algorytmu $O(n + m)$.

A.3. Testy wyznaczania kliki o największej wadze w grafach permutacji

Algorytm max_weight_clique_permutation() został przetestowany na wybranych rodzinach grafów permutacji reprezentujących przypadki rzadkie, gęste oraz losowe. Dla każdej permutacji wierzchołkom przypisano losowe wagi całkowite. Generowanie permutacji oraz przypisanie wag wykonano przed rozpoczęciem pomiaru, dlatego czas tych operacji nie został uwzględniony w wynikach testów.

Testy wykonano dla trzech rodzin grafów permutacji: grafów ścieżkowych, pełnych grafów dwudzielnych oraz grafów losowych. Grafy ścieżkowe stanowiły przypadek rzadki, pełne grafy dwudzielne przypadek gęsty, natomiast grafy losowe przypadek o mniej regularnej strukturze. Permutacje zostały wygenerowane za pomocą funkcji make_path_perm(), make_bipartite_perm() oraz make_random_perm(). Dla każdego rozmiaru n generowano jedną permutację danego typu oraz jeden zestaw wag wierzchołków. Następnie dla tego samego zestawu danych wykonywano pięć niezależnych pomiarów czasu działania funkcji max_weight_clique_permutation(). Jako wynik dla danego rozmiaru przyjmowano medianę uzyskanych czasów.

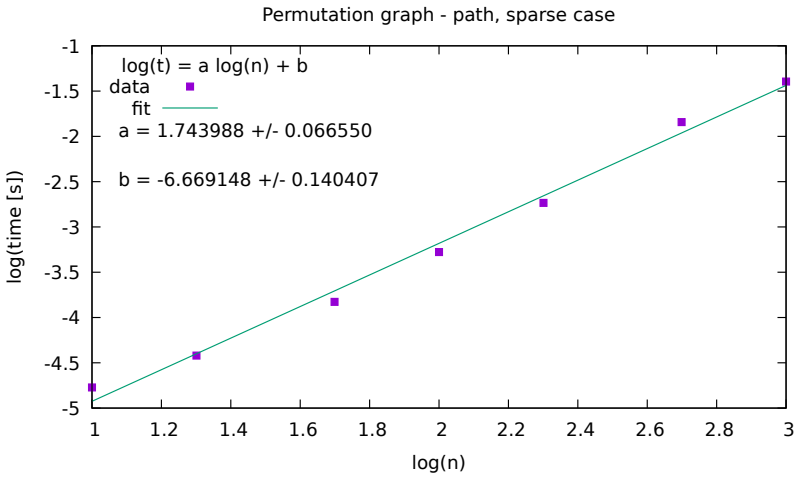
W celu zapewnienia powtarzalności eksperymentu generator liczb losowych inicjalizowano stałą wartością ziarna. Testy wykonano dla rozmiarów $n = 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 7000, 10000$.

Algorytm wykorzystuje programowanie dynamiczne i zawiera dwie zagnieżdżone pętle po elementach permutacji, dlatego jego oczekiwana złożoność czasowa wynosi $O(n^2)$. Otrzymane wyniki, przedstawione na wykresach A.9, A.10, A.11, są zgodne z tą analizą. Dla permutacji losowych uzyskano współczynnik $a = 1.944(27)$, a dla pełnych grafów dwudzielnych $a = 1.930(25)$, czyli wartości bliskie teoretycznemu wykładnikowi 2. Dla grafów ścieżkowych współczynnik jest niższy i wynosi $a = 1.744(67)$, co może wynikać z bardziej regularnej struktury danych oraz krótszego zakresu pomiarów. Nie zmienia to jednak pesymistycznej złożoności algorytmu, ponieważ liczba sprawdzanych par elementów permutacji nadal rośnie kwadratowo względem liczby wierzchołków.

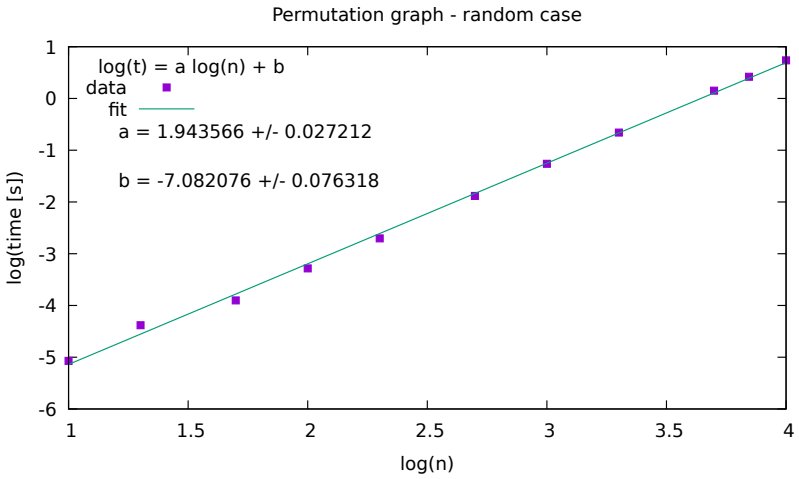
A.4. Testy wyznaczania kliki o największej wadze w grafach kołowych

Algorytm find_maximum_weight_clique_circle() przetestowano eksperymentalnie dla kilku klas grafów kołowych o różnej strukturze. Celem testów było porównanie czasu działania algorytmu dla grafów rzadkich, gęstych oraz grafów generowanych losowo. Każdemu wierzchołkowi przypisano dodatnią wagę całkowitą. Zarówno stworzenie grafu, jak i przydzielenie wag zostały wykonane przed rozpoczęciem właściwego pomiaru, dlatego operacje te nie wpływają na zapisane czasy działania.

Do testów wybrano trzy rodziny grafów kołowych: grafy losowe, grafy pełne oraz grafy cykliczne. Grafy pełne odpowiadały przypadkowi gęstemu. Grafy cykliczne wykorzystano jako przykład grafów rzadkich. Grafy loso-



Rysunek A.9. Wyznaczanie kliki o największej wadze dla grafów permutacji odpowiadających grafom ścieżkowym. Wartość współczynnika dopasowania $a = 1.744(67)$ jest niższa niż w innych przypadkach, ale potwierdza złożoność $O(n^2)$.



Rysunek A.10. Wyznaczanie kliki o największej wadze dla losowych grafów permutacji. Wartość współczynnika dopasowania $a = 1.944(27)$ jest bliska wartości 2, co potwierdza kwadratowy charakter czasu działania algorytmu $O(n^2)$.

we pełniły rolę przypadku pośredniego, pozbawionego tak regularnej struktury jak grafy pełne lub cykliczne. Do generowania grafów użyto funkcji make_random_circle(), make_complete_circle() oraz make_cycle_circle().

Dla ustalonego rozmiaru n tworzono jeden graf danej rodziny oraz jeden odpowiadający mu zestaw wag wierzchołków. Na tak przygotowanych danych wykonywano następnie pięć pomiarów czasu działania algorytmu find_maximum_weight_clique_circle(). Do dalszej analizy przyjęto medianę z pięciu otrzymanych wartości, co ograniczyło wpływ pojedynczych odchyleń czasowych.

Losowanie wag wykonywano z ustalonym ziarnem generatora liczb pseudolosowych, zależnym od rozmiaru grafu. Dzięki temu eksperyment był powtarzalny, a dla tego samego rozmiaru grafu otrzymywano ten sam zestaw wag. Pomiary przeprowadzono dla rozmiarów $n = 10, 20, 50, 100, 200, 500, 1000$.

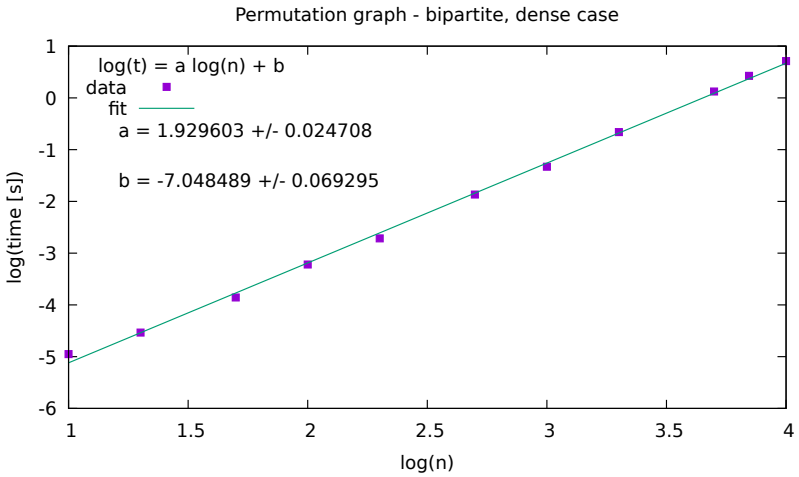
Otrzymane wyniki, przedstawione na wykresach A.12, A.13, A.14, są zgodne z teoretycznym oszacowaniem złożoności czasowej algorytmu, która wynosi $O(n^3)$. Największy współczynnik uzyskano dla grafów pełnych, dla których $a = 2.786(43)$. Wartość ta jest bliska teoretycznemu wykładnikowi 3, co jest naturalne, ponieważ w grafie pełnym domknięte sąsiedztwo każdego wierzchołka obejmuje cały graf. Dla grafów losowych otrzymano $a = 2.171(76)$, czyli wartość pośrednią między przypadkiem rzadkim i gęstym. Dla grafów cyklicznych współczynnik wyniósł $a = 1.618(60)$. Niższa wartość wynika z rzadkiej struktury tych grafów: każdy wierzchołek cyklu ma tylko dwóch sąsiadów, przez co pomocnicze grafy permutacji budowane dla kolejnych wierzchołków mają mały rozmiar.

A.5. Testy algorytmu Szwarcfitera–Barroso do wyznaczania wszystkich klik maksymalnych w grafach kołowych

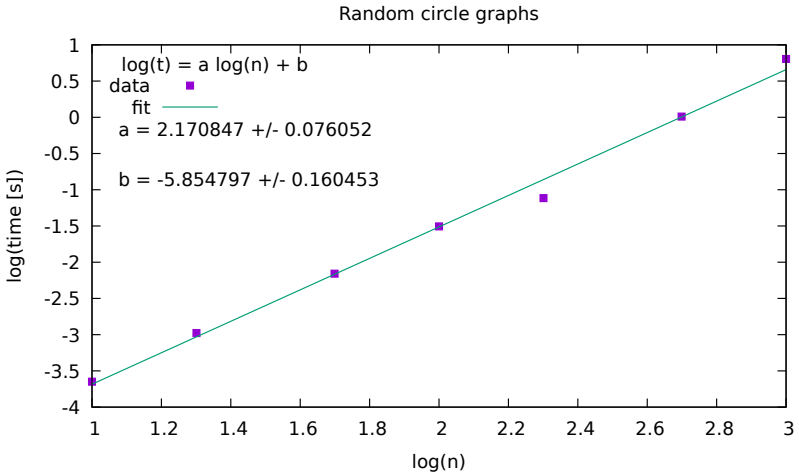
Algorytm przetestowano eksperymentalnie dla kilku rodzin grafów kołowych o różnej strukturze. Celem testów było zbadanie czasu potrzebnego do wyznaczenia wszystkich klik maksymalnych oraz porównanie zachowania algorytmu dla grafów rzadkich, gęstych i generowanych losowo.

Do testów wybrano trzy rodziny grafów kołowych: grafy losowe, grafy cykliczne oraz gęste k-drzewa, w których dla grafu o n wierzchołkach przyjmowano $k = \lfloor n/2 \rfloor$. Grafy cykliczne wykorzystano jako przykład grafów rzadkich. Gęste k-drzewa stanowiły przypadek regularnych grafów o kwadratowej liczbie krawędzi. Grafy losowe pełniły natomiast rolę przypadku nieregularnego. Do generowania grafów użyto funkcji make_random_circle(), make_ktree_circle() oraz make_cycle_circle().

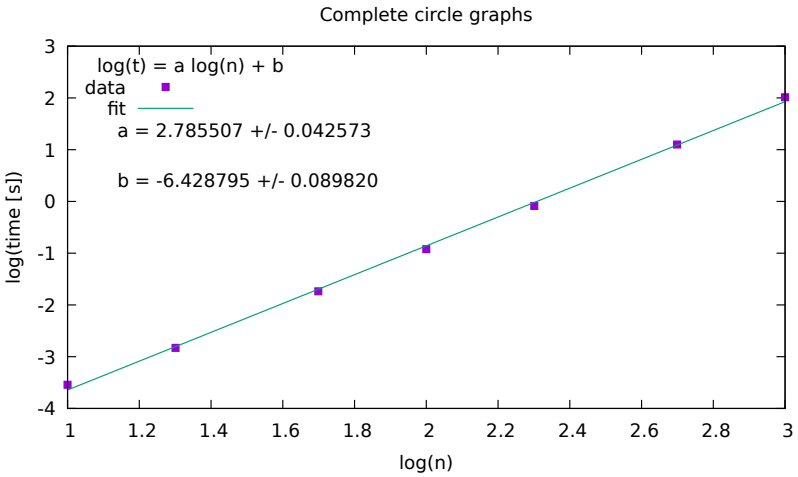
Algorytm został zaimplementowany w klasie SzwarcfiterBarroso, a metoda run() zwraca iterator kolejnych klik maksymalnych. Podczas każdego pomiaru tworzono nowy obiekt algorytmu, a następnie przechodzono przez cały zwrócony iterator. Utworzenie reprezentacji wejściowej grafu i obliczenie liczby jego krawędzi wykonywano przed rozpoczęciem pomiaru.



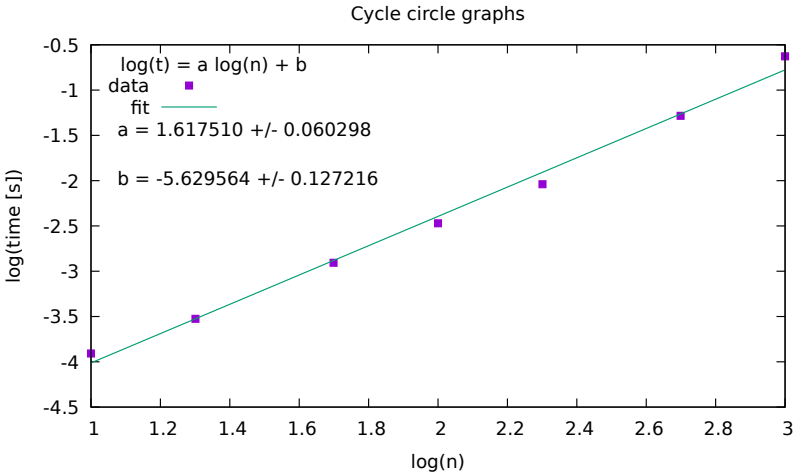
Rysunek A.11. Wyznaczanie kliki o największej wadze dla grafów permutacji odpowiadających pełnym grafom dwudzielnym. Wartość współczynnika dopasowania $a = 1.930(25)$ jest bliska wartości 2, co jest zgodne z teoretyczną złożonością czasową algorytmu $O(n^2)$.



Rysunek A.12. Wyznaczanie kliki o największej wadze dla losowych grafów kołowych. Wartość współczynnika dopasowania $a = 2.171(76)$ wskazuje na wzrost czasu działania wolniejszy niż w przypadku grafów pełnych, ale szybszy niż dla grafów cyklicznych.



Rysunek A.13. Wyznaczanie kliki o największej wadze dla pełnych grafów kołowych. Wartość współczynnika dopasowania $a = 2.786(43)$ jest bliska wartości 3, co jest zgodne z teoretyczną złożonością czasową algorytmu $O(n^3)$.



Rysunek A.14. Wyznaczanie kliki o największej wadze dla cyklicznych grafów kołowych. Wartość współczynnika dopasowania $a = 1.618(60)$ jest niższa niż w pozostałych przypadkach, co wynika z rzadkiej struktury grafów cyklicznych oraz małych domkniętych sąsiedztw wierzchołków.

Dla ustalonego rozmiaru n tworzono jeden graf danej rodziny, na którym wykonywano następnie pięć pomiarów czasu działania algorytmu. Do dalszej analizy przyjęto medianę z pięciu otrzymanych wartości, co ograniczyło wpływ pojedynczych odchyleń czasowych.

W przypadku grafów losowych używano ziarna generatora liczb pseudolosowych zależnego od rozmiaru grafu. Dzięki temu eksperyment był powtarzalny i dla ustalonego n otrzymywano zawsze tę samą reprezentację grafu. Dla grafów losowych i gęstych k-drzew pomiary przeprowadzono dla rozmiarów $n = 10, 20, 50, 70, 100, 150$. Dla grafów cyklicznych, ze względu na znacznie krótszy czas działania, wykorzystano większy zakres: $n = 10, 20, 50, 100, 200, 500, 1000$. Wyniki testów przedstawiono na wykresach A.15, A.16, A.17.

Teoretyczna złożoność czasowa badanej implementacji wynosi

$$O(n^2 + nm + nA),$$

gdzie m oznacza liczbę krawędzi, natomiast A liczbę wypisywanych klik maksymalnych. Jest to zatem algorytm zależny od rozmiaru wyniku. Z tego powodu jego czas działania może istotnie różnić się nawet dla grafów o podobnych wartościach n i m, jeżeli różnią się one liczbą klik maksymalnych lub strukturą redukcji tranzytywnej.

Dla grafów cyklicznych otrzymano współczynnik dopasowania $a = 1.514(70)$. W tej rodzinie zarówno liczba krawędzi, jak i liczba klik maksymalnych rosną liniowo:

$$m = n, \qquad A = n.$$

Po podstawieniu do oszacowania teoretycznego otrzymuje się

$$O(n^2 + nm + nA) = O(n^2).$$

Uzyskany współczynnik jest niższy od wartości 2, wynikającej z ogólnego ograniczenia dla tej rodziny. Oznacza to, że w badanym zakresie czas działania wzrastał wolniej niż kwadratowo. Może to wynikać z regularnej i rzadkiej struktury cykli oraz z wpływu niewielkich czasów wykonania dla mniejszych grafów.

Dla gęstych k-drzew otrzymano współczynnik dopasowania $a = 2.122(59)$. Przy $k = \lfloor n/2 \rfloor$ liczba krawędzi rośnie kwadratowo wraz z liczbą wierzchołków, natomiast liczba klik maksymalnych rośnie liniowo. Teoretyczne oszacowanie złożoności przyjmuje więc w tym przypadku postać

$$O(n^2 + nm + nA) = O(n^3).$$

Uzyskany współczynnik jest jednak wyraźnie mniejszy od 3, co oznacza, że dla badanych k-drzew czas działania wzrastał wolniej, niż wynikałoby z ogólnego oszacowania najgorszego przypadku. Wynika to prawdopodobnie z regularnej struktury tych grafów.

Największą wartość współczynnika otrzymano dla grafów losowych, $a = 4.42(67)$. Dopasowanie było w tym przypadku wyraźnie mniej dokładne niż dla pozostałych rodzin. Względna niepewność współczynnika wyniosła około 15%, podczas gdy dla cykli i k-drzew była mniejsza niż 5%. Szczególnie duży wzrost czasu odnotowano pomiędzy grafami o 100 i 150 wierzchołkach.

Tak duża wartość współczynnika nie jest sprzeczna z oszacowaniem $O(n^2 + nm + nA)$. Wielkości m oraz A nie są stałe, lecz również zależą od liczby wierzchołków. Dla grafów gęstych liczba krawędzi może rosnąć kwadratowo, przez co składnik nm osiąga rząd $O(n^3)$. Ponadto czas działania zależy od liczby wypisywanych klik maksymalnych A, a składnik nA może dominować nad pozostałymi składnikami.

A.6. Testy algorytmu wyznaczania kliki największej w grafach łuków na okręgu

Algorytm max_clique_gavril() przetestowano na różnych rodzajach grafów łuków na okręgu. Celem testów było zbadanie czasu potrzebnego do wyznaczenia kliki największej oraz porównanie zachowania algorytmu dla grafów rzadkich, gęstych i o losowej strukturze.

Do testów wybrano trzy rodziny grafów łuków na okręgu: grafy losowe, grafy pełne oraz grafy cykliczne. Grafy pełne reprezentowały przypadek gęsty, natomiast grafy cykliczne wykorzystano jako przykład grafów rzadkich. Grafy losowe pozwoliły zbadać działanie algorytmu dla struktur nieregularnych. W uzyskanych przypadkach grafy losowe również były stosunkowo gęste.

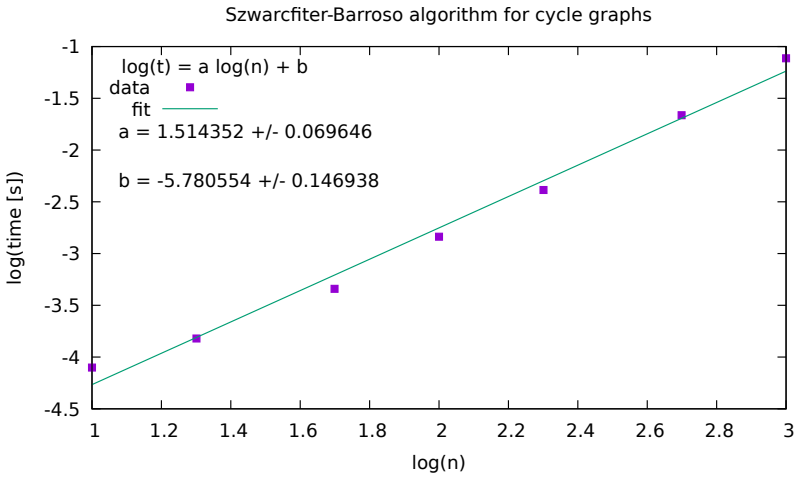
Do tworzenia reprezentacji łukowych wykorzystano trzy istniejące funkcje: make_random_arc(), make_complete_arc() oraz make_cycle_arc(). Generowanie reprezentacji i obliczanie liczby krawędzi wykonywano przed rozpoczęciem właściwego pomiaru. Operacje te nie wpływały więc na zapisane czasy działania algorytmu.

Dla ustalonego rozmiaru n tworzono jeden graf danej rodziny. Na tej samej reprezentacji wykonywano następnie pięć pomiarów czasu działania funkcji max_clique_gavril(). Do dalszej analizy przyjęto medianę z pięciu otrzymanych wartości, co ograniczyło wpływ pojedynczych odchyleń czasowych. Pomiary przeprowadzono dla rozmiarów $n = 50, 100, 200, 300, 400, 500$. Wyniki testów przedstawiono na wykresach A.18, A.19, A.20.

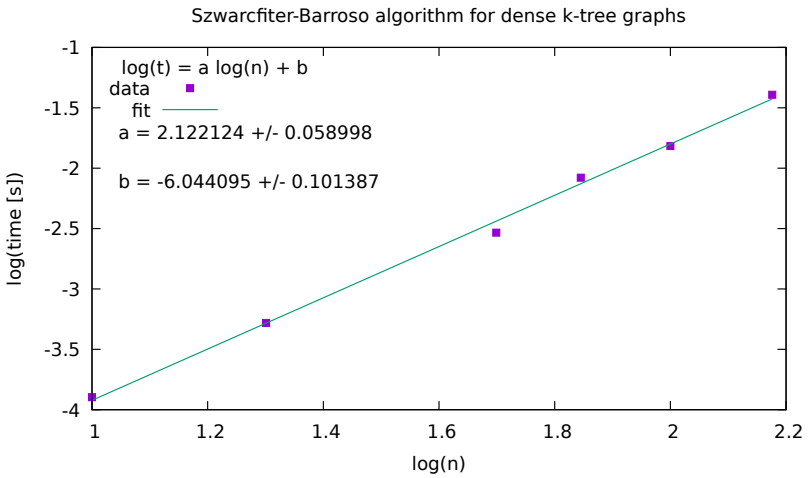
Największą wartość współczynnika dopasowania uzyskano dla grafów pełnych, $a = 3.049(5)$. Wynik wskazuje na wzrost czasu działania bardzo zbliżony do sześciennego. Jest on niższy od ogólnej górnej granicy $O(n^{3.5})$ wyznaczonej dla tej implementacji, lecz zgodny z oszacowaniem $O(n^3)$ podanym przez Gavrila.

Dla grafów losowych otrzymano $a = 2.969(26)$. Wartość ta również jest bliska 3. Wynika to między innymi z dużej gęstości wygenerowanych grafów losowych. Zbiory X_i i Y_i są w takich przypadkach stosunkowo duże, przez co kolejne grafy pomocnicze zawierają znaczną część wierzchołków grafu wejściowego.

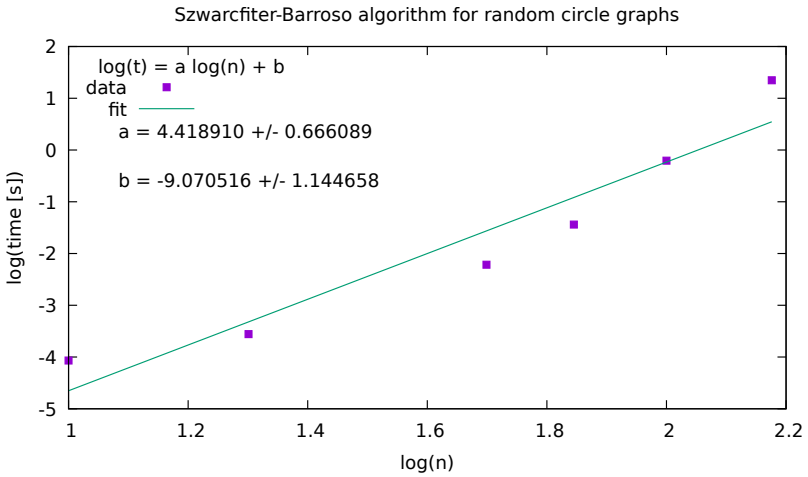
Dla grafów cyklicznych otrzymano znacznie niższą wartość $a = 1.662(67)$. Grafy cykliczne są rzadkie, a każdy ich wierzchołek ma tylko dwóch sąsiadów. W reprezentacji łukowej przekłada się to na niewielkie zbiory X_i i Y_i, a w konsekwencji na małe grafy pomocnicze M_i. Operacje wykonywane na tych grafach są więc znacznie tańsze niż w przypadku grafów pełnych i losowych.



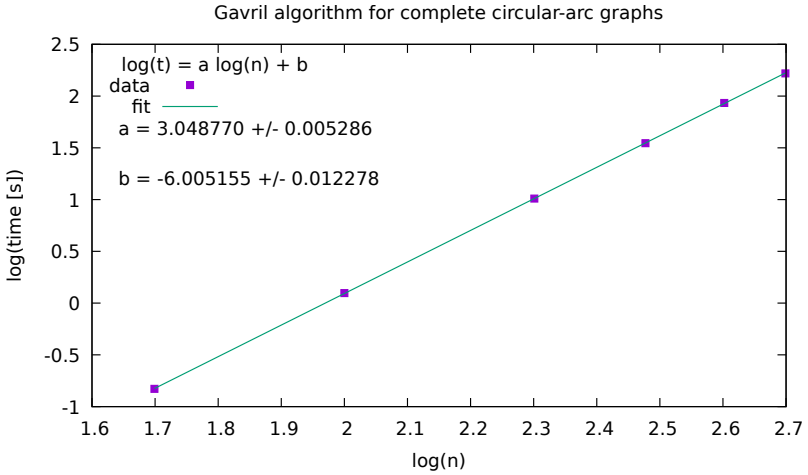
Rysunek A.15. Zależność czasu wyznaczania wszystkich klik maksymalnych od liczby wierzchołków dla grafów cyklicznych. Otrzymany współczynnik dopasowania $a = 1.514(70)$ wskazuje na wolniejszy wzrost czasu niż w pozostałych badanych rodzinach, co jest związane z rzadką i regularną strukturą cykli.



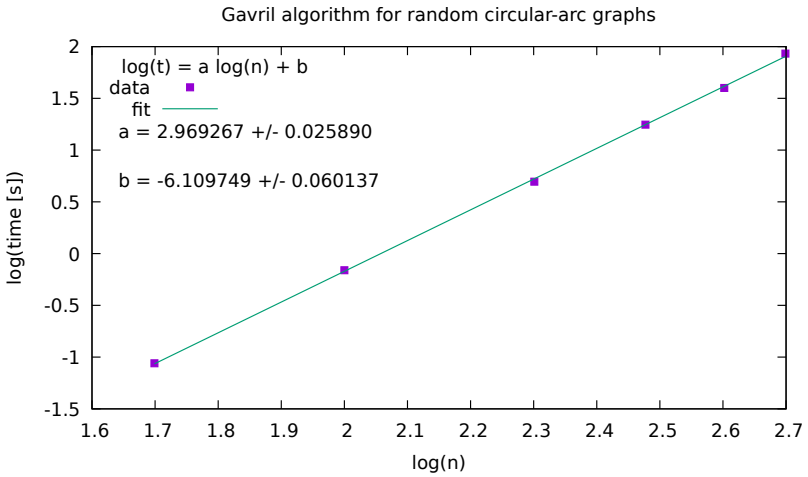
Rysunek A.16. Zależność czasu wyznaczania wszystkich klik maksymalnych od liczby wierzchołków dla gęstych k-drzew, w których przyjmowano $k = \lfloor n/2 \rfloor$. Otrzymano współczynnik dopasowania $a = 2.122(59)$, niższy od wykładnika 3 wynikającego z ogólnego oszacowania złożoności.



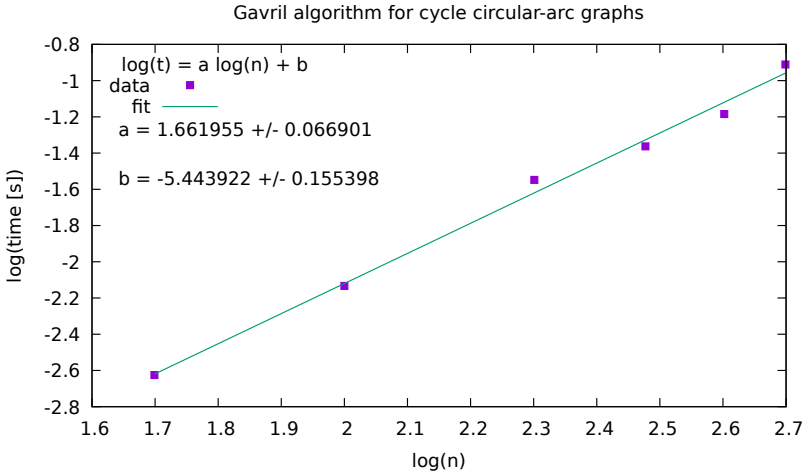
Rysunek A.17. Zależność czasu wyznaczania wszystkich klik maksymalnych od liczby wierzchołków dla losowych grafów kołowych. Współczynnik dopasowania $a = 4.42(67)$ oraz jego większa niepewność wskazują na nieregularny wzrost czasu i silną zależność od struktury wygenerowanego grafu.



Rysunek A.18. Wyznaczanie największej kliki w pełnych grafach łuków na okręgu za pomocą algorytmu Gavrila. Wartość współczynnika dopasowania $a = 3.049(5)$ wskazuje na wzrost czasu działania bardzo zbliżony do sześciennego. Dla każdego wierzchołka budowany jest graf pomocniczy o rozmiarze porównywalnym z rozmiarem całego grafu.



Rysunek A.19. Wyznaczanie największej kliki w losowych grafach łuków na okręgu za pomocą algorytmu Gavrila. Wartość współczynnika dopasowania $a = 2.969(26)$ wskazuje na wzrost czasu działania zbliżony do sześciennego.



Rysunek A.20. Wyznaczanie największej kliki w cyklicznych grafach łuków na okręgu za pomocą algorytmu Gavrila. Wartość współczynnika dopasowania $a = 1.662(67)$ jest niższa niż w pozostałych badanych przypadkach. Wynika to z rzadkiej i regularnej struktury cykli oraz niewielkich rozmiarów grafów pomocniczych M_i.

Bibliografia

[1] Wikipedia, Clique problem, 2026, https://en.wikipedia.org/wiki/Clique_problem.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa 2012.

[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] Dariusz Zdybski, *Badanie problemu kliki z językiem Python*, Uniwersytet Jagielloński, Kraków 2016.

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

[8] Andrzej Kapanowski, graphtheory, GitHub repository, 2026, https://github.com/ufkapano/graphtheory/.

[9] Gergely Palla, Imre Derényi, Illés Farkas, Tamás Vicsek, *Uncovering the Overlapping Community Structure of Complex Networks in Nature and Society*, Nature, vol. 435, pp. 814–818, 2005. https://doi.org/10.1038/nature03607.

[10] Balázs Adamcsek, Gergely Palla, Illés J. Farkas, Imre Derényi, Tamás Vicsek, *CFinder: Locating Cliques and Overlapping Modules in Biological Networks*, Bioinformatics, vol. 22, no. 8, pp. 1021–1023, 2006. https://doi.org/10.1093/bioinformatics/btl039.

[11] Nicholas Rhodes, Peter Willett, Alain Calvet, James B. Dunbar, Christine Humblet, *CLIP: Similarity Searching of 3D Databases Using Clique Detection*, Journal of Chemical Information and Computer Sciences, vol. 43, no. 2, pp. 443–448, 2003. https://doi.org/10.1021/ci025605o.

[12] H. G. Barrow, R. M. Burstall, *Subgraph Isomorphism, Matching Relational Structures and Maximal Cliques*, Information Processing Letters, vol. 4, no. 4, pp. 83–84, 1976. https://doi.org/10.1016/0020-0190(76)90049-1.

[13] Reinhard Diestel, *Graph Theory*, 5th ed., Graduate Texts in Mathematics, vol. 173, Springer, Berlin, Heidelberg 2017. https://doi.org/10.1007/978-3-662-53622-3.

[14] W.-L. Hsu, *Maximum weight clique algorithms for circular-arc graphs and circle graphs*, SIAM Journal on Computing 14, 224-231 (1985).

[15] J.W. Moon and L. Moser, *On cliques in graphs*, Israel J. Math. 3, 23-28 (1965).

[16] Małgorzata Olak, *Badanie grafów cięciwowych z językiem Python*, Uniwersytet Jagielloński, Kraków 2017.

[17] Wikipedia, Permutation graph, 2026, https://en.wikipedia.org/wiki/Permutation_graph.

[18] Albert Surmacz, *Badanie grafów permutacji z językiem Python*, Uniwersytet Jagielloński, Kraków 2021.
[19] Wikipedia, Circle graph, 2026, `https://en.wikipedia.org/wiki/Circle_graph`.
[20] Albert Surmacz, *Badanie grafów kołowych z językiem Python*, Uniwersytet Jagielloński, Kraków 2023.
[21] Oliwia Gil, *Grafy łuków na okręgu*, Uniwersytet Jagielloński, Kraków 2025.
[22] Wikipedia, Circular-arc graph, 2026, `https://en.wikipedia.org/wiki/Circular-arc_graph`.
[23] F. Gavril, *Algorithms on circular-arc graphs*, Networks 4(4), 357-369 (1974).
[24] T. Kashiwabara, S. Masuda, K. Nakajima, and T. Fujisawa, *Generation of Maximum Independent Sets of a Bipartite Graph and Maximum Cliques of a Circular-Arc Graph*, Journal of Algorithms 13, 161-174 (1992).
[25] Andreas Brandstädt, On Improved Time Bounds for Permutation Graph Problems, in: Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science, vol. 657, Springer, 1992, pp. 1–10.
[26] J. L. Szwarcfiter and M. M. A. Barroso, *Enumerating the maximal cliques of a circle graph*, Rio de Janeiro: NCE, UFRJ, 1988, 8 p. (Relatório Técnico, 14/88).