Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Maciej Mularski

Nr albumu: 1154810

Badanie grafów przedziałowych z językiem Python

Praca magisterska na kierunku Informatyka stosowana

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

Kraków 2023

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

Chciałbym serdecznie podziękować wszystkim, którzy udzielili mi wsparcia w trakcie tworzenia tej pracy, a w szczególności Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za nieocenioną pomoc w analizie przedstawionych rozwiązań, jak i przy samej implementacji nowych algorytmów oraz za zaangażowanie przy opracowywaniu tej pracy.

Streszczenie

Grafy przedziałowe są ważną rodziną grafów, która znajduje szerokie zastosowanie w różnych dziedzinach, takich jak planowanie projektów, harmonogramowanie zadań, analiza sieci, bioinformatyka, itp. Wierzchołki grafu przedziałowego odpowiadają przedziałom na osi liczbowej, a krawędzie łączą wierzchołki wtedy i tylko wtedy, gdy odpowiednie przedziały przecinają się.

Praca skupia się na analizie i implementacji wydajnych algorytmów grafów przedziałowych. Zbadano reprezentację grafów przedziałowych w postaci permutacji z podwójnymi wystąpieniami wierzchołków. Najważniejszym wynikiem pracy jest implementacja algorytmu wykrywania grafu przedziałowego (Habib, 2000), gdzie używana jest technika doskonalenia podziału. Dalej stworzono implementacje algorytmów do sprawdzania spójności grafów, kolorowania wierzchołków, wyznaczania największego zbioru niezależnego, wyznaczania wszystkich klik maksymalnych. Wiele algorytmów ma liniową złożoność obliczeniową.

Dodatkowo zostały przeanalizowane algorytmy tworzenia i wykorzystania dekompozycji ścieżkowej dla grafów przedziałowych. Struktury te zostały przetestowane pod względem poprawności i rzeczywistej wydajności, dzięki czemu mogą być używane w programach wymagających wykorzystania grafów przedziałowych. Kod działa w Pythonie 2.7 i Pythonie 3.

Słowa kluczowe: grafy przedziałowe, grafy cięciwowe, dekompozycja ścieżkowa, spójność, zbiór niezależny, kolorowanie wierzchołków, kliki maksymalne

English title: Study of interval graphs with Python

Abstract

Interval graphs are an important family of graphs that find wide applications in various fields such as project planning, task scheduling, network analysis, bioinformatics, etc. The vertices of an interval graph correspond to intervals on the number line, and the edges connect vertices only when the corresponding intervals intersect.

This work focuses on the analysis and implementation of efficient algorithms for interval graphs. The representation of interval graphs as permutations with double occurrences of vertices has been examined. The most important result is implementation of an algorithm for detecting interval graphs (Habib, 2000) where the partition refinement technique is used. Next, algorithms for connectivity testing, for vertex coloring, for finding maximum independent sets, and for finding all maximal cliques are implemented. Many algorithms have linear time computational complexity.

Additionally, algorithms for creating and utilizing path decomposition in interval graphs have been analyzed. These structures have been tested for correctness and actual performance, making them suitable for use in programs that require the utilization of interval graphs. The code is compatible with both Python 2.7 and Python 3.

Keywords: interval graphs, chordal graphs, path decomposition, connectivity, independent set, vertex coloring, maximal cliques

Spis treści

Spis rysunków

Listings

1. Wstęp

Zagadnienia opisane w pracy dotyczą teorii grafów. Jest to dziedzina matematyki i informatyki, która zajmuje się badaniem własności grafów. Ogólnie grafy składają sie z obiektów (wierzchołki) i połączeń między tymi obiektami (krawędzie). Tematem niniejszej pracy jest analiza algorytmów dla grafów przedziałowych [1], które mają bardzo ciekawe własności. Algorytmy zostały zaimplementowane w języku Python [2] przy wykorzystaniu pakietu *graph-theory* rozwijanego na Wydziale FAIS UJ [3].

1.1. Motywacja i cel pracy

Niniejsza praca magisterska ma na celu zgłębienie tematyki grafów przedziałowych i ich zastosowań w informatyce stosowanej. Graf przedziałowy jest strukturą, która znalazła szerokie zastosowanie w różnych dziedzinach. Celem tej pracy jest dokładne zrozumienie teorii grafów przedziałowych oraz badanie ich praktycznych zastosowań w planowaniu zadań, analizie sieciowej i innych obszarach informatyki stosowanej. Praktycznym rezultatem pracy mają być wydajne i dobrze przetestowane implementacje algorytmów grafowych w języku Python, które mogą służyć do nauki algorytmów i do rozwiązywania konkretnych problemów obliczeniowych.

1.2. Organizacja pracy

Praca została zaplanowana w następujący sposób. W rozdziale 1 znajduje się wprowadzenie do grafów przedziałowych i ich zastosowań, a także cele i organizację pracy. W rozdziale 2, znajdują się podstawowe definicje z teorii grafów, które są używane w dalszych częściach pracy. Ponadto, w tym rozdziale znajduje się część poświęcona grafom przedziałowym, opisująca szczegółowo ich własności. Rozdział 3 ukazuje szczegółowy opis wdrażania grafów, wykorzystanych struktur danych i przykładowych sesji interaktywnych. W rozdziale 4 prezentowane są opisy algorytmów i ich implementacje. Rozdział 5 zawiera opis i przykłady algorytmów wykorzystywanych przy de-

kompozycji przedziałowej. Rozdział 6 jest podsumowaniem pracy. Dodatek A zawiera wyniki testów, które zostały wykorzystane w pracy nad algorytmami.

1.3. Historia problemu

W 1957 roku G. Hajós postawił następujący problem: Mając skończoną liczbę przedziałów na prostej, z tym zbiorem przedziałów można powiązać graf w następujący sposób: każdy przedział odpowiada wierzchołkowi grafu, a dwa wierzchołki są połączone krawędzią, jeśli odpowiadające im przedziały mają przynajmniej częściowe nakładanie się. Pytanie brzmi, czy dany graf jest izomorficzny z jednym z opisanych wcześniej grafów (Hajós [1957, s. 65, przetłumaczone przez M.C.G.]).

Niezależnie, znany biolog molekularny, Seymour Benzer, w trakcie badań nad strukturą genów, zadał powiązane pytanie.

Na podstawie klasycznych badań Morgana i jego szkoły, chromosom jest znany jako liniowe ułożenie elementów dziedzicznych, genów. Te elementy muszą mieć swoją własną wewnętrzną strukturę. Na tym bardziej precyzyjnym poziomie, w obrębie genu pojawia się pytanie: "Czy są one [pod-elementy w obrębie genu] połączone w liniowym porządku analogicznym do wyższego poziomu integracji genów w chromosomie?".

1.4. Zastosowania grafów przedziałowych

Grafy przedziałowe mają szerokie zastosowanie w informatyce, szczególnie w obszarach takich jak planowanie zadań i analiza sieciowa. Wykorzystuje się je do modelowania harmonogramów, rozkładów jazdy, zarządzania projektami i innych złożonych struktur. Ich struktura oparta na przedziałach czasowych umożliwia skuteczną reprezentację i analizę danych związanych z czasem. Grafy przedziałowe są stosowane do rozwiązywania problemów analizy sieciowej, takich jak znajdowanie krytycznych ścieżek, szukanie najwcześniejszego lub najdłuższego zakończenia. Również dzięki grafom przedziałowym możliwe jest identyfikowanie zależności, wykrywanie kolizji czasowych oraz relacji między elementami, optymalizacja harmonogramów.

1.5. Grafy przedziałowe a analiza sieciowa

1.5.1. Wprowadzenie do analizy sieciowej

Analiza sieciowa to dziedzina badawcza, która zajmuje się badaniem złożonych struktur sieciowych i zależności między ich elementami. Grafy przedziałowe znajdują zastosowanie w analizie sieciowej, umożliwiając reprezentację i badanie różnych aspektów sieci.

1.5.2. Analiza roli podgrafów przedziałowych w sieciach społecznościowych

Sieci społecznościowe to jeden z obszarów, w którym grafy przedziałowe są szeroko stosowane. Grafy przedziałowe umożliwiają identyfikację grup społecznych o specyficznych właściwościach oraz badanie ich wpływu na całą sieć.

1.5.3. Badanie struktury sieciowej za pomocą grafów przedziałowych

Grafy przedziałowe są również używane do badania struktur sieciowych. Booth i Lueker [10] wskazują, że grafy przedziałowe umożliwiają analizę zależności czasowych między elementami sieci, takich jak interakcje między wierzchołkami i ich kolejność. Dzięki temu, możliwe jest lepsze zrozumienie struktury i dynamiki sieci oraz identyfikacja ważnych wierzchołków i wzorców połączeń.

1.6. Zastosowania grafów przedziałowych w planowaniu zadań

1.6.1. Planowanie zadań

Planowanie zadań jest istotnym zagadnieniem w wielu dziedzinach, takich jak logistyka, zarządzanie projektami i rozkładanie grafików. Grafy przedziałowe znalazły zastosowanie w tej dziedzinie, umożliwiając modelowanie i rozwiązywanie problemów planowania zadań. Booth i Lueker [10] opisują wykorzystanie grafów przedziałowych do reprezentacji zależności między zadaniami oraz rozwiązywania problemów minimalizacji opóźnień i identyfikacji krytycznych ścieżek.

1.6.2. Planowanie produkcji

Zastosowanie grafów przedziałowych w planowaniu produkcji pozwala na efektywne zarządzanie procesem produkcyjnym. Golumbic [9] podkreśla, że grafy przedziałowe mogą być wykorzystane do modelowania sekwencji operacji produkcyjnych oraz analizy zależności między nimi. Dzięki temu, planowanie produkcji może być zoptymalizowane pod kątem minimalizacji czasu, maksymalizacji wydajności i minimalizacji kosztów.

1.6.3. Rozkładanie grafików

Rozkładanie grafików jest kolejnym obszarem, w którym grafy przedziałowe znajdują zastosowanie. Praca nad efektywnym rozłożeniem grafików jest ważna w systemach harmonogramowania zadań, takich jak służba medyczna, transport publiczny, czy obsługa klienta. Booth i Lueker [10] opisują wykorzystanie grafów przedziałowych do reprezentacji godzin pracy, ograniczeń czasowych i zależności między zadaniami w celu optymalnego rozłożenia grafików.

1.6.4. Zarządzanie projektami

Grafy przedziałowe mają również zastosowanie w zarządzaniu projektami. Vizing [8] wskazuje, że grafy przedziałowe mogą być używane do modelowania zadań w projekcie, ich kolejności oraz ograniczeń czasowych. Dzięki temu, zarządzanie projektami staje się bardziej efektywne, umożliwiając lepszą kontrolę nad harmonogramem, zasobami i przebiegiem projektu.

2. Teoria grafów

Teoria grafów łączy w sobie matematykę i informatykę. Jest ona zajmuje się badaniem grafów oraz wdrażaniem algorytmów, które wyznaczają własności grafów. Początki teorii grafów sięgają 1793 roku, kiedy to Leonard Euler opublikował opis zagadnienia mostów królewieckich. Teoria grafów jest narzędziem wykorzystywanym do formułowania i rozwiązywania problemów o znaczeniu praktycznym, a do tego stale się rozwija. Poniżej znajdują się podstawowe definicje i twierdzenia, których zrozumienie jest niezbędne do poznania własności i cech grafów ciągłych. Definicje głównie opierają się na książkach Cormena [6], Wojciechowskiego [5], oraz Wilsona [4].

2.1. Podstawowe definicje

Definicja: Graf nieskierowany to uporządkowana para $G = (V, E)$, gdzie $V(G)$ jest niepustym zbiorem elementów, które nazywamy wierzchołkami, a $E(G)$ jest zbiorem nieuporządkowanych par elementów z $V(G)$, które nazywamy krawędziami [6]. W grafie wierzchołki reprezentują różne obiekty lub punkty, a krawędzie reprezentują relacje między tymi obiektami. Krawędź łączy dwa różne wierzchołki (brak pętli) i może wskazywać na istnienie pewnego rodzaju związku, np. sąsiedztwo, połączenie lub relację między nimi. Przyjmujemy oznaczenia: liczba wierzchołków $n = |V(G)|$, liczba krawędzi $m = |E(G)|$.

Definicja: Graf skierowany to uporządkowana para $G = (V, E)$, gdzie $V(G)$ jest niepustym zbiorem elementów, które nazywamy wierzchołkami, a $E(G)$ jest zbiorem uporządkowanych par różnych elementów z $V(G)$, które nazywamy krawędziami skierowanymi [6]. W grafie skierowanym wierzchołki nadal reprezentują różne obiekty lub punkty, ale krawędzie mają określony kierunek. Krawędź skierowana wskazuje na relację między dwoma wierzchołkami, przy czym jeden wierzchołek jest wierzchołkiem początkowym, a drugi wierzchołek jest wierzchołkiem końcowym.

Definicja: Krawędź skierowana jest reprezentowana jako uporządkowana para (v, w) lub vw, gdzie v i w należą do zbioru wierzchołków $V(G)$. Graficzna reprezentacja to linia ze strzałką skierowaną w kierunku wierzchołka końcowego w [6].

Definicja: Krawędzie wielokrotne to zestaw krawędzi o wspólnym wierzchołku początkowym i wspólnym wierzchołku końcowym [4].

Definicja: Pętla to krawędź, której początkiem i końcem jest ten sam wierzchołek [4].

Definicja: Graf prosty jest to graf (skierowany lub nieskierowany), który nie zawiera pętli i krawędzi wielokrotnych [6].

Definicja: Multigrafy (skierowane lub nieskierowane) to grafy, które mogą posiadać pętle i krawędzie wielokrotne [4].

Definicja: Podgraf H grafu G to graf, który zawiera wybrane wierzchołki grafu G oraz może zawierać niektóre krawędzie łączące wybrane wierzchołki w grafie G. Formalnie $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$ a wszystkie krawędzie z $E(H)$ mają końce w $V(H)$ [6].

Definicja: Podgraf indukowany H grafu G to graf, który zawiera wybrane wierzchołki grafu G oraz musi zawierać wszystkie krawędzie łączące wybrane wierzchołki w grafie G. Formalnie $V(H) \subseteq V(G)$ oraz $E(H) = \{vw \in E(G) : v \in V(H), w \in V(H)\}$ [6].

Definicja: Graf pełny K_n (ang. *complete graph*) to graf prosty nieskierowany z liczbą wierzchołków równą n, w którym dla każdej pary wierzchołków istnieje krawędź je łącząca. Graf K_n jest czasem nazywany kliką.

Definicja: Graf cykliczny C_n (ang. *cycle graph*) to graf nieskierowany spójny zawierający n wierzchołków, w którym każdy wierzchołek ma dwóch sąsiadów.

Definicja: Graf liniowy P_n (ang. *path graph*, *linear graph*) to graf otrzymany z grafu cyklicznego C_n przez usunięcie jednej krawędzi.

Definicja: Zbiór niezależny w grafie nieskierowanym G to taki podzbiór S zbioru wierzchołków $V(G)$, że elementy zbioru S są parami rozłączne, czyli

nie są połączone krawędzią. Maksymalny zbiór niezależny to zbiór niezależny, który nie jest podzbiorem właściwym innego zbioru niezależnego. Największy zbiór niezależny to zbiór niezależny o największej liczności.

Definicja: Klika to podgraf H grafu nieskierowanego G, w którym każde dwa wierzchołki są połączączone ze sobą krawędziami. Inaczej mówiąc jest to podzbiór zbioru wierzchołków $V(G)$, który indukuje podgraf pełny [6].

Definicja: Rozmiar kliki to liczba wierzchołków zawartych w klice.

Definicja: Klika maksymalna jest wtedy, gdy do kliki nie można już dodać kolejnego wierzchołka, który współtworzyłby większą klikę [7].

Definicja: Klika największa to klika z największą liczbą wierzchołków. Rozmiar takiej kliki grafu G oznaczamy $\omega(G)$ [7].

2.2. Spójność

Definicja: Ścieżka w grafie G z wierzchołka p do q jest to ciąg wierzchołków i krawędzi je łączących, które należy przejść w grafie, aby z wierzchołka p dostać się do wierzchołka q [6].

Definicja: Długość ścieżki jest to liczba krawędzi znajdujących się w ścieżce [6].

Definicja: Ścieżka prosta jest to ścieżka składająca się z unikalnych wierzchołków i krawędzi [6].

Definicja: Cykl to ścieżka, w której pierwszy i ostatni wierzchołek jest taki sam.

Definicja: Cykl prosty jest to cykl, w którym wierzchołki nie powtarzają się, poza pierwszym i ostatnim.

Definicja: Graf acykliczny to graf, który nie posiada cykli [6]. Graf skierowany acykliczny nazywa sie dagiem.

Definicja: Graf nieskierowany jest spójny, gdy istnieje ścieżka między dowolną parą wierzchołków w grafie.

Definicja: Liczba chromatyczna to minimalna liczba kolorów potrzebnych do pokolorowania wszystkich wierzchołków grafu w taki sposób, żeby żadne dwa sąsiednie wierzchołki połączone krawędzią nie miały tego samego koloru.

Definicja: Graf doskonały (ang. *perfect graph*) to taki graf nieskierowany, w którym liczba chromatyczna każdego jego podgrafu indukowanego jest równa liczności największej kliki tego podgrafu [9].

2.3. Grafy cięciwowe

Definicja: Graf cięciwowy (ang. *chordal graph*) to graf nieskierowany, w którym wszystkie cykle o długości cztery lub więcej zawierają cięciwę. Cięciwa jest to krawędź nie należąca do cyklu, ale łącząca dwa wierzchołki należące do cyklu. Istnieje kilka innych równoważnych charakteryzacji grafów cięciwowych [9].

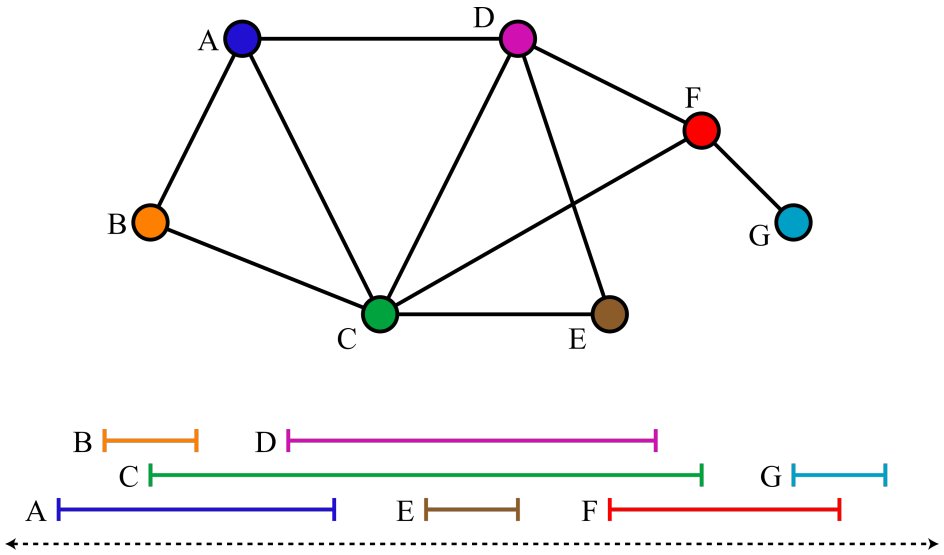
Twierdzenie (Berge, 1960; Hajnal, Surányi, 1958): Każdy graf cięciwowy jest grafem doskonałym. Można powiedzieć, że badania grafów cięciwowych były początkiem teorii grafów doskonałych [9].

Definicja: Doskonałe uporządkowanie eliminacji wierzchołków (ang. *perfect elimination ordering*, PEO) jest to takie uporządkowanie wierzchołków grafu, że dla każdego wierzchołka v, v i jego sąsiedzi, którzy znajdują się za nim w PEO, tworzą klikę.

Twierdzenie (Fulkerson, Gross, 1965): Graf jest grafem cięciwowym wtedy i tylko wtedy, gdy posiada PEO [16].

2.4. Grafy przedziałowe

Definicja: Graf przedziałowy to graf nieskierowany, w którym każdemu wierzchołkowi przypisany jest przedział na osi liczbowej. Dwa wierzchołki są połączone krawędzią, jeżeli odpowiednie przedziały mają niepuste przecięcie [9]. Zwykle przyjmuje się, że przedziały są domknięte, a wszystkie końce przedziałów są różne.



Rysunek 2.1. Siedem przedziałów na osi rzeczywistej i odpowiadający im graf przedziałowy o siedmiu wierzchołkach. Przykład pochodzi z [1].

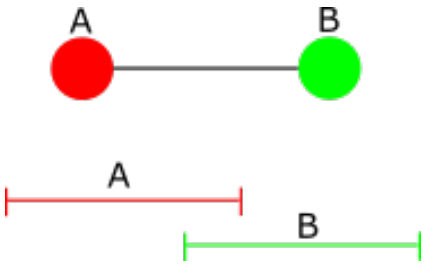
Twierdzenie (Fulkerson, Gross, 1965): Nieskierowany graf G jest grafem przedziałowym wtedy i tylko wtedy, gdy jego macierz klikowa M ma właściwość kolejnych jedynek dla kolumn (ang. *the consecutive 1's property*) [16].

Definicja: Macierz klikowa (ang. *clique matrix*) jest to macierz reprezentująca graf. Każdy wiersz macierzy odpowiada wierzchołkowi grafu, a każda kolumna odpowiada klice. Wartości w macierzy są zwykle binarne i informują, czy dany wierzchołek należy do danej kliki. Jeśli wierzchołek należy do kliki, w odpowiedniej komórce macierzy jest wartość 1, w przeciwnym razie 0.

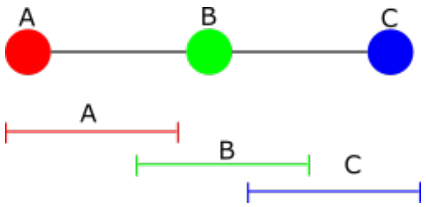
$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Rysunek 2.2. Macierz klikowa dla grafu przedstawionego na rysunku 2.1.

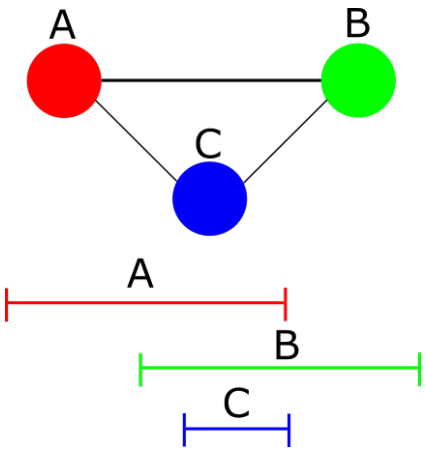
Grafy przedziałowe wykazują interesujące własności, które czynią je przedmiotem dogłębnych badań w teorii grafów. Praca Tarjana i Yannakakisa [11] podkreśla, że każdy graf przedziałowy jest grafem cięciwowym. Ta charakterystyczna cecha wpływa na rozwinięcie różnych technik i algorytmów wykorzystujących grafy przedziałowe.



Rysunek 2.3. Wygląd grafu P_2 i reprezentacji na osi czasu dla dwóch elementów.



Rysunek 2.4. Wygląd grafu P_3 i reprezentacji na osi czasu dla trzech elementów, które się przecinają się w czasie jedna po drugiej.



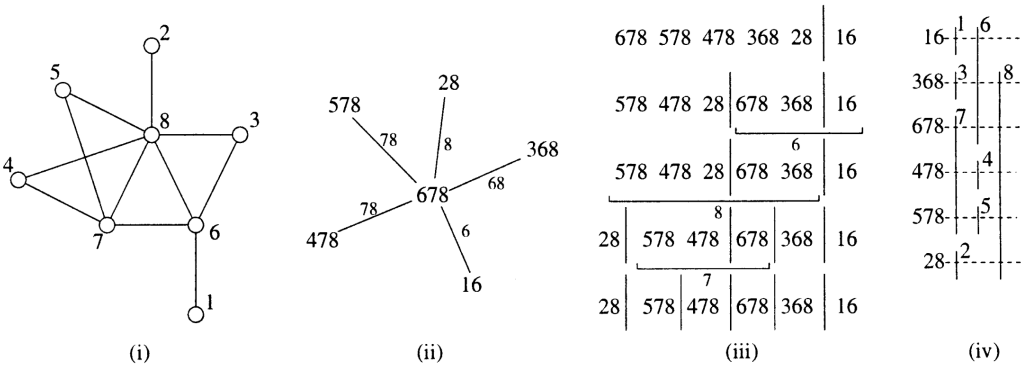
Rysunek 2.5. Wygląd grafu K_3 i reprezentacji na osi czasu dla trzech elementów, które się przecinają w tym samym czasie.

2.5. Rozpoznawanie grafów przedziałowych

Rozpoznawanie grafów przedziałowych jest jednym z kluczowych zagadnień w teorii grafów. Szybkie rozpoznanie grafu przedziałowego pozwala potem na używanie algorytmów dedykowanych tym grafom.

W pracy Michel Habiba, Rossa McConnella, Christophe'a Paula i Laurenta Viennota został przedstawiony sposób na sprawdzenie czy dany graf jest grafem przedziałowym. Wykorzystany został do tego algorytm Lex-BFS i udoskonalanie partycji (partition refinement). Na rysunkach 2.6 został przedstawiony sposób działania opisanego przez nich algorytmu. Samo działanie algorytmu przebiega następująco:

1. Wierzchołki grafu są numerowane zgodnie z porządkiem Lex-BFS.
2. Budowane jest drzewo klik związane z porządkiem Lex-BFS.
3. Kolejnym krokiem jest podzielenie zbioru klik na mniejsze części.
4. Sprawdzenie czy przedziały zawierają kolejno występujące elementy.



Rysunek 2.6. Działanie algorytmu rozpoznawania grafu przedziałowego. (i) Graf przedziałowy. Jego wierzchołki są numerowane zgodnie z porządkiem Lex-BFS. (ii) Drzewo klik związane z porządkiem Lex-BFS. (iii) Rozbicie zbioru klik na mniejsze części. (iv) Reprezentacja odcinkowa związana z obliczanym łańcuchem klik. Przykład pochodzi ze slajdów [17].

W rozdziale algorytmy 4.2 znajduje się implementacja oraz dokładny opis działania tego algorytmu.

Innym spobobem na rozpozanie grafu przedziałowego jest użycie algorytmu Bootha i Luekera [10] opartego na strukturze danych PQ-tree, która umożliwia poprzez testowanie właściwości kolejnych jedynek sprawdzenie grafu. Właściwość kolejnych jedynek oznacza, że w każdym wierszu ma-

cierzy lub wierzchołku grafu, wszystkie jedynki występują w jednym ciągu bez przerw. Rozpoznanie grafu przedziałowego jest możliwe w czasie liniowym [10].

2.6. Kolorowanie grafów przedziałowych

Definicja: Kolorowanie grafu (ang. *graph coloring*) polega na przyporządkowaniu elementom grafu odpowiednich etykiet, zwanych kolorami, przy zachowaniu ściśle określonych reguł etykietowania [23].

Definicja: Kolorowanie wierzchołków grafu (ang. *vertex coloring*) polega na przypisaniu wierzchołkom etykiet w taki sposób, żeby żadne dwa sąsiadujące w grafie wierzchołki nie miały tego samego koloru. Podobnie przebiega proces kolorowania krawędzi grafu. Kolorowanie wierzchołków grafu, które spełnia podaną regułę, nazywane jest kolorowaniem właściwym lub dozwolonym (ang. *proper coloring*). Graf zawierający pętle nie może zostać poprawnie pokolorowany [23].

Badania nad kolorowaniem grafów przedziałowych mają zastosowanie w wielu dziedzinach, takich jak planowanie harmonogramów, alokacja zasobów i optymalizacja procesów.

1. Kolorowanie grafów przedziałowych w czasie liniowym:
   Umożliwia to algorytm zachłanny, który polega na przypisaniu kolorów wierzchołkom grafu w pewnym określonym porządku. Można to zrobić, na przykład, przeglądając wierzchołki w porządku rosnących lub malejących numerów. Dla każdego wierzchołka, przypisuje się najmniejszy dostępny kolor, który nie został jeszcze użyty przez sąsiadujące wierzchołki. Golumbic w swojej pracy z 2004 roku wprowadził algorytmy kolorowania grafów przedziałowych w czasie liniowym, wykorzystując właściwości grafu cięciwowego [9].
2. Minimalne kolorowanie on-line grafów przedziałowych:

   Twierdzenie (Kierstead i Trotter, 1981): Istnieje algorytm kolorowania on-line A taki, że dla dowolnego grafu przedziałowego G, $X_A(G) < 3w(G) - 2$ ponadto, dla dowolnego algorytmu kolorowania on-line A i dowolnej liczby naturalnej k, istnieje graf przedziałowy G taki, że $\omega(G) = k$ i $X_A(G) > 3k - 2$ [22].

2.7. Przeszukiwanie grafów przedziałowych

Algorytmy przeszukiwania grafów przedziałowych odgrywają kluczową rolę w analizie i manipulacji takimi strukturami. Golumbic [9] przedstawia algorytmy głębokiego przeszukiwania grafów przedziałowych, które umożliwiają dokładne badanie ich właściwości i struktur. Jednym z najpopularniejszych algorytmów przeszukiwania grafów przedziałowych jest algorytm DFS (Depth-First Search), który umożliwia badanie spójności grafu i identyfikowanie cykli.

3. Implementacja grafów

Grafy przedziałowe będą reprezentowane jako lista w języku Python zawierająca permutację kolejnych elementów. Narzędzia przygotowane w ramach tej pracy działają zarówno w Pythonie 2.7, jak i Pythonie 3.7+. Poniżej przedstawiamy przykładowe zastosowanie zaimplementowanych algorytmów.

Przykładowa sesja interaktywna zakłada, że pakiet *graphtheory* jest już zainstalowany w systemie (pip install graphtheory).

```
# Importy potrzebne do pracy z grafami abstrakcyjnymi.
>>> from graphtheory.structures.edges import Edge
>>> from graphtheory.structures.graphs import Graph
```

W dalszych częściach zostały przedstawione interaktywne sesje prezentujące wykorzystanie zaimplementowanych algorytmów.

3.1. Generowanie grafów przedziałowych

Graf przedziałowy można wygenerować używając poniżej przedstawionych sposobów. Aby wygenerować losowy graf permutacyjny należy użyć metody make_random_interval(n), gdzie n oznacza liczbę wierzchołków w grafie.

```
>>> perm = make_random_interval_graph(5)
>>> print(perm)
[1, 2, 4, 3, 4, 0, 1, 0, 2, 3]
```

Aby wygenerować losowy graf ścieżkę P_n należy użyć metody make_path_interval(n), gdzie n oznacza liczbę wierzchołków w grafie.

```
>>> perm = make_path_interval(5)
>>> print(perm)
[0, 1, 0, 2, 1, 3, 2, 4, 3, 4]
```

Aby wygenerować losowy graf *tepee* należy użyć metody make_tepee_interval(n), gdzie n oznacza liczbę wierzchołków w grafie.

```
>>> perm = make_tepee_interval(5)
>>> print(perm)
```

```
[4, 0, 1, 0, 2, 1, 3, 2, 3, 4]
```

Aby wygenerować losowy graf przedziałowy będący grafem k-tree z n wierzchołkami należy użyć metody make_ktree_interval(n, k).

```
>>> perm = make_ktree_interval(5, 2)
>>> print(perm)
[0, 1, 2, 0, 3, 1, 4, 2, 3, 4]
```

Aby wygenerować losowy graf dwudzielny $K_{1,n-1}$ należy użyć metody make_star_interval(n), gdzie n oznacza liczbę wierzchołków w grafie.

```
>>> perm = make_star_interval(5)
>>> print(perm)
[0, 1, 1, 2, 2, 3, 3, 4, 4, 0]
```

3.2. Rozpoznawanie grafów przedziałowych

W celu sprawdzenia, czy dany graf G jest grafem przedziałowym, należy utworzyć drzewo klik używając algorytmu algorithm4(G, order), gdzie G oznacza objekt klasy Graph, a *order* to PEO uzyskane przy pomocy algorytmu Lex-BFS dla grafu. PEO można uzyskać używając metody find_peo_lex_bfs(G). Używając metody graph_recognition(tree), gdzie tree oznacza drzewo klik jako klasa Graph, możemy uzyskać informację, czy dany graf jest grafem przedziałowym. Metoda ta zwraca dodatkowo zbiór klik przy pomocy których można uzyskać informację o kolejności elementów w czasie.

```
>>> perm = make_random_interval_graph(5)
>>> print(perm)
[2, 8, 5, 9, 5, 9, 6, 7, 6, 1, 0, 0, 4, 10, 2, 3, 7, 10, 1, 4, 8, 3]
>>> G = find_abstract_graph(perm)
>>> order = find_peo_lex_bfs(G)
>>> print(order)
[3, 10, 4, 0, 1, 7, 6, 9, 5, 8, 2]
>>> is_peo1(G, order)
True
>>> is_peo2(G, order)
True
>>> tree = algorithm4(G, order)
>>> print(tree)
{(2, 5, 8, 9): None, (2, 6, 7, 8): (2, 5, 8, 9), (0, 1, 2, 7, 8): (2, 6, 7, 8),
(1, 2, 4, 7, 8, 10): (0, 1, 2, 7, 8), (1, 3, 4, 7, 8, 10): (1, 2, 4, 7, 8, 10)}
>>> graph_recognition(tree2graph(tree))
```

```
True [[(2, 5, 8, 9)], [(2, 6, 7, 8)], [(0, 1, 2, 7, 8)], [(1, 2, 4, 7, 8, 10)], [(1, 3, 4, 7, 8, 10)]]
```

3.3. Przechodzenie przez grafy przedziałowe

Aby skorzytać z algorytmu BFS należy użyć klasy IntervalBFS, w konstruktorze należy podać graf w reprezentacji permutacyjnej. Używając metody run(source, preaction, postaction), możemy przechodząc od wybranego wierzchołka source, wykonać metody na wierzchołkach w kolejności postorder albo preorder. Używając metody path(source, target) możemy uzyskać informację przez jakie wierzchołki należy przejść chcąc dotrzeć od jedngo do drugiego. Atrybut parent zwraca wygląd drzwa BFS w formie słownika. W podanym przykładzie algorytm wyznacza kolejności preorder i posorder odwiedzania wierzchołków grafu, gdzie source = 0.

```
>>> perm = make_random_interval_graph(5)
>>> print(perm)
[2, 0, 1, 2, 4, 0, 3, 4, 3, 1]
>>> pre_order = []
>>> post_order = []
>>> algorithm = IntervalBFS(perm)
>>> algorithm.run(0, pre_action=lambda node: pre_order.append(node),
                  post_action=lambda node: post_order.append(node))
>>> print(pre_order)
[0, 1, 2, 4, 3]
>>> print(post_order)
[0, 1, 2, 4, 3]
>>> print(algorithm.parent)
{0: None, 1: 0, 2: 0, 4: 0, 3: 1}
>>> print(algorithm.path(0, 3))
[0, 1, 3]
```

Aby skorzytać z algorytmu DFS należy użyć klasy IntervalDFS, w konstruktorze należy podać graf w reprezentacji permutacyjnej. Używając metody run(source, preaction, postaction), możemy przechodząc od wybranego wierzchołka source, wykonać metody na wierzchołkach w kolejności postorder albo preorder. Używając metody path(source, target) możemy uzyskać informację przez jakie wierzchołki należy przejść chcąc dotrzeć od jedngo do drugiego. Atrybut parent zwraca wygląd drzwa DFS w formie słownika.

```
>>> perm = make_random_interval_graph(5)
>>> print(perm)
[2, 0, 1, 2, 4, 0, 3, 4, 3, 1]
```

```
>>> pre_order = []
>>> post_order = []
>>> algorithm = IntervalDFS(perm)
>>> algorithm.run(0, pre_action=lambda node: pre_order.append(node),
                  post_action=lambda node: post_order.append(node))
>>> print(pre_order)
[0, 1, 2, 3, 4]
>>> print(post_order)
[2, 4, 3, 1, 0]
>>> print(algorithm.parent)
{0: None, 1: 0, 2: 1, 3: 1, 4: 3}
>>> print(algorithm.path(0, 3))
[0, 1, 3]
```

3.4. Sprawdzanie spójności grafów przedziałowych

Aby sprawdzić spójność grafu przedziałowego, należy użyć funkcji interval_is_connected(perm), podając jako argument graf w reprezentacji permutacyjnej.

```
>>> perm = make_random_interval_graph(5)
>>> print(perm)
[2, 0, 1, 2, 4, 0, 3, 4, 3, 1]
>>> interval_is_connected(perm)
True
```

3.5. Budowanie grafu abstrakcyjnego z permutacji

Aby z grafu w reprezentacji permutacyjnej stworzyć graf w formie klasy Graph, należy użyć funkcji make_abstract_interval_graph(perm).

```
>>> perm = make_random_interval_graph(5)
>>> print(perm)
[0, 1, 2, 0, 3, 1, 4, 2, 3, 4]
>>> G = make_abstract_interval_graph(perm)
>>> G.show()
0 : 1 2
1 : 0 2 3
2 : 0 1 3 4
3 : 1 2 4
4 : 2 3
```

3.6. Rysowanie przedziałów grafu przedziałowego

Aby wpisać graficznie (w terminalu) wygląd przedziałów z grafu przedziałowego, należy użyć funkcji intevals_drawing(perm).

```
>>> perm = make_random_interval_graph(11)
>>> print(perm)
[5, 2, 4, 0, 5, 7, 10, 2, 6, 8, 9, 4, 1, 0, 8, 7, 1, 10, 3, 9, 3, 6]
>>> intevals_drawing(perm)
5----------5 7------------------------------------7 3----3
   2--------------------2 6-------------------------------6
      4-------------------------4 1-----------1
         0-----------------------------------0
            10-------------------------------------10
               8-----------------8
                  9-----------------------------------9
```

3.7. Kolorowanie wierzchołków grafu przedziałowego

Aby przypisać kolor wierzchołkom w grafie przedziałowym, należy skorzytać z funkcji interval_node_color(perm), zwraca ona słownik zawierający wierzchołek jako klucz i numer koloru jako wartość. Jest to konwencja stosowana w pakiecie graphtheory.

```
>>> perm = make_random_interval_graph(5)
>>> print(perm)
[1, 3, 0, 2, 0, 2, 4, 4, 3, 1]
>>> interval_node_color(perm)
{1: 0, 3: 1, 0: 2, 2: 3, 4: 3}
```

3.8. Wyznaczanie największego zbioru niezależnego

Aby znaleźć największy zbiór niezależny w grafie przedziałowym, należy skorzystać z funkcji interval_max_iset(perm).

```
>>> perm = make_random_interval_graph(5)
>>> print(perm)
[3, 0, 2, 2, 3, 1, 1, 4, 0, 4]
>>> interval_max_iset(perm)
{1, 2, 4}
```

3.9. Dekompozycja ścieżkowa

Aby dokonać dekompozycji ścieżkowej grafu przedziałowego podanego w reprezentacji permutacyjnej, należy skorzystać z funkcji make_path_decomposition(perm). Zwraca ona instancję klasy Graph zawierającą drzewo dekompozycji.

```
>>> perm = make_ktree_interval(5, 2)
>>> print(perm)
[0, 1, 2, 0, 3, 1, 4, 2, 3, 4]
>>> T = make_path_decomposition(perm)
>>> T.show()
(0, 1, 2) : (1, 2, 3)
(1, 2, 3) : (0, 1, 2) (2, 3, 4)
(2, 3, 4) : (1, 2, 3)
```

4. Algorytmy

Rozdział prezentuje wybrane algorytmy dla grafów przedziałowych, wraz ze szczegółowymi opisami i omówieniem złożoności obliczeniowej.

4.1. Generatory wybranych grafów przedziałowych

Do testowania przygotowywanych w ramach pracy programów potrzebne są przykładowe grafy przedziałowe, dlatego stworzono kilka generatorów grafów przedziałowych o różnych właściwościach. Generatory to funkcje, których argumentem jest liczba wierzchołków n, a wartością zwracaną jest permutacja z podwójnymi wystąpieniami etykiet wierzchołków. Dla wygody etykietami wierzchołków grafu przedziałowego są liczby całkowite od 0 do $n-1$, choć dalej przedstawione algorytmy działają dla etykiet wierzchołków będących dowolnymi liczbami lub stringami. Kolejne etykiety w permutacji odpowiadają zdarzeniom na osi czasu, przy czym zdarzeniem jest początek lub koniec przedziału.

— make_random_interval(n) - zwraca przypadkowy graf przedziałowy. Ta sama funkcja może być użyta do wytworzenia przypadkowego grafu kołowego w postaci permutacji.
— make_path_interval(n) - zwraca graf ścieżkę P_n.
— make_tepee_interval(n) - zwraca graf *tepee*. Jest to graf powstały z grafu P_{n-1}, gdzie dodatkowy wierzchołek jest połączony ze wszystkimi poprzednimi wierzchołkami.
— make_ktree_interval(n,k) - zwraca specjalny graf przedziałowy będący grafem k-tree z n wierzchołkami.
— make_star_interval(n) - zwraca graf dwudzielny $K_{1,n-1}$.

4.2. Rozpoznawanie grafów przedziałowych

Implementacja techniki *partition refinement* z pracy Habiba [17] w celu rozpoznania grafu przedziałowego.

Dane wejściowe: Drzewo klik związane z porządkiem Lex-BFS jako klasa Graph.

Opis algorytmu: Algorytm rozpoczynamy od wybrania pierwszych wierzchołków poprzez oddzielenie ostatniej znalezionej kliki za pomocą algorytmu Lex-BFS. Następnie dodajemy wierzchołki z tej kliki do kolejki. Proces ten powtarzamy, dopóki kolejka nie będzie pusta.

Jeśli kolejka zawiera wierzchołki, wybieramy jeden z nich i przeszukujemy zbiory wierzchołków, aby znaleźć te, które zawierają ten wybrany wierzchołek. Przeszukiwanie to jest wykonywane niezależnie od początku i końca zbioru zawierającego zbiory maksymalnych klik. Kończymy je, gdy znajdziemy zbiór klik, który zawiera poszukiwany wierzchołek. Następnie dzielimy ten zbiór na dwa osobne zbiory klik: jeden zawierający poszukiwany wierzchołek i drugi niezawierający go.

Z wybranych zbiorów klik dodajemy do kolejki te wierzchołki, które nie zostały jeszcze użyte w algorytmie. Cały proces trwa, dopóki istnieją zbiory klik zawierające więcej niż jedną klikę.

Po zakończeniu tego algorytmu sprawdzamy, czy kliki dla każdego wierzchołka występują po sobie. Jeśli tak jest, to stwierdzamy, że dany graf jest grafem przedziałowym.

Złożoność: Teoretyczna złożoność czasowa algorytmu wynosi $O(n + m)$.

Uwagi: Algorytm został zmodyfikowany w celu wykrycia błędu spowodowanego znalezieniem takich samych zbiorów na etapie przeszukiwania zbioru klik.

Zwracany jest wybrany wierzchołek na koniec kolejki i wykorzystywany w ponownym przeszukiwaniu. Jeśli takowy wierzchołek podczas kolejnej iteracji spowoduje wykrycie tych samych zbiórów, następuje jego odrzucenie.

Listing 4.1. Rozpoznawanie grafów przedziałowych przy pomocy techniki *partition refinement*.

```
def graph_recognition(tree):
    """Recognizing if graph is interval graph."""
    vertices = set()
    L = [[x for x in tree if not (x in vertices or vertices.update(x))]]
    pivots = deque()
    processed_nodes = set()
    deleyed_nodes = set()
```

```
while exist_non_singleton(L) != {}:
    if len(pivots) == 0:
        Xc, index = list(exist_non_singleton(reversed(L)))
        index = len(L) - 1 - index
        Cl = Xc[-1]
        L[index:index+1] = [Xc[:-1], [Cl]]
        E = [Cl]
        processed_nodes = set([x for x in processed_nodes
                                   if not x in deleyed_nodes])
        deleyed_nodes = set()
    else:
        p = pivots.popleft()
        E = list(set(c for c in tree if p in c))
        for index_x_a, l in enumerate(L):
            if any(e in E for e in l):
                Xa = l
                left_x_a = list(x for x in Xa if not x in E)
                right_x_a = list(x for x in Xa if x in E)
                break
        for index_x_b, l in enumerate(reversed(L)):
            index_x_b = len(L) - 1 - index_x_b
            if any(e in E for e in l):
                Xb = l
                left_x_b = list(x for x in Xb if x in E)
                right_x_b = list(x for x in Xb if not x in E)
                break

        if right_x_a == left_x_b and left_x_a == right_x_b:
            if not p in deleyed_nodes:
                pivots.append(p)
                deleyed_nodes.add(p)
            else:
                processed_nodes.add(p)
            E = []
        else:
            processed_nodes.add(p)
            if left_x_a and right_x_a:
                L[index_x_a:index_x_a+1] = [left_x_a, right_x_a]
                left_x_a, right_x_a = [], []
                index_x_b += 1
            if left_x_b and right_x_b:
                L[index_x_b:index_x_b+1] = [left_x_b, right_x_b]
                left_x_b, right_x_b = [], []
```

```
            for e in E:
                for c in tree[e]:
                    if c in E:
                        continue
                    for v in c:
                        if not v in processed_nodes and not v in pivots and v in e:
                            pivots.append(v)

        for k in vertices:
            isStarted = False
            isEnded = False
            for clique in L:
                if len(clique) <= 0:
                    return False, L
                if k in clique[0]:
                    if not isStarted:
                        isStarted = True
                    elif isEnded:
                        return False, L
                elif isStarted:
                    isEnded = True

        return True, L

def exist_non_singleton(L):
    for index, l in enumerate(L):
        if len(l) > 1:
            return l, index
    return {}
```

4.3. Przechodzenie przez grafy przedziałowe

W przypadku ogólnych grafów, do rozpoznawania ich struktury stosuje się przede wszystkim dwa algorytmy przechodzenia przez graf, BFS i DFS, o złożoności liniowej $O(n + m)$. Dla grafów przedziałowych w reprezentacji permutacyjnej przygotowano specjalne wersje tych algorytmów o złożoności $O(n^2)$. Są to klasy IntervalBFS i Interval DFS. W konstruktorze obu klas wykonuje się preprocesing w czasie $O(n)$, przez co potem w stałym czasie można testować obecność krawędzi między wierzchołkami. Przygotowane klasy mogą być podstawą implementacji innych algorytmów, korzystających z BFS lub DFS. W rozdziale 3.3 został przedstawiony przykład użycia powyższych klas.

4.4. Sprawdzanie spójności grafów przedziałowych

Reprezentacja permutacyjna grafu przedziałowego umożliwia szybkie testowanie spójności grafu w czasie $O(n)$. Przechodząc przez permutację zapisujemy etykiety wierzchołków rozpoczynających się przedziałów, a po ponownym pojawieniu się tej samej etykiety (koniec przedziału) usuwamy ją ze zbioru etykiet. Jeżeli zbiór etykiet stanie się pusty przed dojściem do końca permutacji, to jest to sygnał nieciągłości grafu.

Listing 4.2. Sprawdzanie spójności grafu przedziałowego.

```
def interval_is_connected(perm):
    """Testing connectivity using double perm in O(n) time."""
    used = set()
    for (idx, node) in enumerate(perm):
        if node in used:
            used.remove(node)
            if len(used) == 0 and idx < len(perm)-1:
                return False
        else:
            used.add(node)
    return True
```

4.5. Wyznaczanie klik maksymalnych

W grafie przedziałowym w reprezentacji permutacyjnej można w czasie $O(n)$ znaleźć wszystkie kliki maksymalne, a jednocześnie wyznaczyć PEO (graf przedziałowy jest grafem cięciwowym). Funkcja find_peo_cliques(perm) zwraca krotkę składającą się z PEO (lista etykiet wierzchołków) i listy klik maksymalnych (lista zbiorów). Algorytm przechodzi przez permutację. Rozpoczynające się przedziały powiększają bieżącą klikę, która jest maksymalna, gdy następnie wierzchołki są z niej usuwane. Usuwane wierzchołki są wstawiane do PEO jako wierzchołki simplicjalne (ich sąsiedzi tworzą klikę).

Listing 4.3. Wyznaczanie klik maksymalnych.

```
def find_peo_cliques(perm):
    """Finding PEO and maximal cliques for the interval graph."""
    growing = True
    peo = []
    cliques = [] # list of maximal cliques
    used = set() # current clique
    for node in perm:
```

```
        if node in used:
            if growing:
                cliques.append(set(used)) # new maximal clique
            used.remove(node)
            peo.append(node)
            growing = False
        else: # clique is growing
            used.add(node)
            growing = True
    return peo, cliques
```

4.6. Budowanie grafu abstrakcyjnego z permutacji

W grafie przedziałowym w reprezentacji permutacyjnej można w czasie $O(n+m)$ utworzyć graf abstrakcyjny. Funkcja make_abstract_interval_graph(perm) zwraca klasę Graph reprezentującą graf abstrakcyjny. Algorytm przechodzi przez permutację. Sprawdzane jest to, czy wierchołek został już utworzony. Jeśli tak, to usuwany jest z listy przechowującej użyte wierzchołki, oznacza to również, że algorytm doszedł do końca przedziału dla danego elementu z reprezentacji permutacyjnej. W przeciwnym przypadku dodajemy nowy wierzchołek do grafu i tworzymy krawędzie między nim a wszystkimi wierzchołkami przechowywanymi w liście wierzchołków, przy których nie został osiągnięty koniec przedziału w reprezentacji permutacyjnej.

Listing 4.4. Budowanie grafu abstrakcyjnego z permutacji.

```
def make_abstract_interval_graph(perm):
    """Finding an abstract interval graph from double perm in O(n+m) time."""
    graph = Graph(n=len(perm) // 2)
    used = set() # current nodes
    for source in perm:
        if source in used: # bedzie usuwanie node
            used.remove(source)
        else: # dodajemy nowy node
            graph.add_node(source) # isolated node is possible!
            for target in used:
                graph.add_edge(Edge(source, target))
            used.add(source)
    return graph
```

4.7. Rysowanie przedziałów grafu przedziałowego

Przygotowano algorytm do rysowania przedziałów w grafie przedziałowym w formie permutacyjnej. Przygotowany algorytm działa w czasie $O(n)$. Po jego użyciu na konsoli zostaną graficznie przedstawione przedziały. Przykładowy wynik pokazano w rozdziale 3.6.

Listing 4.5. Graficzne przedstawienie grafu przedziałowego.

```
def intevals_drawing(perm):
    """Interval graph drawing."""
    lenght = len(perm) // 2
    used = [-1] * lenght
    str_lines = [""] * lenght
    symbol_len = max(len(str(node)) for node in perm)
    empty = ' '
    line = '--'
    for node in perm: # idziemy wzdluz permutacji
        if node in used:
            idx = used.index(node)
            str_lines[idx] += str(node)
            used[idx] = -1
        else:
            free_line_idx = -1
            for i in range (lenght):
                if used[i] == -1:
                    free_line_idx = i
                    break
            str_lines[free_line_idx] += str(node)
            used[free_line_idx] = node

        for i in range(lenght):
            if used[i] == -1:
                str_lines[i] += (empty * symbol_len)
            else:
                str_lines[i] += (line * symbol_len)

    for line in str_lines:
        if line.strip() == "":
            break
        print(line)
```

4.8. Kolorowanie wierzchołków grafu przedziałowego

Kolorowanie wierzchołków grafu nieskierowanego polega na przyporządkowaniu wierzchołkom grafu kolorów (etykiet, liczb naturalnych) w taki sposób, aby końce każdej krawędzi miały różne kolory. Znalezienie optymalnej liczby kolorów (liczba chromatyczna) to klasyczny problem NP-zupełny. W przypadku grafów cięciwowych można znaleźć optymalną liczbę kolorów w czasie $O(n + m)$. Okazuje się, że w przypadku grafów przedziałowych ten czas można zredukować do $O(n)$ przy wykorzystaniu reprezentacji permutacyjnej.

W podanym algorytmie każdy nowy przedział otrzymuje pierwszy wolny kolor (algorytm zachłanny), ponieważ wiadomo, że nowy przedział powiększa istniejącą klikę i nie ma potrzeby sprawdzania jego sąsiadów. Usuwany przedział zwraca swój kolor do puli wolnych kolorów. Jest jasne, że algorytm wykorzysta liczbę kolorów równą liczności największej kliki, a to potwierdza, że grafy przedziałowe są grafami doskonałymi.

W podanej implementacji pula wolnych kolorów jest realizowana przez listę Pythona, która emuluje stos. Pobieranie wolnego koloru i zwracanie go do puli wykonywane jest w stałym czasie $O(1)$. Funkcja zwraca słownik z parami wierzchołek i jego kolor. Jest to zgodne z konwencją stosowaną w pakiecie graphtheory.

Listing 4.6. Kolorowanie wierzchołków grafu przedziałowego.

```
def interval_node_color(perm):
    """Vertex coloring of an interval graph (double perm) in O(n) time."""
    n = len(perm) // 2
    free = list(range(n-1,-1,-1)) # wolne kolory O(n)
    color = dict()
    used = set() # aktywne przedzialy
    for node in perm: # idziemy wzdluz permutacji
        if node in used: # bedzie usuwanie przedzialu, klika zmaleje
            used.remove(node)
            free.append(color[node]) # zwracamy kolor
        else: # dodajemy nowy przedzial, klika rosnie
            used.add(node)
            color[node] = free.pop() # pobieramy wolny kolor
    return color
```

4.9. Wyznaczanie największego zbioru niezależnego

Posiadanie reprezentacji permutacyjnej grafu przedziałowego pozwala na szybkie wyznaczenie największego zbioru niezależnego [18]. W pierwszym etapie należy posortować przedziały wg rosnących prawych końców przedziałów, co zajmuje czas $O(n \log n)$. W reprezentacji permutacyjnej przedziały już są posortowane, więc ten etap można pominąć. Niech S oznacza początkowo pusty największy zbiór niezależny. W drugim etapie pobieramy pierwszy przedział I z lewej strony i wstawiamy go do S. Ze zbioru przedziałów usuwamy wszystkie przedziały sąsiadujące z I. Czynności powtarzamy aż do wyczerpania przedziałów. Drugi etap można zrealizować w czasie $O(n)$, jeżeli przedziały będą przechowywane na liście podwójnie powiązanej i będą usuwane w stałym czasie.

Listing 4.7 przedstawia prostszą implementację algorytmu wyznaczania największego zbioru niezależnego, gdzie przedziały (etykiety) nie są usuwane, tylko oznaczane jako zabronione. Powoduje to wielokrotne oznaczanie tych samych przedziałów jako zabronione, przez co dla pewnych grafów przedziałowych złożoność obliczeniowa jest gorsza niż $O(n)$.

Warto zauważyć podobieństwo podanego algorytmu do algorytmu Gavrila działającego dla ogólnych grafów cięciwowych [19]. Posortowane przedziały odpowiadają uporządkowaniu wierzchołków grafu wg pewnego PEO. Usuwanie przedziałów sąsiadujących z przedziałem I odpowiada zaznaczaniu w PEO sąsiadujących (zabronionych) wierzchołków, które łącznie z danym wierzchołkiem tworzą klikę.

Listing 4.8 przedstawia szybszą wersję algorytmu wyznaczania największego zbioru niezależnego, gdzie etykiety sąsiadów są wymazywane z permutacji (zastępowane przez None), przez co nie będą analizowane w późniejszych etapach algorytmu. W ten sposób została uzyskana optymalna złożoność obliczeniowa $O(n)$.

Listing 4.7. Największy zbiór niezależny grafu przedziałowego.

```
def interval_max_iset(perm):
    """Finding a maximum independent set of an interval graph (double perm)."""
    iset = set()
    used = set() # aktywne przedzialy
    forbidden = set()
    for node in perm: # idziemy wzdluz permutacji
        if node in used: # bedzie usuwanie node, klika zmaleje
            used.remove(node)
```

```
            if node not in forbidden:
                iset.add(node)
                forbidden.update(used)
        else: # dodajemy nowy node, klika rosnie
            used.add(node)
    return iset
```

Listing 4.8. Szybkie wyznaczanie największego zbioru niezależnego grafu przedziałowego.

```
def interval_max_iset2(perm):
    """Finding maximum iset of an interval graph (double perm)."""
    perm = list(perm) # O(n) time, bedzie modyfikacja perm
    iset = set()
    used = set() # aktywne przedzialy
    # Zapisuje indeksy koncow przedzialow.
    pairs = dict((node, []) for node in set(perm)) # O(n) time
    for idx, node in enumerate(perm): # O(n) time
        pairs[node].append(idx)
    for node in perm: # idziemy wzdluz permutacji
        if node is None: # wymazany przedzial
            continue
        elif node in used: # bedzie usuwanie node, klika zmaleje
            used.remove(node)
            iset.add(node)
            for source in used: # usuwam sasiadow z perm, sumarycznie O(n)
                perm[pairs[source][0]] = None
                perm[pairs[source][1]] = None
            used.clear() # przedzialy wymazane
        else: # dodajemy nowy node, klika rosnie
            used.add(node)
    return iset
```

5. Dekompozycja ścieżkowa

Z grafami cięciwowymi wiążą się takie pojęcia jak dekompozycja drzewowa (ang. *tree decomposition, TD*) i szerokość drzewowa (ang. *treewidth*) [20]. Dla grafów przedziałowych mamy analogiczne pojęcia, takie jak dekompozycja ścieżkowa (ang. *path decomposition, PD*) i szerokość ścieżkowa (ang. *pathwidth*) [21]. Nieformalnie można powiedzieć, że chodzi o przedstawienie danego grafu cięciwowego jako 'pogrubionego' drzewa (lub ścieżki), przy czym wierzchołkami tego drzewa są kliki maksymalne w pierwotnym grafie.

W przypadku ogólnych grafów również wyznacza się dekompozycję drzewową (ścieżkową), ale wcześniej należy dodać nowe krawędzie w taki sposób, aby powstał graf cięciwowy (przedziałowy). Nowy graf nazywa się uzupełnieniem cięciwowym (przedziałowym). Generalnie jest to problem NP-trudny, dlatego zwykle stosuje się heurystyki. Posiadanie dobrej dekompozycji drzewowej (ścieżkowej) umożliwia wydajne rozwiązywanie wielu problemów, zwykle techniką programowania dynamicznego.

5.1. Definicje i oznaczenia

Definicja: Mamy dany graf nieskierowany G. Sekwencja $(X_1, \ldots, X_r)$ podzbiorów $V(G)$ jest *dekompozycją ścieżkową (PD)* grafu G, jeżeli spełnione są warunki:

1. Suma wszystkich zbiorów X_i równa się $V(G)$. Zbiory X_i nazywa się workami (ang. *bags*).
2. Dla każdej krawędzi $vw \in E(G)$, pewien zbiór X_i $(1 \leqslant i \leqslant r)$ zawiera oba końce krawędzi v i w.
3. Dla $1 \leqslant i \leqslant j \leqslant k \leqslant r$, $X_i \cap X_k$ zawiera się w X_j. Inaczej można powiedzieć, że dla każdego wierzchołka $v \in V(G)$ worki X_i zawierające v występują obok siebie w sekwencji $(X_1, \ldots, X_r)$.

Definicja: Szerokością ścieżkową (ang. *pathwidth*) grafu nieskierowanego G jest najmniejsza liczba $k \geqslant 0$ taka, że G posiada dekompozycję ścieżkową

$(X_1, \ldots, X_r)$, przy czym $|X_i| \leqslant k+1$ dla każdego $1 \leqslant i \leqslant r$. Oznaczamy $\mathrm{pw}(G) = k$.

Definicja: Jeżeli G jest dowolnym grafem nieskierowanym, to *uzupełnieniem przedziałowym* grafu G (ang. *interval completion*) jest graf H taki, że $V(H) = V(G)$ oraz G jest podgrafem H.

5.2. Wyznaczanie PD dla grafu przedziałowego

Posiadanie grafu przedziałowego w reprezentacji permutacyjnej pozwala na szybkie wyznaczenie dekompozycji ścieżkowej grafu.

Dane wejściowe: Graf przedziałowy G w reprezentacji permutacyjnej.

Problem: Wyznaczenie PD grafu G.

Opis algorytmu: Algorytm rozpoczynamy od wyznaczenia listy klik maksymalnych (worków), zapisanych jako krotki posortowanych wierzchołków. Lista worków połączonych kolejno krawędziami tworzy PD.

Złożoność: Złożoność czasowa wyznaczania r klik maksymalnych (jako zbiorów) wynosi $O(n)$, przy czym trzeba jeszcze doliczyć czas sortowania elementów zbiorów w celu otrzymania posortowanych krotek. W drugim etapie algorytmu tworzymy instancję klasy Graph z r wierzchołkami i $r-1$ krawędziami w czasie $O(r)$.

Uwagi: Warto zauważyć, że klasa Graph z pakietu graphtheory jest na tyle elastyczna, że może przechowywać zarówno graf, jak i jego dekompozycję ścieżkową.

Listing 5.1. Wyznaczanie dekompozycji ścieżkowej grafu przedziałowego.

```
def make_path_decomposition(perm):
    """Finding a path decomposition of an iterval graph (double perm)."""
    growing = True
    bags = [] # list of maximal cliques/bags
    used = set() # current clique
    for node in perm:
        if node in used:
            if growing: # new maximal clique
                bags.append(tuple(sorted(used)))
```

```
            used.remove(node)
            growing = False
        else: # clique is growing
            used.add(node)
            growing = True
    T = Graph(n=len(bags)) # path decomposition
    for bag in bags:
        T.add_node(bag)
    for i in range(len(bags)-1):
        T.add_edge(Edge(bags[i], bags[i+1]))
    return T
```

5.3. Testowanie dekompozycji ścieżkowej

Dekompozycję ścieżkową danego grafu można uzyskać ręcznie lub za pomocą funkcji make_path_decomposition(perm). W każdym przypadku dobrze jest sprawdzić poprawność dekompozycji. Została przygotowana klasa testująca TestPathDecomposition, wykorzystująca moduł unittest i technikę *test-driven development*. Metody klasy testującej sprawdzają kolejne punkty definicji dekompozycji ścieżkowej. Warto zauważyć, że analogiczna klasa TestTreeDecomposition w istotny sposób różni się od naszej.

Listing 5.2. Testowanie dekompozycji ścieżkowej.

```
class TestPathDecomposition(unittest.TestCase):

    def setUp(self):
        self.G = ... # abstract graph
        self.T = ... # path decomposition

    def test_is_path(self):
        self.assertTrue(is_acyclic(self.T))
        self.assertTrue(is_connected(self.T))
        degree_dict = dict()
        for bag in self.T.iternodes():
            deg = self.T.degree(bag)
            degree_dict[deg] = degree_dict.get(deg, 0) + 1
        self.assertEqual(degree_dict[1], 2)
        self.assertEqual(degree_dict[2], self.T.v()-2)

    def test_bags_cover_vertices(self):
        set1 = set(self.G.iternodes())
        set2 = set()
```

```
        for bag in self.T.iternodes():
            set2.update(bag)
        self.assertEqual(set1, set2)

    def test_bags_cover_edges(self):
        # Budujemy 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_path_property(self):
        # Szukamy krancowego worka degree 1.
        for bag in self.T.iternodes():
            if self.T.degree(bag) == 1:
                root_bag = bag
                break
        bag_order = [] # kolejnosc odkrywania przez BFS
        algorithm = SimpleBFS(self.T)
        algorithm.run(root_bag, pre_action=lambda node: bag_order.append(node))
        self.assertEqual(root_bag, bag_order[0])
        # Dla kazdego wierzcholka grafu G budujemy subpath (list).
        subpath = dict((node, []) for node in self.G.iternodes())
        # Zaczynamy pierwsze subpath.
        for node in root_bag:
            subpath[node].append(root_bag)
        # Przetwarzamy nastepne bags.
        is_pd = True # flaga sprawdzajaca poprawnosc PD
        for bag in bag_order[1:]:
            for node in bag:
                if len(subpath[node]) == 0: # new subpath
                    subpath[node].append(bag)
                elif algorithm.parent[bag] == subpath[node][-1]: # kontynuacja
                    subpath[node].append(bag)
                else: # rozlaczne subpaths, wlasnosc nie jest spelniona
```

```
                    is_pd = False
        self.assertTrue(is_pd)

    def test_pathwidth(self):
        pathwidth = max(len(bag) for bag in self.T.iternodes()) -1
        self.assertEqual(pathwidth, 2)
```

---

6. Podsumowanie

Celem niniejszej pracy było przedstawienie grafów przedziałowych i zbadanie ich właściwości. Przytoczono odpowiednie definicje z ogólnej teorii grafów, jak i zagadnienia dotyczące samych grafów przedziałowych. Dużą trudnością okazała się implementacja algorytmu rozpoznawania grafu przedziałowego. Pseudokod przedstawiony w pracy Habiba zawierał pewne uproszczenia, które należało wypełnić w trakcie implementacji działającego algorytmu. Rozpoznanie grafu przedziałowego umożliwia zastosowanie wielu algorytmów działających w czasie $O(n)$, natomiast dla ogólnych grafów dostępne algorytmy mają złożoność wykładniczą.

W pierwszej części pracy (rozdz. 2) przedstawiono wstęp do teorii grafów, pojęcia przydatne do zrozumienia dalszej części pracy, oraz opis grafu przedziałowego. Informacje zostały zaczerpnięte z arytkułu Bootha [10], Habiba [17] oraz z innych artykułów naukowych i prezentacji osób zajmujących się tą tematyką. Dodatkowo w celu lepszego zrozumienia wyglądu oraz działania algorytmów grafu przedziałowego zamieszczono ilustracje graficzne.

W części drugiej (rozdział 3) przedstawiona została implementacja algorytmów grafu przedziałowego. Opisano metody, podano przykłady użycia oraz zwracane wartości. W osobnym rozdziale 4 opisano algorytmy, które stanowią podstawę działania tych metod, a także przedstawiono ich złożoność obliczeniową. Dodatkowo w rozdziale 5 zostały przedstawione definicje oraz opisano algorytmy dekompozycji ścieżkowej grafu przedziałowego. Optymalna dekompozycja ścieżkowa pozwala na efektywne rozwiązywanie wielu problemów, zazwyczaj poprzez wykorzystanie techniki programowania dynamicznego.

Praca pozwala czytelnikowi zrozumieć podstawy grafów przedziałowych oraz sposoby ich wykorzystania w różnych dziedzinach. Stworzona implementacja została starannie przetestowana (dodatek A) i skutecznie sprawdza się w rozpoznawaniu i wykonywaniu różnych obliczeń na grafach przedziałowych. Kod działa poprawnie zarówno w Pythonie 2.7, jak i Pythonie 3.

A. Testy algorytmów

W tym rozdziale opracowane są wyniki testów złożoności algorytmów. Wynikiem testu wydajności jest wykres zależności czasu działania algorytmu od liczności zbioru punktów przekazanych do struktury. Testy były przeprowadzone na komputerze z procesorem Intel Core i5 3.7GHz.

Algorytmy zostały dodatkowo przetestowane przy pomocy testów jednostkowych z użyciem modułu unittest [25].

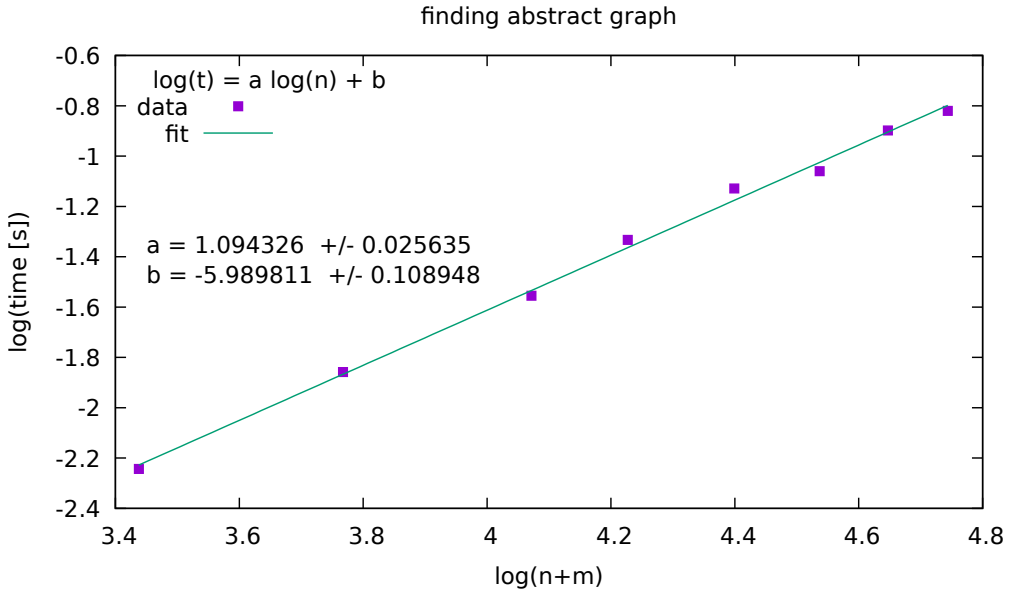
A.1. Testy rozpoznawania grafów przedziałowych

A.1.1. Testy na losowych grafach przedziałowych

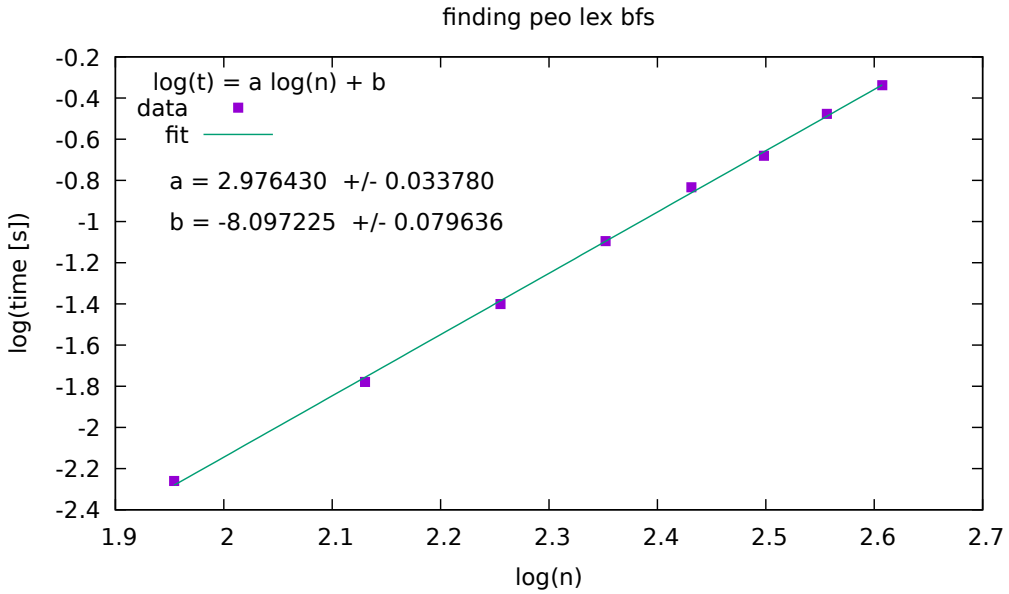
W celu przetestowania wydajności zmierzono czasy wykonania algorytmu. Badano wywołania dla grafów o coraz większych liczbach wierzchołków. Do testów wykorzystano w pełni losowe grafy przedziałowe, aby sprawdzić szybkość działania funckji niezależnie od wyglądu grafu.

Tworzenie reprezentacji grafu przedziałowego w formie klasy Graph. Teoretyczna złożoność algorytmu tworzenia reprezentacji grafu przedziałowego w formie klasy Graph wynosi $O(n + m)$. Widoczny na wykresie A.1 współczynnik $a = 1.094(26)$ oznacza, że uzyskana złożoność spełnia założenia teoretyczne.

Znajdowanie PEO przy pomocy algorytmu Lex-BFS. Teoretyczna złożoność algorytmu znajdowania PEO przy pomocy Lex-BFS wynosi $O(n^2)$. Widoczny na wykresie A.2 współczynnik $a = 2.976(34)$ oznacza, że uzyskana złożoność praktyczna jest bliższa do złożoności $O(n^3)$. Przyczyną jest implementacja bazująca na pseudokodzie, przez co kod jest czytelny. Optymalna implementacja wymaga zastosowania złożonej struktury danych i techniki *partition refinement*.

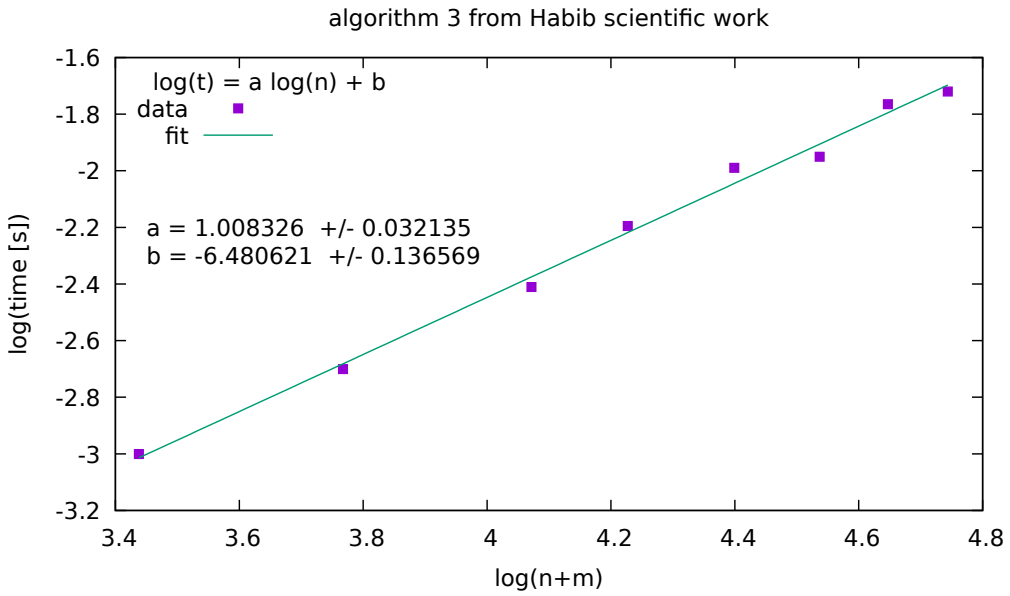


Rysunek A.1. Wyniki pomiarów tworzenia reprezentacji grafu przedziałowego w formie klasy Graph z reprezentacji permutacyjnej. Współczynnik a bliski jedności wskazuje na złożoność liniową $O(n+m)$.



Rysunek A.2. Wyniki pomiarów znajdowanie PEO przy pomocy algorytmu Lex-BFS. Współczynnik a bliski 3 wskazuje na złożoność $O(n^3)$.

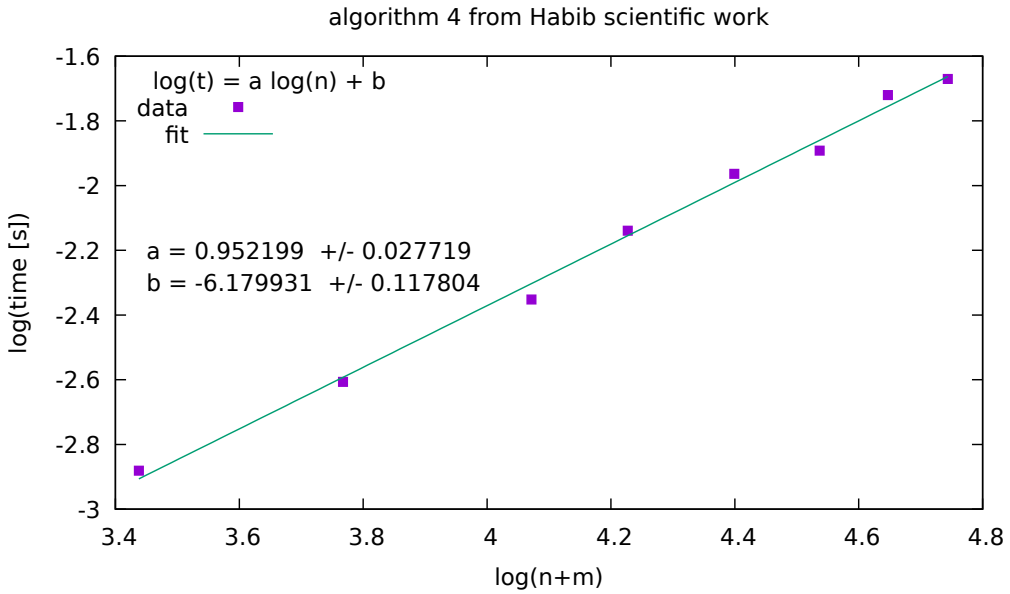
Algorytm sprawdzania czy PEO jest poprawne. Teoretyczna złożoność algorytmu sprawdzającego czy PEO jest poprawne wynosi $O(n+m)$. Widoczny na wykresie A.3 współczynnik $a = 1.008(33)$ oznacza, że uzyskana złożoność spełnia założenia teoretyczne.



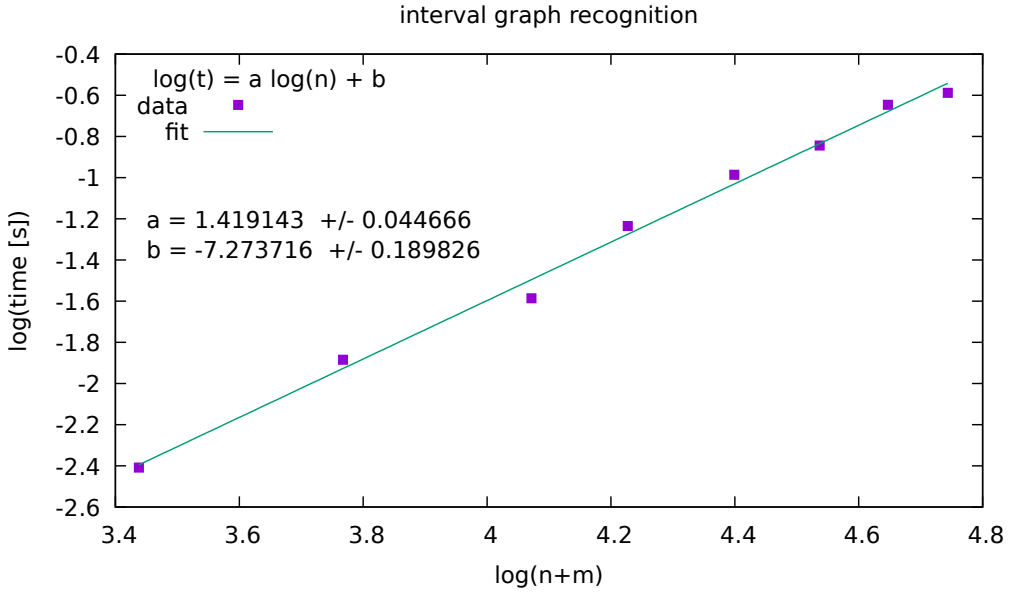
Rysunek A.3. Wyniki pomiarów sprawdzania czy graf jest grafem cięciwowym. Współczynnik a bliski 1 wskazuje na złożoność $O(n+m)$.

Algorytm tworzący drzewo klik. Teoretyczna złożoność algorytmu tworzącego drzewo klik wynosi $O(n+m)$. Widoczny na wykresie A.4 współczynnik $a = 0.952(28)$ oznacza, że uzyskana złożoność spełnia założenia teoretyczne.

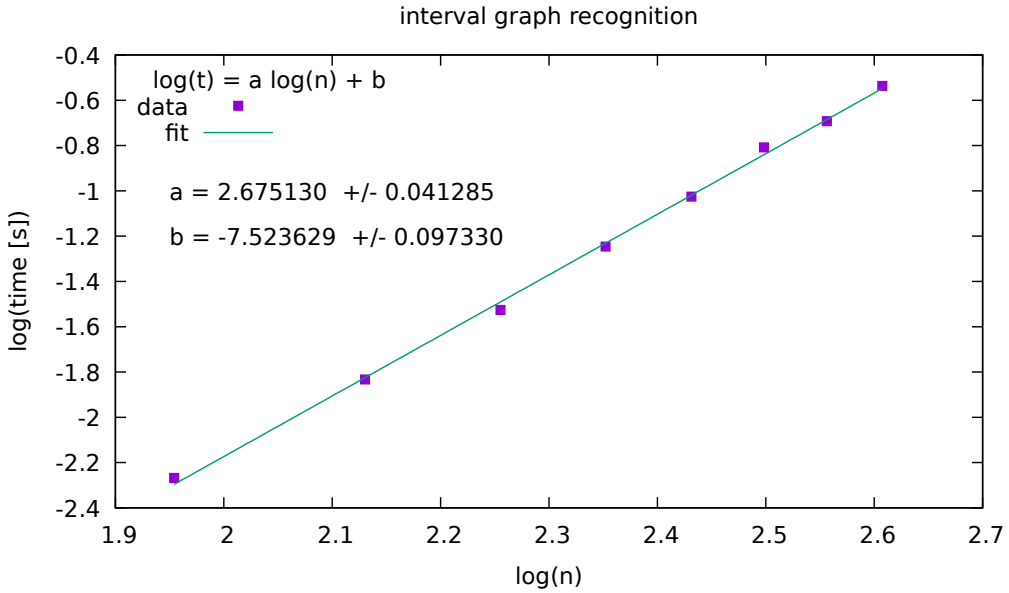
Algorytm sprawdzający czy graf jest grafem przedziałowym. Teoretyczna złożoność algorytmu sprawdzającego czy graf jest grafem przedziałowym wynosi $O(n+m)$. Widoczny na wykresie A.5 współczynnik $a = 1.419(45)$ oznacza, że uzyskana złożoność jest ponad złożoność $O(n+m)$.



Rysunek A.4. Wyniki pomiarów tworzenia drzewa klik. Współczynnik a bliski 1 wskazuje na złożoność $O(n+m)$.



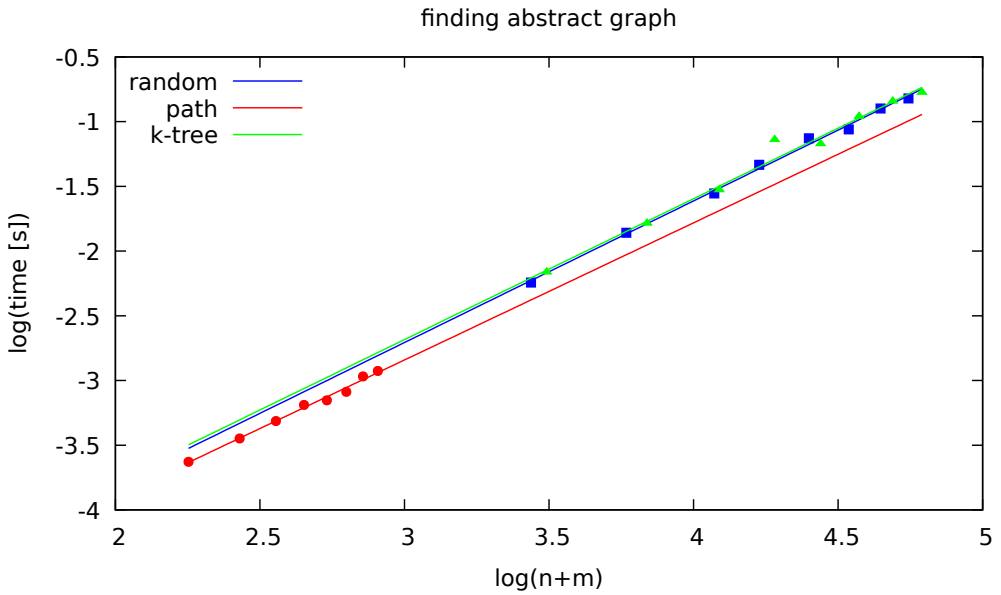
Rysunek A.5. Wyniki pomiarów sprawdzania czy graf jest grafem przedziałowym. Współczynnik a wskazuje na złożoność nieco większą niż $O(n+m)$.



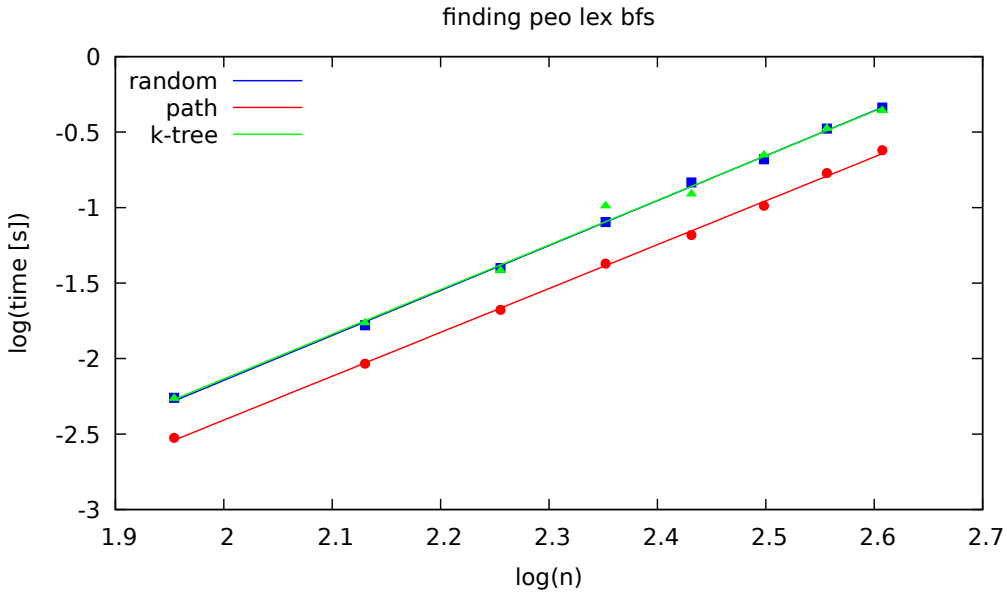
Rysunek A.6. Wyniki pomiarów sprawdzania czy graf jest grafem przedziałowym. Współczynnik a będący ponad 2 wskazuje na złożoność większą niż $O(n^2)$.

A.1.2. Testy porównawcze dla różnych typów grafów przedziałowych

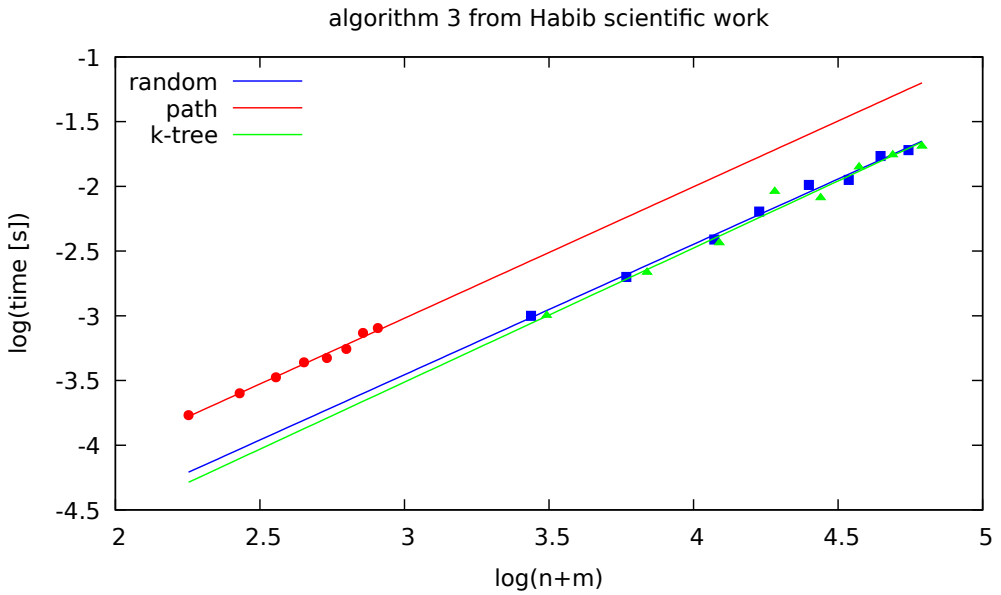
W tym rozdziale znajduje porównanie wydajności poszczególnych algorytmów względem różnych typów grafów. Badano wywołania dla grafów o coraz większych liczbach wierzchołków. Do testów wykorzystano w pełni losowe grafy przedziałowe, losowe grafy k-tree i grafy liniowe. Graf k-tree zapewnia, że ilość krawędzi w grafie będzie rzędu n^2. Na widocznych wykresach widać zależność pomiędzy czasem wykonania, a ilością krawędzi w grafie. Operacje na grafach liniowych, które posiadają najmniejszą ich ilość wykonują się w krótszym czasie.



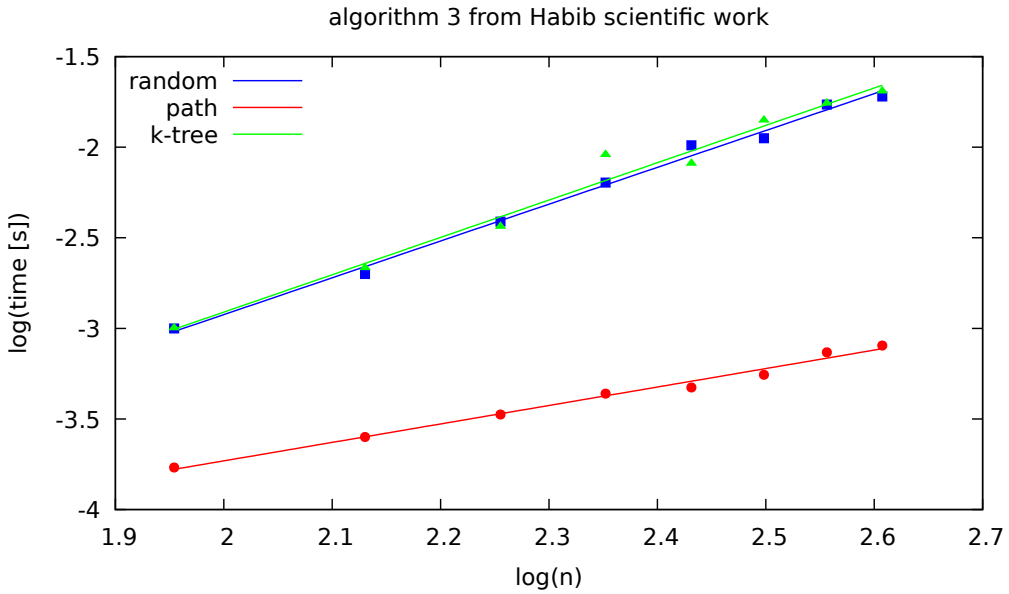
Rysunek A.7. Wyniki porównania tworzenia reprezentacji grafu przedziałowego w formie klasy Graph z reprezentacji permutacyjnej.



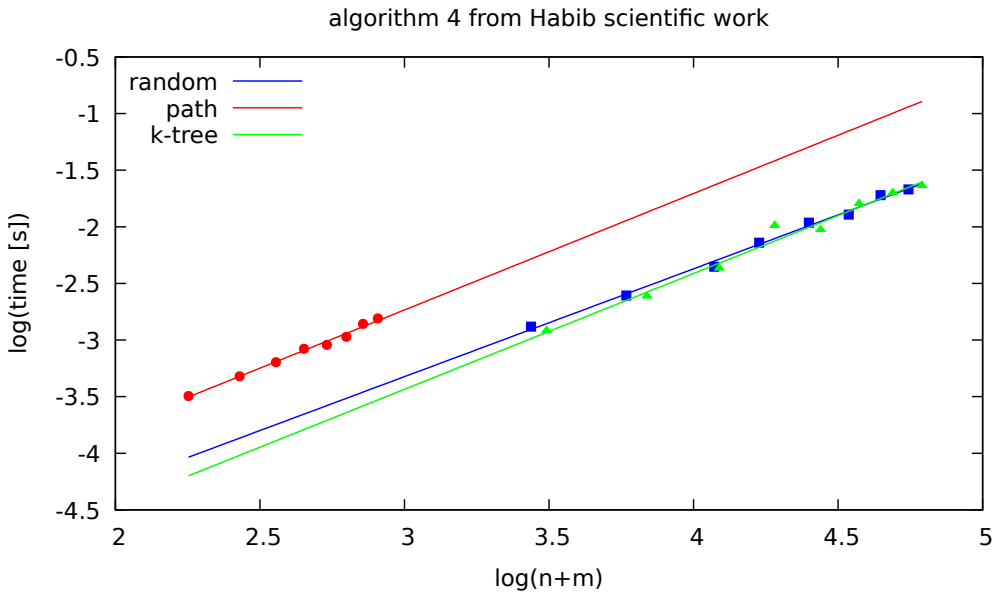
Rysunek A.8. Wyniki porównania znajdowanie PEO przy pomocy algorytmu Lex-BFS.



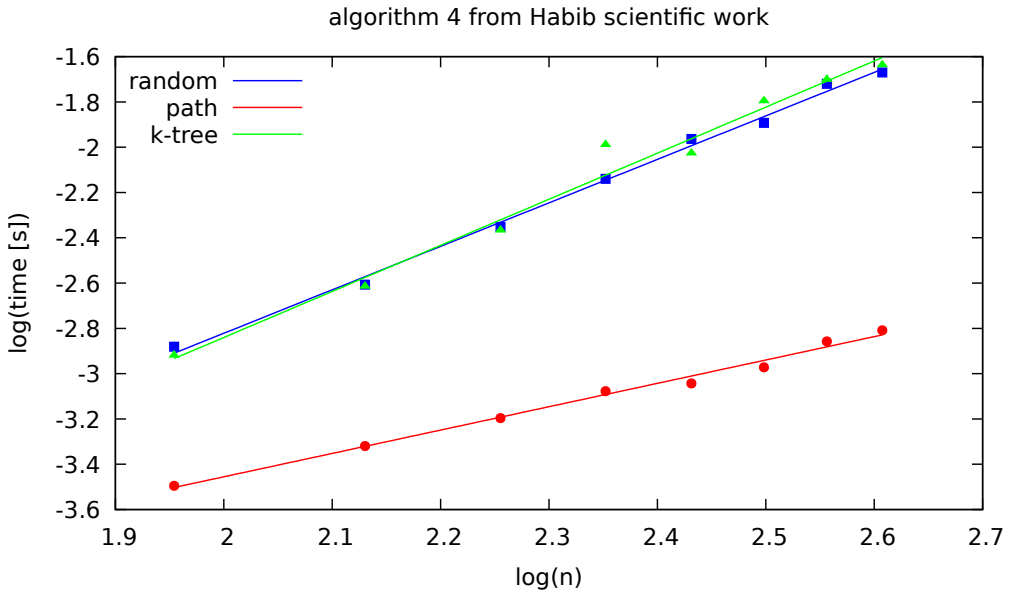
Rysunek A.9. Wyniki pomiarów sprawdzania czy PEO jest poprawne względem ilości wierzchołków i krawędzi.



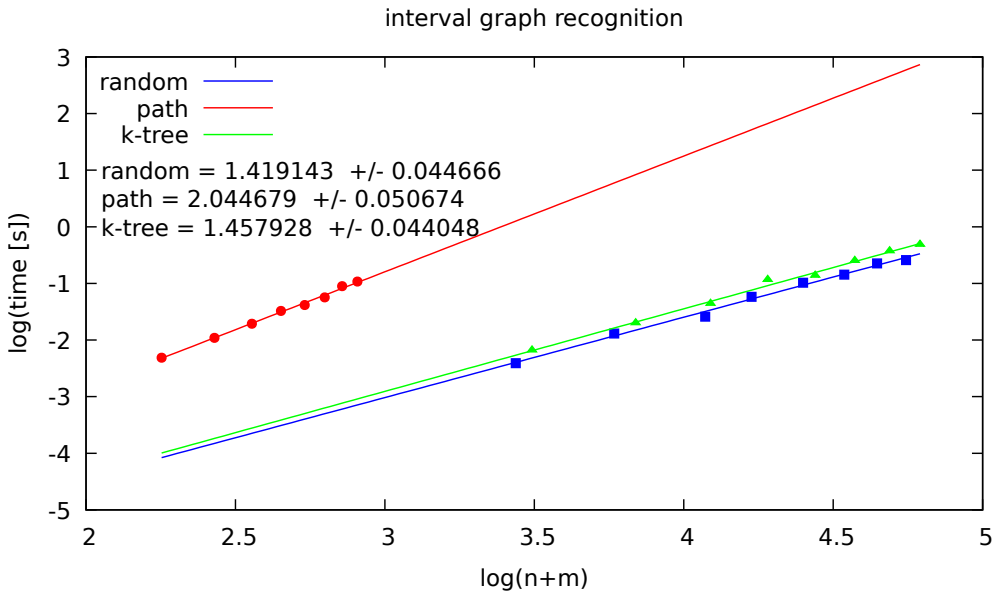
Rysunek A.10. Wyniki pomiarów sprawdzania czy PEO jest poprawne względem ilości wierzchołków.



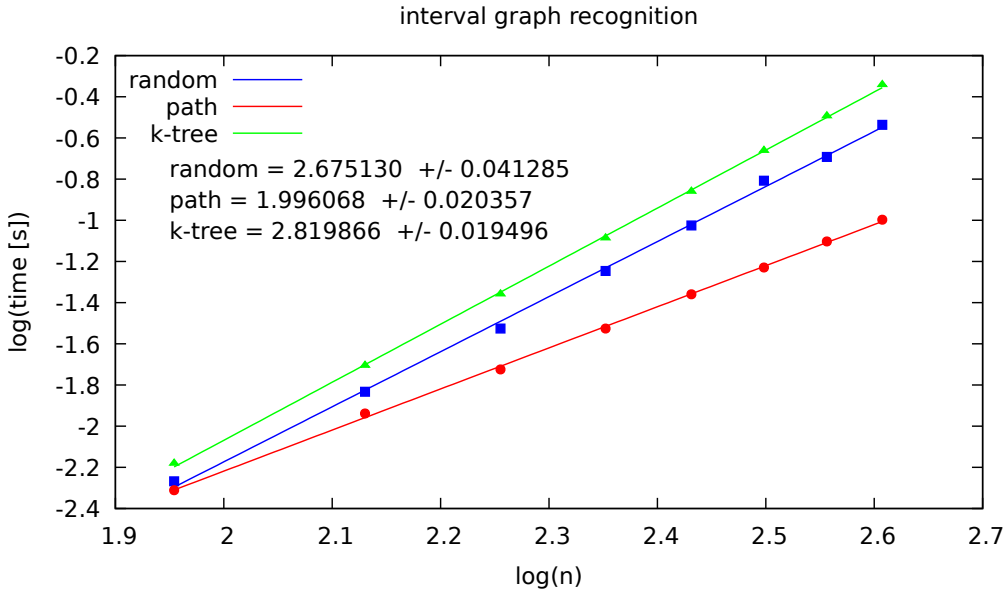
Rysunek A.11. Wyniki pomiarów tworzenia drzewa klik względem ilości wierzchołków i krawędzi.



Rysunek A.12. Wyniki pomiarów tworzenia drzewa klik względem ilości wierzchołków.



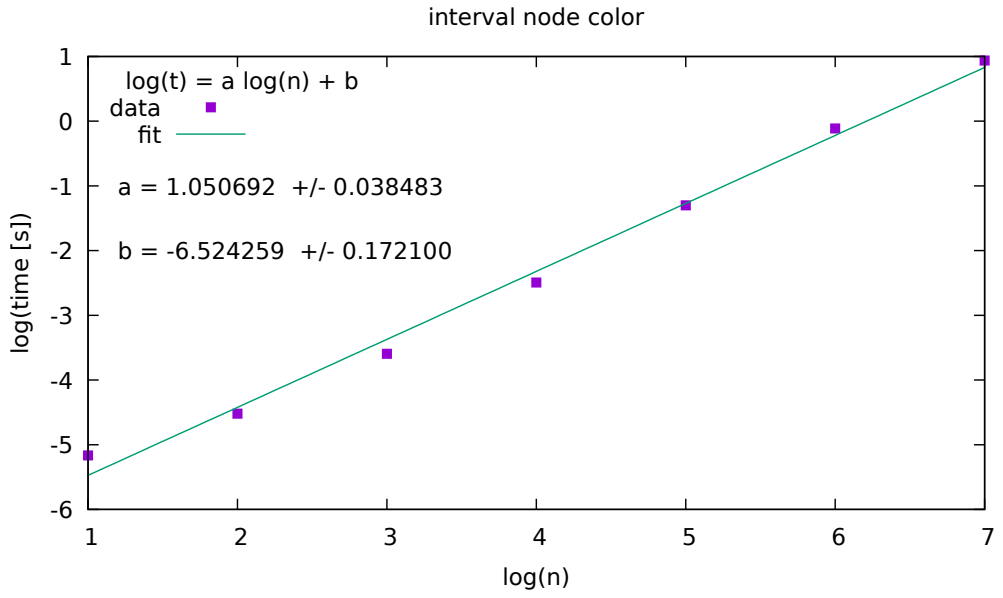
Rysunek A.13. Wyniki pomiarów sprawdzania czy graf jest grafem przedziałowym względem ilości wierzchołków i krawędzi.



Rysunek A.14. Wyniki pomiarów sprawdzania czy graf jest grafem przedziałowym względem ilości wierzchołków.

A.2. Testy kolorowania grafu przedziałowego

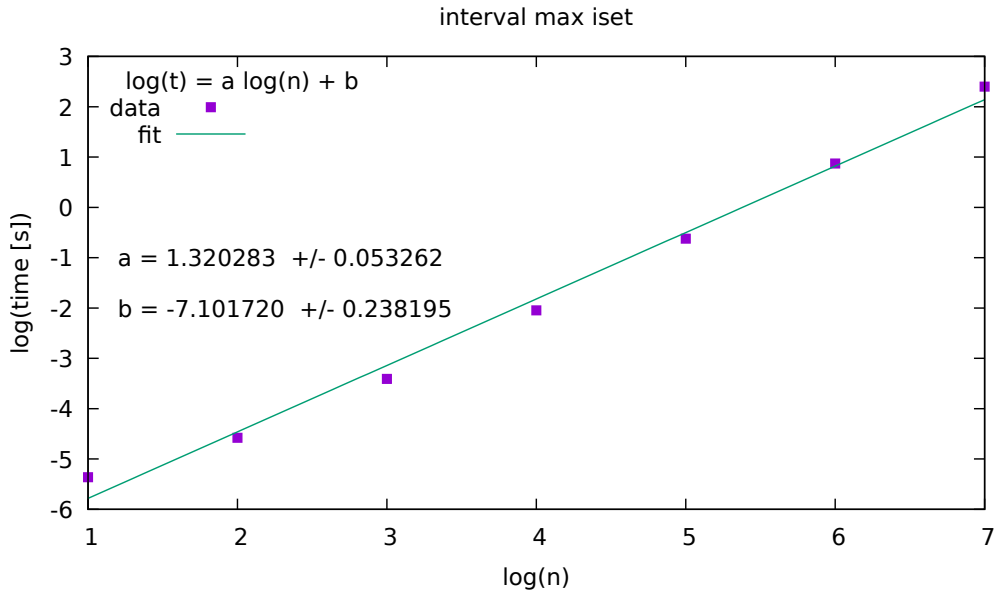
Algorytm kolorowania grafu przedziałowego. Teoretyczna złożoność algorytmu kolorowania grafu przedziałowego wynosi $O(n)$. Widoczny na wykresie A.15 współczynnik $a = 1.05(4)$ oznacza, że uzyskana złożoność spełnia założenia teoretyczne.



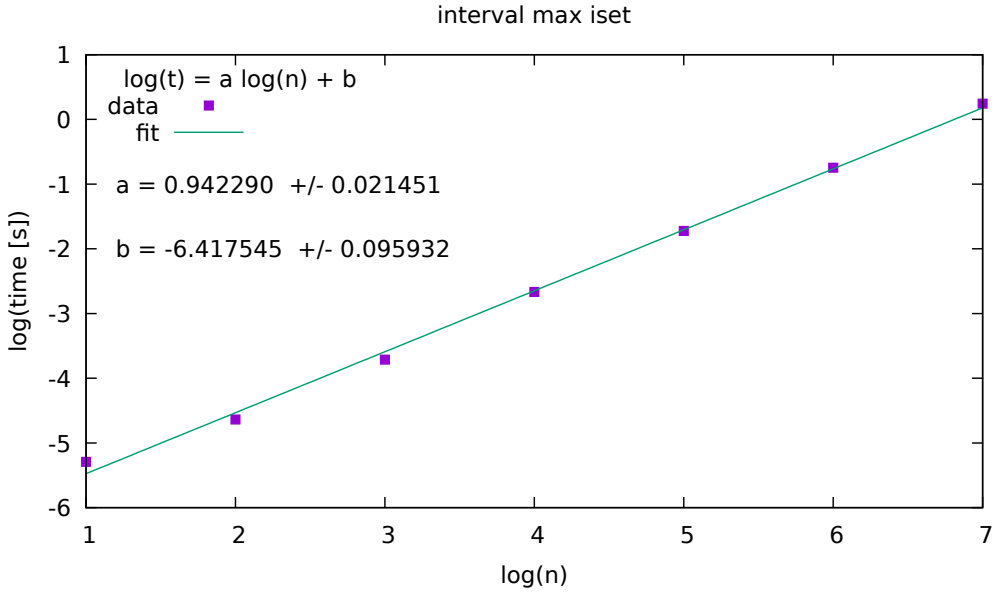
Rysunek A.15. Wyniki pomiarów algorytmu kolorowania grafu przedziałowego. Współczynnik a będący blisko 1 wskazuje na złożoność $O(n)$.

A.3. Testy znajdowania największego zbioru niezależnego grafu przedziałowego

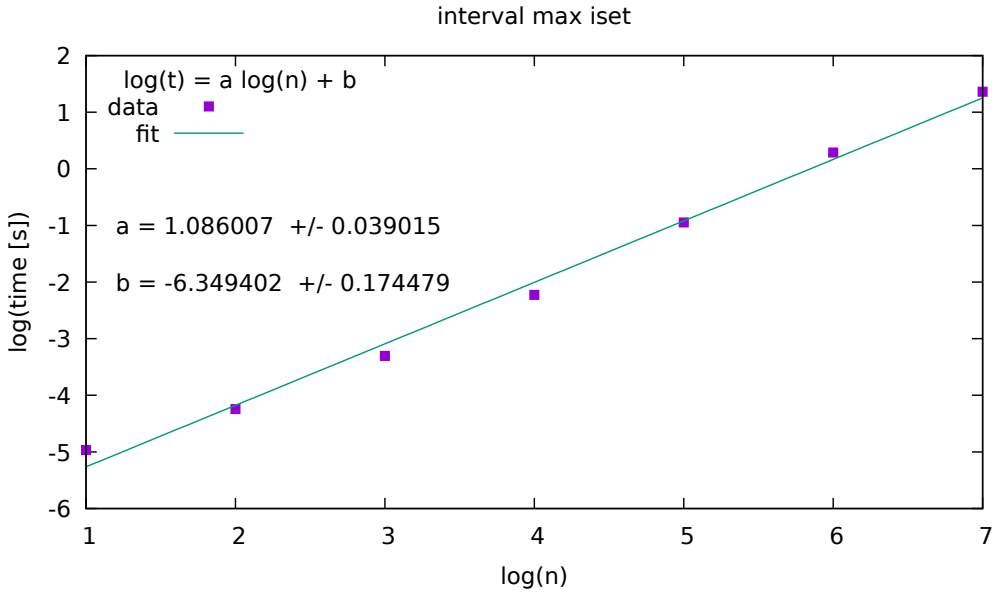
Teoretyczna złożoność algorytmu znajdowania największego zbioru niezależnego grafu przedziałowego wynosi $O(n)$. Widoczny na wykresie A.16 współczynnik $a = 1.320(53)$ dla grafu losowego oraz na wykresie A.17 $a = 0.942(22)$ dla grafu P_n oznacza, że uzyskana złożoność dla grafów losowych przekracza założenia teoretyczne. Natomiast widoczny na wykresie A.18 współczynnik $a = 1.086(39)$ oznacza, że zastosowanie usuwania przedziałów przyspieszyło działanie algorytmu.



Rysunek A.16. Wyniki pomiarów algorytmu znajdowania największego zbioru niezależnego grafu przedziałowego dla grafu losowego bez usuwania przedziałów. Współczynnik a będący ponad 1 wskazuje na złożoność większą niż $O(n)$.



Rysunek A.17. Wyniki pomiarów algorytmu znajdowania największego zbioru niezależnego grafu przedziałowego dla grafu P_n bez usuwania przedziałów. Współczynnik a będący blisko 1 wskazuje na złożoność $O(n)$.



Rysunek A.18. Wyniki pomiarów algorytmu znajdowania największego zbioru niezależnego grafu przedziałowego z usuwaniem przedziałów dla grafu losowego. Współczynnik a będący blisko 1 wskazuje na złożoność $O(n)$.

Bibliografia

[1] Wikipedia, Interval graph, 2023, `https://en.wikipedia.org/wiki/Interval_graph`.

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

[3] Andrzej Kapanowski, graphtheory, GitHub repository, 2023, `https://github.com/ufkapano/graphtheory/`.

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

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

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

[7] Wikipedia, Clique (graph theory), 2023, `https://en.wikipedia.org/wiki/Clique_(graph_theory)`.

[8] Vizing, V. G. *On an estimate of the chromatic class of a p-graph* Diskret Analiz, 3(25), 25-30, 1964.

[9] Golumbic, M. C. *Algorithmic graph theory and perfect graphs* Academic Press, 2004.

[10] Booth, K. S., Lueker, G. S. *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms* Journal of Computer and System Sciences, 13(3), 335-379, 1976.

[11] Tarjan, R. E., & Yannakakis, M. *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.

[12] Gilmore, P. C., & Hoffman, A. J. *A characterization of comparability graphs and of interval graphs* Canadian Journal of Mathematics, 16(04), 539-548, 1964.

[13] Kratsch, S., & Spinrad, J. *Strong recognition and isomorphism of interval graphs* Discrete Applied Mathematics, 92(1-3), 189-200, 1999.

[14] Köbler, J., Kuhnert, S., & Watanabe, O. *Interval graph representation with given interval and intersection lengths* Journal of Discrete Algorithms, 34, 108-117, 2015.

[15] Brandstadt, A., Le, V. B., & Spinrad, J. P. *Graph classes: a survey* SIAM monographs on discrete mathematics and applications, 1999.

[16] Fulkerson, D. R., & Gross, O. A. *Incidence matrices and interval graphs*, Pacific Journal of Mathematics, 15(3), 835-855, 1965.

[17] M. Habib, R. McConnell, C. Paul, L. Viennot, *Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing*, Theoretical Computer Science 234, 59-84, 2000.

[18] U. I. Gupta, D. T. Lee, and J. Y. T. Leung, *Efficient algorithms for interval graphs and circular-arc graphs*, Networks 12(4), 459-467, 1982.

[19] 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.

[20] Wikipedia, Treewidth, 2023, `https://en.wikipedia.org/wiki/Treewidth`.

[21] Wikipedia, Pathwidth, 2023, `https://en.wikipedia.org/wiki/Pathwidth`.

[22] H. A. Kierstead and W. T. Trotter. An extremal problem in recursive combinatorics. Congressus Numerantium, 33:143-153, 1981.

[23] Wikipedia, Graph coloring, 2023, `https://en.wikipedia.org/wiki/Graph_coloring`.

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

[25] Python Docs, *unittest - Unit testing framework*, 2021, `https://docs.python.org/3/library/unittest.html`.

[26] Harary F., Norman R. Z., Cartwright D., Structural Models: An Introduction to the Theory of Directed Graphs. John Wiley & Sons, 1965

[27] Bertossi A. A., Raskin J., Interval graphs and temporal data models. ACM Transactions on Database Systems (TODS), 27(3), 277-306, 2002

[28] Bertossi A. A., Raskin J., Interval temporal logic for the specification and verification of database-driven systems. Annals of Mathematics and Artificial Intelligence, 50(1-2), 157-195, 2007

[29] Dourado M. C., Cardoso E., Cruz D., An overview of applications of interval graphs. Discrete Applied Mathematics, 159(21), 2514-2524, 2011

[30] Meduna A., Sekanina L., Maximal and minimal interval graphs. Theoretical Computer Science, 498, 3-12, 2013

[31] Brandstadt A., Hammer P. L., Simeone B. Graphs and Order: The Role of Graphs in the Theory of Ordered Sets and Its Applications. Springer Science & Business Media, 1999

[32] Farber M., Interval Orders and Interval Graphs: A Study of Partially Ordered Sets. Wiley-Interscience, 1993