Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Małgorzata Olak

Nr albumu: 1087242

Badanie grafów cięciwowych w języku Python

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2017

1:1091434197

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

2:1098025528

Pragnę złożyć serdeczne podziękowania Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za nieocenioną pomoc, wsparcie, poświęcony czas, ogromne zaangażowanie i uwagi merytoryczne, bez których niniejsza praca nie mogłaby powstać.

3:4037949122

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów dla grafów cięciwowych. Graf nieskierowany jest cięciwowy, jeżeli każdy cykl o długości większej niż trzy zawiera cięciwę, czyli krawędź łączącą dwa niekolejne wierzchołki cyklu. Grafy cięciwowe należą do rodziny grafów doskonałych i można je rozpoznać w czasie liniowym. Niektóre problemy trudne dla ogólnych grafów mogą być rozwiązane w czasie wielomianowym dla grafów cięciwowych.

W pracy opisano wiele właściwości grafów cięciwowych i wykorzystano je do implementacji wydajnych algorytmów, w większości przypadków uzyskano liniową złożoność obliczeniową. Zaimplementowano trzy algorytmy wyszukiwania doskonałego uporządkowania wierzchołków (PEO): algorytm wierzchołków simplicjalnych, przeszukiwanie największej liczności (MCS), leksykograficzne przeszukiwanie wszerz. PEO zastosowano do rozpoznawania i generowania grafów cięciwowych, kolorowania grafów, znajdowania największej kliki, znajdowania wszystkich klik maksymalnych, znajdowania największego zbioru niezależnego.

Zaimplementowano algorytm znajdowania uporządkowania najmniejszego stopnia (MDO) dla grafu ogólnego. MDO wykorzystano do kolorowania grafów cięciwowych, znajdowania największej kliki w grafie cięciwowym, oraz w algorytmie MMD. Opisano problem uzupełnienia cięciwowego dowolnego grafu nieskierowanego i podano dwie metody przybliżone wyznaczania szerokości drzewowej: algorytm MMD (dolne ograniczenie) i heurystykę najmniejszego stopnia (górne ograniczenie).

W celu zapewnienia wysokiej jakości kodu źródłowego wykorzystano narzędzia języka Python do stworzenia testów jednostkowych. Sprawdzono praktyczną złożoność obliczeniową wszystkich algorytmów.

Słowa kluczowe: grafy cięciwowe, doskonałe uporządkowanie eliminacji, dekompozycja drzewowa, szerokość drzewowa, problem kliki, kolorowanie grafów, problem zbioru niezależnego, uzupełnienie cięciwowe

4:3502763717

English title: Study of chordal graphs with Python

Abstract

Python implementation of selected graph algorithms for chordal graphs is presented. An undirected graph is chordal if every cycle of length greater than three has a chord - an edge joining two nonconsecutive vertices of the cycle. Chordal graphs belong to the family of perfect graphs and they can be recognized in polynomial time. Several problems that are hard for general graphs may be solved in polynomial time for chordal graphs.

Many properties of chordal graphs are described and used to implement efficient algorithms, mostly with the linear complexity. Three algorithms for finding a perfect elimination ordering (PEO) of a chordal graph are presented: the simplicial vertices algorithm, maximum cardinality search (MCS), and lexicographic breadth-first search. PEO is used for chordal graph recognition and generation, graph coloring, finding a maximum clique, finding all maximal cliques, finding a maximum independent set.

The algorithm for finding a minimum degree ordering (MDO) is implemented for general graphs. MDO is used for chordal graph colouring, finding a maximum clique, and in the MMD algorithm. A chordal completion problem is discussed and two approximate methods for finding the treewidth of graphs are shown: the MMD algorithm (a lower bound) and the minimal degree heuristic (an upper bound).

The Python unit testing framework is used to assure high quality of source code. Real computational complexity of all algorithms was checked.

Keywords: chordal graphs, perfect elimination ordering, tree decomposition, treewidth, clique problem, graph coloring, independent set problem, chordal completion

5:9611828584

Spis treści

Spis tabel 4

Spis rysunków 5

Listings 7

1. Wstęp 8

1.1. Zastosowania grafów cięciwowych 8
1.2. Cel pracy 9
1.3. Organizacja pracy 10

2. Teoria grafów 11

2.1. Podstawowe definicje 11
2.2. Spójność 12
2.3. Zbiory niezależne 13
2.4. Zbiory dominujące 13
2.5. Pokrycie wierzchołkowe 13
2.6. Wybrane rodziny grafów 14
2.7. Grafy dwudzielne 14
2.8. Grafy przecięć 15
2.9. Grafy przedziałowe 15
2.10. Grafy cięciwowe 16
2.11. Galeria grafów cięciwowych 19
2.12. Grafy doskonałe 20
2.13. Dekompozycja drzewowa 20
2.14. Kolorowanie grafów 21

3. Implementacja grafów 23

3.1. Struktury danych dla grafów abstrakcyjnych 23
3.2. Struktury danych dla grafów cięciwowych 24
3.3. Przykładowe sesje interaktywne 24

4. Algorytmy 28

4.1. Generowanie grafów cięciwowych 28
4.2. Rozpoznawanie grafu cięciwowego przez wierzchołki simplicjalne . . 29
4.3. Znajdowanie PEO dla grafu cięciwowego przez Lex-BFS 30
4.4. Znajdowanie PEO dla grafu cięciwowego przez MCS 31
4.5. Znajdowanie MDO dla dowolnego grafu 32
4.6. Kolorowanie grafu cięciwowego z MDO 33
4.7. Znajdowanie największej kliki w grafie cięciwowym 34
4.8. Znajdowanie klik maksymalnych w grafie cięciwowym 35
4.9. Znajdowanie dekompozycji drzewowej dla grafu cięciwowego 36
4.10. Poprawność dekompozycji drzewowej 37
4.11. Algorytm MMD 38
4.12. Heurystyka najmniejszego stopnia 39
4.13. Heurystyka najmniejszego uzupełnienia 40

6:9068621470

4.14. Największy zbiór niezależny dla drzew 40
4.15. Największy zbiór niezależny dla grafów cięciwowych 42
4.16. Najmniejsze pokrycie wierzchołkowe dla drzew 42
4.17. Najmniejszy zbiór dominujący dla drzew 44

5. Podsumowanie 46

A. Testy algorytmów 48
A.1. Testy generatorów grafów cięciwowych 48
A.2. Testy rozpoznawania grafów cięciwowych 50
A.3. Testy zbiorów niezależnych w grafach cięciwowych 50
A.4. Testy zbiorów dominujących w grafach cięciwowych 55
A.5. Testy pokrycia wierzchołkowego w grafach cięciwowych 55
A.6. Testy znajdowania MDO 55
A.7. Testy znajdowania największej kliki 60
A.8. Testy znajdowania klik maksymalnych 60
A.9. Testy algorytmu MCS 60
A.10. Testy algorytmu MMD 67
A.11. Testy heurystyki najmniejszego stopnia 67

Bibliografia 72

7:1029295147

Spis tabel

2.1 Liczba grafów cięciwowych spójnych. 19

8:2488681955

Spis rysunków

2.1 Graf przedziałowy jest grafem cięciwowym. 16
2.2 Pięć grafów cięciwowych z $n = 4$. 20

A.1 Porównanie wydajności generatorów grafów cięciwowych. 49
A.2 Wydajność generatora grafów cięciwowych bazującego na PEO. 49
A.3 Wydajność rozpoznawania grafów cięciwowych metodą wierzchołków simplicjalnych - wersja z kopią grafu. 51
A.4 Porównanie wydajności rozpoznawania grafów cięciwowych metodą wierzchołków simplicjalnych. 51
A.5 Wykres wydajności rozpoznawania grafów cięciwowych metodą wierzchołków simplicjalnych dla k-drzew. 52
A.6 Porównanie wydajności sprawdzania poprawności PEO dla k-drzew. . 52
A.7 Wydajność sprawdzania PEO dla k-drzew. 53
A.8 Wydajność wyznaczania największego zbioru niezależnego dla drzew. . 53
A.9 Wydajność wyznaczania największego zbioru niezależnego dla k-drzew. 54
A.10 Porównanie wydajności wyznaczania największego zbioru niezależnego. 54
A.11 Wydajność wyznaczania najmniejszego zbioru dominującego dla drzew. 56
A.12 Wydajność wyznaczania pokrycia wierzchołkowego dla drzew. 56
A.13 Porównanie wydajności znajdowania MDO (zależność kwadratowa). . 57
A.14 Wydajność znajdowania MDO dla grafu gęstego (zależność kwadratowa). 57
A.15 Wydajność znajdowania MDO dla grafu rzadkiego (zależność kwadratowa). 58
A.16 Porównanie wydajności znajdowania MDO (sortowanie bukietowe). . . 58
A.17 Wydajność znajdowania MDO dla grafu gęstego (sortowanie bukietowe). 59
A.18 Wydajność znajdowania MDO dla grafu rzadkiego (sortowanie bukietowe). 59
A.19 Wydajność znajdowania największej kliki w grafie cięciwowym (MDO). 61
A.20 Wydajność znajdowania największej kliki dla k-drzewa (MDO). 61
A.21 Wydajność znajdowania największej kliki w grafie cięciwowym (PEO). 62
A.22 Wydajność znajdowania największej kliki dla k-drzewa (PEO). 62
A.23 Wydajność znajdowania klik maksymalnych dla grafu cięciwowego. . . 63
A.24 Wydajność znajdowania klik maksymalnych dla k-drzew. 63
A.25 Porównanie wydajności znajdowania klik maksymalnych. 64
A.26 Wydajność algorytmu MCS dla grafów cięciwowych (zależność kwadratowa). 64
A.27 Wydajność algorytmu MCS dla k-drzew (zależność kwadratowa). . . . 65
A.28 Wydajność algorytmu MCS dla grafów cięciwowych (sortowanie bukietowe). 65
A.29 Wydajność algorytmu MCS dla k-drzew (sortowanie bukietowe). . . . 66
A.30 Porównanie wydajności algorytmów MCS dla grafów cięciwowych. . . 66
A.31 Wydajność algorytmu MMD dla grafu gęstego (zależność kwadratowa). 68
A.32 Porównanie wydajności algorytmu MMD dla różnych grafów (zależność kwadratowa). 68

9:8195290674

A.33 Wydajność algorytmu MMD dla grafu gęstego (sortowanie bukietowe). 69
A.34 Porównanie wydajności algorytmu MMD dla różnych grafów (sortowanie bukietowe). 69
A.35 Wydajność heurystyki najmniejszego stopnia dla grafów gęstych (wersja 1 z listami zbiorów sąsiedztwa). 70
A.36 Wydajność heurystyki najmniejszego stopnia dla grafów rzadkich (wersja 1 z listami zbiorów sąsiedztwa). 70
A.37 Wydajność heurystyki najmniejszego stopnia dla grafów gęstych (wersja 2 z kopią grafu). 71
A.38 Wydajność heurystyki najmniejszego stopnia dla grafów rzadkich (wersja 2 z kopią grafu). 71

10:5021764848

Listings

3.1 Budowanie prostego grafu. 24
3.2 Budowanie grafu cięciwowego. 25
3.3 Generowanie k-drzewa. 26
3.4 Znajdowanie PEO za pomocą algorytmu MCS. 26
3.5 Obliczenia z dowolnym grafem. 27
4.1 Generator grafów cięciwowych. 29
4.2 Moduł chordal2. 29
4.3 Znajdowanie PEO algorytmem Lex-BFS. 30
4.4 Znajdowanie PEO algorytmem MCS. 31
4.5 Znajdowanie MDO dla dowolnych grafów. 32
4.6 Moduł nodecolorsl. 33
4.7 Największa klika w grafie cięciwowym - PEO. 34
4.8 Największa klika w grafie cięciwowym - MDO. 34
4.9 Znajdowanie klik maksymalnych w grafie cięciwowym. 35
4.10 Poprawność dekompozycji drzewowej. 37
4.11 Algorytm MMD. 38
4.12 Heurystyka najmniejszego stopnia. 39
4.13 Moduł treeiset 41
4.14 Największy zbiór niezależny w grafie cięciwowym. 42
4.15 Moduł treecover. 43
4.16 Moduł treedset. 44

11:3656945994

1. Wstęp

Tematem niniejszej pracy jest analiza grafów cięciwowych i ich zastosowanie w algorytmach rozwiązujących konkretne problemy kombinatoryczne. Zagadnienia opisane w pracy dotyczą teorii grafów. Jest to dział matematyki i informatyki zajmujący się badaniem i stosowaniem grafów. Grafy cięciwowe są interesującym przedmiotem badań ze względu na szereg własności jakie posiadają. Problemy, które dla ogólnych grafów są NP-trudne, w przypadku grafów cięciwowych można rozwiązać w czasie wielomianowym [1]. Pierwsze prace naukowe związane z grafami cięciwowymi powstały w latach 50-tych XX wieku. Odkrycie grafów cięciwowych pozwoliło na zdefiniowanie grafów doskonałych, przy czym grafy cięciwowe jako pierwsze zostały zakwalifikowane do klasy grafów doskonałych [2]. W polskiej literaturze istnieje niewiele źródeł opisujących ten rodzaj grafów. Najdokładniejszy opis grafów cięciwowych sporządził Golumbic [3]. Graf cięciwowy to graf nieskierowany prosty, który definiowany jest na kilka równoważnych sposobów [1].

- Graf, w którym wszystkie indukowane cykle mają długość równa trzy.
- Graf, w którym każdy cykl dłuższy bądź równy cztery posiada cięciwę, czyli krawędź łączącą dwa niesąsiednie w cyklu wierzchołki.
- Graf, w którym każdy minimalny separator jest kliką.
- Graf, który posiada doskonałe uporządkowanie wierzchołków.
- Graf, który posiada wierzchołki simplicjalne.
- Graf, który jest grafem przecięć poddrzew pewnego grzewa.

Dokładna definicja grafów cięciwowych wraz z opisem własności znajduje się w rozdziale 2.10. W literaturze anglojęzycznej grafy cięciwowe można znaleźć pod kilkoma różnymi pojęciami, np. *triangulated graphs*, *rigid-circuit graphs*, *decomposable graphs*, *monotonne-transitive graphs*, oraz *perfect elimination graphs* [3].

1.1. Zastosowania grafów cięciwowych

Grafy cięciwowe mają wiele zastosowań. Jednym z nich jest problem alokacji rejestrów [4]. Problem polega na alokowaniu/przydzieleniu skończonej liczby rejestrów maszyny do nieograniczonej liczby tymczasowych zmiennych z interferującym czasem życia. Problem redukuje się do problemu kolorowania wierzchołków pewnego grafu, czyli jest to problem NP-zupełny. Okazuje się jednak, że w rzeczywistych programach graf interferencji jest cięciwowy, co pozwala na zastosowanie szybkich i optymalnych algorytmów dla grafów cięciwowych (sprawdzono np. bibliotekę standardową języka Java).

Grafy cięciwowe są wykorzystywane w modelowaniu rzadkiej struktury macierzy podczas faktoryzacji Choleskiego. Współczynniki niezerowe (ang.

12:2477646806

sparsity pattern) symetrycznej rzadkiej macierzy mogą być reprezentowane przez graf nieskierowany. Eliminacja wierzchołków opisuje uzupełnienie, które zachodzi podczas faktoryzacji Choleskiego na rzadkiej dodatnio określonej macierzy. Z tego wynika, że współczynnik niezerowy faktoryzacji Choleskiego jest cięciwowy i dodatnio określone macierze z cięciwowym współczynnikiem niezerowym ulegają faktoryzacji z uzupełnieniem zerami. Ten fakt leży u podstaw wielu następstw dekompozycji, odkrytych w 1980 roku [2].

Kolejne zastosowanie grafów cięciwowych można odnaleźć w biologii. Drzewo filogenetyczne jest grafem acyklicznym przedstawiającym ewolucyjne zależności pomiędzy sekwencjami lub gatunkami wszystkich organizmów żywych. Dużym problemem jest zrekonstruowanie drzewa filogenetycznego bazując na zbiorze pewnych specyficznych organizmów (połączonych ze sobą w grafie). Każdy organizm jest liściem drzewa. Graf cięciwowy jest używany do budowy drzewa filogenetycznego, na jego podstawie rekonstruuje się drzewo filogenetyczne [5], [6].

Inne zastosowania grafów cięciwowych wspomniano w pracy [6]. Grafy cięciwowe znalazły zastosowanie w obliczeniach na macierzach rzadkich, również w badaniach związanych z zaawansowanymi architekturami komputerowymi (superkomputery wektorowe).

1.2. Cel pracy

Celem niniejszej pracy jest zbadanie własności grafów cięciwowych oraz ich implementacja i wykorzystanie w wybranych algorytmach. Grafy cięciwowe posiadają wiele własności i ważnym aspektem pracy jest ich właściwe wykorzystanie do szybszego rozwiązywania problemów algorytmicznych. Pierwszym zagadnieniem poruszonym w pracy jest omówienie teorii związanej z grafami cięciwowymi. Następnym krokiem jest zbudowanie grafu cięciwowego oraz przetestowanie czy spełnia podstawowe własności. Kluczowe jest uzyskanie doskonałego uporządkowania wierzchołków. To specyficzne uporządkowanie charakteryzujące grafy cięciwowe prowadzi do szybszego rozwiązania wielu problemów, takich jak np. kolorowanie wierzchołków, znalezienie największego zbioru niezależnego wierzchołków, znalezienie największej kliki.

Kolejnym celem pracy jest zbadanie związku grafów cięciwowych oraz dekompozycji drzewiastej. Zamiast narzucać ograniczenia na rozmiar danych wejściowych, wymaga się na wejściu parametrów o określonej strukturze. W takim przypadku wykorzystuje się programowanie dynamiczne. Drzewa łatwo rozdziela się na rozłączne elementy (podproblemy), dzięki temu można rozwiązywać problemy mniejszych rozmiarów i z nich obliczać ostateczne rozwiązanie.

Następnym elementem pracy jest przeprowadzenie uzupełnienia grafu ogólnego do grafu cięciwowego. Ważne jest zastosowanie heurystyk dla dowolnych grafów, dzięki którym znajdziemy dolne oraz górne ograniczenie na szerokość drzewową grafu. Posiadając dane o szerokości drzewowej można w czasie wielomianowym rozwiązać problemy trudne [7].

Do implementacji algorytmów i struktur danych został użyty język programowania Python, wersja 2.7. Python jest językiem interpretowanym, któ-

13:3556833760

rego składnia jest przejrzysta i intuicyjna, a biblioteka standardowa jest bogata. Język wspiera obiektowy paradygmat programowania, który ułatwia implementację nowych pomysłów [8].

1.3. Organizacja pracy

Praca została zorganizowana w następujący sposób. Rozdział 1 zawiera wprowadzenie do grafów cięciwowych i ich zastosowań, cele oraz organizację pracy. W rozdziale 2 znajdują się podstawowe definicje z teorii grafów, używane w dalszej części pracy. W tym rozdziale znajduje się również część poświęcona grafom cięciwowym, opisująca szczegółowo ich własności. Rozdział 3 opisuje szczegóły implementacji grafów, użyte struktury danych, wraz z przykładowymi sesjami interaktywnymi. W rozdziale 4 przedstawione są opisy algorytmów oraz ich implementacje. Rozdział 5 jest podsumowaniem pracy. W dodatku A zostały przedstawione wyniki testów zaimplementowanych w pracy algorytmów.

2. Teoria grafów

Teoria grafów jest dziedziną łączącą matematykę i informatykę. Zajmuje się badaniem grafów oraz implementacją algorytmów wyznaczających własności grafów. Początki teorii grafów datuje się na rok 1793, w którym Leonard Euler opublikował opis zagadnienia mostów królewieckich. Teoria grafów pomaga w formułowaniu i rozwiązywaniu problemów o znaczeniu praktycznym, jest to dział nauki który stale się rozwija. Poniżej znajdują się podstawowe definicje i twierdzenia, które są istotne do zrozumienia grafów cięciwowych i ich własności. Definicje w większości są oparte na książki Cormena [9] oraz Wilsona [10].

2.1. Podstawowe definicje

Graf skierowany, zwany także digrafem (ang. *directed graph*), to para $G = (V, E)$ złożona z niepustego zbioru wierzchołków $V(G)$, oraz ze zbioru krawędzi skierowanych $E(G)$. *Krawędź skierowana* jest formalnie reprezentowana jako uporządkowana para (v, w), zwyczajowo zapisywana jako vw, gdzie v i w należą do zbioru wierzchołków $V(G)$ [9], [11]. Za pomocą grafów skierowanych reprezentujemy kierunek przechodzenia po wierzchołkach grafu. Graficzna reprezentacja to linia ze strzałką skierowaną do wierzchołka końcowego w, wychodząca od wierzchołka początkowego v. O wierzchołku v mówimy, że jest wierzchołkiem *wychodzącym* - to znaczy początkowym, od którego wychodzi krawędź. Wierzchołek w jest zwany wierzchołkiem *wchodzącym* - końcowym, krawędź wchodzi do wierzchołka w, jest do niego skierowana [9].

Krawędź vw jest *incydentna* z wierzchołkami v i w, co oznacza że łączy je ze sobą. Wierzchołek w w grafie skierowanym jest *wierzchołkiem sąsiednim* dla wierzchołka v. Zbiór sąsiadów wierzchołka v w grafie G oznaczamy $N_G(v)$. *Stopniem wierzchołka* v nazywamy liczbę krawędzi, które są incydentne z danym wierzchołkiem - w przypadku grafów skierowanych sumujemy ilość krawędzi wchodzących i wychodzących danego wierzchołka [9]. Formalnie stopień wierzchołka v oznaczany jest jako $\deg(v)$. W grafach skierowanych możemy mówić o *stopniu wejściowym* jako o liczbie krawędzi skierowanych do danego wierzchołka, oraz o *stopniu wyjściowym* jako o liczbie krawędzi wychodzących z danego wierzchołka.

Graf nieskierowany to para $G = (V, E)$ złożona z niepustego zbioru wierzchołków $V(G)$, oraz ze zbioru krawędzi $E(G)$. Krawędź w grafie nieskierowanym to nieuporządkowany zbiór wierzchołków $\{v, w\}$, który oznacza zarówno krawędź vw, jak i krawędź wv. W reprezentacji graficznej nie mamy zaznaczonego kierunku krawędzi, graf ilustruje połączenia pomiędzy wierzchołkami. Sąsiedztwo wierzchołków w grafie nieskierowanym jest symetryczne, w przy-

15:1088893689

padku krawędzi vw wierzchołek v jest sąsiadem w, jak i wierzchołek w jest sąsiedni dla v.

W zastosowaniach teorii grafów czasem pojawia się potrzeba określenia pętli i krawędzi wielokrotnych. *Pętla* jest to krawędź, której początkiem i końcem jest ten sam wierzchołek. *Krawędzie wielokrotne (równoległe)* to zestaw krawędzi o wspólnym wierzchołku wychodzącym i wspólnym wierzchołku wchodzącym. *Grafy proste* nie mają pętli i krawędzi wielokrotnych. *Multigrafy (skierowane lub nieskierowane)* mogą posiadać pętle lub krawędzie wielokrotne. Przy formalnym opisie matematycznym trzeba wtedy $E(G)$ rozumieć jako multizbiór, w którym elementy mogą się powtarzać. Krawędź nieskierowana będzie wtedy dwuelementowym multizbiorem. Samo określenie $graf$ w naszej pracy oznacza graf prosty, ale w literaturze spotyka się też inne definicje.

Liczby wierzchołków i krawędzi w grafie G są oznaczane odpowiednio jako $|V(G)| = n$, oraz $|E(G)| = m$. *Największym stopniem grafu* $\Delta(G)$ jest największy stopień wierzchołka w grafie $\Delta(G) = \max\{\deg(v) : v \in V(G)\}$. *Najmniejszym stopniem grafu* $\delta(G)$ nazywamy najmniejszy stopień wierzchołka w grafie $\delta(G) = \min\{\deg(v) : v \in V(G)\}$ [12].

Podgraf H grafu G to graf, który zawiera wybrane wierzchołki grafu G, oraz może zawierać niektóre krawędzie łączące wybrane wierzchołki w grafie G. Formalnie zapisujemy, że $V(H) \subseteq V(G)$, oraz $E(H) \subseteq E(G)$ [9], [11]. Inaczej można powiedzieć, że graf H powstaje po usunięciu niektórych wierzchołków i krawędzi z grafu G. Usuwając wierzchołek należy pamiętać o usunięciu wszystkich krawędzi incydentnych z danym wierzchołkiem. Nie wymaga się, by krawędzie grafu G łączące wierzchołki znajdujące się również w podgrafie należały do niego.

Podgraf indukowany H, w przeciwieństwie do zwykłego podgrafu, wymaga aby krawędzie łączące wierzchołki znajdujące się w $V(G)$, a przy tym należące do $V(H)$, znalazły się w podgrafie. Formalnie jest to graf H, którego krawędzie spełniają warunek $E(H) = \{vw \in E(G) : v \in V(H), w \in V(H)\}$ [13], [11].

Klika to podgraf H grafu nieskierowanego G, w którym wszystkie wierzchołki są połączone ze sobą nawzajem krawędziami. Innymi słowy jest to podzbiór zbioru wierzchołków $V(G)$, który indukuje podgraf pełny [9]. *Rozmiarem kliki* nazywamy liczbę wierzchołków zawartych w klice. *Klika maksymalna* nie może być rozszerzona przez dodanie kolejnego wierzchołka, który współtworzyłby klikę [14]. *Klika największa* jest to klika najliczniejsza, czyli w grafie nie istnieją kliki o większej liczbie wierzchołków. Rozmiar największej kliki grafu G oznaczamy $\omega(G)$ [14].

2.2. Spójność

Ścieżka w grafie $G = (V, E)$ z wierzchołka p do q to ciąg wierzchołków, które należy przejść w grafie, aby dotrzeć z wierzchołka p do wierzchołka q. Wierzchołek p jest wierzchołkiem startowym $p = v_0$, wierzchołek q jest wierzchołkiem końcowym $q = v_k$ w ścieżce $(v_0, v_1, ..., v_k)$. Ścieżka zawiera też krawędzie grafu G, po których można dojść do następnych wierzchołków :

16:8460880382

v_0v_1, v_1v_2, ..., $v_{k-1}v_k$. *Długość ścieżki* wyznacza liczbę krawędzi znajdujących się w ścieżce. Jeśli ścieżka składa się z jednego wierzchołka, to długość ścieżki wynosi zero. *Ścieżka prosta* składa się z wierzchołków, które się nie powtarzają, występują tylko raz [9].

Cykl jest to ścieżka, w której pierwszy i ostatni wierzchołek są takie same, $v_0 = v_k$. W przypadku grafu skierowanego musi istnieć przynajmniej jedna krawędź. Pętla stanowi cykl o długości 1. *Cykl prosty* to cykl, w którym żaden z wierzchołków, poza wierzchołkiem początku i końca cyklu, nie powtarza się: $v_1, v_2, v_3, ..., v_k$ są różne. W grafie nieskierowanym cykl zdefiniowany jest jako cykl prosty z przynajmniej trzema krawędziami. Graf, który nie posiada cykli, zwany jest grafem *acyklicznym* [9].

Graf nieskierowany jest *spójny*, jeśli każda para wierzchołków jest połączona ścieżką. Inaczej mówiąc z każdego wierzchołka da się dojść do innego z wierzchołków. *Spójna składowa* to spójny podgraf grafu, który da się wydzielić z całego grafu bez konieczności usuwania krawędzi. Graf spójny ma jedną spójną składową [9].

2.3. Zbiory niezależne

Zbiór niezależny to zbiór S wierzchołków grafu G, które nie są połączone ze sobą krawędziami. Krawędzie mogą być incydentne najwyżej z jednym wierzchołkiem ze zbioru S. Dwa podzbiory A i B wierzchołków grafu G są niezależne, jeśli ich część wspólna jest zbiorem pustym. *Problem największego zbioru niezależnego* polega na znalezieniu w grafie zbioru niezależnego o największej liczności. Jest to znany problem NP-trudny. Moc maksymalnego zbióru niezależnego oznaczamy $\alpha(G)$ [15].

2.4. Zbiory dominujące

Zbiór dominujący to podzbiór D wierzchołków grafu G, taki że każdy wierzchołek nienależący do zbioru D jest połączony przynajmniej jedną krawędzią ze zbiorem D. Innymi słowy każdy wierzchołek w grafie albo należy do zbioru dominującego, albo ma tam sąsiada. *Liczbą dominowania* $\gamma(G)$ w grafie G nazywamy liczbę wierzchołków w najmniejszym zbiorze dominującym. *Problem minimalnego zbioru dominującego* polega na znalezieniu najmniejszej liczby dominowania w grafie. Znalezienie najmniejszego zbioru dominującego jest problemem NP-trudnym [16].

2.5. Pokrycie wierzchołkowe

Pokrycie wierzchołkowe grafu G to podzbiór wierzchołków C, taki że wszystkie krawędzie grafu G są incydentne przynajmniej z jednym wierzchołkiem należącym do zbioru C. *Rozmiarem pokrycia wierzchołkowego* nazywamy liczbę wierzchołków w zbiorze C. *Problem najmniejszego pokrycia wierzchołkowego* polega na znalezieniu pokrycia wierzchołkowego o najmniej-

17:3525613028

szym rozmiarze. Problem najmniejszego pokrycia wierzchołkowego należy do problemów NP-trudnych [17].

2.6. Wybrane rodziny grafów

Graf pełny jest grafem prostym nieskierowanym, w którym każda para wierzchołków jest ze sobą połączona krawędzią. Graf pełny o n wierzchołkach oznacza się jako K_n. Każdy graf pełny stanowi swoją klikę. Graf K_n posiada $n(n-1)/2$ krawędzi. Z każdego wierzchołka grafu wychodzi $n-1$ krawędzi, dlatego K_n jest określany jako graf regularny stopnia $n-1$. Grafy pełne stopnia conajmniej 5 nie są planarne [10].

Graf cykliczny jest grafem spójnym, regularnym stopnia drugiego. Graf cykliczny jest oznaczany jako C_n, gdzie n oznacza liczbę wierzchołków [10].

Graf liniowy powstaje po usunięciu z grafu cyklicznego jednej krawędzi, oznaczany jest jako P_n, gdzie n oznacza liczbę wierzchołków [10].

Graf regularny stopnia k (w skrócie *graf k-regularny*) to graf, w którym każdy wierzchołek jest tego samego stopnia k. Przykładem grafów regularnych są *grafy kubiczne*, czyli grafy regularne stopnia 3 [10].

Grafem planarnym nazywamy graf, którego krawędzie w graficznej reprezentacji na płaszczyźnie można narysować tak, aby nie przecinały się ze sobą. Rysunek, który przedstawia graf z krawędziami nieprzecinającymi się nazywamy *rysunkiem płaskim*, natomiast taki graf można nazwać *grafem płaskim*. Każdy podgraf grafu planarnego jest również grafem planarnym. Twierdzenie Kuratowskiego (1930) formułuje warunek konieczny do tego, aby graf był grafem planarnym: dany graf jest planarny wtedy i tylko wtedy, gdy nie zawiera podgrafu homeomorficznego z grafem pełnym K_5 lub z grafem pełnym dwudzielnym $K_{3,3}$. Dwa grafy są *homeomorficzne*, jeśli można je zbudować na podstawie tego samego grafu wstawiając wierzchołki stopnia drugiego wewnątrz jego krawędzi [10].

Graf kołowy powstaje z połączenia grafu cyklicznego C_{n-1} z nowym wierzchołkiem v, który na rysunkach znajduje się w środku grafu cyklicznego. Graf kołowy o n wierzchołkach oznaczamy W_n [10].

2.7. Grafy dwudzielne

Grafy dwudzielne (ang. *bipartite graphs*) [18] są to grafy nieskierowane, w których zbiór wierzchołków można podzielić na dwa rozłączne zbiory w taki sposób, że krawędzie grafu zawsze łączą wierzchołki z różnych zbiorów. Te dwa zbiory wierzchołków można traktować jako prawidłowe kolorowanie grafu dwoma kolorami. Każdy graf dwudzielny jest grafem doskonałym.

Jeżeli pomiędzy każdą parą wierzchołków z różnych zbiorów istnieje krawędź, to taki graf nazywamy *grafem pełnym dwudzielnym* $K_{p,q}$, gdzie p, q oznaczają liczności zbiorów wierzchołków.

18:5703440266

2.8. Grafy przecięć

Grafy przecięć (ang. *intersection graphs*) [19] są to grafy nieskierowane, formalnie zdefiniowane następująco. Niech będzie dana rodzina zbiorów S_i, $i = 0, 1, 2, \ldots$ Zbiór wierzchołków grafu $V(G)$ zawiera po jednym wierzchołku v_i dla każdego zbioru S_i. Dwa wierzchołki v_i, v_j są połączone krawędzią z $E(G)$, jeżeli część wspólna zbiorów S_i, S_j jest niepusta.

Wiele ważnych rodzin grafów może być opisanych jako grafy przecięć.

— Każdy graf nieskierowany G może być przedstawiony jako graf przecięć. Dla każdego wierzchołka v_i tworzymy zbiór S_i z krawędzi mających koniec w v_i.
— Graf przedziałowy ma zbiory S_i określone jako przedziały na osi rzeczywistej, albo jako spójne podgrafy grafu liniowego P_n.
— *Circular-arc graph* to graf, w którym zbiory S_i odpowiadają łukom na okręgu. Takie grafy są wykorzystywane w problemie ustawiania świateł ulicznych na skrzyżowaniu [3].
— Graf cięciwowy ma zbiory S_i określone jako spójne podgrafy pewnego drzewa T [20].
— Graf liniowy P_n ma po jednym zbiorze S_i dla każdej krawędzi, przy czym S_i zawiera dwa końce danej krawędzi.
— *Graf permutacji* (ang. *permutation graph*) ma wierzchołki odpowiadające elementom pewnej permutacji, a krawędzie reprezentują pary elementów, które są odwracane przez permutację [21]. Geometrycznie można to zobrazować przez odcinki na płaszczyźnie (odpowiadające wierzchołkom), których końce leżą na dwóch prostych równoległych. Jeżeli dwa odcinki się przecinają, to znaczy że istnieje krawędź między wierzchołkami.

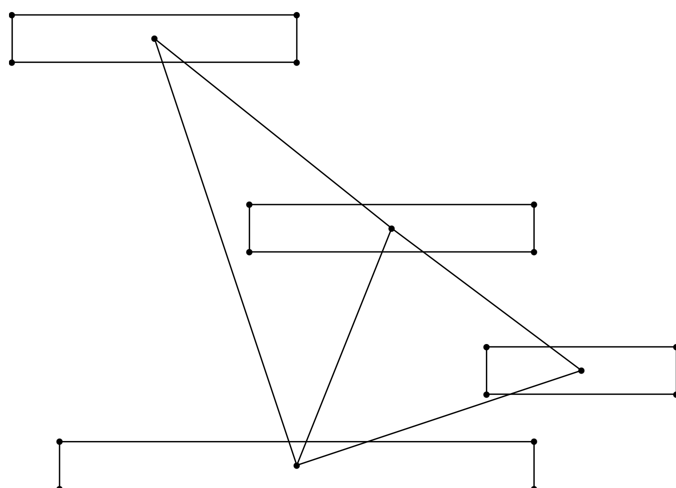
2.9. Grafy przedziałowe

Grafy przedziałowe (ang. *interval graphs*) [22] to grafy przecięć dla zbioru przedziałów na osi rzeczywistej. Grafy przedziałowe można rozpoznać w czasie liniowym $O(V+E)$ poprzez szukanie uporządkowania klik maksymalnych. Bycie grafem przedziałowym jest własnością dziedziczną, czyli przenosi się na podgrafy indukowane.

Twierdzenie (Hajös, 1958): Każdy graf przedziałowy jest grafem cięciwowym. Można to łatwo pokazać próbując rysować przedziały odpowiadające pewnemu cyklowi o długości cztery lub więcej (rysunek 2.1).

Zastosowania: Grafy przedziałowe w naturalny sposób mogą reprezentować problem układania harmonogramu dla zdarzeń odbywających się w pewnych odcinkach czasu. Przykładowo pewne kursy na uniwersytecie mogą odbywać się w tym samym czasie, ale nie w tej samej sali. Odpowiada to poprawnemu kolorowaniu wierzchołków grafu przedziałowego. Dla ogólnych grafów problem jest NP-zupełny, ale dla grafów przedziałowych znane są algorytmy działające w czasie liniowym.

19:9838592742



Rysunek 2.1. Pokrywanie się przedziałów (prostokątów) można zilustrować przez połączenie środków prostokątów. Oś czasu biegnie z lewa na prawo. Cztery przedziały narysowane od góry do dołu tworzą cykl. Czwarty przedział łączy pierwszy i trzeci przedział. Z rysunku widać, że czwarty przedział musi mieć część wspólną z drugim przedziałem, a to odpowiada cięciwie cyklu.

2.10. Grafy cięciwowe

Grafy cięciwowe (ang. *chordal graphs*) [1] są to grafy nieskierowane proste, w których każdy cykl o długości cztery lub więcej posiada cięciwę (ang. *chord*), czyli krawędź nie będącą częścią cyklu, ale łączącą dwa wierzchołki należące do cyklu. Istnieją także inne, równoważne definicje grafów cięciwowych. Grafy cięciwowe trywialne to drzewa (nie mają cykli) lub grafy zawierające jedynie cykle długości trzy.

W grafie cięciwowym każdy cykl indukowany (ang. *induced cycle*) ma maksymalnie trzy wierzchołki. Cykl indukowany jest to cykl, który jest jednocześnie podgrafem indukowanym. Warto zauważyć, że bycie grafem cięciwowym jest własnością dziedziczną (ang. *hereditary property*), to znaczy wszystkie podgrafy indukowane grafu cięciwowego są grafami cięciwowymi.

Twierdzenie (Berge, 1960; Hajnal, Surányi, 1958): Każdy graf cięciwowy jest grafem doskonałym. Można powiedzieć, że badania grafów cięciwowych były początkiem teorii grafów doskonałych [3].

Twierdzenie (Fulkerson, Gross, 1965): Graf jest grafem cięciwowym wtedy i tylko wtedy, gdy posiada *doskonałe uporządkowanie/schemat eliminacji wierzchołków* (ang. *perfect vertex elimination ordering/scheme, PEO*) [23]. Jest to takie uporządkowanie wierzchołków grafu, że dla każdego wierzchołka v, jego sąsiedzi pojawiający się za nim w PEO indukują klikę. Warto dodać, że PEO nie jest unikalne, różne podejścia mogą wyznaczyć różne

20:7215354235

ciągi wierzchołków [6]. Znany jest również algorytm wyznaczający wszystkie możliwe PEO dla danego grafu cięciwowego [24].

Wierzchołki simplicjalne: Wierzchołek v grafu nazywamy wierzchołkiem *simplicjalnym* (ang. *simplicial*), jeżeli jego zbiór sąsiadów $N(v)$ tworzy klikę (niekoniecznie maksymalną). Tak więc wierzchołki w PEO są simplicjalne w indukowanych podgrafach opartych na wierzchołkach będących za nimi w PEO.

Podane powyżej twierdzenie sugeruje prosty sposób rozpoznawania grafów cięciwowych. Należy w kolejnych krokach szukać pewnego wierzchołka simplicjalnego, usuwać go z grafu i dodawać do PEO. Tylko dla grafów cięciwowych ta procedura pozwoli usunąć wszystkie wierzchołki grafu. Szacowana złożoność obliczeniowa tego algorytmu to $O(V^4)$, dlatego w praktyce używa się algorytmów działających w czasie liniowym. Implementacja algorytmu znajduje się w rozdziale 4.

Twierdzenie (Dirac, 1961): Każdy graf cięciwowy G posiada wierzchołek simplicjalny, a jeżeli graf G nie jest kliką, to posiada dwa niesąsiednie wierzchołki simplicjalne [25]. Dowód jest prowadzony przez indukcję ze względu na liczbę wierzchołków.

Lex-BFS: Rose, Tarjan i Lueker [26] podali algorytm wyznaczania PEO o nazwie *leksykograficzne przeszukiwanie wszerz* (ang. *lexicographic breadth first search, Lex-BFS, LBFS*), który działa w czasie liniowym $O(V + E)$ [27]. Wierzchołki wybierane są od końca. Niech $S = \{v_{i+1}, \ldots, v_n\}$ będzie zbiorem już wybranych wierzchołków. Jako następny do S będzie wybrany wierzchołek v_i ze zbioru $V - S$, który ma największą etykietę leksykograficzną. Etykieta jest zbudowana jako malejący ciąg indeksów wierzchołków ze zbioru $N(v) \cap S$. Porządek leksykograficzny to porządek stosowany w słowniku, czyli przykładowo $9761 < 985$, $643 < 6432$ [3].

Warto zauważyć, że Lex-BFS w pewien sposób wynika z twierdzenia Diraca. Mianowicie mając dwa wierzchołki simplicjalne niesąsiednie w grafie cięciwowym możemy jeden z nich usunąć na bok, aby nie brał udziału w procesie wyszukiwania wierzchołków simplicjalnych. Znajdzie się on na ostatniej pozycji w PEO. Dalej możemy jego dowolnego sąsiada umieścić na przedostatniej pozycji w PEO. Kontynuując ten proces uzyskamy PEO od końca. Podobna idea pojawia się w algorytmie MCS.

Maximum Cardinality Search (MCS): Tarjan i Yannakakis [28] podali jeszcze inny algorytm wyznaczania PEO o nazwie *przeszukiwanie największej liczności* (ang. *maximum cardinality search*), który również działa w czasie liniowym $O(V + E)$. Wierzchołki są wybierane od końca. Niech S będzie zbiorem już wybranych wierzchołków. Jako następny do S będzie wybrany wierzchołek v ze zbioru $V - S$, taki że zbiór $N(v) \cap S$ ma największą liczność.

21:9374878246

Jeżeli kilka wierzchołków ma największą liczność, to można z nich wybrać dowolny wierzchołek.

Twierdzenie (Fulkerson, Gross, 1965): Każdy graf cięciwowy $G = (V, E)$ zawiera co najwyżej $|V(G)|$ klik maksymalnych. Dokładnie $|V(G)|$ klik maksymalnych zawiera graf bez krawędzi.

Ogólne grafy mogą mieć wykładniczą liczbę klik maksymalnych. Z podanego twierdzenia wynika, że w grafach cięciwowych można znaleźć największą klikę w czasie wielomianowym, ponieważ największa klika będzie jedną z klik maksymalnych.

Z definicji PEO wynika, że każdy wierzchołek v tworzy klikę razem ze swoimi sąsiadami ze zbioru $N(v)$, którzy znajdują się za nim w PEO. Niektóre z tych klik są klikami maksymalnymi, a jedna z nich to klika największa. Ta analiza leży u podstaw algorytmu znajdowania największej kliki w grafie cięciwowym, który przedstawimy w rozdziale 4.

k-drzewa: *k-drzewo* (ang. *k-tree*) jest to graf nieskierowany $G = (V, E)$ w którym liczba wierzchołków wynosi $|V| = n$, liczba krawędzi jest równa $|E| = nk - k(k + 1)/2$. K-drzewa definiujemy rekurencyjnie w następujący sposób ($n > k$) [29].
(1) Graf pełny K_{k+1} jest k-drzewem z $k + 1$ wierzchołkami.
(2) Niech będzie dane k-drzewo H z n wierzchołkami. Budujemy k-drzewo G z $n + 1$ wierzchołkami przez wybranie k wierzchołków z $V(H)$ tworzących k-klikę, a następnie przez połączenie tych wierzchołków z nowym wierzchołkiem [powstaje $(k + 1)$-klika].

Uporządkowanie degeneracji: Dla dowolnego grafu nieskierowanego G definiujemy *degenerację* d (ang. *degeneracy*). Jest to najmniejsza liczba d taka, że każdy podgraf grafu G ma wierzchołek stopnia co najwyżej d [30]. *Uporządkowanie degeneracji* (ang. *degeneracy ordering*) jest to uporządkowanie wierzchołków, w którym każdy wierzchołek ma co najwyżej d sąsiadów w grafie indukowanym przez wierzchołki będące za nim w uporządkowaniu.

Minimum Degree Ordering (MDO): Szczególnym rodzajem uporządkowania degeneracji jest *uporządkowanie najmniejszego stopnia* (ang. *minimum degree odering*). Jest to uporządkowanie, w którym każdy wierzchołek ma najmniejszy stopień w grafie indukowanym przez wierzchołki będące za nim w uporządkowaniu [31].

Uporządkowanie MDO stosuje się m.in. w algorytmie SL (ang. *Smallest Last*) kolorowania wierzchołków (kolejność wierzchołków odwrotna do MDO), w algorytmie Brona-Kerboscha wyznaczającym wszystkie kliki maksymalne. Prosta implementacja wyznacza uporząkowania MDO w czasie $O(V^2)$, natomiast implementacja wykorzystująca sortowanie bukietowe działa w czasie liniowym $O(V + E)$. Analiza złożoności wykorzystuje znany związek na sumę stopni wierzchołków $\sum_{v \in V(G)} \deg(v) = 2|E(G)|$. Przedstawimy tą implementację w rozdziale 4.

W przypadku grafów cięciwowych pokazano dwa ważne zastosowania uporządkowania MDO [31]. Po pierwsze, uporządkowanie MDO pozwala znaleźć

22:5803462482

Tabela 2.1. Liczba grafów cięciwowych spójnych z n wierzchołkami.

n	1	2	3	4	5
spójne	1	1	2	6	21
cięciwowe	1	1	2	5	15
drzewa	1	1	1	2	3

największą klikę w grafie cięciwowym. Po drugie, można optymalnie pokolorować wierzchołki grafu cięciwowego algorytmem SL.

Uzupełnienie cięciwowe: W praktycznych zastosowaniach dużą rolę odgrywa *uzupełnienie cięciwowe* (ang. *chordal completion*) grafu nieskierowanego [32]. Dla danego grafu nieskierowanego H uzupełnienie cięciwowe jest to graf cięciwowy G, taki że $V(G) = V(H)$, oraz H jest podgrafem grafu G. Potrzebne są różne uzupełnienia cięciwowe: (1) najmniejsze uzupełnienie cięciwowe (ang. *minimum chordal completion*) różniące się od pierwotnego grafu najmniejszą liczbą krawędzi, (2) uzupełnienie cięciwowe dające najmniejszy rozmiar największej kliki, co jest używane przy definiowaniu szerokości drzewowej.

Problem znalezienia najmniejszego uzupełnienia cięciwowego (ang. *the minimum fill-in problem*) jest NP-zupełny [33]. Tak samo jest z wyznaczaniem szerokości drzewowej, dlatego znane algorytmy dokładne działają w czasie wykładniczym [34]. Stworzono również pewne algorytmy aproksymacyjne i szereg różnych heurystyk. Kilka algorytmów heurystycznych wyznaczających przybliżoną wartość szerokości drzewowej pokażemy w rozdziale 4.

Kolorowanie wierzchołków: Optymalne kolorowanie wierzchołków grafu cięciwowego można otrzymać przy użyciu algorytmu zachłannego, przy czym wierzchołki grafu należy kolorować w kolejności odwrotnej do PEO [1]. Druga możliwość to wykorzystanie uporządkowania MDO i algorytmu SL [35].

Samą liczbę chromatyczną grafu cięciwowego można otrzymać w czasie liniowym $O(V + E)$ przez znalezienie największej kliki [3].

Zbiory niezależne: Największy zbiór niezależny w grafie cięciwowym można znaleźć na bazie PEO w czasie liniowym $O(V + E)$ [36]. Implementacja algorytmu będzie omówiona w rozdziale 4.

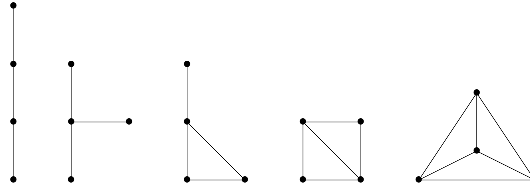
Zbiory dominujące: Problem znalezienia najmniejszego zbioru dominującego dla grafu cięciwowego jest NP-zupełny [37].

2.11. Galeria grafów cięciwowych

W tym rozdziale pokażemy rysunki najprostszych grafów cięciwowych. Liczby grafów cięciwowych spójnych o danej liczbie wierzchołków zostały zebrane w tabeli 2.1.

Dla $n = 1$ mamy jeden wierzchołek. Dla $n = 2$ mamy jedną krawędź, czyli graf P_2. Dla $n = 3$ mamy ścieżkę P_3 i cykl C_3. Dla $n = 4$ mamy pięć grafów

23:3081685669



Rysunek 2.2. Pięć grafów cięciwowych z $n = 4$, w tym dwa drzewa: ścieżka P_4 i gwiazda $K_{1,3}$. Nie należy tutaj cykl C_4.

(rysunek 2.2). Dla $n = 5$ mamy piętnaście grafów, które zostały podzielone według liczby krawędzi.

2.12. Grafy doskonałe

Grafy doskonałe (ang. *perfect graphs*) [38] są to grafy nieskierowane, w których liczba chromatyczna każdego podgrafu indukowanego jest równa rozmiarowi największej kliki tego podgrafu. Odkryto algorytm wielomianowy rozpoznawania tych grafów, ale jest on bardzo skomplikowany. Grafy doskonałe zawierają wiele ważnych rodzin grafów, łącznie ponad dwieście specjalnych klas grafów:
— grafy dwudzielne,
— grafy krawędziowe grafów dwudzielnych (ang. *line graphs*),
— grafy cięciwowe, w tym grafy przedziałowe,
— *comparability graphs*, w tym grafy permutacji,
— *perfectly orderable graphs*,
— *trapezoid graphs*.
Istnieją monografie poświęcone tym grafom, np. [3].

2.13. Dekompozycja drzewowa

Dekompozycja drzewowa (ang. *tree decomposition*) grafu $G = (V, E)$ jest to drzewo T z wierzchołkami/workami X_i, gdzie każdy worek (ang. *bag*) jest podzbiorem $V(G)$, przy czym spełnione są następujące warunki [39]:
— Suma worków X_i jest równa $V(G)$.
— Dla każdej krawędzi ze zbioru $E(G)$ istnieje worek X_i, taki że oba końce krawędzi należą do X_i.

24:7677737470

— Dla każdego wierzchołka v ze zbioru $V(G)$, worki zawierające v tworzą spójne poddrzewo w drzewie T.

Szerokość dekompozycji drzewowej to rozmiar największego worka X_i minus 1. Dekompozycja drzewowa nie jest jednoznaczna. Trywialna dekompozycja drzewowa to jeden worek równy $V(G)$. *Szerokość drzewowa* (ang. *treewidth*) $\mathrm{tw}(G)$ grafu G to minimalna szerokość po wszystkich dekompozycjach drzewowych grafu. Każda dekompozycja drzewowa wyznacza pewien graf cięciwowy, którego podgrafem jest pierwotny graf G.

Można powiedzieć, że dekompozycja drzewowa jest mapowaniem grafu G na drzewo T, które jest wykorzystywane do obliczenia szerokości drzewowej i przyspieszenia rozwiązywania pewnych problemów obliczeniowych na grafie. Koncepcja dekompozycji drzewowej pojawiała się pod różnymi nazwami u kilku autorów (Halin, Robertson i Seymour, Arnborg i Proskurowski), a obecnie jest intensywnie badana. Problem wyznaczenia szerokości drzewowej jest w ogólności NP-zupełny [40], dlatego znaleziono wiele algorytmów przybliżonych, które szybko wyznaczają dolne lub górne ograniczenie na szerokość drzewową [41], [42]. Znane algorytmy dokładne mają złożoność wykładniczą [43].

Istnieją rodziny grafów o małej (ograniczonej) szerokości drzewowej (ang. *bounded treewidth*). Okazuje się, że wiele problemów trudnych dla ogólnych grafów może być w tym przypadku wydajnie rozwiązanych przy pomocy *programowania dynamicznego*.

W przypadku grafów cięciwowych szerokość drzewowa jest równa wielkości największej kliki minus jeden. Największą klikę można znaleźć w czasie liniowym $O(V + E)$.

2.14. Kolorowanie grafów

Kolorowanie grafu (ang. *graph coloring*) polega na przyporządkowaniu elementom grafu odpowiedniej etykiety, zwanej kolorem, przestrzegając pewnych ściśle określonych reguł etykietowania [44].

Kolorowanie wierzchołków grafu (ang. *vertex coloring*) jest to przypisanie wierzchołkom etykiet, w taki sposób, że każde dwa sąsiadujące w grafie wierzchołki nie mają tego samego koloru. W podobny sposób przebiega kolorowanie krawędzi grafu. Kolorowanie wierzchołków grafu spełniające podaną regułę nazywa się *kolorowaniem właściwym, dozwolonym* (ang. *proper coloring*). Graf zawierający pętle nie może być właściwie pokolorowany [44].

Graf opisujemy jako *k-kolorowalny wierzchołkowo* jeśli istnieje kolorowanie właściwe, które wykorzystuje k kolorów do etykietowania wierzchołków. Kolorując graf staramy się o wykorzystanie jak najmniejszej liczby kolorów, określonej jako *liczba chromatyczna* $\chi(G)$. Jest to optymalne kolorowanie wierzchołków. Problem kolorowania wierzchołków jest NP-trudny.

Dla grafów doskonałych, a więc i dla grafów cięciwowych, możemy otrzymać optymalne rozwiązanie w czasie wielomianowym. Algorytmy zachłanne, które pozwalają na kolorowanie w czasie wielomianowym, korzystają z odwróconego uporządkowania wierzchołków PEO. Rozwiązania mogą również wykorzystywać uporządkowanie wierzchołków ze względu na stopień. Istnie-

25:1123928692

ją również algorytmy sekwencyjne kolorowania wierzchołków oparte na algorytmach zachłannych. Wykorzystują dynamiczne lub statyczne strategie porządkowania wierzchołków do pokolorowania [44]. W rozdziale 4.6 przedstawiamy algorytm sekwencyjny *Smallest Last*. Usuwamy z grafu po kolei wierzchołki o najmniejszym stopniu, zapamiętując kolejność. Gdy graf jest już pusty kolorujemy wierzchołki zachłannie, zaczynając od ostatnio usuniętego wierzchołka. Dla grafów ogólnych algorytm SL dla wynik przybliżony, natomiast dla grafów cięciwowych otrzymamy dokładne rozwiązanie. Do uzyskania uporządkowania wierzchołków według stopni wykorzystano algorytm MDO (wykorzystujemy porządek odwrócony do wygenerowanego przez MDO).

26:1115885166

3. Implementacja grafów

Graf można reprezentować w programie komputerowym na wiele sposobów: macierz sąsiedztwa, lista sąsiedztwa, lista krawędzi, itp. W pracy została wykorzystana biblioteka grafowa, która jest rozwijana w Instytucie Fizyki UJ. Wszystkie struktury oraz algorytmy zostały zaimplementowane w języku Python (wersja 2.7). Biblioteka definiuje jednolity interfejs grafów i dostarcza kilka implementacji klasy Graph. Zwykle korzystamy z implementacji, w której graf jest słownikiem, w którym kluczami są wierzchołki, natomiast wartościami są słowniki zawierające jako klucz wierzchołek sąsiedni, oraz krawędź (instancja klasy Edge). Opisane rozwiązanie łączy zalety list sąsiedztwa i macierzy sasiedztwa. Dodatkowo korzystając z mechanizmu języka Python możemy w prosty sposób zebrać informacje na temat krawędzi grafu.

3.1. Struktury danych dla grafów abstrakcyjnych

Wierzchołek jest obiektem hashowalnym, który umożliwia porównywanie. W praktyce najczęściej wierzchołek jest stringiem lub liczbą. W przedstawionych w tej pracy generatorach grafów wierzchołki są liczbami całkowitymi od 0 do $n-1$, gdzie n oznacza liczbę wierzchołków w grafie.

Krawędź jest instancją klasy Edge, która przechowuje informacje o wierzchołku początkowym (atrybut source), wierzchołku końcowym (atrybut target), oraz o wadze krawędzi (atrybut weight). Domyślnie krawędź jest skierowana, natomiast w grafach nieskierowanych krawędź nieskierowaną zapisujemy wewnętrznie jako parę krawędzi skierowanych przeciwnie.

Graf jest instancją klasą Graph, której interfejs zawiera niezbędne operacje wykonywane na grafach, m.in. dodanie/usunięcie wierzchołka, dodanie/usunięcie krawędzi, iteracja po wszystkich wierzchołkach, krawędziach, czy sąsiadach podanego wierzchołka. Graf jest reprezentowany jako słownik słowników, w którym kluczami są wierzchołki, natomiast wartościami zagnieżdżone słowniki z kluczem – wierzchołkiem sąsiednim, oraz wartością – obiektem klasy Edge reprezentującym krawędź. Klasa Graph udostępnia metodę is_directed(), która informuje nas o tym, czy badany graf jest grafem skierowanym. Jest to pierwsza informacja, którą w trakcie badania grafów cięciwowych należy sprawdzić. Fabryki grafów (moduł factory) zostały w pracy

27:9852240157

wykorzystane pomocniczo do badania zachowania algorytmów dla grafów rzadkich i gęstych.

Klika jest reprezentowana jako zbiór wierzchołków. Wykorzystujemy wbudowaną w języku Python kolekcję nieuporządkowanych elementów **set**.

Zbiór potęgowy jest otrzymywany za pomocą funkcji iter_power_set z modułu powersets, która generuje wszystkie podzbiory danego zbioru. Podzbiory są tu reprezentowane jako krotki (typ **tuple**).

3.2. Struktury danych dla grafów cięciwowych

Graf cięciwowy jest zwyczajną instancją klasy Graph. Do badania własności grafów cięciwowych został stworzony moduł chordaltools. Zawiera on generator k-drzew make_random_ktree(n, k), którego autorem jest Konrad Gałuszka [45], oraz generator przypadkowych grafów cięciwowych make_random_chordal(n).

Została też zaimplementowana klasa ChordalGraph do rozpoznawania grafów cięciwowych przez wierzchołki simplicjalne. Wykorzystuje do tego podstawowe własności grafów cięciwowych.

Do zbadania problemów trudnych w przypadku grafów cięciwowych konieczne było przekształcenie algorytmów dla drzew, które były opracowane w pracy magisterskiej Aleksandra Krawczyka [46]. Badane problemy dla drzew to: problem zbioru niezależnego (klasa TreeIndependentSet), problem zbioru dominującego (klasa TreeDominatingSet), problem pokrycia wierzchołkowego (klasa TreeNodeCover).

3.3. Przykładowe sesje interaktywne

Poniższe sesje interaktywne przedstawiają przykłady wykorzystania zaimplementowanych struktur danych i algorytmów. Każdy przykład potrzebuje importu klasy Graph z modułu graphs i klasy Edge z modułu edges. Pierwszy przykład pokazuje utworzenie prostego grafu przez ręczne dodawanie wierzchołków i krawędzi, a następnie uzyskanie pewnych podstawowych informacji o grafie.

Listing 3.1. Budowanie prostego grafu.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> G = Graph()
>>> G.add_node(0)
>>> G.add_node(1)
>>> G.add_node(2)
>>> G.add_edge(Edge(0, 1))
>>> G.add_edge(Edge(1, 2))
>>> G.show()
0 : 1
1 : 0 2
2 : 1
>>> list(G.iternodes())    # lista wierzcholkow
```


28:7068456939

```
[0, 1, 2]
>>> list(G.iteroutedges(1))
[Edge(1, 0), Edge(1, 2)]
>>> G.is_directed()
False
>>> H = G.copy()
>>> H.show()
0 : 1
1 : 0 2
2 : 1
>>> print H.v(), H.e()
3 2
>>> H.has_edge(Edge(1, 2))
True
>>> H.del_edge(Edge(1, 2))
>>> list(H.iteredges())
[Edge(0, 1)]
```

W kolejnym przykładzie korzystamy z generatora grafów cięciwowych z modułu chordaltools. Następnie algorytmem z klasy ChordalGraph rozpoznajemy graf cięciwowy, co potwierdza poprawność generatora. W przypadku grafu innego niż cięciwowy algorytm rzuciłby wyjątek ValueError po uruchomieniu metody run(). Z atrybutu order można odczytać PEO grafu.

Listing 3.2. Budowanie grafu cięciwowego.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from chordaltools import make_random_chordal
>>> from chordal1 import ChordalGraph
>>> G = make_random_chordal(5)
>>> G.show()
0 : 1 3 4
1 : 0 3 4
2 : 4
3 : 0 1 4
4 : 0 1 2 3
>>> list(G.iteredges())    # lista krawedzi
[Edge(0, 1), Edge(0, 3), Edge(0, 4), Edge(1, 3),
Edge(1, 4), Edge(2, 4), Edge(3, 4)]
>>> algorithm = ChordalGraph(G)
>>> algorithm.run()
>>> algorithm.order    # odczytujemy PEO
[0, 1, 2, 3, 4]
```


29:7011141381

Listing 3.3. Generowanie k-drzewa.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from chordaltools import make_random_ktree
>>> from chordal2 import ChordalGraph
>>> G = make_random_ktree(5, 2)    # 2-tree
>>> print G.v(), G.e()
5 7
>>> G.show()
0 : 2 4
1 : 2 3
2 : 0 1 3 4
3 : 1 2 4
4 : 0 2 3
>>> list(G.iteredges())
[Edge(0, 2), Edge(0, 4), Edge(1, 2), Edge(1, 3),
Edge(2, 3), Edge(2, 4), Edge(3, 4)]
>>> algorithm = ChordalGraph(G)
>>> algorithm.run()
>>> algorithm.order    # odczytujemy PEO
[0, 1, 4, 2, 3]
```

W następnym przykładzie po wygenerowaniu grafu cięciwowego znajdujemy PEO korzystając z algorytmu MCS i potwierdzamy poprawność PEO. Następnie znajdujemy największą klikę przez PEO, wszystkie kliki maksymalne, największy zbiór niezależny, uporządkowanie MDO wierzchołków, największą klikę przez MDO, optymalne kolorowanie wierzchołków.

Listing 3.4. Znajdowanie PEO za pomocą algorytmu MCS.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from chordaltools import make_random_chordal
>>> from chordaltools import find_maximum_clique_mdo, find_mdo
>>> from clique1 import find_peo_mcs, is_peo1
>>> from clique1 import find_maximum_clique_peo
>>> from clique1 import find_all_maximal_cliques
>>> from clique1 import find_maximum_independent_set
>>> from nodecolorsl import SmallestLastNodeColoring
>>> G = make_random_chordal(5)
>>> G.show()
0 : 1
1 : 0 4
2 : 3 4
3 : 2 4
4 : 1 2 3
>>> list(G.iteredges())
[Edge(0, 1), Edge(1, 4), Edge(2, 3), Edge(2, 4), Edge(3, 4)]
>>> order = find_peo_mcs(G)
>>> order
[3, 2, 4, 1, 0]
>>> is_peo1(G, order)
True
>>> max_clique = find_maximum_clique_peo(G, order)
>>> max_clique
set([2, 3, 4])
>>> treewidth = len(max_clique)-1
```


30:3791746835

```
>>> find_all_maximal_cliques(G, order)
[set([2, 3, 4]), set([1, 4]), set([0, 1])]
>>> find_maximum_independent_set(G, order)
set([1, 3])
>>> find_mdo(G)
[0, 1, 2, 3, 4]
>>> find_maximum_clique_mdo(G)
set([2, 3, 4])
>>> algorithm = SmallestLastNodeColoring(G)
>>> algorithm.run()
>>> algorithm.color
{0: 0, 1: 1, 2: 2, 3: 1, 4: 0}
```

W następnym przykładzie pokażemy zastosowanie algorytmu MMD i heurystyki najmniejszego stopnia do wyznaczania dolnego i górnego ograniczenia na szerokość drzewową dowolnego grafu spójnego. Jeżeli dolne i górne ograniczenie na szerokość drzewową są równe, to jest to dokładnie szerokość drzewowa danego grafu.

Listing 3.5. Obliczenia z dowolnym grafem.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> from factory import GraphFactory
>>> from connected import is_connected
>>> from twtools import find_treewidth_mmd
>>> from twtools import find_treewidth_min_deg
>>> gf = GraphFactory(Graph)
>>> G = gf.make_random(5, False, 0.5)
>>> G.show()
>>> is_connected(G)
True
>>> G.show()
0 : 2(2)
1 : 2(4) 3(3)
2 : 0(2) 1(4) 3(7) 4(9)
3 : 1(3) 2(7)
4 : 2(9)
>>> list(G.iteredges())
[Edge(0, 2, 2), Edge(1, 2, 4), Edge(1, 3, 3), Edge(2, 3, 7), Edge(2, 4, 9)]
>>> find_treewidth_mmd(G)    # algorytm MMD
2
>>> find_treewidth_min_deg(G) # heurystyka najmniejszego stopnia
2
```

4. Algorytmy

Zaprezentujemy implementacje algorytmów dla dwóch zagadnień. Pierwsze zagadnienie dotyczy badania grafów cięciwowych, czyli generowanie, rozpoznawanie, znajdowanie PEO i drzewa dekompozycji, znajdowanie największej kliki, kolorowanie wierzchołków.

Drugie zagadnienie dotyczy dekompozycji drzewowej i wyznaczania szerokości drzewowej dla ogólnych grafów. Przedstawimy algorytmy wyznaczające dolne i górne ograniczenia na szerokość drzewową: algorytm MMD, algorytm MCS. Pokażemy algorytmy rozwiązujące problemy trudne dla grafów o znanej dekompozycji drzewowej.

W tym rozdziale zakładamy, że grafy są spójne. Jeżeli dany graf nie jest spójny, to należy osobno analizować każdą składową spójną tego grafu.

4.1. Generowanie grafów cięciwowych

Przygotowano dwa generatory przypadkowych grafów cięciwowych. Oba generatory tworzą graf cięciwowy, w którym PEO to ciąg kolejnych liczb całkowitych, od 0 do $n-1$.

Pierwszy generator utrzymuje zbiór wszystkich klik grafu, z którego w kolejnych krokach losowana jest pewna klika, a następnie klika ta jest powiększana o nowy wierzchołek. Do grafu dodawane są nowe krawędzie, a do zbioru klik dołączają nowe kliki zawierające nowy wierzchołek. Kliki są przechowywane jako krotki z wierzchołkami grafu posortowanymi rosnąco.

Drugi generator nie utrzymuje zbioru wszystkich klik grafu, tylko przechowuje rosnące PEO budowane od końca. Losowanie kliki do rozszerzenia z nowym wierzchołkiem przebiega dwuetapowo. W pierwszym etapie losowany jest wierzchołek v z PEO, co wyznacza pewną klikę, nie zawsze maksymalną (sąsiedzi v na prawo w PEO). Następnie z tej kliki losowana jest mniejsza klika, która jest rozszerzana o nowy wierzchołek.

Zaletą pierwszego generatora jest losowanie każdej kliki w grafie z jednakowym prawdopodobieństwem. Wadą jest (1) zapotrzebowanie na pamięć związaną z dużym zbiorem klik, oraz (2) czas działania rosnący prawdopodobnie wykładniczo z liczbą wierzchołków. Zaletą drugiego generatora jest szybkość i zużycie pamięci rzędu $O(V)$. Pewną wadą jest częstsze losowanie małych klik, ponieważ powtarzają się one w różnych większych klikach. W konsekwencji generowane grafy cięciwowe są dość rzadkie, zwykle $|E| \approx 2|V|$. Wyniki testów obu generatorów opisano w dodatku A.1. Listing 4.1 prezentuje kod drugiego generatora.

32:1072952150

Listing 4.1. Generator grafów cięciwowych.

```
def make_random_chordal(n):
    """Make a connected chordal graph with n vertices."""
    graph = Graph(n)
    if n < 1:
        raise ValueError("bad n")
    elif n == 1:
        graph.add_node(0)
    else:
        for node in xrange(n):
            graph.add_node(node)
        graph.add_edge(Edge(n-2, n-1))
        node = n-3
        while node >= 0:
            source = random.choice(xrange(node+1, n-1))
            neighbors = list(target for target in
                graph.iteradjacent(source) if source < target)
            neighbors.append(source)    # closed neighborhood
            k = random.randrange(1, len(neighbors)+1)
            neighbors = random.sample(neighbors, k)
            for target in neighbors:
                graph.add_edge(Edge(node, target))
            node -= 1
    return graph
```

4.2. Rozpoznawanie grafu cięciwowego przez wierzchołki simplicjalne

Graf cięciwowy można rozpoznać przez procedurę usuwania kolejnych wierzchołków simplicjalnych. Jeżeli na pewnym etapie algorytmu nie znajdziemy wierzchołka simplicjalnego, to graf nie jest cięciwowy. Jeżeli graf jest cięciwowy, to otrzymany ciąg usuwanych wierzchołków tworzy PEO.

Przygotowano dwie implementacje podanej procedury. W pierwszej wersji usuwanie wierzchołków simplicjalnych wykonywane jest na osobnej strukturze zbiorów sąsiedztwa. W drugiej wersji algorytm operuje na kopii grafu i korzystamy z interfejsu grafów w naszej implementacji. Testy pokazują (dodatek A.2), że obie wersje mają taka samą złożoność obliczeniową, ale wersja z kopią grafu jest zawsze wolniejsza. Listing 4.2 przedstawia klasę realizującą rozpoznawanie grafu cięciwowego z kopią grafu.

Listing 4.2. Moduł chordal2.

```
#!/usr/bin/python

from Queue import Queue
from edges import Edge
from connected import is_connected

class ChordalGraph:
    """Chordal graphs detection with simplicial nodes."""

    def __init__(self, graph):
        """The algorithm initialization."""
```


33:1091250150

```
        if graph.is_directed():
            raise ValueError("the graph is directed")
        if not is_connected(graph):
            raise ValueError("the graph is not connected")
        self.graph = graph
        self._graph_copy = self.graph.copy()   # O(V+E) time
        self.order = list()   # PEO

    def run(self):
        """Executable pseudocode."""
        Q = Queue()   # queue for simplicial nodes
        used = set()   # nodes detected as simplicial
        for node in self._graph_copy.iternodes():   # O(V^3) time
            if self._is_simplicial(node):
                Q.put(node)
                used.add(node)
        while not Q.empty():   # O(V^4) lub O(E*V^2)
            source = Q.get()
            self.order.append(source)
            self._graph_copy.del_node(source)
            for target in self.graph.iteradjacent(source):
                if target not in used and self._is_simplicial(target):
                    Q.put(target)
                    used.add(target)
        if len(self.order) < self.graph.v():
            raise ValueError("the graph is not chordal")

    def _is_simplicial(self, node):   # O(V^2) time
        """Test if a node is simplicial."""
        for source in self._graph_copy.iteradjacent(node):
            for target in self._graph_copy.iteradjacent(node):
                if source < target:
                    if not self._graph_copy.has_edge(Edge(source, target)):
                        return False
        return True
```

4.3. Znajdowanie PEO dla grafu cięciwowego przez Lex-BFS

Algorytm Lex-BFS można uruchomić dla dowolnego grafu, ale tylko dla grafu cięciwowego znajdziemy PEO. Dość szczegółowy opis algorytmu Lex-BFS znajduje się w książce Golumbica [3]. Według teorii algorytm działa w czasie liniowym $O(V + E)$. Nasza implementacja nie osiąga takiej wydajności, ale zwięźle prezentuje idee algorytmu.

Listing 4.3. Znajdowanie PEO algorytmem Lex-BFS.

```
def find_peo_lex_bfs(graph):
    """Find a lexicographic ordering."""
    labels = dict((node, []) for node in graph.iternodes())
    order = []
    i = graph.v()
    while i > 0:
        source = max((node for node in graph.iternodes()
```


34:9411742097

```
            if node not in order), key=labels.__getitem__)
        order.append(source)
        for target in graph.iteradjacent(source):
            if target not in order:
                labels[target].append(i)
        i -= 1
    order.reverse()
    return order
```

4.4. Znajdowanie PEO dla grafu cięciwowego przez MCS

Algorytm MCS wprowadzili Tarjan i Yannakakis [28] do rozpoznawania grafów cięciwowych. Lucena pokazał, że algorytm MCS można wykorzystać jako heurystykę do znajdowania dolnego ograniczenia na szerokość drzewową dowolnych grafów [47], [48]. Inni autorzy wykorzystali algorytm MCS jako heurystykę do znajdowania górnego ograniczenia na szerokość drzewową dowolnych grafów [42].

Algorytm został zaimplementowany dwoma metodami. Pierwsza metoda działa w czasie $O(V^2)$, ponieważ w każdym kroku wybieramy wierzchołek o największym stopniu odwiedzenia, a robimy to sprawdzając stopnie wszystkich wierzchołków.

Druga metoda działa w praktyce w czasie $O(V+E)$, ponieważ używamy sortowania bukietowego wierzchołków ze względu na stopień odwiedzenia. Problemem jest jeszcze znalezienie ostatniego niepustego bukietu z wierzchołkami. Sprawdzanie bukietów od końca nie jest szybkie, dlatego wprowadziliśmy pomocniczą zmienną maxi sugerującą pozycję ostatniego niepustego bukietu. Poszukiwanie niepustego bukietu rozpoczynamy nie od końca, ale od wartości z tej zmiennej. Po zaliczeniu danego wierzchołka do częściowego PEO, wierzchołek jest usuwany z bukietu, a jego sąsiadom uaktualniamy stopień odwiedzenia, czyli przesuwamy je do wyższych bukietów. Drugim usprawnieniem jest pomocniczy zbiór used do szybkiego sprawdzania, czy dany wierzchołek jest już w częściowym PEO. Listing 4.4 przedstawia implementację metody z sortowaniem bukietowym. Testy algorytmu MCS omówiono w dodatku A.9.

Listing 4.4. Znajdowanie PEO algorytmem MCS.

```
def find_peo_mcs(graph):
    """Finding PEO using maximum cardinality search."""
    if graph.is_directed():
        raise ValueError("the graph is directed")
    order = list()
    used = set()
    visited_degree = dict((node, 0) for node in graph.iternodes())
    bucket = list(set() for deg in xrange(graph.v()))
    bucket[0].update(graph.iternodes())
    maxi = 0
    for step in xrange(graph.v()):
        while not bucket[maxi]:    # szukanie niepustego bukietu
            maxi -= 1
```


35:2071914845

```
        source = bucket[maxi].pop()
        order.append(source)
        used.add(source)
        for target in graph.iteradjacent(source):
            if target in used:
                continue
            deg = visited_degree[target]
            bucket[deg].remove(target)
            bucket[deg+1].add(target)
            visited_degree[target] = deg+1
            maxi = max(maxi, deg+1)
    order.reverse()
    return order
```

4.5. Znajdowanie MDO dla dowolnego grafu

Znajdowanie uporządkowania MDO dowolnego grafu nieskierowanego zaimplemenowano w dwóch wersjach. Mamy metodę prostą działająca w czasie $O(V^2)$ i metodę z sortowaniem bukietowym, działającą w czasie liniowym $O(V+E)$ [31]. Dla uzyskania podanych złożoności ważne było wykorzystanie dodatkowego słownika do sprawdzania, czy dany wierzchołek jest już na liście usuniętych wierzchołków. Bez tego mielibyśmy sprawdzanie należenia do listy w czasie $O(V)$, co pogorszyłoby całkowitą złożoność. Listing 4.5 przedstawia najszybszą wersję wyznaczania MDO z sortowaniem bukietowym. Testy omówiono w dodatku A.6.

Listing 4.5. Znajdowanie MDO dla dowolnych grafów.

```
def find_mdo(graph):
    """Finding a minimum degree ordering in linear time."""
    if graph.is_directed():
        raise ValueError("the graph is directed")
    order = list()
    used = set()
    degree_dict = dict((node, graph.degree(node))
        for node in graph.iternodes())
    bucket = list(set() for deg in xrange(graph.v()))
    for node in graph.iternodes():
        bucket[graph.degree(node)].add(node)
    for step in xrange(graph.v()):
        for deg in xrange(graph.v()):
            if bucket[deg]:
                source = bucket[deg].pop()
                break
        order.append(source)
        used.add(source)
        for target in graph.iteradjacent(source):
            if target in used:
                continue
            deg = degree_dict[target]
            bucket[deg].remove(target)
            bucket[deg-1].add(target)
            degree_dict[target] = deg-1
    return order
```


36:1051378530

4.6. Kolorowanie grafu cięciwowego z MDO

Uporządkowanie MDO umożliwia optymalne kolorowanie wierzchołków grafu cięciwowego algorytmem sekwencyjnym SL. Wierzchołki grafu są kolorowane zachłannie w kolejności odwrotnej do MDO. Listing 4.6 przedstawia algorytm kolorowania grafu cięciwowego, którego wydajność jest liniowa $O(V + E)$. Jest to ulepszona wersja algorytmu, który pojawił się w pracy magisterskiej Igora Samsona [49]. Po pierwsze, metoda _greedy_color() nie musi za każdym razem alokować pamięci na listę używanych kolorów, tylko używana jest jedna lista w przestrzeni nazw instancji klasy. Po drugie, wyodrębniono metodę _find_mdo() do wyznaczania MDO i zastosowano szybkie sortowanie bukietowe do sortowania wierzchołków względem stopni. Dzięki temu wreszcie uzyskano kod o złożoności liniowej opisywanej w teorii.

Listing 4.6. Moduł nodecolorsl.

```
#!/usr/bin/python

class SmallestLastNodeColoring:
    """Find a smallest last (SL) node coloring."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.color = dict((node, None) for node in self.graph.iternodes())
        for edge in self.graph.iteredges():
            if edge.source == edge.target:
                raise ValueError("a loop detected")
        self._color_list = [False] * self.graph.v()

    def run(self):
        """Executable pseudocode."""
        order = self._find_mdo()
        for source in reversed(order):
            self._greedy_color(source)

    def _find_mdo(self):
        """Finding a minimum degree ordering in linear time."""
        n = self.graph.v()
        order = list()
        used = set()
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())    # O(V) time
        bucket = list(set() for deg in xrange(n))    # O(V) time
        for node in self.graph.iternodes():    # O(V) time
            bucket[self.graph.degree(node)].add(node)
        for step in xrange(n):    # O(V) time
            for deg in xrange(n):
                if bucket[deg]:
                    source = bucket[deg].pop()
                    break
            order.append(source)
            used.add(source)
            for target in self.graph.iteradjacent(source):
```


37:5925785762

```
                if target in used:
                    continue
                deg = degree_dict[target]    # stary stopien
                bucket[deg].remove(target)
                bucket[deg-1].add(target)
                degree_dict[target] = deg-1    # nowy stopien
        return order

    def _greedy_color(self, source):
        """Give node the smallest possible color."""
        for target in self.graph.iteradjacent(source):
            if self.color[target] is not None:
                self._color_list[self.color[target]] = True
        for c in xrange(self.graph.v()):    # check colors
            if not self._color_list[c]:
                self.color[source] = c
                break
        for target in self.graph.iteradjacent(source):
            if self.color[target] is not None:
                self._color_list[self.color[target]] = False
        return c
```

4.7. Znajdowanie największej kliki w grafie cięciwowym

Największą klikę w grafie cięciwowym można znaleźć co najmniej dwoma metodami. Pierwsza metoda korzysta z PEO do znalezienia klik powiązanych z wierzchołkami simplicjalnymi w podgrafach indukowanych. Wśród tych klik jest klika największa (listing 4.7).

Druga metoda korzysta z własności MDO. Należy znaleźć w MDO ostatni na prawo wierzchołek v o największym stopniu w odpowiednim podgrafie indukowanym. Wierzchołek v razem ze wszystkimi wierzchołkami na prawo od niego tworzą klikę największą [31]. Zaimplementowano dwie wersje algorytmu z MDO, na bazie dwóch wersji wyznaczania MDO. Listing 4.8 przedstawia szybszą wersję działającą w czasie liniowym $O(V + E)$. Testy obu metod wyznaczania kliki największej opisano w dodatku A.7.

Listing 4.7. Największa klika w grafie cięciwowym - PEO.

```
def find_maximum_clique_peo(graph, order):
    """Find a maximum clique in a chordal graph using PEO."""
    max_clique = set()
    M = dict((node, i) for (i, node) in enumerate(order))
    for source in order:
        clique = set([source])
        for target in graph.iteradjacent(source):
            if M[source] < M[target]:
                clique.add(target)
        max_clique = max(clique, max_clique, key=len)
    return max_clique
```

Listing 4.8. Największa klika w grafie cięciwowym - MDO.

```
def find_maximum_clique_mdo(graph):
```


38:5468108953

```
    """Finding a maximum clique in a chordal graph using MDO."""
    order = list()
    used = set()
    degree_dict = dict((node, graph.degree(node))
        for node in graph.iternodes())
    bucket = list(set() for deg in xrange(graph.v()))
    for node in graph.iternodes():
        bucket[graph.degree(node)].add(node)
    max_deg = 0
    max_idx = 0
    for step in xrange(graph.v()):
        for deg in xrange(graph.v()):
            if bucket[deg]:
                source = bucket[deg].pop()
                break
        order.append(source)
        used.add(source)
        if max_deg <= degree_dict[source]:
            max_deg = degree_dict[source]
            max_idx = step
        for target in graph.iteradjacent(source):
            if target in used:
                continue
            deg = degree_dict[target]
            bucket[deg].remove(target)
            bucket[deg-1].add(target)
            degree_dict[target] = deg-1
        max_clique = set(order[max_idx:])
    return max_clique
```

4.8. Znajdowanie klik maksymalnych w grafie cięciwowym

Znajdowanie wszystkich klik maksymalnych w grafie cięciwowym jest dość złożone. Na wejściu mamy graf cięciwowy i PEO, na wyjściu listę klik maksymalnych. Algorytm jest modyfikacją algorytmu do sprawdzania, czy dane uporządkowanie wierzchołków jest PEO. Algorytm działa w czasie liniowym $O(V + E)$ i jest to implementacja Algorytmu 4.3 z książki Golumbica [3].

Listing 4.9. Znajdowanie klik maksymalnych w grafie cięciwowym.

```
def find_all_maximal_cliques(graph, order):
    """Find all maximal cliques in a chordal graph."""
    cliques = []    # lista klik maksymalnych
    M = dict((node, i) for (i, node) in enumerate(order))
    S = dict((node, 0) for node in order)
    for source in order:
        X = set()
        for target in graph.iteradjacent(source):
            if M[source] < M[target]:
                X.add(target)
        if graph.degree(source) == 0:
            cliques.append(set([source]))
        if not X:
```


39:1097135260

```
            continue
        node = min(X, key=M.__getitem__)
        S[node] = max(S[node], len(X)-1)
        if S[source] < len(X):
            cliques.append(X | set([source]))
    return cliques
```

4.9. Znajdowanie dekompozycji drzewowej dla grafu cięciwowego

Dla grafu cięciwowego $G = (V, E)$ optymalne drzewo dekompozycji T czasem jest nazywane *drzewem klik* (ang. *clique tree*) [50]. Optymalne drzewo dekompozycji T zwykle nie jest wyznaczone jednoznacznie. Worki X_i są wszystkimi klikami maksymalnymi w G. Jest spełniona własność przecięć klik (ang. *clique-intersection property*): dla każdej pary różnych klik maksymalnych X_i, X_j, część wspólna $X_i \cap X_j$ jest zawarta w każdej klice na ścieżce łączącej X_i i X_j w drzewie T. Dla grafu cięciwowego ta własność jest równoważna własności indukowanego poddrzewa (ang. *induced-subtree property*): dla każdego wierzchołka v ze zbioru $V(G)$, worki zawierające v tworzą podrzewo w T.

Optymalne drzewo dekompozycji T dla grafu cięciwowego G może być wyznaczone przy pomocy *ważonego grafu przecięć klik* (ang. *weighted clique intersection graph*) W_G [50]. Zbiór wierzchołków grafu W_G to zbiór klik maksymalnych grafu G. Dwie różne kliki X_i, X_j są połączone krawędzią w W_G, jeżeli część wspólna $X_i \cap X_j$ jest niepusta. Krawędź (X_i, X_j) ma wagę równą $|X_i \cap X_j|$. Optymalne drzewo dekompozycji T jest *drzewem rozpinającym o największej wadze* dla grafu W_G [50].

W tej sytuacji już widać jak należy poszukiwać drzewa dekompozycji. Dla grafu cięciwowego G i jego PEO znajdujemy w czasie liniowym wszystkie kliki maksymalne. Następnie budujemy graf ważony W_G. Wreszcie znajdujemy drzewo rozpinające o największej wadze dla grafu W_G. Można wykorzystać dowolny algorytm znajdujący minimalne drzewo rozpinające, jeżeli zmienimy znak na ujemny dla wszystkich wag w grafie W_G. Mamy do dyspozycji algorytmy Prima, Kruskala, Borůvki (tu wszystkie wagi muszą być różne).

W pracy [50] opisano zmodyfikowany algorytm MCS, który wyznacza jednocześnie PEO i drzewo klik dla grafu cięciwowego. Algorytm z tej pracy może być traktowany jako implementacja algorytmu Prima, zastosowana do grafu W_G.

W pracy [2] podano algorytm rekurencyjny na drzewo klik (Algorytm 3.1). Działa on również dla grafu niespójnego, kiedy składowe spójne są grafami cięciwowymi. Z analizy algorytmu wynika, że spójny graf cięciwowy może mieć co najwyżej $n-1$ klik maksymalnych. Krawędzie drzewa klik dostarczają kompletą listę minimalnych separatorów wierzchołków i jest ich co najwyżej $n-1$. Należy jednak zauważyć, że minimalnych separatorów może być mniej niż krawędzi w drzewie klik.

40:3315410472

4.10. Poprawność dekompozycji drzewowej

Przygotowano klasę testującą poprawność dekompozycji drzewowej (listing 4.10). Na wejściu mamy dowolny graf nieskierowany G i jego drzewo dekompozycji T, niekoniecznie optymalne. Sprawdzane są trzy punkty występujące w definicji dekompozycji drzewowej. Korzysta się z modułu unittest, przy czym drzewo dekompozycji jest instancją klasy Graph, worki drzewa T są krotkami zawierającymi posortowane ciągi wierzchołków grafu G.

Listing 4.10. Poprawność dekompozycji drzewowej.

```
class TestTreeDecomposition(unittest.TestCase):

    def setUp(self):
        self.G = Graph()    # badany graf
        self.T = Graph()    # jego drzewo dekompozycji
        # Budowanie G i T ...

    def test_is_tree(self):
        self.assertTrue(is_acyclic(self.T))
        self.assertTrue(is_connected(self.T))

    def test_bags_cover_vertices(self):
        set1 = set(self.G.iternodes())
        set2 = set()
        for bag in self.T.iternodes():
            set2.update(bag)    # po worku moge iterowac
        self.assertEqual(set1, set2)

    def test_bags_cover_edges(self):
        # Buduje graf wiekszy niz G.
        H = Graph()
        for node in self.G.iternodes():
            H.add_node(node)
        for bag in self.T.iternodes():
            for node1 in bag:
                for node2 in bag:
                    if node1 < node2:
                        edge = Edge(node1, node2)
                        if not H.has_edge(edge):
                            H.add_edge(edge)
        # Kazda krawedz G ma zawierac sie w H.
        for edge in self.G.iteredges():
            self.assertTrue(H.has_edge(edge))

    def test_tree_property(self):
        bag_order = []    # kolejnosc odkrywania przez BFS
        algorithm = SimpleBFS(self.T)
        algorithm.run(pre_action=lambda node: bag_order.append(node))
        # Bede korzystal z drzewa BFS zapisanego jako parent (dict).
        # Dla kazdego wierzcholka grafu G buduje poddrzewo (dict).
        subtree = dict((node, dict()) for node in self.G.iternodes())
        root_bag = bag_order[0]
        # Zaczynam pierwsze poddrzewa.
        for node in root_bag:
            subtree[node][root_bag] = None
        # Przetwarzam nastepne bags.
```


41:2761773361

```
    is_td = True    # flaga sprawdzajaca poprawnosc td
    for bag in bag_order[1:]:
        for node in bag:
            if len(subtree[node]) == 0:    # new subtree
                subtree[node][bag] = None
            elif algorithm.parent[bag] in subtree[node]:    # kontynuacja
                subtree[node][bag] = algorithm.parent[bag]
            else:    # rozlaczne poddrzewa, wlasnosc nie jest spelniona
                is_td = False
    self.assertTrue(is_td)
```

4.11. Algorytm MMD

Algorytm MMD (ang. *Maximum Minimum Degree*) znajduje dolne ograniczenie na szerokość drzewową dowolnego grafu przy wykorzystaniu uporządkowania MDO. Jest to heurystyka, która bazuje na następującej obserwacji. Jeżeli wierzchołek v należy do worka, który jest liściem w drzewie dekompozycji, to zachodzi nierówność $\mathrm{tw}(G) \geqslant |N(v)|$. Algorytm MMD polega na sukcesywnym usuwaniu z grafu wierzchołków o najmniejszym stopniu i znalezieniu największego stopnia wśród tych wierzchołków.

Przygotowano dwie implementacje algorytmu MMD. Pierwsza wersja działa w czasie $O(V^2)$, ponieważ w każdym kroku wyszukiwany jest wierzchołek o najmniejszym stopniu spośród wszystkich dostępnych wierzchołków. Druga wersja algorytmu wykorzystuje sortowanie bukietowe, tak że szybko można znaleźć wierzchołek o najmniejszym stopniu. Po usunięciu danego wierzchołka z grafu położenie jego sąsiadów w bukietach jest uaktualniane w łącznym czasie $O(E)$. Ta wersja działa w czasie liniowym $O(V+E)$. Testy obu wersji znajdują się w dodatku A.10. Listing 4.11 przedstawia szybszą wersję z sortowaniem bukietowym. W obu wersjach potrzebne jest szybkie sprawdzanie, czy dany wierzchołek został już usunięty z grafu. Do tego celu uzywany jest pomocniczy zbiór used.

Listing 4.11. Algorytm MMD.

```
def find_treewidth_mmd(graph):
    """Finding a width using the maximum minimum degree heuristic."""
    if graph.is_directed():
        raise ValueError("the graph is directed")
    treewidth = 0
    used = set()
    degree_dict = dict((node, graph.degree(node))
        for node in graph.iternodes())
    bucket = list(set() for deg in xrange(graph.v()))
    for node in graph.iternodes():
        bucket[graph.degree(node)].add(node)
    for step in xrange(graph.v()):
        for deg in xrange(graph.v()):
            if bucket[deg]:
                source = bucket[deg].pop()
                break
        used.add(source)
        # Szacowanie treewidth.
```


42:4201514969

```
        treewidth = max(treewidth, degree_dict[source])
        # Usuwanie source.
        for target in graph.iteradjacent(source):
            if target in used:
                continue
            deg = degree_dict[target]
            bucket[deg].remove(target)
            bucket[deg-1].add(target)
            degree_dict[target] = deg-1
    return treewidth
```

4.12. Heurystyka najmniejszego stopnia

Heurystyka najmniejszego stopnia (ang. *minimum degree heuristic*) pozwala szybko znaleźć górne ograniczenie na szerokość drzewową dowolnego grafu nieskierowanego i wyznaczyć przykładowe drzewo dekompozycji o tej szerokości drzewowej. Heurystyka polega na (1) kolejnym znajdowaniu w grafie wierzchołka v o najmniejszym stopniu, (2) uzupełnieniu krawędzi tak, aby sąsiedzi ze zbioru $N(v)$ tworzyli klikę, (3) usuwaniu wierzchołka v z grafu. Wierzchołek v wraz ze zbiorem $N(v)$ tworzy osobny worek w drzewie dekompozycji. Usuwane wierzchołki są kolejnymi elementami w *uporządkowaniu eliminacji* dla budowanego drzewa dekompozycji.

Mamy dwie implementacje, obie działają w czasie $O(V^3)$. Pierwsza implementacja manipuluje słownikiem, który przechowuje zbiory sąsiadów dla wierzchołków. Druga implementacja manipuluje kopią grafu, przez co bardziej można wykorzystać standardowy interfejs grafów. Testy obu wersji znajdują się w dodatku A.11. Listing 4.12 przedstawia wersję z kopią grafu.

Listing 4.12. Heurystyka najmniejszego stopnia.

```
def find_treewidth_min_deg(graph):
    """Finding a width using the minimum degree heuristic."""
    if graph.is_directed():
        raise ValueError("the graph is directed")
    treewidth = 0
    used = set()
    graph_copy = graph.copy()
    for step in xrange(graph.v()):
        source = min((node for node in graph_copy.iternodes()
            if node not in used), key=graph_copy.degree)
        used.add(source)
        # Szacowanie treewidth.
        treewidth = max(treewidth, graph_copy.degree(source))
        for (node, target) in itertools.combinations(
            graph_copy.iteradjacent(source), 2):
            edge = Edge(node, target)
            if not graph_copy.has_edge(edge):
                graph_copy.add_edge(edge)
        graph_copy.del_node(source)
    return treewidth
```


43:5069187369

4.13. Heurystyka najmniejszego uzupełnienia

Heurystyka najmniejszego uzupełnienia (ang. *minimum fill-in heuristic*) jest podobna do heurystyki najmniejszego stopnia, wyznacza ona górne ograniczenie na szerokość drzewową dowolnego grafu i przykładowe drzewo dekompozycji o tej szerokości drzewowej. Heurystyka polega na kolejnym wybieraniu wierzchołka v o najmniejszym uzupełnieniu fill-in(v), czyli liczbie krawędzi które trzeba uzupełnić w grafie, aby sąsiedzi ze zbioru $N(v)$ tworzyli klikę. Uzupełnienie dla danego wierzchołka v z grafu G można ściśle zdefiniować jako $\text{fill-in}(v) = |\{st : s, t \in N(v), st \notin E(G)\}|$.

Warto zauważyć, że dla grafów cięciwowych heurystyka najmniejszego uzupełnienia wyznaczy dokładną szerokość drzewową, ponieważ w każdym kroku będzie wybierany wierzchołek simplicjalny o uzupełnieniu zerowym. Występowanie samych zerowych uzupełnień można wykorzystać do rozpoznania grafu cięciwowego.

Bezpieczne reguły: Przy tworzeniu heurystyk znajdujących górne ograniczenie na szerokość drzewową określono pewne *bezpieczne reguły* (ang. *safe rules*), których stosowanie zabezpiecza przed niepotrzebnym powiększeniem obliczanej szerokości drzewowej. Staramy się przestrzegać bezpiecznych reguł podczas przetwarzania/redukcji grafu. Przy przetwarzaniu utrzymuje się zmienną low, taką że przy danej bezpiecznej operacji nie zmienia się wielkość $\max(\text{tw}(G), \text{low})$, która daje wielkość szerokości drzewowej oryginalnego grafu.

Pierwszą bezpieczną regułą jest usuwanie wierzchołka simplicjalnego v i ustawienie $\text{low} = \max(\text{low}, \deg(v))$. Drugą bezpieczną regułą jest usuwanie wierzchołka prawie simplicjalnego v stopnia $\deg(v) \geqslant 2$, o ile $\text{low} \geqslant \deg(v)$. Wierzchołek v jest *prawie simplicjalny*, jeżeli ma takiego sąsiada u, że zbiór $N(v) - \{u\}$ tworzy klikę.

Bachoore i Bodlaender wykorzystali m.in. drugą bezpieczną regułę do stworzenia ulepszonego algorytmu minimalnego uzupełnienia [51]. Oryginalny algorytm nie precyzuje kolejności wybierania wierzchołków o jednakowym uzupełnieniu. Ulepszony algorytm najpierw wybiera wierzchołki simplicjalne i prawie simplicjalne stopnia co najwyżej równego szerokości drzewowej. Przydatny jest jeszcze jeden parametr
$\text{fill-in-excl-one}(v) = \min_{u \in N(v)} |\{st : s, t \in N(v) - \{u\}, st \notin E(G)\}|$,
który dla wierzchołków prawie simplicjalnych wynosi zero.

4.14. Największy zbiór niezależny dla drzew

Drzewa są szczególnymi grafami cięciwowymi nie zawierającymi cykli. Największe kliki w drzewach mają dwa wierzchołki i są to końce jednej krawędzi. Wierzchołki simplicjalne w drzewach to liście, a liścia można wykryć w czasie $O(1)$ przez sprawdzenie, czy stopień wierzchołka jest równy jeden.

W pracy magisterskiej Aleksandra Krawczyka [46] pokazano algorytm wyznaczający największy zbiór niezależny dla drzew (lasów), który polegał na odrywaniu liści i ich rodziców z grafu. Podczas tych operacji drzewo mogło stać się niespójne. Przedstawimy modyfikację tego algorytmu polegającą

44:2006542742

na tym, że z grafu będą odrywane tylko liście, przez co drzewo cały czas będzie spójne. Jeżeli dany liść zostanie zaliczony do zbioru niezależnego, to jego rodzica należy oznaczyć jako zabronionego. Kiedy obecny rodzic stanie się nowym liściem, to będzie on oderwany z grafu, ale nie będzie zaliczony do zbioru niezależnego. W algorytmie należy również prawidłowo obsłużyć wierzchołki izolowane, które pojawiają się przy przetwarzaniu lasów, ale też na końcu przetwarzania pojedyńczego drzewa (listing 4.13). Testy algorytmu opisano w dodatku A.3.

Podany algorytm jest interesujący między innymi dlatego, że sugeruje uogólnienie na dowolne grafy cięciwowe. Odrywanie liści jest odrywaniem z grafu wierzchołków simplicjalnych wg kolejności z pewnego PEO. Kolejny odrywany wierzchołek jest zaliczany do zbioru niezależnego, o ile nie jest sąsiadem wierzchołka zaliczonego wcześniej do zbioru niezależnego. Poprawność algorytmu została udowodniona przez Gavrila [36].

Listing 4.13. Moduł treeiset.

```
#!/usr/bin/python

from Queue import Queue

class TreeIndependentSet:
    """Find a maximum independent set for trees."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.independent_set = set()
        self.cardinality = 0
        self._used = set()

    def run(self):
        """Executable pseudocode."""
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())
        Q = Queue()    # for leafs
        for node in self.graph.iternodes():
            if degree_dict[node] == 0:
                self.independent_set.add(node)
                self._used.add(node)
                self.cardinality += 1
            elif degree_dict[node] == 1:
                Q.put(node)
        while not Q.empty():
            source = Q.get()
            if degree_dict[source] == 0:
                if source not in self._used:
                    self.independent_set.add(source)
                    self._used.add(source)
                    self.cardinality += 1
                continue
            assert degree_dict[source] == 1
            for target in self.graph.iteradjacent(source):
```


45:2211107488

```
            if degree_dict[target] > 0:    # this is parent
                if source not in self._used:
                    self.independent_set.add(source)
                    self._used.add(source)
                    self._used.add(target)
                    self.cardinality += 1
                degree_dict[target] -= 1
                degree_dict[source] -= 1
                if degree_dict[target] == 1:
                    Q.put(target)
                break
```

4.15. Największy zbiór niezależny dla grafów cięciwowych

Wyznaczanie największego zbioru niezależnego w grafie cięciwowym na bazie PEO opisane jest w książce Golumbica [3], ale po raz pierwszy było podane przez Gavrila [36]. Przy okazji w dowodzie poprawności algorytmu jest opisane wyznaczenie najmniejszego pokrycia grafu przez kliki (ang. *clique cover*). Jest też referencja do rozwiązywania problemu największego ważonego zbioru niezależnego (Frank, 1976).

Algorytm wyznaczania największego zbioru niezależnego w grafie cięciwowym jest uogólnieniem algorytmu z odrywaniem liści dla drzew. Zakładamy, że dany jest graf cięciwowy i jego dowolne PEO. Przetwarzamy wierzchołki grafu zgodnie z PEO.

Listing 4.14. Największy zbiór niezależny w grafie cięciwowym.

```
def find_maximum_independent_set(graph, order):
    """Find a maximum independent set in a chordal graph."""
    M = dict((node, i) for (i, node) in enumerate(order))
    iset = set()    # maximum independent set
    used = set()
    for source in order:
        if source in used:
            continue
        iset.add(source)
        used.add(source)
        for target in graph.iteradjacent(source):
            if M[source] < M[target]:
                used.add(target)
    return iset
```

4.16. Najmniejsze pokrycie wierzchołkowe dla drzew

W pracy magisterskiej Aleksandra Krawczyka [46] pokazano algorytm wyznaczający najmniejsze pokrycie wierzchołkowe dla lasów, który polegał na odrywaniu liści i ich rodziców z grafu. Podczas tych działań drzewo mogło stać się niespójne. Przedstawimy modyfikację tego algorytmu zachowującą spójność drzewa. Z grafu będą odrywane tylko liście, a do pokrycia zaliczamy

46:2134474109

rodziców, chyba że liść należy już do pokrycia. W algorytmie należy również prawidłowo obsłużyć wierzchołki izolowane (listing 4.15). Testy algorytmu opisano w dodatku A.5.

Podany algorytm sugeruje sposób postępowania dla ogólnych grafów cięciwowych. Odrywanie liści to odrywanie wierzchołków simplicjalnych wg kolejności z pewnego PEO. Na początku najlepiej zaliczać do pokrycia wszystkie pozostałe wierzchołki połączone z wierzchołkiem simplicjanym, ponieważ krawędzie k-kliki można pokryć $k-1$ wierzchołkami. Jednak w późniejszych etapach algorytmu kolejny wierzchołek simplicjalny może należeć do pokrycia i nie wiadomo, który wierzchołek z kliki nie zaliczać do pokrycia (może wszystkie wierzchołki powinny należeć do pokrycia). Widać potrzebę stosowania metody programowania dynamicznego i przechowywania kilku kandydatów na optymalne rozwiązanie.

Listing 4.15. Moduł treecover.

```
#!/usr/bin/python

from Queue import Queue

class TreeNodeCover:
    """Find a minimum cardinality node cover for forests."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.node_cover = set()
        self.cardinality = 0

    def run(self):
        """Executable pseudocode."""
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())
        Q = Queue()    # for leafs
        for node in self.graph.iternodes():
            if degree_dict[node] == 1:
                Q.put(node)
        while not Q.empty():
            source = Q.get()
            if degree_dict[source] == 0:
                continue
            assert degree_dict[source] == 1
            for target in self.graph.iteradjacent(source):
                if degree_dict[target] > 0:    # this is parent
                    if (source not in self.node_cover and
                        target not in self.node_cover):
                        self.node_cover.add(target)
                        self.cardinality += 1
                    degree_dict[source] -= 1
                    degree_dict[target] -= 1
                    if degree_dict[target] == 1:
                        Q.put(target)
                    break
```


47:1070961014

4.17. Najmniejszy zbiór dominujący dla drzew

W pracy magisterskiej Aleksandra Krawczyka [46] pokazano algorytm wyznaczający najmniejszy zbiór dominujący dla lasów, który polegał na odrywaniu liści i ich rodziców z grafu. Podczas tych operacji drzewo mogło stać się niespójne. Przedstawimy modyfikację tego algorytmu polegającą na tym, że z grafu będą odrywane tylko liście, przez co drzewo cały czas będzie spójne. Jeżeli rodzic danego liścia zostanie zaliczony do zbioru dominującego, to wszystkich sąsiadów rodzica (i samego rodzica) należy zaznaczyć jako zabronionych. Kiedy wierzchołek zabroniony stanie się liściem, to będzie on oderwany z grafu bez wykonywania dodatkowych czynności. W algorytmie należy również prawidłowo obsłużyć wierzchołki izolowane (listing 4.16). Testy algorytmu opisano w dodatku A.4.

Podany algorytm warto przedyskutować w kontekście uogólnienia na dowolne grafy cięciwowe. Odrywanie liści jest odrywaniem z grafu wierzchołków simplicjalnych wg kolejności z pewnego PEO. Jednak do zbioru dominującego w przypadku drzew nie zaliczamy liścia, tylko jego rodzica, który jest drugim wierzchołkiem w klice związanej z liściem. W przypadku ogólnych grafów cięciwowych można wydedukowć, że do zbioru dominującego nie zaliczymy wierzchołka simplicjalnego (chyba że został zaliczony we wcześniejszych fazach algorytmu), tylko dokładnie jednego z pozostałych wierzchołków z jego kliki (o ile wcześniej nie zaliczono jakichś sąsiadów). Niestety nie wiadomo z góry jaki wybór będzie najkorzystniejszy, więc potrzebna jest metoda programowania dynamicznego, w której budujemy kilka alternatywnych rozwiązań, a dopiero na końcu wybieramy najlepsze rozwiązanie. Liczba alternatywnych rozwiązań jest funkcją szerokości drzewowej, stąd widać znaczenie grafów o małej szerokości drzewowej.

Listing 4.16. Moduł treedset.

```
#!/usr/bin/python

from Queue import Queue

class TreeDominatingSet:
    """Find a maximum dominating set for forests."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.dominating_set = set()
        self.cardinality = 0
        self._used = set()

    def run(self):
        """Executable pseudocode."""
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())
        Q = Queue()    # for leafs
        for node in self.graph.iternodes():
            if degree_dict[node] == 0:
```


48:9565652479

```
        self.dominating_set.add(node)
        self._used.add(node)
        self.cardinality += 1
    elif degree_dict[node] == 1:
        Q.put(node)
while not Q.empty():
    source = Q.get()
    if degree_dict[source] == 0:
        if source not in self._used:
            self.dominating_set.add(source)
            self._used.add(source)
            self.cardinality += 1
        continue
    assert degree_dict[source] == 1
    for target in self.graph.iteradjacent(source):
        if degree_dict[target] > 0:    # this is parent
            if source not in self._used:
                self.dominating_set.add(target)
                self._used.add(target)
                self.cardinality += 1
                for node in self.graph.iteradjacent(target):
                    self._used.add(node)
            degree_dict[target] -= 1
            degree_dict[source] -= 1
            if degree_dict[target] == 1:
                Q.put(target)
            break
```

5. Podsumowanie

Głównym celem niniejszej pracy było zbadanie własności grafów cięciwowych, oraz pokazanie szybkich rozwiązań dla problemów, które są trudne i bardziej kosztowne obliczeniowe dla ogólnych grafów. W pracy została przedstawiona część teoretyczna opisująca własności grafów cięciwowych i ich powiązania z innymi klasami grafów. Zgromadzenie dużej liczby wyników teoretycznych pozwoliło na powiązanie ze sobą wielu zagadnień i zastosowanie ich w implementacji grafów cięciwowych, oraz w pozostałych algorytmach.

Stworzono bibliotekę odpowiadającą za generowanie grafów cięciwowych. Wykorzystując pojęcie wierzchołków simplicjalnych zaimplementowano rozpoznawanie grafów cięciwowych. Ważnym zagadnieniem charakteryzującym grafy cięciwowe jest doskonałe uporządkowanie wierzchołków (PEO). Uporządkowanie PEO wykorzystano m.in. do rozpoznawania i generacji grafów cięciwowych, znajdowania największej kliki, oraz do znajdowania wszystkich klik maksymalnych grafu cięciwowego. PEO można znaleźć m.in. za pomocą dwóch algorytmów: Lex-BFS i MCS. W pracy został wykorzystany algorytm MCS, który może być również przydatny jako heurystyka do znajdowania dolnego i górnego ograniczenia na szerokość drzewową dowolnych grafów. Znajomość PEO umożliwiła rozwiązanie niektórych problemów bez konieczności znajdowania dekompozycji drzewowej, oraz powiązanej z nią szerokości drzewowej.

Kolejnym istotnym punktem było zastosowanie w pracy uporządkowania najmniejszego stopnia (MDO). Jest to uporządkowanie, które stosuje się dla dowolnego grafu. W pracy MDO znalazło zastosowanie do poszukiwania największej kliki w grafie cięciwowym, a także do optymalnego kolorowania wierzchołków grafu cięciwowego.

Grafy cięciwowe umożliwiają rozwiązywanie problemów w czasie krótszym niż dla innych klas grafów. Potwierdzają to zaimplementowane w pracy algorytmy, które w większości mają złożoność liniową. Potrzebne były testy, aby potwierdzić czas działania algorytmów. Wyniki testów, wraz z ilustrującymi czas wykonania wykresami, znajdują się w dodatku A.

Ze względu na przydatne właściwości grafów cięciwowych ważny w praktyce jest problem uzupełnienia ogólnego grafu do grafu cięciwowego, najlepiej bez powiększania szerokości drzewowej. Jest to problem trudny, dlatego stosuje się metody przybliżone. W pracy zostały pokazane algorytmy wyznaczające dolne (algorytm MMD) i górne (heurystyka najmniejszego stopnia) ograniczenie na szerokość drzewową. Jeżeli te dwa ograniczenia są równe, to oznacza dokładne wyznaczenie szerokości drzewowej.

W polskiej literaturze nie ma monografii na temat grafów cięciwowych, dlatego istotne informacje o tych grafach zostały znalezione w zagranicznych źródłach naukowych. Niektóre implementacje algorytmów są oparte o książ-

50:1140987094

kę Golumbica [3], prace naukowe Blaira i Peytona [50], Luceny [48], oraz Bachoore i Bodlandera [51].

51:2744091374

A. Testy algorytmów

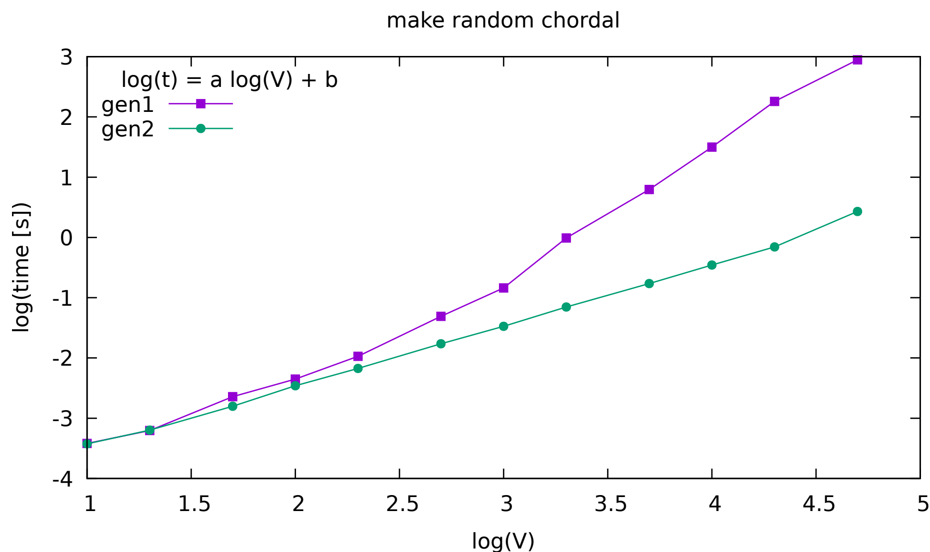
W niniejszej pracy zebrano opisy i przygotowano implementacje wielu algorytmów związanych z grafami cięciwowymi. Ostatnim elementem jest potwierdzenie w praktyce oczekiwanych wydajności przygotowanych implementacji poprzez odpowiednio dobrane testy.

A.1. Testy generatorów grafów cięciwowych

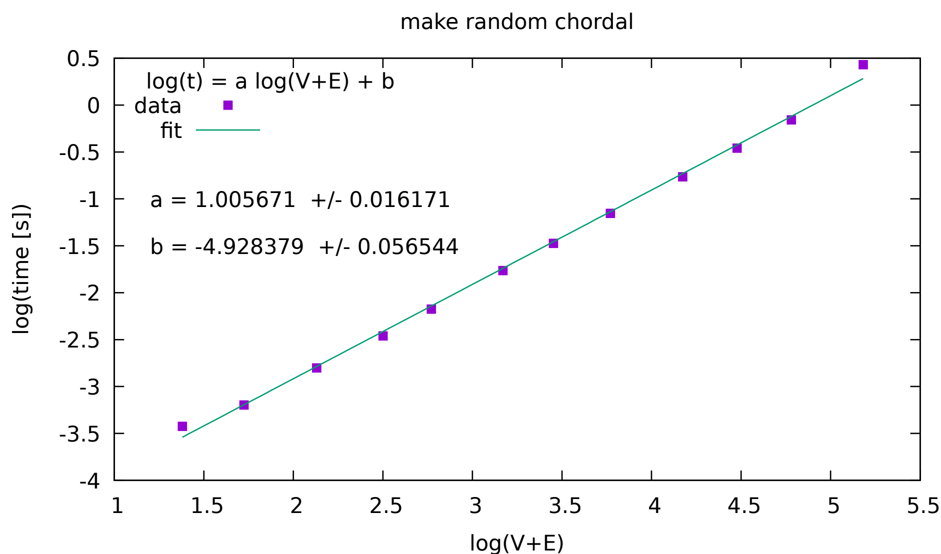
Testowanie generatorów grafów cięciwowych. Przygotowano dwa generatory przypadkowych grafów cięciwowych. Pierwszy generator przechowuje zbiór wszystkich klik w powstającym grafie cięciwowym, z którego w każdym kroku powiększania grafu losuje dowolną klikę z jednakowym prawdopodobieństwem. Drugi generator bazuje na powstającym PEO, a w każdym kroku powiększania grafu następują dwa losowania. Pierwsze losowanie wybiera pewną dużą klikę (czasem maksymalną), a drugie losowanie wybiera z niej mniejszą klikę do rozszerzenia.

Rysunek A.1 pokazuje większą szybkość drugiego generatora, szczególnie dla dużych grafów. Rysunek A.2 pokazuje, że drugi generator w praktyce działa w czasie rosnącym liniowo z liczbą wierzchołków.

52:1137086009



Rysunek A.1. Porównanie wydajności działania generatorów grafów cięciwowych. Generator pierwszy wykorzystuje zbiór klik, natomiast drugi generator korzysta z budowanego PEO. Na wykresie widać pogorszenie wydajności pierwszego generatora wraz ze wzrostem liczby wierzchołków (i liczby klik).



Rysunek A.2. Wykres wydajności generatora grafów cięciwowych bazującego na PEO. Współczynnik a bliski jedności sugeruje złożoność liniową algorytmu.

53:6043706377

A.2. Testy rozpoznawania grafów cięciwowych

Testowanie rozpoznawania grafów cięciwowych. Pierwsza metoda polega na usuwaniu z grafu kolejnych wierzchołków simplicjalnych, aż do wyczerpania zbioru wierzchołków grafu. Jeżeli przed wyczerpaniem zbioru wierzchołków nie można znaleźć wierzchołka simplicjalnego, to graf nie jest cięciwowy.

Druga metoda ma dwa etapy. W pierwszym etapie wyznaczamy uporządkowanie wierzchołów, które jest kandydatem na PEO (algorytmy Lex-BFS lub MCS). W drugim etapie faktycznie sprawdzamy, czy znalezione uporządkowanie jest PEO. Oba etapy mają w teorii liniową złożoność obliczeniową.

Pierwszy test wydajności metody wierzchołków simplicjalnych pokazał liniową zależność czasu obliczeń od liczby wierzchołków A.3, przy czym implementacja z kopią grafu jest wolniejsza od implementacji ze zbiorami sąsiedztwa A.4. Można to wyjaśnić właściwościami naszego generatora grafów cięciwowych. Generator tworzy grafy rzadkie, w których dominują małe kliki. Wtedy sprawdzanie klik i usuwanie wierzchołków z grafu jest szybkie, a czas pracy algorytmu staje się proporcjonalny do wielkości grafu (liczby wierzchołków).

W celu lepszego przetestowania algorytmu musieliśmy wymusić obecność większych klik. Wykorzystaliśmy generator k-drzew z innego projektu, ponieważ wtedy wszystkie kliki mają taką samą wielkość, którą można z góry zadać generatorowi. Przy wyborze $k = n/2$ ($n = |V|$), co daje gęste grafy z $|E| = n(3n-2)/8 = O(V^2)$, testy sugerują złożoność algorytmu rzędu $O(V^3)$ A.5.

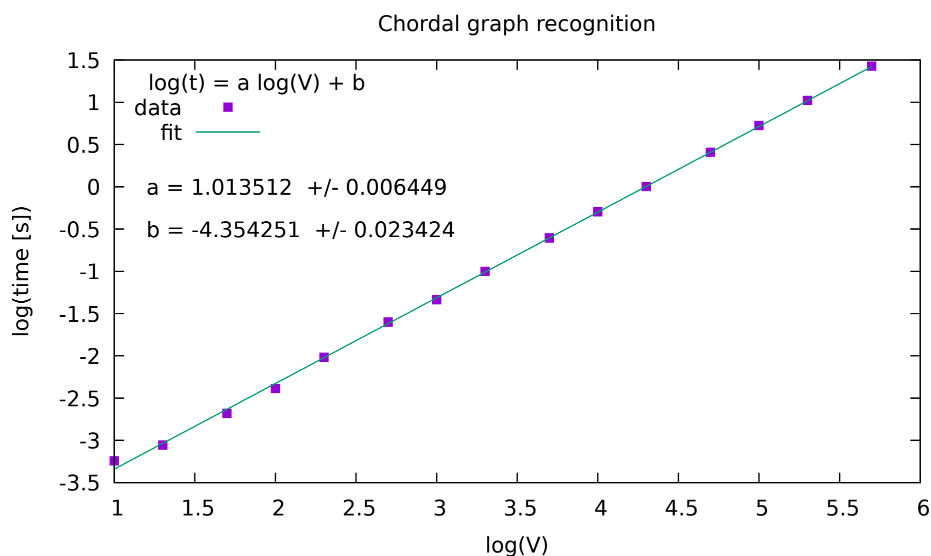
Sprawdzanie poprawności wyznaczenia PEO wykonano dla przypadkowych grafów cięciwowych i k-drzew [funkcja `is_peo()` w dwóch wersjach]. Wykresy A.6, A.7 potwierdzają złożoność $O(V+E)$. Odrobinę szybciej wykonywała się wersja wykorzystująca pythonowe operacje na zbiorach, a nie wersja wykorzystująca pomocniczy słownik.

A.3. Testy zbiorów niezależnych w grafach cięciwowych

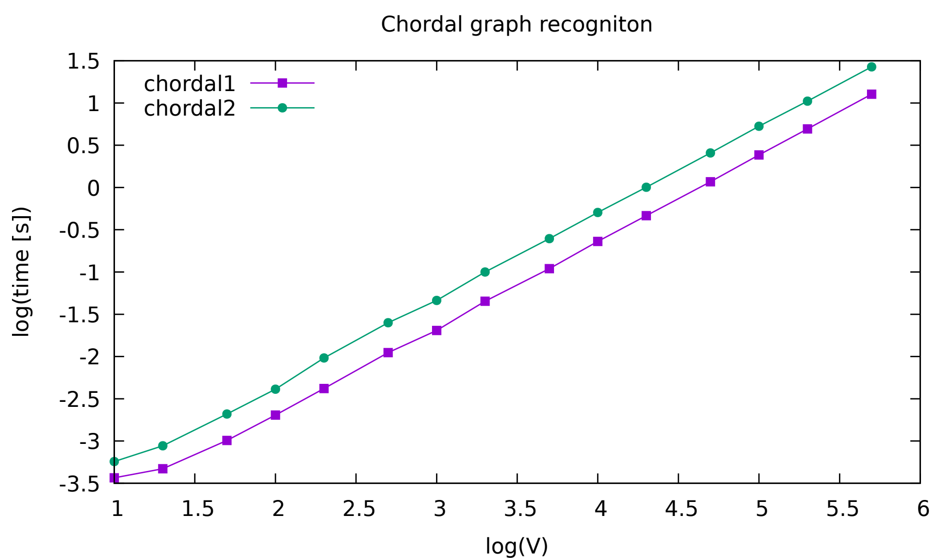
Testowanie algorytmu wyznaczającego największy zbiór niezależny dla drzew metodą odrywania liści. Wyniki przedstawia wykres A.8. Dostajemy zależność $O(V)$ dla drzew przypadkowych.

Testowanie algorytmu wyznaczającego największy zbiór niezależny dla dowolnego grafu cięciwowego. Testy z generatorem przypadkowych grafów cięciwowych pokazują złożoność $O(V)$, ale jednocześnie też $O(V+E)$. Testy z wykorzystaniem k-drzew potwierdzają, że właściwe oszacowane czasu to $O(V + E)$ (wykres A.9).

54:9041085889

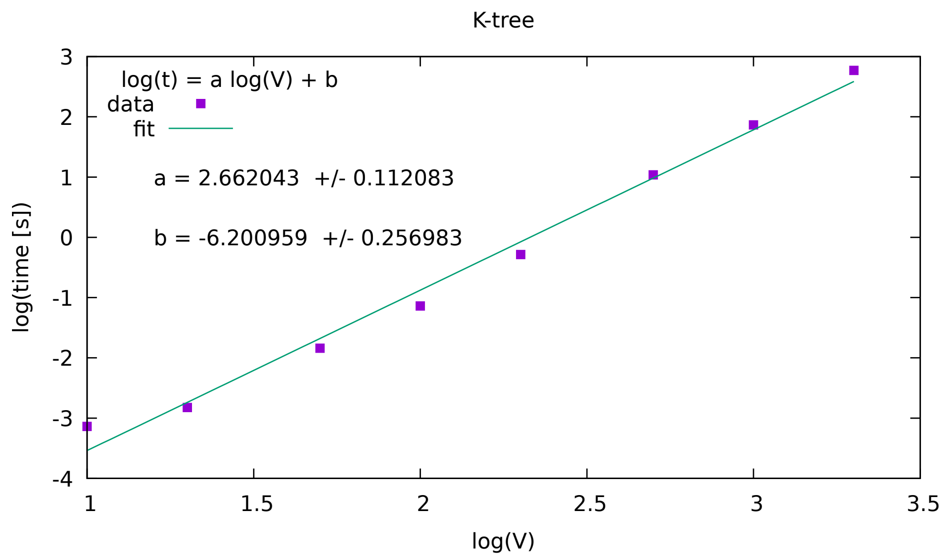


Rysunek A.3. Wykres wydajności rozpoznawania grafów cięciwowych metodą wierzchołków simplicjalnych - wersja z kopią grafu. Współczynnik a bliski jeden wynika z małej gęstości grafów cięciwowych tworzonych przez generator.

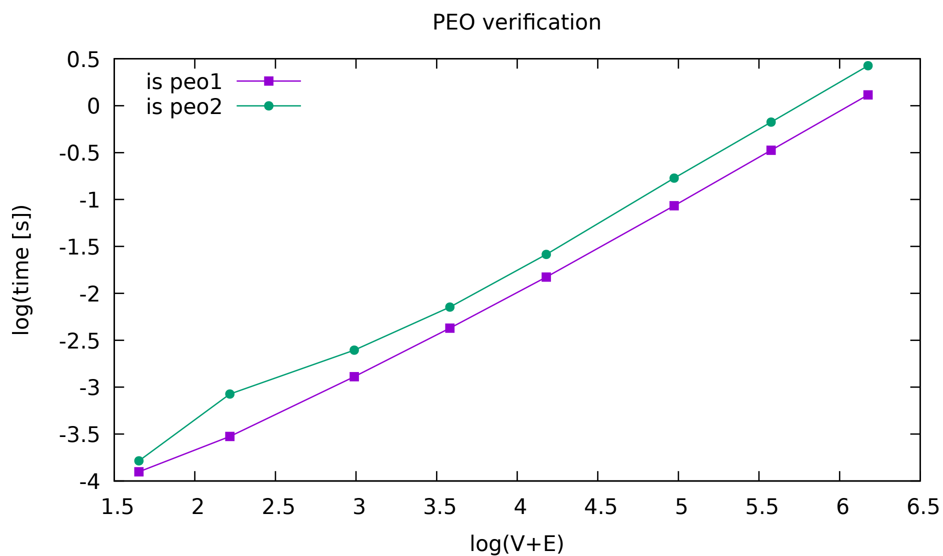


Rysunek A.4. Wykres porównujący wydajność rozpoznawania grafów cięciwowych metodą wierzchołków simplicialnych. Pierwsza metoda operuje na słowniku ze zbiorem sąsiadów. Druga metoda wykorzystuje operacje na kopii grafu.

55:1126548493

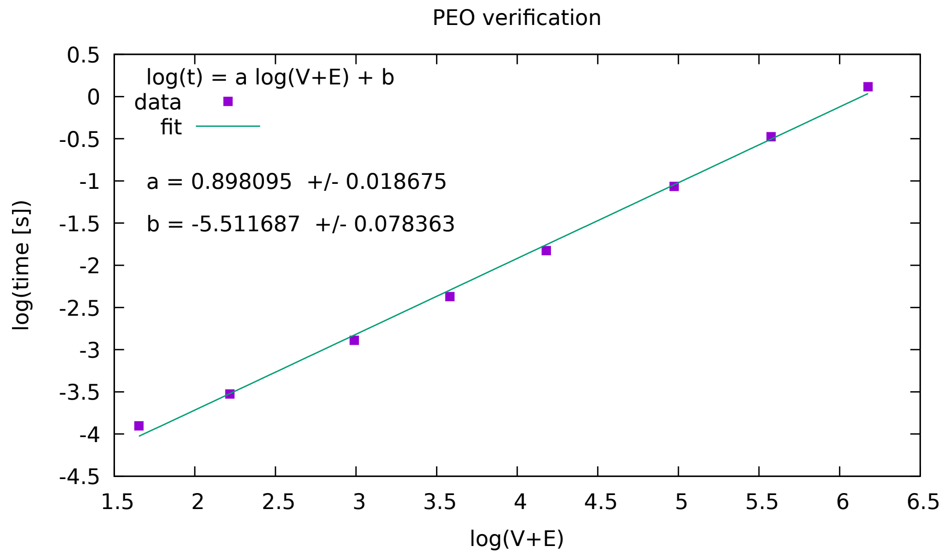


Rysunek A.5. Wykres wydajności rozpoznawania grafów cięciwowych metodą wierzchołków simplicialnych dla k-drzew. Wydajność jest bliska $O(V^3)$.

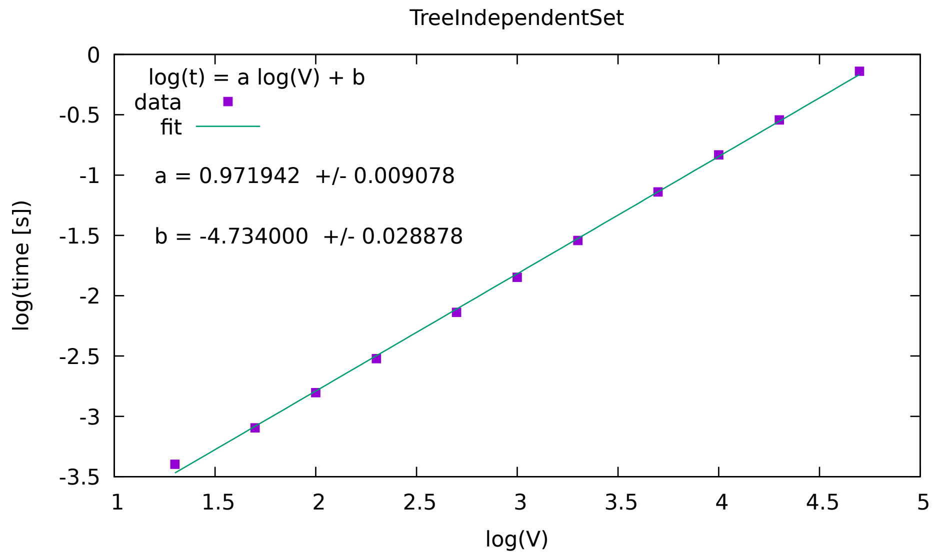


Rysunek A.6. Wykres porównujący wydajność sprawdzania poprawności PEO dla k-drzew.

56:6198065973

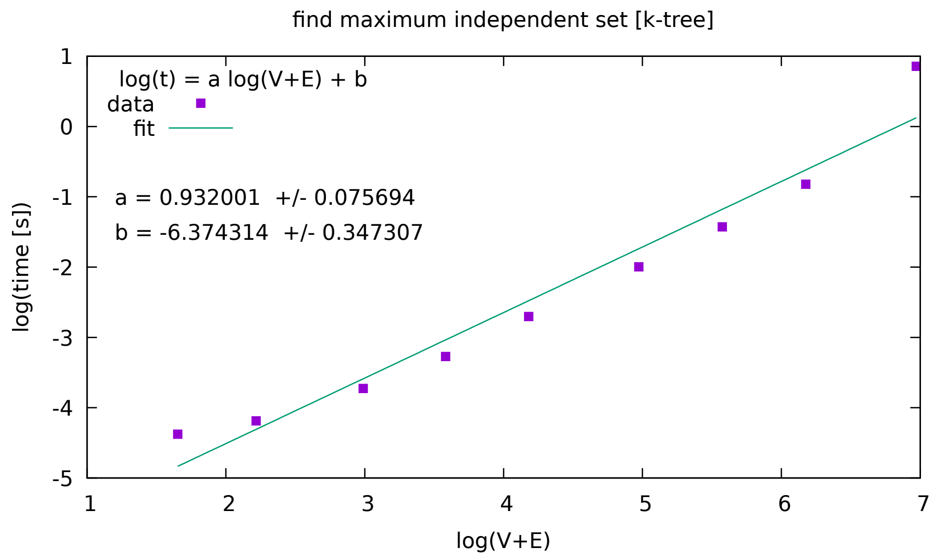


Rysunek A.7. Wykres sprawdzający PEO dla k-drzew, wersja wykorzystująca pythonowe operacje na zbiorach.

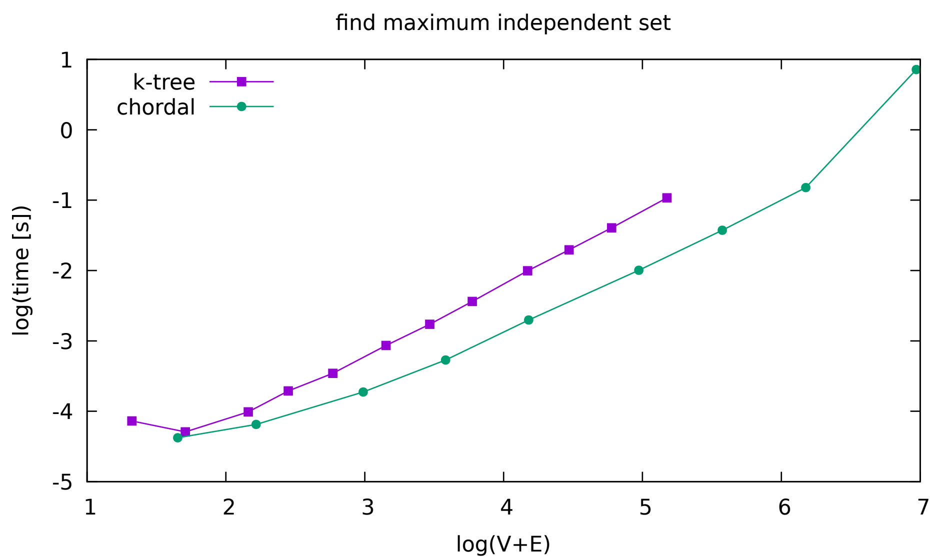


Rysunek A.8. Wykres wydajności wyznaczania największego zbioru niezależnego dla drzew metodą odrywania liści. Współczynnik a bliski 1 potwierdza zależność $O(V)$.

57:7934299423



Rysunek A.9. Wykres wydajności wyznaczania największego zbioru niezależnego dla k-drzew. Współczynnik a bliski 1 potwierdza złożoność $O(V+E)$ algorytmu.



Rysunek A.10. Porównanie wydajności wyznaczania największego zbioru niezależnego dla k-drzew i przypadkowych grafów cięciwowych. Widać drobne anomalie dla bardzo małych i bardzo dużych grafów.

58:8212043687

A.4. Testy zbiorów dominujących w grafach cięciwowych

Testowanie algorytmu wyznaczającego najmniejszy zbiór dominujący dla drzew metodą odrywania liści. Wyniki przedstawia wykres A.11. Dostajemy zależność $O(V)$ dla drzew przypadkowych.

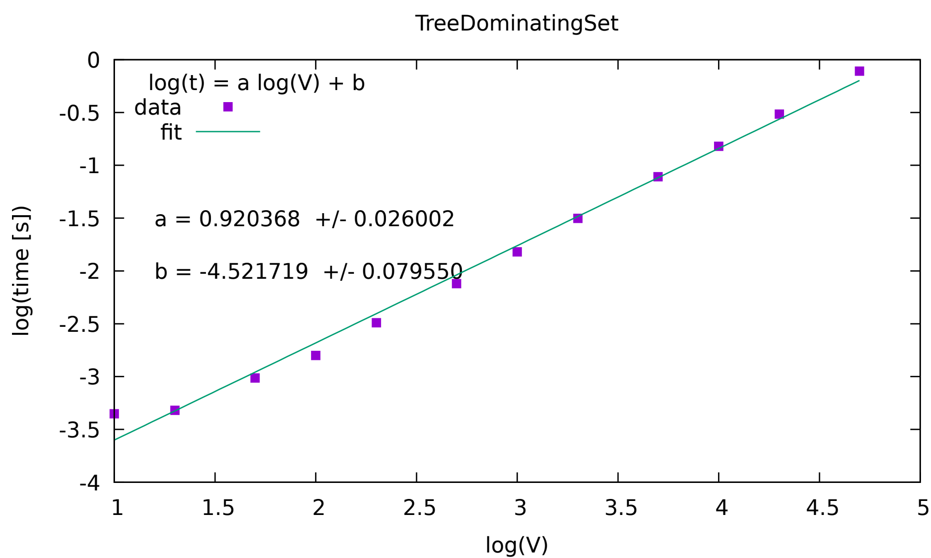
A.5. Testy pokrycia wierzchołkowego w grafach cięciwowych

Testowanie algorytmu wyznaczającego najmniejsze pokrycie wierzchołkowe dla drzew metodą odrywania liści. Wyniki przedstawia wykres A.12. Dostajemy zależność $O(V)$ dla drzew przypadkowych.

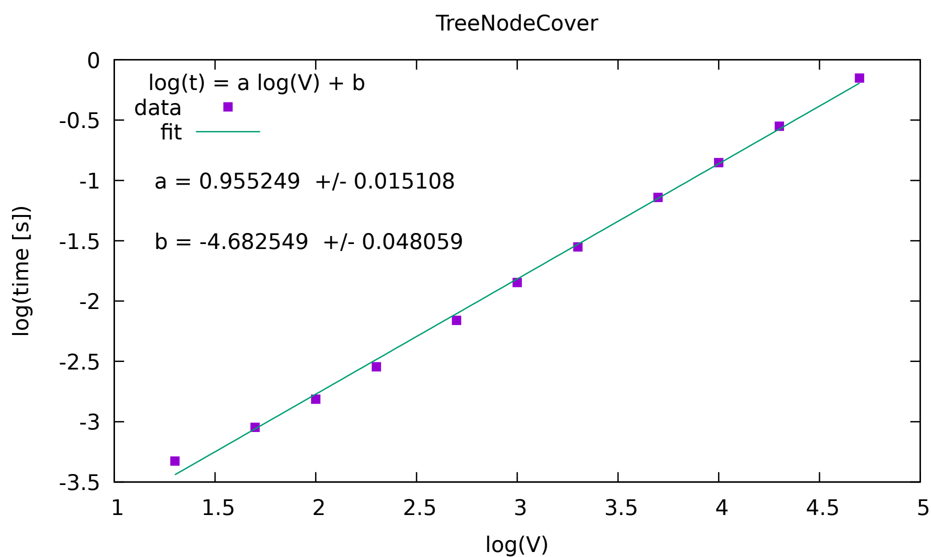
A.6. Testy znajdowania MDO

Testy znajdowania MDO dla dowolnego grafu. Wykorzystano grafy przypadkowe gęste ($p = 0.5$) i rzadkie ($p = 0.1$), przy czym p jest prawdopodobieństwem wystąpienia krawędzi pomiędzy dwoma wierzchołkami. Wykresy A.13, A.14, A.15 potwierdzają zależność $O(V^2)$ prostej metody. Wykresy A.16, A.17, A.18 potwierdzają zależność $O(V + E)$ dla wersji z sortowaniem bukietowym.

59:9010549601

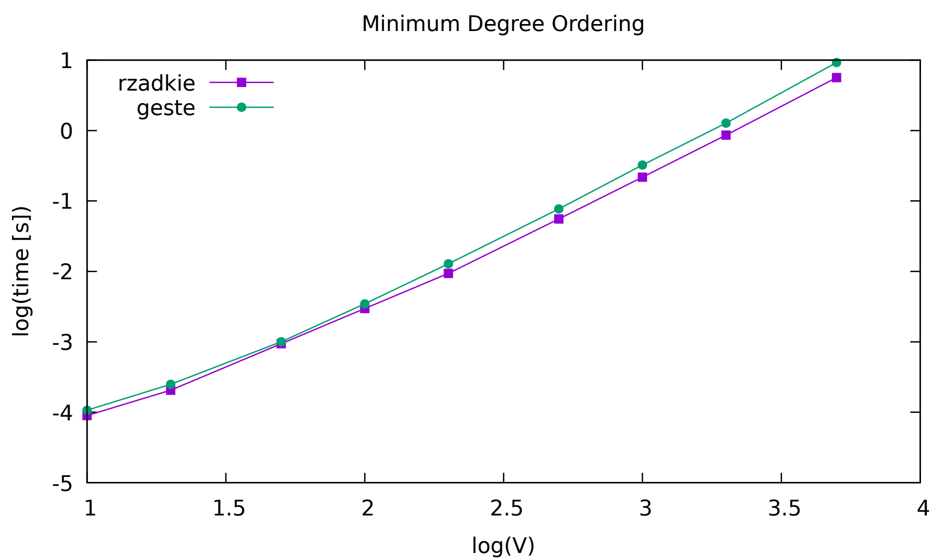


Rysunek A.11. Wykres wydajności wyznaczania najmniejszego zbioru dominującego dla drzew. Współczynnik a bliski 1 potwierdza złożoność $O(V)$.

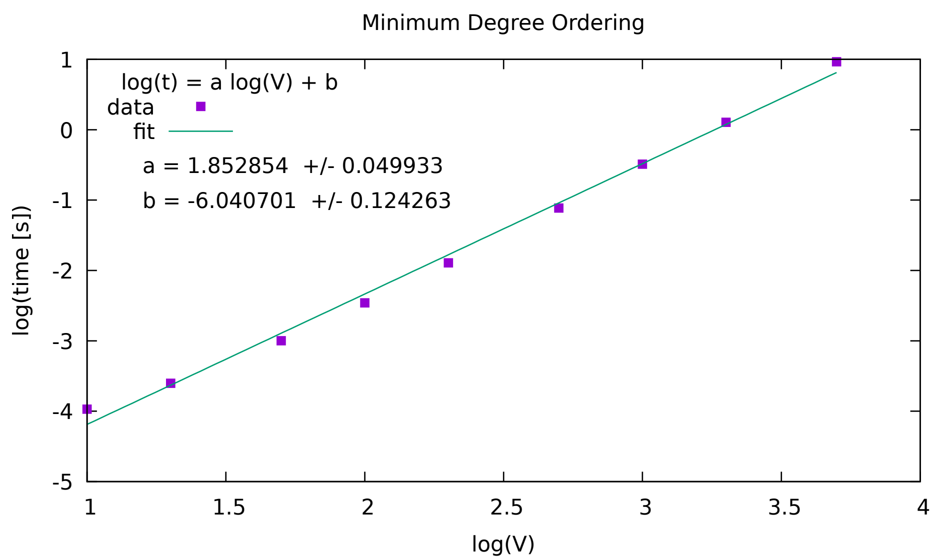


Rysunek A.12. Wykres wydajności wyznaczania pokrycia wierzchołkowego dla drzew. Współczynnik a bliski 1 potwierdza złożoność $O(V)$.

60:4133780318

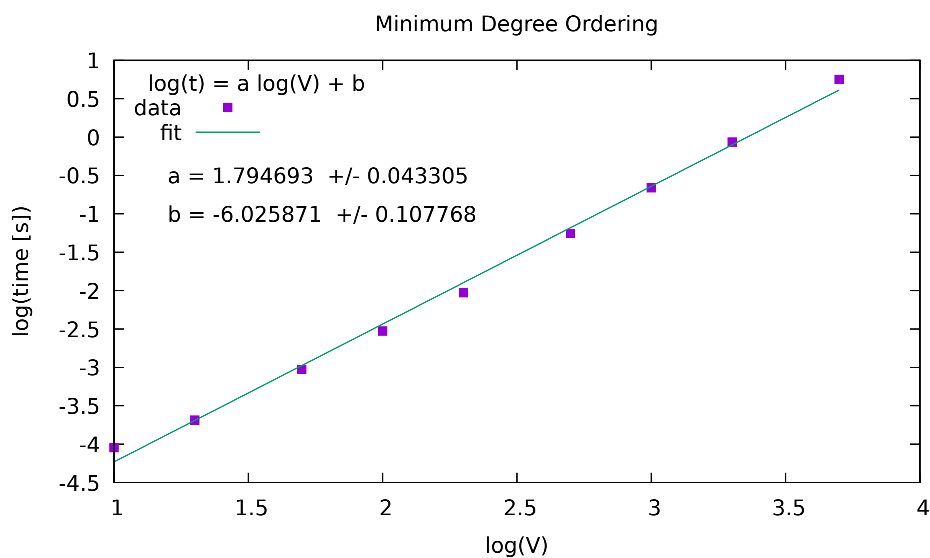


Rysunek A.13. Porównanie wydajności znajdowania MDO dla grafów rzadkich i gęstych.

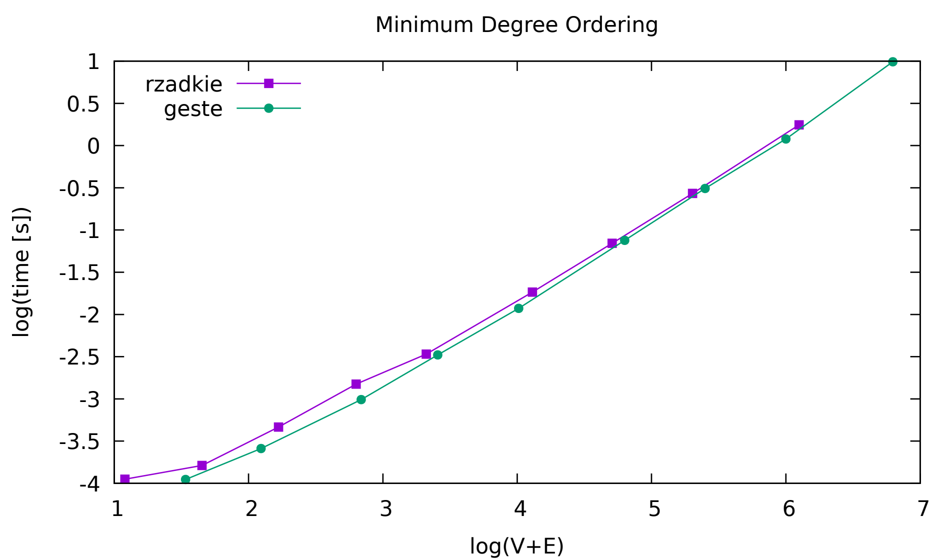


Rysunek A.14. Wykres wydajności znajdowania MDO dla dowolnego gęstego grafu. Współczynnik a bliski 2 potwierdza złożoność $O(V^2)$.

61:5431555327

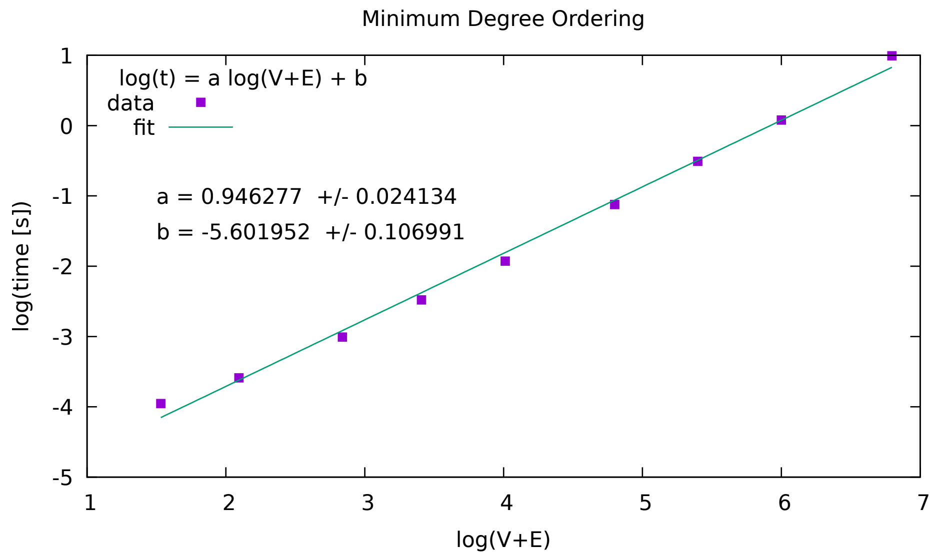


Rysunek A.15. Wykres wydajności znajdowania MDO dla dowolnego rzadkiego grafu. Współczynnik a bliski 2 potwierdza złożoność $O(V^2)$.

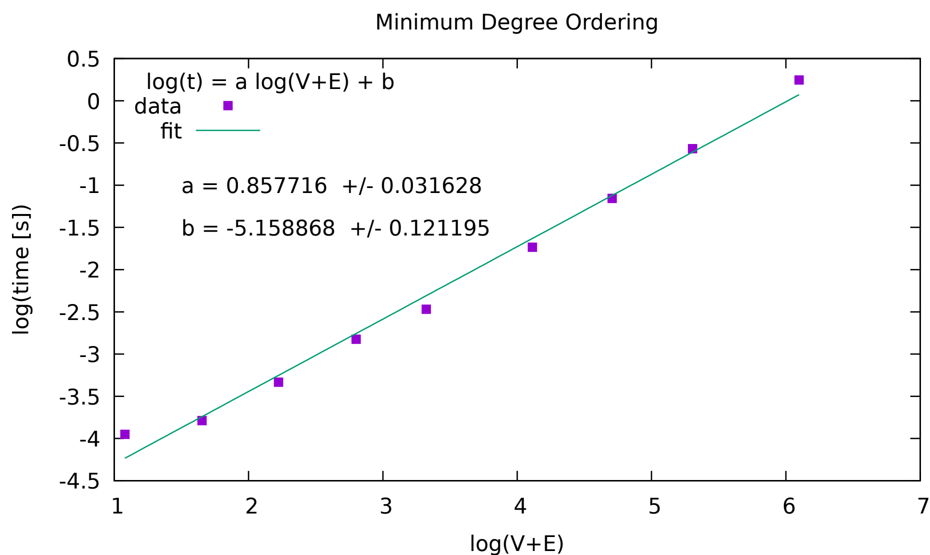


Rysunek A.16. Porównanie wydajności znajdowania MDO z wykorzystaniem sortowania bukietowego dla grafów rzadkich i gęstych.

62:2819908349



Rysunek A.17. Wykres wydajności znajdowania MDO z wykorzystaniem sortowania bukietowego dla dowolnego gęstego grafu. Współczynnik a bliski 1 potwierdza złożoność $O(V + E)$.



Rysunek A.18. Wykres wydajności znajdowania MDO z wykorzystaniem sortowania bukietowego dla dowolnego rzadkiego grafu. Współczynnik a bliski 1 potwierdza złożoność $O(V + E)$.

63:1086528287

A.7. Testy znajdowania największej kliki

Testy znajdowania największej kliki w grafie cięciwowym za pomocą MDO. Algorytm jednocześnie wyznacza MDO i największą klikę. Do testów wykorzystano generator grafów cięciwowych przypadkowych i generator k-drzew. Wykresy A.19, A.20 wskazują na drobne odchylenie od zależności $O(V+E)$ dla rzadkich grafów cięciwowych.

Testy znajdowania największej kliki w grafie cięciwowym za pomocą PEO. Algorytm na wejściu otrzymuje graf cięciwowy i jego PEO. Do testów wykorzystano generator grafów cięciwowych przypadkowych i generator k-drzew. Wykresy A.21, A.22 potwierdzają zależność $O(V+E)$.

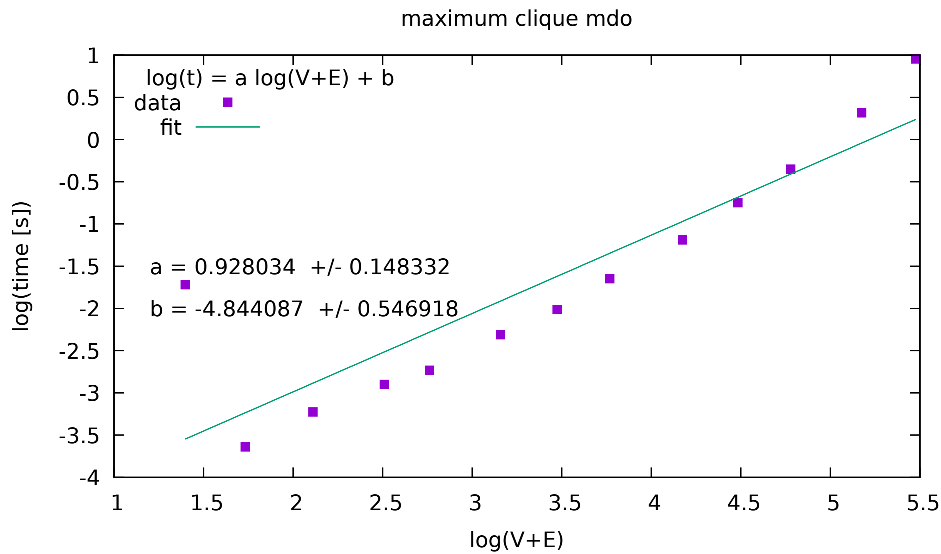
A.8. Testy znajdowania klik maksymalnych

Testy znajdowania wszystkich klik maksymalnych w grafie cięciwowym [funkcja find_all_maximal_cliques()]. Algorytm na wejściu otrzymuje graf cięciwowy i jego PEO. Do testów wykorzystano generator grafów cięciwowych przypadkowych i generator k-drzew. Wykresy A.23, A.24 potwierdzają zależność $O(V+E)$. Wykres A.25 pokazuje porównanie działania algorytmu dla k-drzew i grafów cięciwowych.

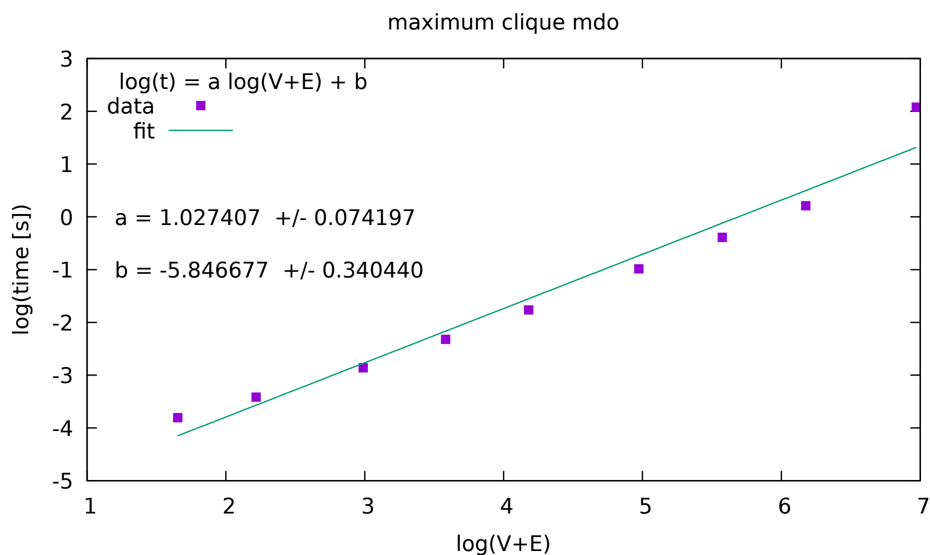
A.9. Testy algorytmu MCS

Testy algorytmu MCS znajdującego PEO dla grafu cięciwowego. Do testów wykorzystano generator grafów cięciwowych przypadkowych i generator k-drzew. Wykresy A.26, A.27 potwierdzają zależność $O(V^2)$ prostej metody. Wykresy A.28, A.29 potwierdzają zależność $O(V+E)$ dla wersji z sortowaniem bukietowym. Wykres A.30 pokazuje różnicę czasu wykonywania różnych wersji algorytmu MCS.

64:6010050316

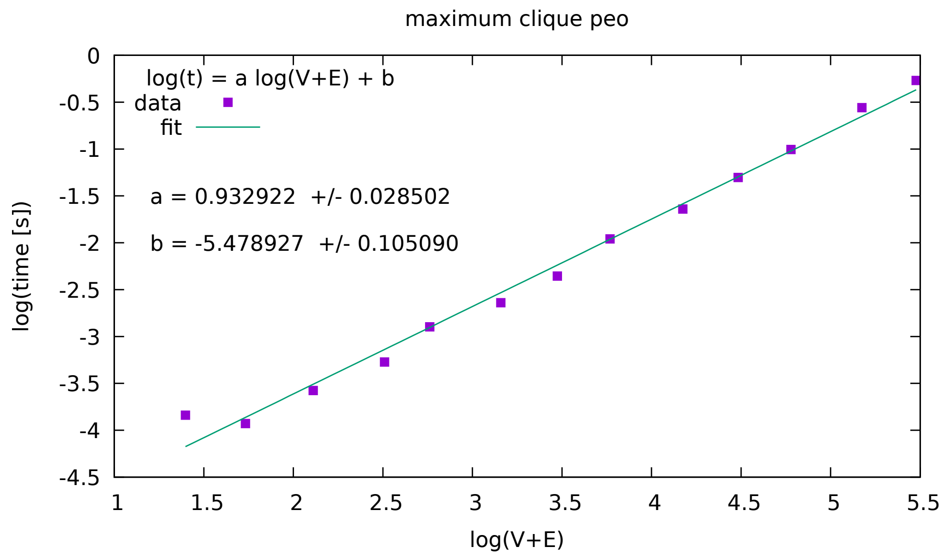


Rysunek A.19. Wykres wydajności znajdowania największej kliki w grafie cięciwowym za pomocą MDO. Współczynnik a bliski 1 potwierdza złożoność $O(V+E)$.

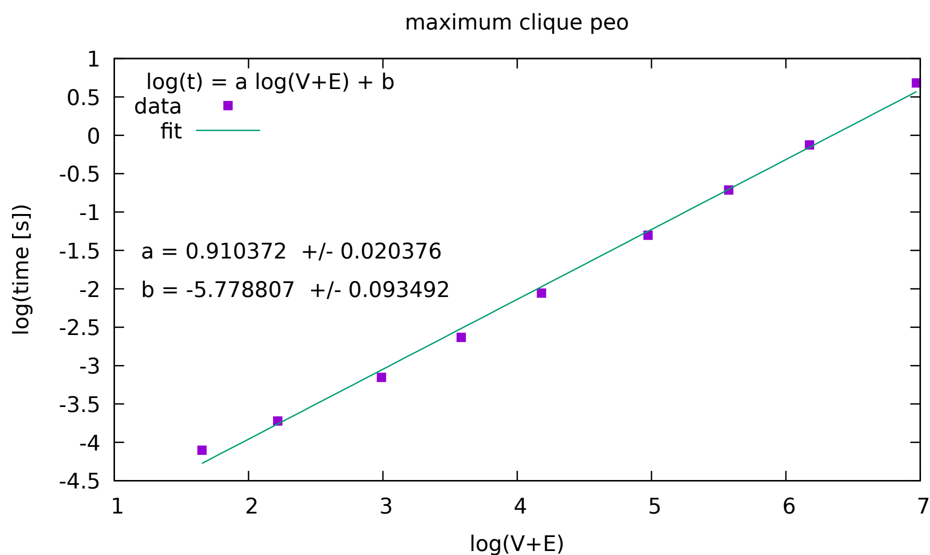


Rysunek A.20. Wykres wydajności znajdowania największej kliki dla k-drzewa za pomocą MDO. Współczynnik a bliski 1 potwierdza złożoność $O(V+E)$.

65:2477856284

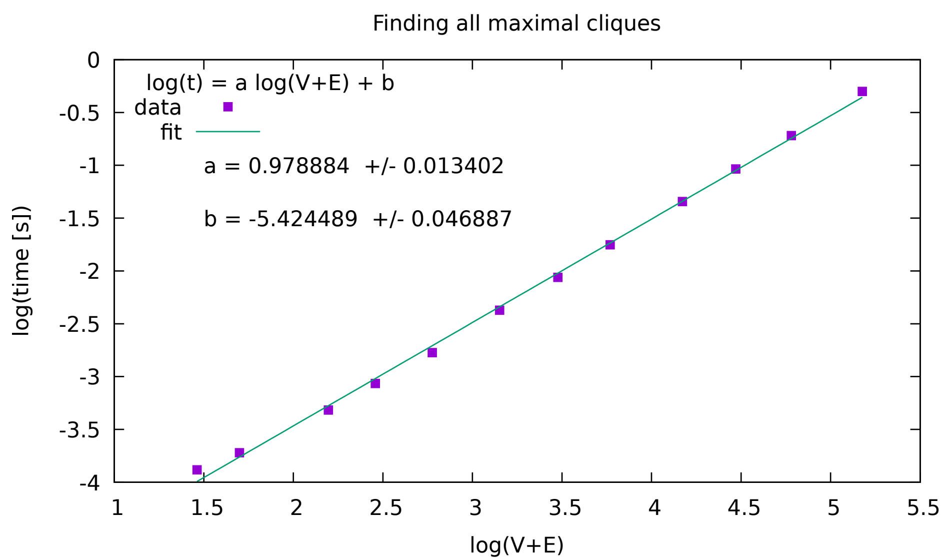


Rysunek A.21. Wykres wydajności znajdowania największej kliki w grafie cięciwowym za pomocą PEO. Współczynnik a bliski 1 potwierdza złożoność $O(V + E)$.

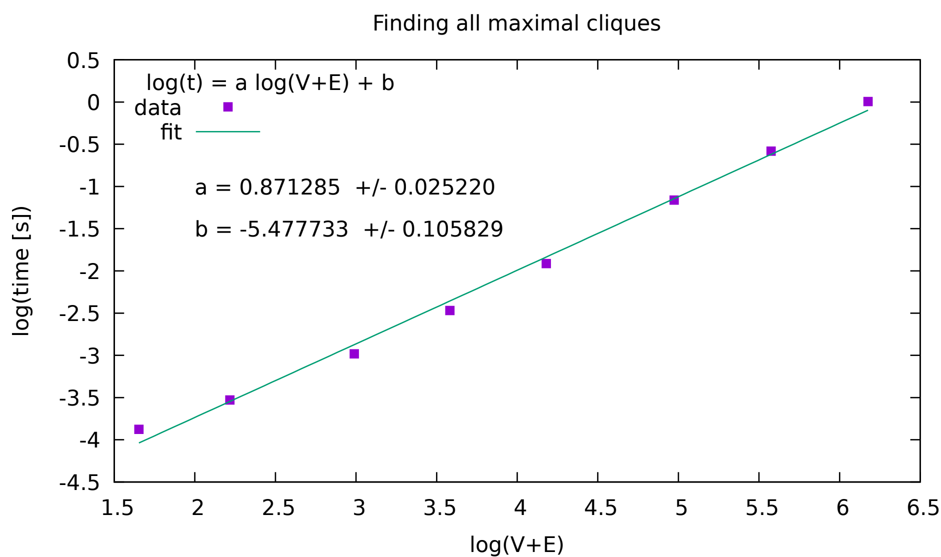


Rysunek A.22. Wykres wydajności znajdowania największej kliki dla k-drzewa za pomocą PEO. Współczynnik a bliski 1 potwierdza złożoność $O(V + E)$.

66:2559956576

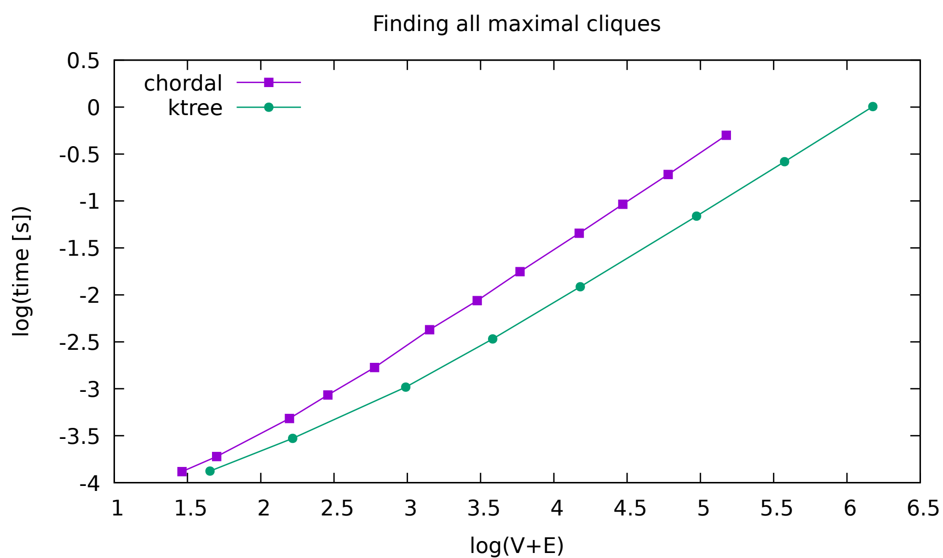


Rysunek A.23. Wykres wydajności znajdowania klik maksymalnych dla grafu cięciwowego. Współczynnik a bliski 1 potwierdza złożoność $O(V + E)$.

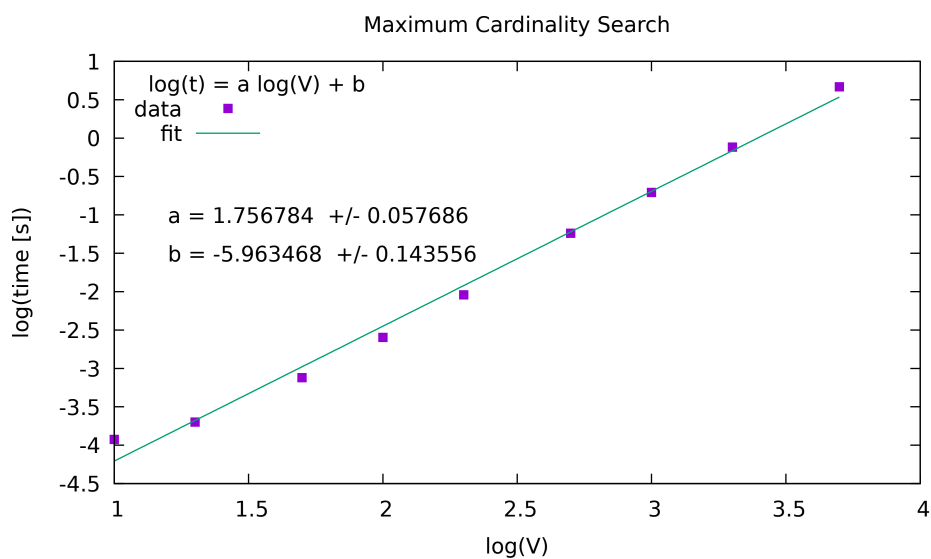


Rysunek A.24. Wykres wydajności znajdowania klik maksymalnych dla k-drzew. Współczynnik a bliski 1 potwierdza złożoność $O(V + E)$.

67:4601243853

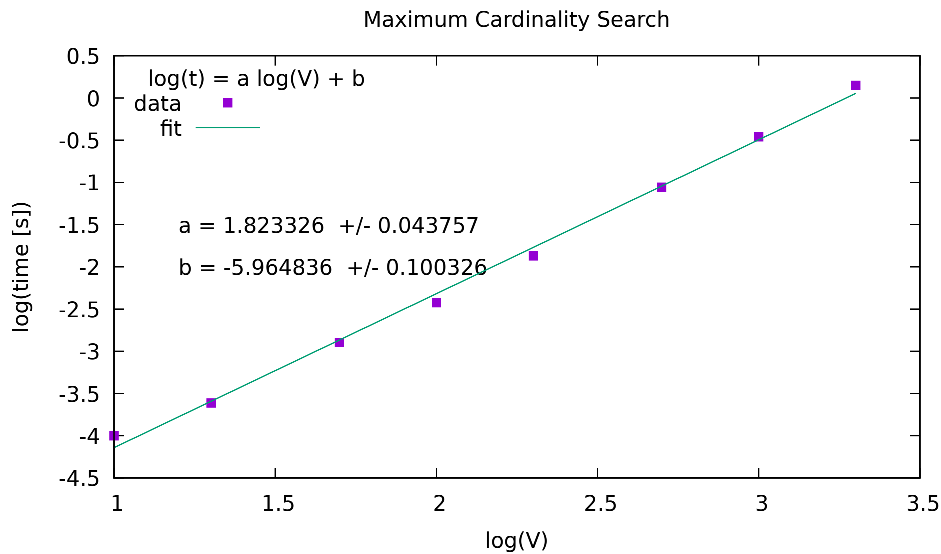


Rysunek A.25. Wykres porównujący wydajności znajdowania klik maksymalnych.

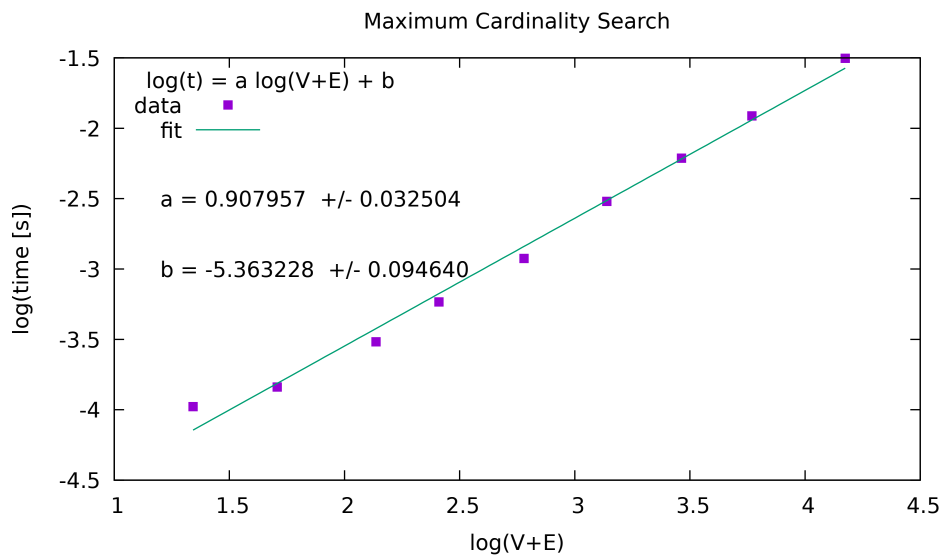


Rysunek A.26. Wykres wydajności algorytmu MCS dla grafów cięciwowych. Współczynnik a bliski 2 potwierdza złożoność $O(V^2)$.

68:5135368180

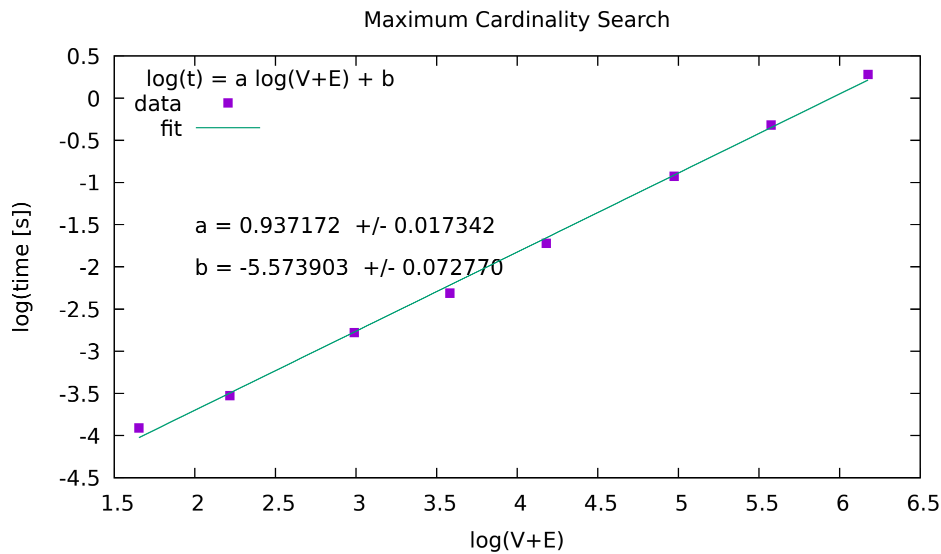


Rysunek A.27. Wykres wydajności algorytmu MCS dla k-drzew. Współczynnik a bliski 2 potwierdza złożoność $O(V^2)$.

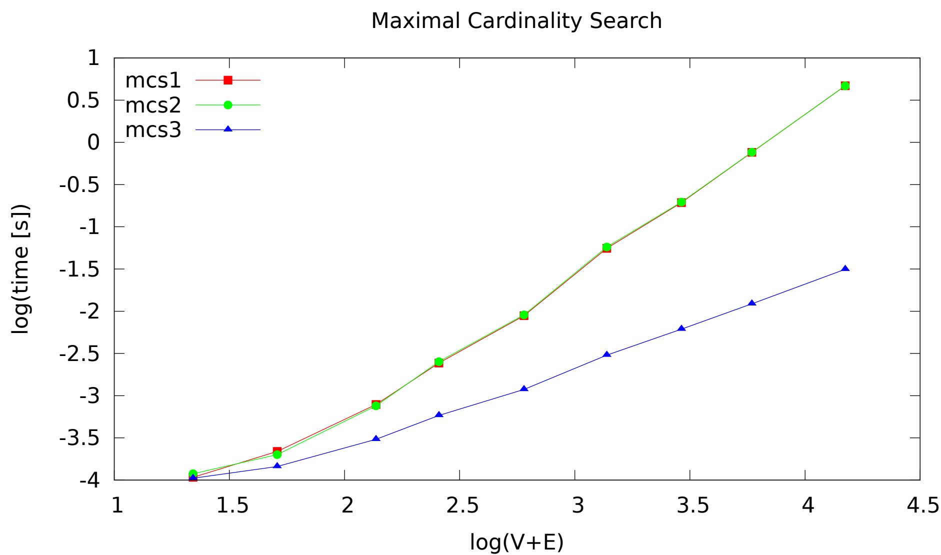


Rysunek A.28. Wykres wydajności algorytmu MCS z sortowaniem bukietowym dla grafów cięciwowych. Współczynnik a bliski 1 potwierdza złożoność $O(V + E)$.

69:2604886448



Rysunek A.29. Wykres wydajności algorytmu MCS z sortowaniem bukietowym dla k-drzew. Współczynnik a bliski 1 potwierdza złożoność $O(V+E)$.



Rysunek A.30. Wykres porównania wydajności algorytmów MCS dla grafów cięciwowych. Pierwsze dwie wersje działają w czasie $O(V^2)$, trzecia wersja z sortowaniem bukietowym działa w czasie liniowym $O(V+E)$.

70:2880449175

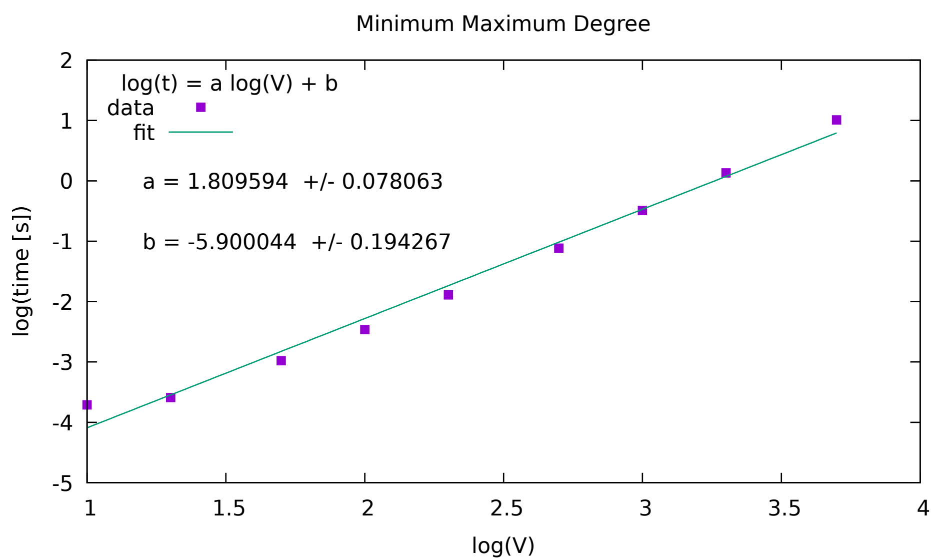
A.10. Testy algorytmu MMD

Testy algorytmu MMD znajdującego dolne ograniczenie na szerokość drzewową dowolnego grafu przy wykorzystaniu uporządkowania MDO [funkcja find_treewidth_mmd() w dwóch wersjach]. Wykorzystano grafy przypadkowe gęste ($p = 0.5$) i rzadkie ($p = 0.1$), przy czym p jest prawdopodobieństwem wystąpienia krawędzi pomiędy dwoma wierzchołkami. Wykresy A.31, A.32 potwierdzają zależność $O(V^2)$ prostej metody. Wykresy A.33, A.34 potwierdzają zależność $O(V + E)$ dla wersji z sortowaniem bukietowym.

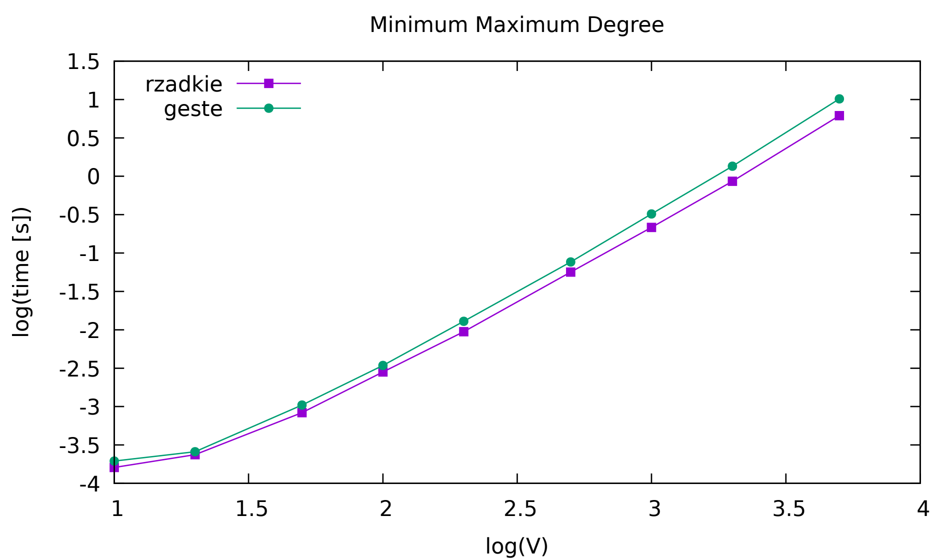
A.11. Testy heurystyki najmniejszego stopnia

Testy heurystyki najmniejszego stopnia znajdującej górne ograniczenie na szerokość drzewową dowolnego grafu [funkcja find_treewidth_min_deg() w dwóch wersjach]. Wykorzystano grafy przypadkowe gęste ($p = 0.5$) i rzadkie ($p = 0.1$), przy czym p jest prawdopodobieństwem wystąpienia krawędzi pomiędy dwoma wierzchołkami. Wykresy A.35, A.36 potwierdzają zależność $O(V^3)$ dla wersji ze zbiorami sąsiedztwa. Wykresy A.37, A.38 potwierdzają zależność $O(V^3)$ dla wersji z kopią grafu. Można zaobserwować, że współczynnik dopasowania prostej a maleje ze wzrostem gęstości grafu w obu wersjach heurystyki. Możliwe, że dla grafów rzadkich algorytm musi uzupełniać więcej krawędzi w grafie, aby otrzymać kliki pomiędzy sąsiadami.

71:8176863433

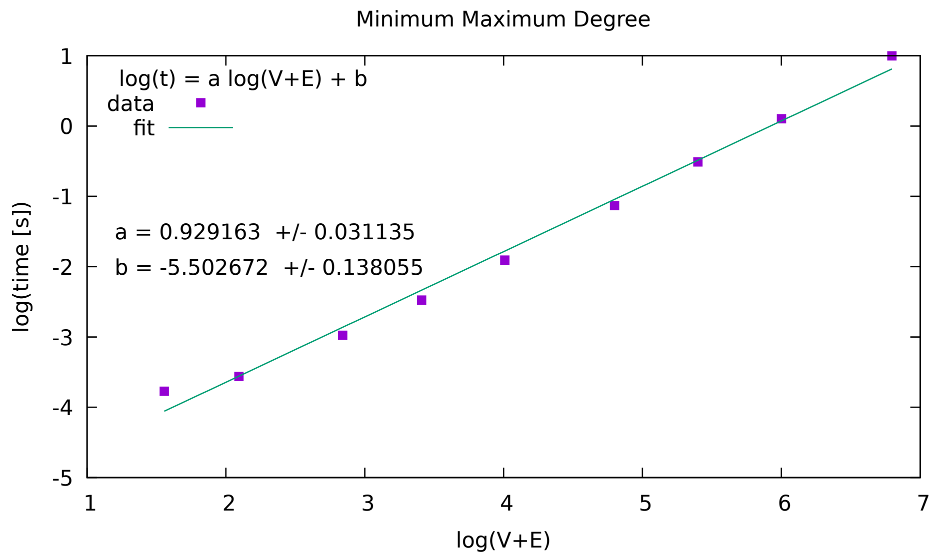


Rysunek A.31. Wykres wydajność algorytmu MMD dla grafu gęstego. Współczynnik a bliski 2 potwierdza złożoność $O(V^2)$.

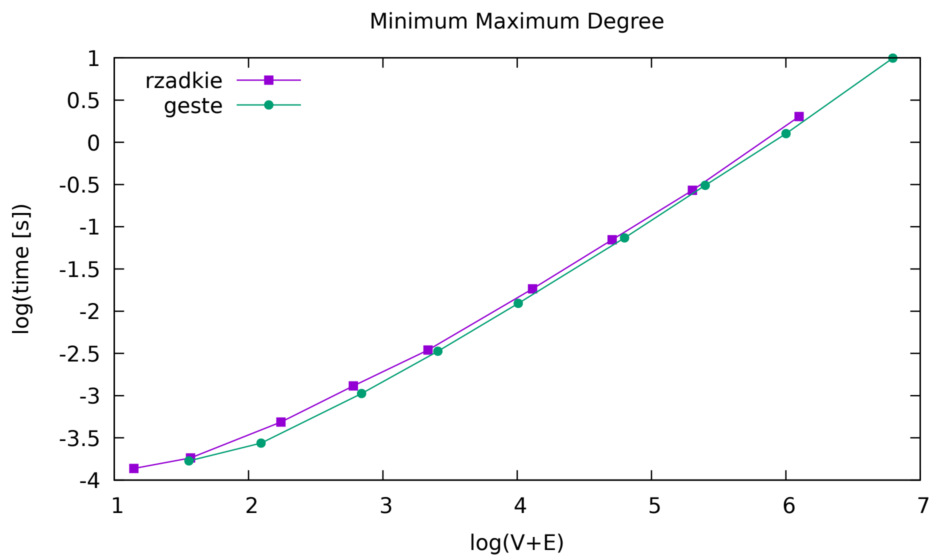


Rysunek A.32. Wykres porównania wydajności algorytmu MMD dla różnych grafów. Zależność kwadratowa wydajności nie zależy od gęstości grafu.

72:3914775627

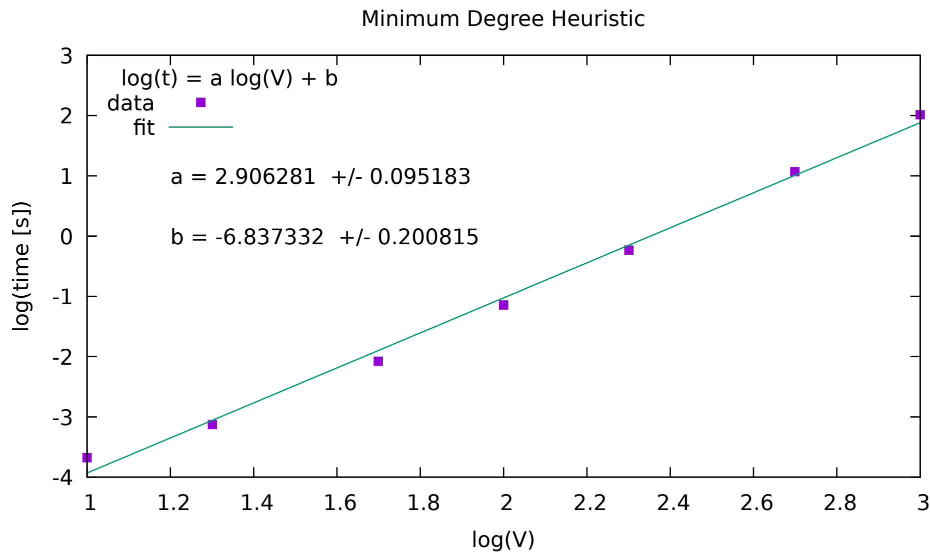


Rysunek A.33. Wykres wydajności algorytmu MMD z sortowaniem bukietowym dla grafu gęstego. Współczynnik a bliski 1 potwierdza złożoność $O(V+E)$.

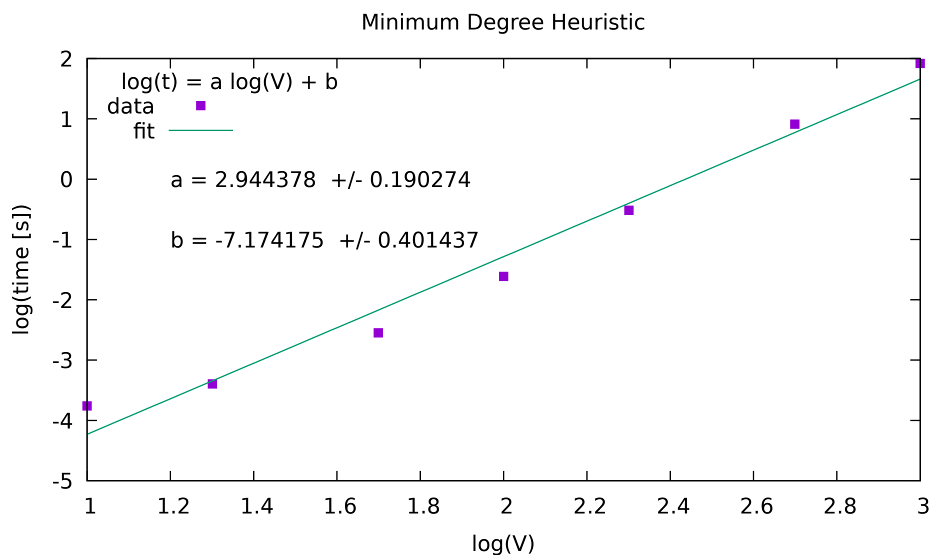


Rysunek A.34. Wykres porównania wydajności algorytmu MMD z sortowaniem bukietowym dla różnych grafów. Złożoność liniowa wydajności nie zależy od gęstości grafu.

73:7916490760

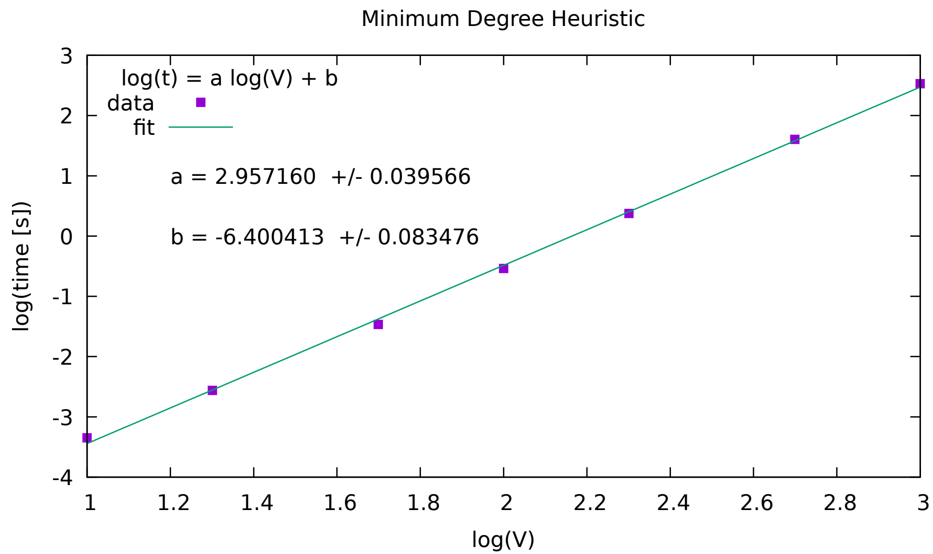


Rysunek A.35. Wykres wydajności heurystyki najmniejszego stopnia dla grafów gęstych (wersja 1 z listami zbiorów sąsiedztwa). Współczynnik a bliski 3 potwierdza złożoność $O(V^3)$.

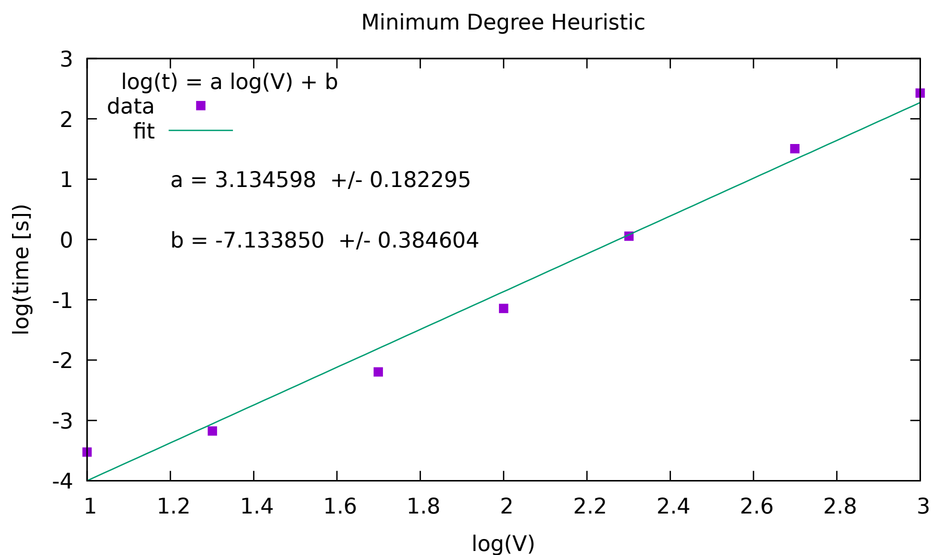


Rysunek A.36. Wykres wydajności heurystyki najmniejszego stopnia dla grafów rzadkich (wersja 1 z listami zbiorów sąsiedztwa). Współczynnik a bliski 3 potwierdza złożoność $O(V^3)$.

74:8084966871



Rysunek A.37. Wykres wydajności heurystyki najmniejszego stopnia dla grafów gęstych (wersja 2 z kopią grafu). Współczynnik a bliski 3 potwierdza złożoność $O(V^3)$.



Rysunek A.38. Wykres wydajności heurystyki najmniejszego stopnia dla grafów rzadkich (wersja 2 z kopią grafu). Współczynnik a bliski 3 potwierdza złożoność $O(V^3)$.

75:4964560537

Bibliografia

[1] Wikipedia, Chordal graph, 2017, https://en.wikipedia.org/wiki/Chordal_graph.

[2] Lieven Vandenberghe, Martin S. Andersen, *Chordal Graphs and Semidefinite Optimization*, Foundations and Trends® in Optimalization 1, No. 4, 242-244 (2014).

[3] Martin Charles Golumbic, *Algorithmic Graph Theory and Perfect Graphs, Second Edition*, Annals of Discrete Mathematics, Volume 57, Elsevier 2004 [First edition 1980].

[4] F. M. Q. Pereira, J. Palsberg, *Register Allocation Via Coloring of Chordal Graphs.* In: Yi K. (eds) Programming Languages and Systems. APLAS 2005. Lecture Notes in Computer Science, vol 3780. Springer, Berlin, Heidelberg, pp. 315-329, 2005.

[5] Magnus Bordewich, Katharina T. Huber, Charles Semple, *Identifying phylogenetic trees*, Discrete Mathematics, Vol. 300, Issues 1-3, 30-45 (2005).

[6] D. R. Shier, *Some aspects of perfect elimination orderings in chordal graphs*, Discrete Applied Mathematics 7, 325-331 (1984).

[7] H.L. Bodlaender, *A Tourist Guide Through Treewidth*, Acta Cybernetica, Vol. 11, No. 1-2, 1-21 (1993).

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

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

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

[11] Adam Drozdek, *C++ Algorytmy i Struktury Danych, Rozdział 8. Grafy*, Wydawnictwo Helion, Gliwice 2004.

[12] Reinhard Diestel, *Graph Theory*, The Basics, 5-th Electronic Edition 2016, http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/preview/GrTh5_Ch1.pdf.

[13] Bill Cherowitzo, *Applied Graph Theory*, Lecture Notes: Subgraphs and Graph Operations, 2001 http://www-math.ucdenver.edu/~wcherowi/courses/m4408/m4408f.html.

[14] Wikipedia, Clique (graph theory), 2017, https://en.wikipedia.org/wiki/Clique_(graph_theory).

[15] Wikipedia, Independent set (graph theory), 2017 https://en.wikipedia.org/wiki/Independent_set_(graph_theory).

[16] Wikipedia, Dominating set, 2017, https://en.wikipedia.org/wiki/Dominating_set.

[17] Wikipedia, Vertex cover, 2017, https://en.wikipedia.org/wiki/Vertex_cover.

[18] Wikipedia, Bipartite graph, 2017, https://en.wikipedia.org/wiki/Bipartite_graph.

76:3680858345

[19] Wikipedia, Intersection graph, 2017,
https://en.wikipedia.org/wiki/Intersection_graph.

[20] Fanica Gavril, *The Intersection Graphs of Subtrees in Trees Are Exactly the Chordal Graphs*, Journal of Combinatorial Theory (B) 16, 47-56 (1974).

[21] Wikipedia, Permutation graph, 2017,
https://en.wikipedia.org/wiki/Permutation_graph.

[22] Wikipedia, Interval graph, 2017,
https://en.wikipedia.org/wiki/Interval_graph.

[23] D. R. Fulkerson, O. A. Gross, *Incidence matrices and interval graphs*, Pacific Journal of Mathematics 15, 835-855 (1965).

[24] L. S. Chandran, L. Ibarra, F. Ruskey, J. Sawada, *Generating and characterizing the perfect elimination orderings of a chordal graph*, Theoretical Computer Science 307, 303-317 (2003).

[25] G. A. Dirac, *On rigid circuit graphs*, Abh. Math. Sem. Univ. Hamburg 25, 71-76 (1961).

[26] Donald J. Rose, Robert Endre Tarjan, George S. Lueker, *Algorithmic Aspects of Vertex Elimination on Graphs*, SIAM Journal on Computing 5(2), 266-283 (1976).

[27] Wikipedia, Lexicographic breadth-first search, 2017,
https://en.wikipedia.org/wiki/Lexicographic_breadth-first_search.

[28] Robert E. Tarjan, Mihalis Yannakakis, *Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs*, SIAM Journal on Computing 13(3), 566-579 (1984).

[29] Wikipedia, K-tree, 2017,
https://en.wikipedia.org/wiki/K-tree.

[30] Wikipedia, Degeneracy (graph theory), 2017,
https://en.wikipedia.org/wiki/Degeneracy_%28graph_theory%29.

[31] Sudipta Bhaduri, *Finding a maximum clique of a chordal graph by removing vertices of minimum degree*, MSc Thesis, Kent State University, 2008.

[32] Wikipedia, Chordal completion, 2017,
https://en.wikipedia.org/wiki/Chordal_completion.

[33] Mihalis Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM Journal on Algebraic Discrete Methods 2 (1), 77-79 (1981).

[34] F. V. Fomin, D. Kratsch, I. Todinca, *Exact (Exponential) Algorithms for Treewidth and Minimum Fill-In*. In: Díaz J., Karhumäki J., Lepistö A., Sannella D. (eds) Automata, Languages and Programming. ICALP 2004. Lecture Notes in Computer Science, vol 3142. Springer, Berlin, Heidelberg, 2004.

[35] David W. Matula, Leland L. Beck, *Smallest-Last Ordering and Clustering and Graph Coloring Algorithms*, Journal of the Associacion for Computing Machinery 30(3), 417-427 (1983).

[36] Fanica Gavril, *Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph*, SIAM Journal on Computing 1(2), 180-187 (1972).

[37] Kellogg S. Booth, J. Howard Johnson, *Dominating Sets in Chordal Graphs*, SIAM Journal on Computing 11, 191-199 (1982).

[38] Wikipedia, Perfect graph, 2017,
https://en.wikipedia.org/wiki/Perfect_graph.

[39] Wikipedia, Tree decomposition, 2017,
https://en.wikipedia.org/wiki/Tree_decomposition.

[40] Stefan Arnborg, Derek G. Corneil, Andrzej Proskurowski, *Complexity of finding embeddings in a k-tree*, SIAM J. Algeb. Disc. Meth. 8, 277-284 (1987).

77:9539682569

[41] Hans L. Bodlaender, Arie M.C.A. Koster, *Treewidth computations II. Lower bounds*, Information and Computation 209, 1103-1119 (2011).

[42] Hans L. Bodlaender, Arie M.C.A. Koster, *Treewidth computations I. Upper bounds*, Information and Computation 208, 259-275 (2010).

[43] Hans L. Bodlaender, Fedor V. Fomin, Dieter Kratsch, Arie M. C. A. Koster, Dimitrios M Thilikos, *On exact algorithms for treewidth*, ACM Transactions on Algorithms 9, No. 1, Article 12, 23 pages (2012).

[44] Wikipedia, Graph coloring, 2017,
`https://en.wikipedia.org/wiki/Graph_coloring`.

[45] Konrad Gałuszka, *Badanie grafów szeregowo-równoległych z językiem Python*, Praca magisterska, Uniwersytet Jagielloński, Kraków, 2017.

[46] Aleksander Krawczyk, *Badanie grafów Halina z językiem Python*, Praca magisterska, Uniwersytet Jagielloński, Kraków, 2016.

[47] Brian Lucena, *Dynamic programming, tree-width, and computation on graphical models*, PhD Thesis, Brown University, Providence, RI, USA, 2002.

[48] Brian Lucena, *A new lower bound for tree-width using maximum cardinality search*, SIAM Journal on Discrete Mathematics 16, 345-353 (2003).

[49] Igor Samson, *Kolorowanie grafów z językiem Python*, Uniwersytet Jagielloński, Kraków 2016.

[50] Jean R. S. Blair, Barry Peyton, *An introduction to chordal graphs and clique trees*, In: A. George, J. R. Gilbert, J. W. H. Liu (eds), Graph Theory and Sparse Matrix Computation, The IMA Volumes in Mathematics and its Applications, Springer-Verlag, New York, NY, Volume 56, pp. 1-29, 1993.

[51] E. H. Bachoore, H. L. Bodlaender, *New Upper Bound Heuristics for Treewidth*, pp. 216-227. In: Sotiris E. Nikoletseas (Ed.), Experimental and Efficient Algorithms: 4th International Workshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005, Proceedings. LNCS 3503. Springer-Verlag, Berlin, Heidelberg, 2005.

78:5741715565