Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Albert Surmacz

Nr albumu: 1160401

Badanie grafów kołowych z językiem Python

Praca magisterska na kierunku Informatyka gier komputerowych

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

Serdecznie dziękuję Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za okazaną życzliwość i zaangażowanie podczas tworzenia tej pracy.

Streszczenie

W ramach niniejszej pracy omówiono zagadnienia związane z grafami kołowymi, które można zdefiniować jako grafy przecięć dla zbioru cięciw okręgu. Wiele problemów NP-trudnych dla grafów ogólnych, w przypadku grafów kołowych może być rozwiązywanych w czasie wielomianowym. Powodem dla zainteresowania grafami kołowymi jest ich zastosowanie w wielu dziedzinach. Przedstawiono różne wykorzystania grafów kołowych, przykładowo w naukach przyrodniczych do wizualizacji drugorzędowej struktury RNA lub w problemach optymalizacyjnych, w planowaniu transportu publicznego, czy składowania kontenerów na statku.

Przedstawiono relację z innymi klasami grafów, przede wszystkim grafów permutacji, które są szczególnym przypadkiem grafów kołowych, co w niektórych algorytmach pozwala wykorzystywać wydajne rozwiązania wykorzystujące właściwości grafów permutacji. Zaprezentowano również graficzny dowód, że nie każdy graf permutacji jest grafem kołowym.

W pracy opisano właściwości grafów kołowych, które stanowiły podstawę do stworzenia wydajnych algorytmów grafowych. Przedstawiono implementację w języku Python wybranych algorytmów dla grafów kołowych. Algorytmy zostały opisane wraz z przebiegiem ich działania i analizą złożoności obliczeniowej. Zaprezentowano po dwa algorytmy wyznaczające liczność największego zbioru niezależnego oraz znajdujące największą klikę. Przestawiono także algorytmy przeszukiwania grafów kołowych w głąb i wszerz.

Ponadto, w pracy przedstawiono algorytm do wykrywania grafu permutacji w reprezentacji permutacyjnej grafu kołowego, uzyskując to w czasie liniowym. Zaprezentowano różne reprezentacje grafów kołowych oraz algorytmy przekształcające pomiędzy nimi. Dodatkowo, zaimplementowano generatory różnych typów grafów kołowych, zarówno grafów losowych, jak i tych spełniających określone kryteria.

Przetestowano poprawność wszystkich implementacji za pomocą testów jednostkowych. Czasowa złożoność algorytmów została zweryfikowana w sposób eksperymentalny przy wykorzystaniu narzędzi dostępnych w języku Python, a wyniki zostały zobrazowane na wykresach.

Słowa kluczowe: grafy kołowe, grafy permutacji, problem kliki, zbiory niezależne, przeszukiwanie grafu, spójność

English title: Study of circle graphs with Python

Abstract

This paper discusses topics related to circle graphs, which can be defined as intersection graphs of a set of chords on a circle. Many NP-hard problems for general graphs can be solved in polynomial time in the case of circle graphs. The motivation behind the interest in circle graphs lies in their diverse applications. Various applications of circle graphs are presented, such as their use in natural sciences for visualizing the secondary structure of RNA, as well as in optimization problems related to public transport planning and container storage on ships.

The relationship with other classes of graphs, primarily permutation graphs, has been presented. Permutation graphs constitute a special case of circle graphs, which enables the utilization of efficient solutions based on the properties of permutation graphs in certain algorithms. Additionally, a graphical proof was presented that not every permutation graph is a circle graph.

The properties of circle graphs are described in the paper, forming the basis for the development of efficient graph algorithms. An implementation of selected algorithms for circle graphs is presented using the Python programming language. The algorithms are described along with their execution steps and computational complexity analysis. Two algorithms for determining the size of the largest independent set and finding the maximum clique are showcased. Depth-first search and breadth-first search algorithms for circle graphs are also presented.

Furthermore, an algorithm for detecting a permutation graph in permutation representation of a circle graph, achieving it in linear time. Different representations of circle graphs and algorithms for transformations among them are presented. Additionally, generators for various types of circle graphs are implemented, covering both random graphs and those that fulfill specific criteria.

The correctness of all implementations is tested through unit tests. The time complexity of the algorithms is experimentally verified using tools available in the Python programming language, and the results are visualized through diagrams.

Keywords: circle graphs, permutation graphs, clique problem, independent sets, graph traversal, connectivity

Spis treści

Spis tabel

Spis rysunków

Listings

1. Wstęp

Tematem pracy jest analiza grafów kołowych, które można zdefiniować jako grafy przecięć dla zbioru cięciw okręgu [1]. Dokładniej, graf kołowy jest to graf nieskierowany, którego wierzchołki odpowiadają cięciwom okręgu, natomiast dwa wierzchołki są połączone krawędzią wtedy i tylko wtedy, gdy odpowiednie cięciwy przecinają się.

Grafy kołowe zostały po raz pierwszy zdefiniowane w 1971 roku w publikacji Evena i Itaia, w której autorzy przeprowadzili badania możliwości realizowania permutacji przy pomocy stosów i kolejek [2]. Udowodnione zostało, że analizowany problem sprowadza się do kolorowania wierzchołków grafu odpowiedniej klasy. Dla równoległych kolejek jest to rodzina grafów permutacji, a w przypadku stosów - grafy permutacji lub grafy kołowe, w zależności od sposobu przenoszenia elementów pomiędzy stosami i kolejkami. Dla grafu ogólnego problem kolorowania jest NP-trudny, a zawężając się do poszczególnych klas możemy uzyskać wielomianowe algorytmy. We wspomnianej pracy zostały również opisane dwa algorytmy: jeden do zapisu danego grafu kołowego jako permutacji i drugi służący do zapisu permutacji jako graf kołowy. Wykorzystanie specyficznych właściwości grafów kołowych daje możliwość rozwiązania problemów z teorii grafów w czasie wielomianowym, podczas gdy te problemy dla grafu ogólnego są NP-trudne lub NP-zupełne. W 1973 roku opisane zostały algorytmy znajdowania największej kliki i największego zbioru niezależnego dla grafów kołowych [3]. Oba zaproponowane algorytmy posiadają złożoność wielomianową. Algorytm do znajdowania największej kliki grafu kołowego polega na utworzeniu i zbadaniu n podgrafów permutacji, które należą do grafów przechodnich (ang. *transitive graphs*). Dla każdego z nich możemy znaleźć największą klikę w czasie $O(n^2)$, gdzie n jest liczbą wierzchołków grafu. Do znajdowania największego zbioru niezależnego, zostało wykorzystane utożsamienie grafu kołowego z grafem nakładania się zbioru przedziałów na prostej (ang. *overlap graph*). Zauważmy, że nie jest to graf przedziałowy (ang. *interval graph*), bo w grafie nakładania przedział zawierający się całkowicie w drugim nie tworzy krawędzi w grafie abstrakcyjnym. Złożoność obliczeniowa tego algorytmu również wynosi $O(n^3)$.

Rozwiązanie problemu rozpoznania grafu kołowego jest możliwe w czasie wielomianowym, a pierwszy taki algorytm został opisany w roku 1985 [4]. Wykorzystuje on dekompozycję grafu i posiada złożoność $O(n^7)$. Inny algorytm rozpoznania grafu, wykorzystujący charakterystykę grafów kołowych w kategoriach lokalnego dopełnienia, został zaproponowany w 1987 roku przez Boucheta [5]. Jego złożoność wynosi $O(n^5)$. Kolejny algorytm wykorzystujący dekompozycję grafu został przedstawiony w roku 1989, a jego złożoność obliczeniowa wynosi $O(n^3)$ [6]. W końcu w roku 1994 Spinrad podał algorytm rozpoznawania grafu kołowego w czasie $O(n^2)$ [7].

W roku 2009 Geelen i Oum podali nową charakteryzację grafów kołowych przez *pivoting* [8]. Autorzy znaleźli komputerowo 15 grafów, które są zabronione w odniesieniu do grafów kołowych. Jest to uogólnienie twierdzenia Kuratowskiego, które opisuje grafy planarne.

Należy zauważyć, że wykorzystanie grafów kołowych nie zawsze daje możliwość szybkiego rozwiązania danego problemu grafowego. Przykładowo w publikacji z roku 1980 autorzy przedstawili dowód, że dla grafu kołowego problem kolorowania wierzchołków jest NP-trudny [9]. Podobnie w roku 1989 Damaschke pokazał, że problem znajdowania cyklu Hamiltona jest NP-zupełny dla grafów kołowych [10]. W roku 1993 Keil pokazał, że problem znajdowania najmniejszego zbioru dominującego jest NP-zupełny [11].

1.1. Cel i zakres pracy

Niniejsza praca stanowi przegląd wybranych zagadnień dotyczących tematu grafów kołowych. Jej głównym celem jest wykorzystanie zebranej teorii w praktyce poprzez implementację i opis algorytmów działających na grafach kołowych.

Algorytmy zostały zaimplementowane w języku Python [12] w ramach rozwoju pakietu *graphtheory* [13]. Implementacja algorytmów pozwala na stworzenie praktycznych narzędzi, umożliwiając użytkownikowi łatwe i efektywne operowanie na grafach kołowych. Rozwój biblioteki rozszerza jej funkcjonalność oraz ułatwia przyszłe badania i prace związane z grafami kołowymi.

Praca skupia się nie tylko na implementacji algorytmów, ale także na ich analizie. Każdy zastosowany algorytm został szczegółowo opisany pod kątem jego danych wejściowych, procedur oraz złożoności obliczeniowej, umożliwiając czytelnikowi zrozumienie ich działania i praktyczne zastosowanie. W celu weryfikacji poprawności algorytmów, przeprowadzone zostały testy jednostkowe. Zbadano również praktyczną wydajność zaimplementowanych algorytmów. Wyniki testów zostały przedstawione na wykresach w dodatku A.

1.2. Struktura pracy

Niniejsza praca została podzielona na pięć rozdziałów oraz jeden dodatek. Rozdział 1 zawiera wprowadzenie przedstawiające cel, zakres i strukturę pracy. Jego zadaniem jest przybliżenie czytelnikowi istoty pracy i tego, czego można się spodziewać w kolejnych rozdziałach. Rozdział 2 przedstawia kluczowe pojęcia związane z grafami, właściwości grafów kołowych oraz sposoby ich reprezentacji. Rozdział 3 skupia się na opisie zastosowanej implementacji grafów, metody reprezentacji grafów kołowych oraz podstawowych algorytmów służących do ich obsługi. Rozdział 4 prezentuje implementacje i opisy algorytmów rozwiązujących problemy z teorii grafów. Algorytmy te mają na celu efektywne rozwiązywanie różnorodnych problemów wykorzystując implementację grafów kołowych. Rozdział 5 stanowi podsumowanie pracy, zawierające wnioski oraz możliwości dalszych badań i rozwoju w tej dziedzinie. W dodatku A znajdują się testy algorytmów, które zostały przeprowadzone

dla zaprezentowanych implementacji. Są to konkretne przykłady wykorzystania algorytmów i prezentacja ich działania w praktyce. Testy te mają na celu potwierdzenie rzeczywistej wydajności algorytmów.

2. Teoria grafów

W tym rozdziale, w celu zapewnienia jednoznaczności, przedstawione zostaną definicje fundamentalnych pojęć niezbędnych do analizy i badania struktur grafowych, które będą wykorzystane w tej pracy. Poniższe definicje opierają się przede wszystkim na opisach zawartych w podręcznikach Cormena [14], Wilsona [15] i Golumbica [16].

2.1. Podstawowe definicje

Definicja: Graf prosty $G = (V, E)$ to uporządkowana para składającą się z niepustego skończonego zbioru wierzchołków $V(G)$ oraz ze skończonego zbioru krawędzi $E(G)$ (par wierzchołków), łączących te wierzchołki [15]. Na ilustracjach graficznych, wierzchołki (węzły) są przedstawiane jako kółka z unikalnymi etykietami, służącymi do identyfikacji poszczególnych wierzchołków, natomiast krawędzie stanowią proste odcinki łączące te kółka. Przyjmujemy następujące oznaczenia: $n = |V(G)|$ to liczba wierzchołków, $m = |E(G)|$ to liczba krawędzi.

Definicja: Graf nieskierowany (ang. *undirected graph)* to graf prosty, w którym zbiór krawędzi jest zbiorem nieuporządkowanych par wierzchołków [14]. Oznacza to, że każda krawędź łączy dwa różne wierzchołki, a zbiory $\{u, v\}$ oraz $\{v, u\}$, gdzie $u, v \in V(G)$ i $u \neq v$ oznaczają tę samą krawędź.

Definicja: Graf nieskierowany jest spójny (ang. *connected*), jeżeli istnieje ścieżka (ciąg wierzchołków i krawędzi) łącząca dowolną parę wierzchołków. Graf niespójny jest złożony z pewnej liczby spójnych składowych (ang. *connected components*), które są osobnymi spójnymi podgrafami [16].

Definicja: Graf pełny K_n (ang. *complete graph*) jest to graf nieskierowany, w którym dla każdej pary wierzchołków istnieje krawędź je łącząca.

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

Definicja: Graf koło W_n (ang. *wheel graph*) jest to graf nieskierowany z n wierzchołkami, który powstaje przez połączenie centralnego wierzchołka z pozostałymi wierzchołkami, tworzącymi graf cykliczny C_{n-1}. Najmniejszy graf koło to graf pełny $K_4 = W_4$.

Definicja: Graf permutacji (ang. *permutation graph*) jest to graf nieskierowany, którego wierzchołki reprezentują elementy permutacji, a krawędzie reprezentują pary elementów tworzących inwersję w permutacji [17].

Definicja: Graf kołowy (ang. *circle graph*) jest to graf nieskierowany, którego wierzchołki odpowiadają cięciwom okręgu, przy czym dwa wierzchołki są połączone krawędzią wtedy i tylko wtedy, gdy odpowiednie cięciwy przecinają się. Bez zmniejszenia ogólności można przyjąć, że cięciwy nie mają wspólnych końców. Warto zauważyć, że czasem w literaturze graf koło W_n też jest nazywany grafem kołowym.

Ciekawym sposobem opisu danej rodziny grafów jest podanie pewnej zabronionej podstruktury, która nie może się pojawić w danej rodzinie. Przykładowo dla grafów planarnych są to graf pełny K_5 i graf pełny dwudzielny $K_{3,3}$ (twierdzenie Kuratowskiego). W przypadku grafów kołowych są to grafy koło W_6, W_8, oraz graf W_7, z którego usunięto trzy niesąsiednie krawędzie wychodzące z wierzchołka centralnego [18].

Definicja: Graf przedziałowy (ang. *interval graph*) to nieskierowany graf utworzony ze zbioru przedziałów na prostej, z wierzchołkiem dla każdego przedziału i krawędzią między wierzchołkami, których przedziały się przecinają [19].

Definicja: Graf nakładania się zbioru przedziałów na prostej (ang. *overlap graph*) to graf, którego wierzchołki można utożsamić z przedziałami na prostej w następujący sposób: dwa wierzchołki są połączone krawędzią, jeśli ich przedziały częściowo zachodzą na siebie (mają niepuste przecięcie), ale żaden z nich nie zawiera drugiego w całości [20].

Definicja: Grafy G_1 i G_2 są izomorficzne, jeśli istnieje jednoznaczne przekształcenie węzłów tych grafów, takie że liczba krawędzi łączących dowolną parę wierzchołków w grafie G_1 jest równa liczbie krawędzi łączących odpowiadające im wierzchołki w grafie G_2 [15].

Definicja: Przeszukiwanie grafu to algorytmiczny proces badania struktury grafu poprzez systematyczne przechodzenie wzdłuż krawędzi, w celu odwiedzenia wszystkich wierzchołków [14]. Istnieją różne metody przeszukiwania grafu, np. przeszukiwanie wszerz BFS (ang. *Breadth-First Search*) i przeszukiwanie w głąb DFS (ang. *Depth-First Search*), które prezentują różne techniki badania grafu, co przekłada się na odmienną kolejność odwiedzania wierzchołków.

Definicja: Graf dwudzielny (ang. *bipartite graph*) to graf, w którym zbiór wierzchołków można podzielić na dwa rozłączne zbiory w taki sposób, że żadna krawędź nie łączy wierzchołków należących do tego samego zbioru.

Definicja: Podział grafu nieskierowanego (ang. *split*) to podział, którego zbiór elementów wyciętych tworzy kompletny graf dwudzielny [21].

Definicja: Dekompozycja grafu (ang. *split decomposition, join decomposition*) to podział grafu przechowywany w strukturze drzewa [21].

Definicja: Graf pierwszy/główny (ang. *prime graph*) to graf, który nie ma podziału w odniesieniu do *split decomposition* [22].

Definicja: Klika (ang. *clique*) jest podzbiorem C zbioru wierzchołków grafu nieskierowanego, w którym każda para wierzchołków z C jest połączona krawędzią. Klikę składającą się z k wierzchołków nazywamy k-kliką. Klikę nazywamy maksymalną (ang. *maximal clique*) w danym grafie, jeśli nie można dodać do niej kolejnego wierzchołka w taki sposób, aby utworzyć większą klikę [16]. Klikę nazywamy największą (ang. *maximum clique*), jeżeli w grafie nie ma innej kliki o większej liczbie wierzchołków.

Definicja: Zbiór niezależny (ang. *independent set*) grafu to podzbiór jego wierzchołków, w którym żadne dwa wierzchołki nie są połączone krawędzią. Zbiór niezależny może być maksymalny (ang. *maximal*), jeśli nie można dodać żadnego innego wierzchołka do tego zbioru [16]. Największy zbiór niezależny (ang. *maximum*) to zbiór niezależny o największej liczności w danym grafie.

Definicja: Graf cięciwowy (ang. *chordal graph*) to graf nieskierowany, w którym każdy cykl o długości większej niż trzy zawiera cięciwę (ang. *chord*), czyli krawędź łączącą dwa niekolejne wierzchołki cyklu.

2.2. Zastosowania grafów kołowych

Powodem szerokiego zainteresowania rodziną grafów kołowych są ich możliwe zastosowania w wielu dziedzinach. W naukach przyrodniczych stosuje się je do wizualizacji drugorzędowej struktury RNA [23]. Rozkładając cząsteczkę RNA i zapisując kolejne nukleotydy na okręgu, można przedstawić cięciwy tego okręgu jako równoważne z wiązaniami występującymi w prezentowanej cząsteczce. Po utworzeniu grafów kołowych, dla wszystkich możliwych wiązań pomiędzy nukleotydami, można zauważyć, że każda drugorzędowa struktura RNA odpowiada zbiorowi niezależnemu tego grafu, ze względu na to, że wiązania nukleotydów nie mogą się krzyżować. Zatem, algorytmy badające proces zwijania cząsteczek RNA [24], [25] właściwie znajdują maksymalny ważony zbiór niezależny w grafie kołowym [26].

Even i Itai przedstawili dowód, że problem kolorowania grafu kołowego jest równoważny problemowi sortowania permutacji przy pomocy równoległych stosów [2]. W praktyce ten problem pojawia się w planowaniu rozkładów publicznego transportu, gdzie istnieje potrzeba efektywnego rozmieszczenia i optymalizacji zasobów. Przykładowo, dotyczy to optymalnego ustalenia miejsc dla tramwajów w zajezdni, aby rano podczas wyjazdu na trasę się wzajemnie nie blokowały [27]. Kolejnym przypadkiem jest wybór toru dla pociągu wjeżdżającego na stację tak, aby nie blokował innego pociągu i mógł opuścić stację na czas [28], [29]. Podobne problemy pojawiają się również na

zajezdniach autobusowych [30]. Problem kolorowania powiązany jest również z problemem składowania kontenerów na statku [31], [32].

Nash i Gregg rozważali grafy kołowe pod względem optymalizacji kompilacji [33]. Autorzy zaproponowali wykorzystanie algorytmów grafów kołowych do alokacji rejestrów podczas kompilacji programowych pętli potokowych. Opisywany przez nich algorytm do znajdowania największego zbioru niezależnego znalazł swoje zastosowanie również w geometrii obliczeniowej [34].

Algorytmy grafów kołowych znalazły swoje zastosowanie także w projektowaniu układów scalonych VLSI (ang. *Very Large Scale Integration*). Sherwani podaje, że różne klasy grafów, a w tym właśnie grafy kołowe są wykorzystywane do reprezentowania układów VLSI [35]. W projektowaniu układów elektronicznych, wierzchołki grafu mogą reprezentować różne komponenty układu, takie jak tranzystory, rezystory czy kondensatory, a krawędzie między nimi będą odzwierciedlać połączenia elektryczne. Dzięki temu w projektowaniu tych układów możliwe jest wykorzystanie wydajnych algorytmów.

Innym obszarem, w którym znane są zastosowania grafów kołowych, jest chemia obliczeniowa [36]. Grafy kołowe mogą być wykorzystywane do modelowania struktur chemicznych i analizy związków chemicznych. W takim kontekście wierzchołki grafu mogą reprezentować atomy, a krawędzie między nimi oznaczają wiązania chemiczne.

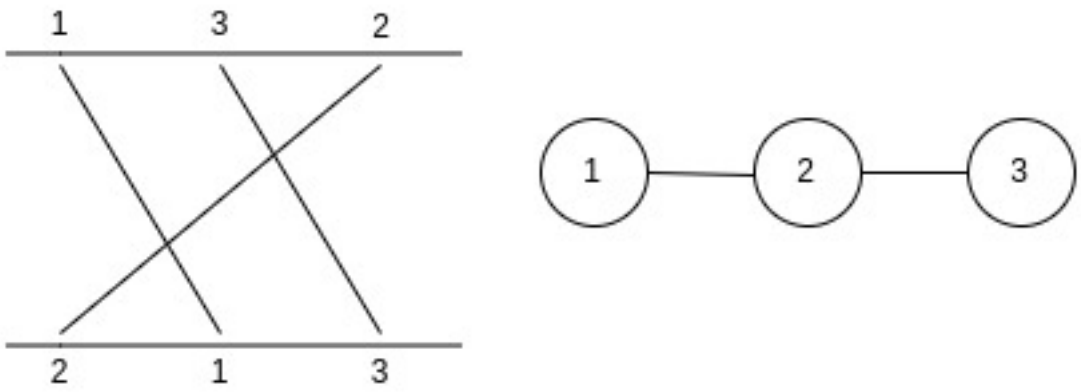
2.3. Relacje z innymi klasami grafów

Szczególnym przypadkiem grafów kołowych są grafy permutacji, które zostały przeze mnie zbadane w pracy licencjackiej [37]. Przedstawiłem w niej następującą analizę: graficzna interpretacja grafów permutacji to ciągi liczb, które reprezentują wierzchołki danego grafu - pierwszy ciąg bazowy oraz drugi będący permutacją tych liczb. Po umieszczeniu ich nad równoległymi prostymi, a następnie połączeniu tych samych liczby nowymi odcinkami, przecinające się odcinki oznaczają krawędź w grafie pomiędzy wierzchołkami, z których wychodziły te krawędzie. Przykład graficznej reprezentacji grafu permutacji znajduje się na rysunku 2.1. Zauważyć można, że ten sam zapis grafu jest możliwy na okręgu, pierwszy ciąg nad górną częścią, a drugi nad dolną, co przedstawia rysunek 2.2. Jest to procedura działająca tylko w jedną stronę, ponieważ nie każdy graf kołowy jest grafem permutacji. Nie zawsze możliwy jest podział okręgu z opisanymi na nim liczbami na dwie części w taki sposób, aby każdy zawierał te same liczby. Najmniejszym grafem kołowym, który nie jest grafem permutacji, jest graf cykliczny C_5, zaprezentowany na rysunku 2.3.

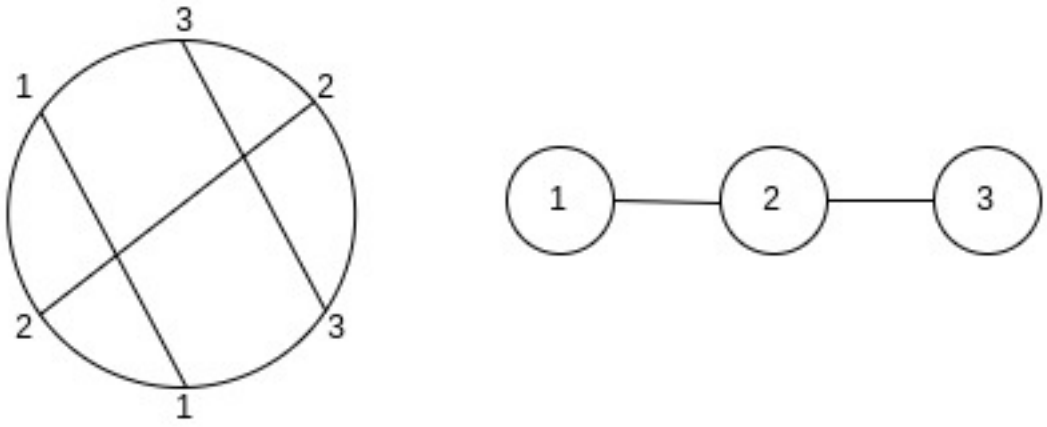
Autorzy Even i Itai przedstawili graficzny dowód na to, że nie każdy graf nieskierowany może być przedstawiony jako graf kołowy [2]. Analogicznie do ich analizy na rysunku 2.4 przedstawiono przykład takiego grafu. Jest to graf koło W_6. Próbując narysować okrąg reprezentujący ten graf otrzymano okrąg cykliczny pokazany na rysunku 2.3. Jest to jedyny sposób na uzyskanie grafu składającego się z tych pięciu wierzchołków. W tym przypadku nie ma możli-

wości dodania szóstego wierzchołka, czyli szóstej cięciwy, która jednocześnie przecinałaby wszystkie pozostałe, co dowodzi, że dla tego grafu nie istnieje reprezentacja w postaci grafu kołowego.

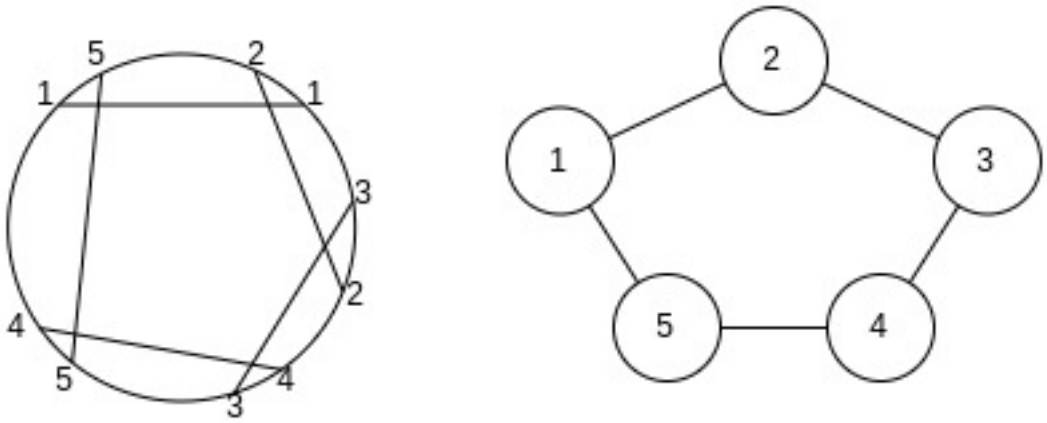
Gavril w swojej pracy zaprezentował, że graf jest grafem kołowym wtedy i tylko wtedy, gdy jest grafem nakładania się zbioru przedziałów na prostej [3]. Grafy te są reprezentowane jako przedziały na prostej, a więc analizując graf kołowy rozważa się styczną do okręgu w dowolnym punkcie, który nie jest punktem początkowym żadnej z cięciw. Następnie wyznaczono punkt znajdujący się po przeciwległej stronie okręgu do punktu styczności. Wyprowadzając z tego punktu półproste przechodzące przez punkty początkowe cięciw zapisano punkty w miejscach przecięcia tych półprostych ze styczną. Każda cięciwa określa w ten sposób dwa punkty na stycznej, które można rozważyć jako przedział na tej prostej. Uzyskany w ten sposób zbiór przedziałów interpretowany jako graf nakładania się zbioru przedziałów na prostej jest tożsamy z grafem kołowym wyznaczonym przez ten okrąg z cięciwami. Graficzne przedstawienie powyższego rozumowania na przykładzie grafu kołowego 2.2 znajduje się na rysunku 2.5.



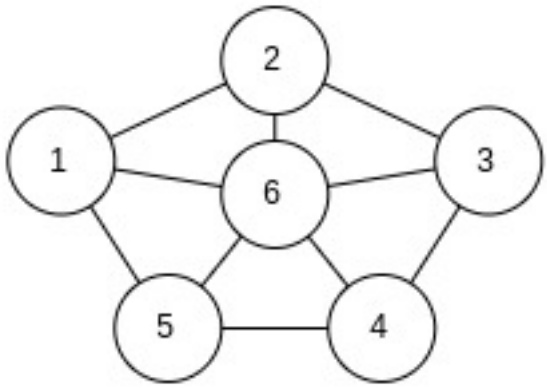
Rysunek 2.1. Reprezentacja graficzna grafu permutacji.



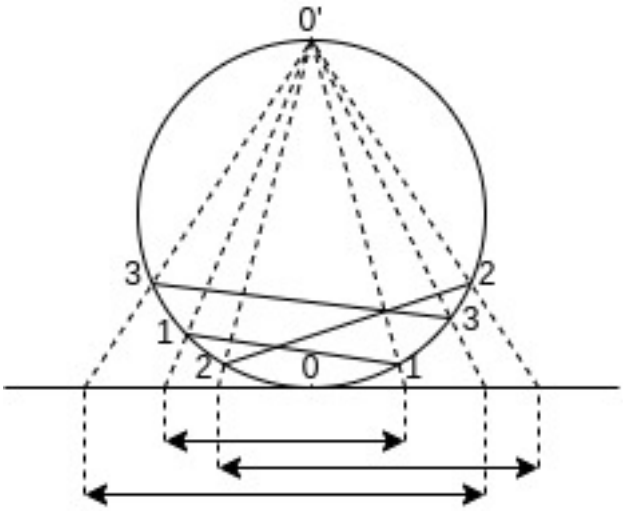
Rysunek 2.2. Reprezentacja graficzna grafu kołowego.



Rysunek 2.3. Graf cykliczny C_5 jako graf kołowy.



Rysunek 2.4. Graf W_6, który nie jest grafem kołowym.



Rysunek 2.5. Graf kołowy jako graf nakładania się zbioru przedziałów na prostej. Przedział 1 zawiera się w całości w przedziale 3, a więc nie ma w grafie krawędzi łączącej wierzchołek 1 z wierzchołkiem 3.

3. Implementacja grafów

Na wstępie warto zauważyć, że grafy kołowe opisuje się używając kilku reprezentacji. Reprezentację dobiera się w zależności od problemu, a przekształcenia między reprezentacjami nie zawsze są proste. Reprezentacje będą bardziej szczegółowo opisane w dalszej części pracy. Pewne reprezentacje nie są wyznaczone jednoznacznie, jeden graf abstrakcyjny może mieć kilka form w danej reprezentacji.

1. Reprezentacja jako graf abstrakcyjny nieskierowany, gdzie mamy jedynie informację o istnieniu krawędzi między wierzchołkami. W kodzie graf będzie instancją klasy Graph.
2. Reprezentacja geometryczna przez skończony zbiór cięciw okręgu. Różne reprezentacje geometryczne mogą prowadzić do tego samego grafu abstrakcyjnego, co zostanie pokazane w przykładowyh obliczeniach.
3. Reprezentacja przez zbiór odcinków na prostej, gdzie odcinki odpowiadają wierzchołkom grafu, a wierzchołki są połączone krawędzią wtedy i tylko wtedy, gdy odpowiednie przedziały nakładają się, ale jeden odcinek nie jest w całości zawarty w drugim. W kontekście tej reprezentacji używa się nazwy *overlap graphs* zamiast *circle graphs*. Tą reprezentację uzyskuje się przez odpowiednie rzutowanie cięciw na prostą.
4. Reprezentacja przez permutację etykiet cięciw okręgu, przy czym każda etykieta cięciwy występuje dwa razy (permutacja z podwójnymi wystąpieniami). Permutację odczytujemy z reprezentacji geometrycznej przez przejście wzdłuż okręgu i zapisanie etykiet końców napotkanych cięciw. Zapis przez permutację nie jest jednoznaczny, możemy zrobić cykliczną rotację etykiet lub odwrócić kolejność etykiet, a mimo to dalej opisujemy ten sam graf kołowy.
5. σ-reprezentacja używająca permutacji σ, wykorzystywana w pracy [33].
6. Reprezentacja przez permutację etykiet cięciw (z primem i bez) i etykiet pewnych sztucznych punktów umieszczonych na okręgu [2].

3.1. Graf kołowy w reprezentacji permutacyjnej

Graf kołowy można przedstawiać jako permutację kolejnych etykiet z podwójnymi wystąpieniami każdej z nich poprzez odczytanie kolejnych etykiet cięciw z modelu kołowego. Przykładowo, dla grafu z rysunku 2.2 taka permutacja to $[1, 3, 2, 3, 1, 2]$. Możemy odczytać te liczby w przeciwnym kierunku otrzymując $[1, 2, 1, 3, 2, 3]$ lub rozpocząć odczytywanie od dowolnego innego miejsca na okręgu, co przykładowo daje nam permutację $[2, 3, 1, 2, 1, 3]$, wciąż wyznaczając ten sam graf. Wykorzystanie takiej postaci grafu może ułatwić wykonanie pewnych algorytmów grafowych. Dla prostoty reprezentację z po-

dwójnymi wystąpieniami liczb będziemy nazywać *reprezentacją permutacyjną*.

Dane wejściowe: Graf kołowy w reprezentacji permutacyjnej.

Problem: Utworzenie grafu abstrakcyjnego z pakietu graphtheory na podstawie permutacji z podwójnymi wystąpieniami.

Opis algorytmu: Algorytm tworzy obiekt grafu abstrakcyjnego z pakietu graphtheory, startując z funkcji create_graph_from_double_occurences_permutation. Wykorzystuje ona funkcję find_overlaped_numbers, która dla każdego wierzchołka zwraca słownik mapujący ten wierzchołek z wyznaczoną przez niego listą. Na kolejnym etapie wykorzystywana jest funkcja count_occurences, która dla takiej listy oblicza wystąpienia każdego z wierzchołków, po czym wyszukiwane są te wierzchołki, które mają dokładnie jedno wystąpienie na takiej liście. Oznacza to, że należy dodać krawędź w tworzonym grafie.

Złożoność: Złożoność algorytmu wynosi $O(n^2)$, gdzie n to liczba wierzchołków grafu, co zostało potwierdzone w dodatku A.

Uwagi: Implementacja algorytmu znajduje się na listingu 3.1.

Listing 3.1. Moduł create_graph_from_permutation.

```
from graphtheory.structures.graphs import Graph
from graphtheory.structures.edges import Edge


def find_overlaped_numbers(double_perm):
    '''
    :param double_perm: Double occurences permutation
    :return: A dictionary mapping each vertex with
        a sub-list contained between the ends of
        the interval that defines this vertex
    '''
    nodes = set(double_perm)
    result_lists = {n: [] for n in nodes}

    for i in range(len(double_perm)):
        for j in range(i + 1, len(double_perm)):
            if double_perm[j] == double_perm[i]:
                # add sub-list
                result_lists[double_perm[i]].append(
                    double_perm[i + 1:j])
                break
    return result_lists


def count_occurences(lst):
    '''
    :param lst: List of nodes
    :return: The number of occurrences of each vertex on the list l
    '''
    counter = {}
```

```
    for node in lst:
        if node in counter:
            counter[node] += 1
        else:
            counter[node] = 1
    return counter


def create_graph_from_double_occurences_permutation(double_perm):
    '''
    :param double_perm: Graph as a double occurences permutation
    :return: Graph as object from graphtheory package
    '''
    nodes = set(double_perm)
    graph = Graph(len(nodes))
    for node in nodes:
        graph.add_node(node)

    lists = find_overlaped_numbers(double_perm)
    for i, sublist in lists.items():
        count = count_occurences(sublist[0])
        for j, c in count.items():
            if c == 1:
                edge = Edge(i, j)
                if not graph.has_edge(edge):
                    graph.add_edge(edge)
    return graph
```

3.2. Utworzenie σ-reprezentacji grafu kołowego

Kolejnym ze sposobów reprezentacji grafu kołowego jest σ-reprezentacja, która została opisana w publikacji Nasha i Gregga [33]. Ten sposób przedstawienia grafu bazuje na właściwości, że grafy kołowe są tożsame z przedziałami na prostej. Permutację σ liczb $\{1, \ldots, 2n\}$ definiujemy przez pary oznaczające zakończenia przedziałów $(\sigma_{2k-1}, \sigma_{2k})$, dla $1 \leqslant k \leqslant n$, gdzie n to liczba wierzchołków grafu. Przykładowo, kolejne punkty końcowe przedziałów grafu z rysunku 2.5 jako permutacja z podwójnymi wystąpieniami to $[3, 1, 2, 1, 3, 2]$. Aby zapisać ten sam graf jako σ-reprezentację, indeksujemy kolejne końce przedziałów (jak zdarzenia na osi liczbowej) zaczynając od 1, a następnie zgodnie z definicją zapisujemy je w kolejności $[1, 5, 2, 4, 3, 6]$. Para 1 i 5 stanowi indeksy lewego i prawego końca pierwszego przedziału, 2 i 4 wyznaczają parę zakończeń kolejnego przedziału, a 3 i 6 ostatniego.

Dane wejściowe: Graf kołowy w reprezentacji permutacyjnej.

Problem: Przekształcenie reprezentacji permutacyjnej w σ-reprezentację.

Opis algorytmu: Algorytm rozpoczynamy od utworzeniu słowników, które będą przechowywać informacje o indeksach pierwszych i drugich wystąpień poszczególnych liczb w podwójnej permutacji. Wypełniamy je, przechodząc przez każdy element podwójnej permutacji i sprawdzając czy jest to pierwsze,

czy już drugie wystąpienie tej wartości, co decyduje o wyborze słownika, do którego zostanie on dodany. Ostatnim krokiem algorytmu jest iteracja po kluczach słownika z pierwszymi wystąpieniami, dodając pary zakończeń kolejnych przedziałów do tworzonej σ-reprezentacji.

Złożoność: Wyniki badań wydajności algorytmu wskazują na złożoność liniową $O(n)$ i znajdują się w dodatku A.

Uwagi: Implementacja algorytmu znajduje się na listingu 3.2.

Listing 3.2. Moduł sigma_representation.

```
def get_sigma_representation(double_perm):
    '''
    :param double_perm: Circle graph as
        a double occurences permutation
    :return: Sigma representation of Circle graph
    '''
    occurrences = {}
    for i, number in enumerate(double_perm, start=1):
        if number in occurrences:
            occurrences[number].append(i)
        else:
            occurrences[number] = [i]
    sigma = []
    for key in occurrences:
        sigma.extend(occurrences[key])
    return sigma
```

3.3. Generowanie przypadkowych grafów kołowych

Aby przeprowadzać badania na grafach kołowych, używamy funkcji, które umożliwiają generowanie ich w sposób losowy. Są one przydatne przy badaniu wydajności algorytmów grafowych. Pierwszy algorytm polega na utworzeniu permutacji z podwójnymi wystąpieniami, a następnie wymieszaniu jej. W drugim algorytmie chcemy uzyskać graf spójny, więc mieszamy tę permutację aż do momentu uzyskania spójnego grafu. Implementacja tych funkcji znajduje się na listingu 3.3.

Listing 3.3. Moduł generate_random_circle_graph.

```
from random import shuffle
from is_connected import is_connected
from create_graph_from_permutation import *


def generate_random_connected_circle_graph(n):
    '''
    :param n: num of vertices
    :return: Connected circle graph
    '''
    doubled_perm = list(range(n)) * 2
    while not is_connected(doubled_perm):
```

```
        shuffle(doubled_perm)
    return create_graph_from_double_occurences_permutation(doubled_perm)


def generate_random_circle_graph(n):
    '''
    :param n: num of vertices
    :return:  Random circle graph
    '''
    doubled_perm = list(range(n)) * 2
    shuffle(doubled_perm)
    return create_graph_from_double_occurences_permutation(doubled_perm)
```

3.4. Generowanie nieprzypadkowych grafów kołowych

W badaniach grafów kołowych, oprócz rozważenia przypadkowych grafów, warto również zbadać te, które mają jakąś szczególną własność. Przykładem może być graf liniowy (ang. *path graph*), czyli taki, który składa się z wierzchołków kolejno połączonych krawędziami, tworząc pojedynczą linię. Kolejnym przykładem są grafy cykliczne (ang. *cycle graph*) czyli graf liniowy składający się z co najmniej trzech wierzchołków, gdzie dodatkowo pierwszy wierzchołek jest sąsiadem ostatniego wierzchołka. Poniżej znajdują się algorytmy do generowania wspomnianych grafów.

Listing 3.4. Moduł generate_graphs.

```
import random


def make_isolated_vetrices(n):
    perm = []
    for i in range(n):
        perm.extend([i, i])
    return perm


def make_path_perm(n):
    '''
    :param n: Size
    :return: Double occur perm that represents path graph of size n
    '''
    perm = make_isolated_vetrices(n)
    for i in range(1, 2 * n - 1, 2):
        perm[i], perm[i + 1] = perm[i + 1], perm[i]
    return perm


def make_cycle_perm(n):
    '''
    :param n: Size
    :return: Double occur perm that represents cycle graph of size n
    '''
    if n > 2:
        perm = make_path_perm(n)
```

```
        perm[0], perm[-1] = perm[-1], perm[0]
        return perm
    else:
        raise ValueError(
            "cycle graph must have at least 3 vertices")


def make_teepe_perm(n):
    '''
    :param n: Size
    :return: Double occur perm that represents tepee graph of size n
    '''
    if n < 3:
        raise ValueError(
            "Teepe graph must have at least 3 vertices.")
    a = 0  # top
    l = 1  # left base
    r = n - 1  # right base
    path = list(range(l, r + 1))
    p_reversed = path[::-1]
    for i in range(0, len(path) - 1, 2):
        path[i], path[i + 1] = path[i + 1], path[i]
    even_length = False
    if len(p_reversed) % 2 == 0:
        p_reversed.remove(r)
        even_length = True
    for i in range(0, len(p_reversed) - 1, 2):
        p_reversed[i], p_reversed[i + 1] = p_reversed[i + 1], p_reversed[i]
    if even_length:
        C = [a] + path + [a] + [r] + p_reversed
    else:
        C = [a] + path + [a] + p_reversed
    return C


def generate_circle_graph_which_is_a_permutation_graph(n):
    '''
    :param n: Size
    :return: Double occur perm that represents permutation graph of size n
    '''
    permutation = list(range(n))
    random.shuffle(permutation)
    reversed_perm = permutation[::-1]
    doubled_perm = permutation + reversed_perm
    return doubled_perm
```

3.5. Sprawdzenie czy dany graf kołowy jest grafem permutacji

Posiadając informację, że dany graf kołowy jest grafem permutacji, można zastosować algorytmy specyficzne dla tej rodziny grafów. Wiemy, że każdy graf permutacji jest grafem kołowym, jednak nie każdy graf kołowy jest grafem permutacji. Zaimplementowany algorytm sprawdza czy dany graf kołowy w reprezentacji permutacyjnej, można przełożyć bezpośrednio na permutację

wyznaczającą graf permutacji. Jeśli rzeczywiście tak jest to badany graf jest grafem permutacji i algorytm zwraca permutację oraz odpowiednie słowniki mapujące wierzchołki, co może przyczynić się do szybszego rozwiązania problemów grafowych. Należy pamiętać, że gdy ten algorytm nie wykryje grafu permutacji w danej reprezentacji to nie możemy stwierdzić, że dany graf abstrakcyjny nie może być grafem permutacji. Implementacja algorytmu znajduje się na listingu 3.5.

Dane wejściowe: Graf kołowy w reprezentacji permutacyjnej.

Problem: Sprawdzenie czy dany graf kołowy jest grafem permutacji.

Opis algorytmu: Algorytm polega na obserwacji, że gdy istnieje możliwość rozcięcia okręgu z opisanymi końcami cięciw na dwie części w taki sposób, że w każdej z obu części będą dokładnie te same etykiety, to mamy do czynienia z grafem permutacji. Algorytm polega na utworzeniu okienka o rozmiarze n, które przesuwa się po permutacji reprezentującej graf kołowy sprawdzając, czy w okienku znalazło się dokładnie n etykiet, co oznacza, że można rozdzielić tę permutację na dwie części. Jeżeli zostanie znalezione okno z liczbą etykiet równą n, to mamy do czynienia z grafem permutacji, a wtedy należy znaleźć jego reprezentację w postaci permutacji liczb od 0 do $n-1$. Ciąg etykiet w znalezionym oknie wyznacza mapowanie od etykiet do liczb od 0 do $n-1$ (i mapowanie odwrotne). Ciąg etykiet poza oknem wyznacza szukaną permutację, którą należy przeczytać od końca.

Złożoność: Złożoność obliczeniowa algorytmu wynosi $O(n)$.

Uwagi: Zauważmy, że etykiety w reprezentacji permutacyjnej grafu kołowego mogą być pewnymi liczbami lub stringami, natomiast permutacja reprezentująca graf permutacji zawiera liczby od 0 do $n-1$. Implementacja obok permutacji zwraca dwa słowniki opisując mapowanie etykiet cięciw na liczby od 0 do $n-1$.

Listing 3.5. Moduł detect_permutation_graph.

```
def is_permutation_graph(double_perm):
    '''
    Algorithm to detect a permutation graph from given circle graph.
    :param double_perm: Circle graph as a doubled permutation
    :return: If it can detect a permutation graph from given
    representationreturns the permutation and labels mappings otherwise
    raises ValueError("permutation graph not detected")
    '''
    window = set()
    start_idx = -1
    n = len(double_perm) // 2
    for i in range(n):  # make initial window
        node = double_perm[i]
        if node in window:
            window.remove(node)
        else:
            window.add(node)
```

```
    if len(window) == n:
        start_idx = 0
    else:
        for i in range(n):
            node = double_perm[i]
            if node in window:
                window.remove(node)
            else:
                window.add(node)
            node = double_perm[i + n]
            if node in window:
                window.remove(node)
            else:
                window.add(node)
            if len(window) == n:
                start_idx = i + 1
                break
    if start_idx == -1:
        raise ValueError("permutation graph not detected")

    label2number = dict()
    number2label = dict()
    for i in range(n):
        label2number[double_perm[start_idx + i]] = i
        number2label[i] = double_perm[start_idx + i]

    perm = [label2number[double_perm[
        (start_idx + n + i) % (2 * n)]] for i in range(n)]
    perm.reverse()
    return perm, number2label, label2number
```

3.6. Przykładowa sesja interaktywna

Oto przykładowa sesja interaktywna, która demonstruje sposoby wykorzystania zaimplementowanych algorytmów:

```
>>> from generate_graphs import make_teepe_perm
>>> from max_clique_from_doubled_perm import max_clique_from_double_perm
>>> from circlebfs import CircleBFS
>>> from circledfs import CircleDFS
>>> from sigma_representation import get_sigma_representation
>>> from sensitive_maximum_independent_set import sensitiveMIS
>>> from detect_permutation_graph import is_permutation_graph
>>> from create_graph_from_permutation import *
>>> from max_clique import max_clique

>>> double_perm = make_teepe_perm(5)
>>> print("Tepee graph as a double permutation =", double_perm)
Tepee graph as a double permutation = [0, 2, 1, 4, 3, 0, 4, 2, 3, 1]

>>> maximum_clique = max_clique_from_double_perm(double_perm)
>>> print("Maximum clique =", maximum_clique)
Maximum clique = {0, 1, 2}
>>> print("Maximum clique size =", len(maximum_clique))
Maximum clique size = 3
```

```
>>> bfs_pre_order = []
>>> bfs_post_order = []
>>> BFS = CircleBFS(double_perm)
>>> BFS.run(0, pre_action=lambda node: bfs_pre_order.append(node),
>>>     post_action=lambda node: bfs_post_order.append(node))

>>> print("bfs_pre_order =", bfs_pre_order)
bfs_pre_order = [0, 1, 2, 3, 4]
>>> print("bfs_post_order =", bfs_post_order)
bfs_post_order = [0, 1, 2, 3, 4]

>>> dfs_pre_order = []
>>> dfs_post_order = []
>>> DFS = CircleDFS(double_perm)
>>> DFS.run(0, pre_action=lambda node: dfs_pre_order.append(node),
>>>     post_action=lambda node: dfs_post_order.append(node))

>>> print("dfs_pre_order =", dfs_pre_order)
dfs_pre_order = [0, 1, 2, 3, 4]
>>> print("dfs_post_order =", dfs_post_order)
dfs_post_order = [4, 3, 2, 1, 0]

>>> sigma = get_sigma_representation(double_perm)
>>> iset = sensitiveMIS(sigma)
>>> print("Sigma representation =", sigma)
Sigma representation = [1, 6, 2, 8, 3, 10, 4, 7, 5, 9]
>>> print("Maximum independent set size =", iset)
Maximum independent set size = 2

>>> perm, number2label, label2number = is_permutation_graph(double_perm)
>>> print("permutation =", perm)
permutation = [2, 4, 1, 3, 0]
>>> print("number2label =", number2label)
number2label = {0: 0, 1: 2, 2: 1, 3: 4, 4: 3}
>>> print("label2number =", label2number)
label2number = {0: 0, 2: 1, 1: 2, 4: 3, 3: 4}

>>> graph = create_graph_from_double_occurences_permutation(double_perm)
>>> graph.show()
0 : 2 1 4 3
1 : 0 2
2 : 0 1 3
3 : 0 2 4
4 : 0 3
>>> clique = max_clique(graph)
>>> print("Max clique =", clique)
Max clique = {0, 1, 3}
```

3.7. Obliczenia dla grafów kołowych

Dobrym zwyczajem jest, aby przy formułowaniu twierdzeń dotyczących danej rodziny grafów zacząć badania od grafów o małej liczbie wierzchołków, aby lepiej zrozumieć tę rodzinę i sprawdzić, czy dane twierdzenie jest spełnione dla jej wybranych przedstawicieli.

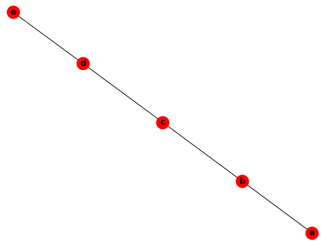
Przykładowo na listingu 3.6 przedstawiamy kod, w którym realizujemy badania grafów kołowych dla ustalonego $n = 5$. Stawiamy hipotezę, że dany graf abstrakcyjny może mieć co najmniej dwie różne reprezentacje w postaci okręgu z cięciwami, gdzie jedna z nich pozwoli na rozcięcie tworzące graf permutacji, a druga nie. Wskazana implementacja przyjmuje jako etykiety wierzchołków litery alfabetu. Następnie wykorzystuje narzędzia dostępne w itertools do wygenerowania z tych etykiet możliwych kombinacji, a w kolejnych krokach odfiltrowuje grafy spójne, oraz te, dla których jest możliwe rozcięcie wyznaczające graf permutacji. Z pozostałych kombinacji etykiet budujemy instancję grafu z biblioteki networkx, dzięki której w następnym kroku generujemy z nich obrazki. W wyniku obliczeń uzyskano 768 grafów o pięciu wierzchołkach, które są grafami kołowymi, ale ich reprezentacja jako cięciwy na okręgu nie pozwala na rozcięcie tworzące graf permutacji. Po przeanalizowaniu uzyskanych rysunków dowiadujemy się, że pod względem izomorfizmu są to tylko 4 grafy, dla których przykłady znajdują się na obrazkach 3.1, 3.4, 3.7 i 3.10.

Kolejno na obrazkach 3.2, 3.5, 3.8, 3.11 znajdują się odpowiadające im reprezentacje jako cięciwy na okręgu. Natomiast obrazki 3.3, 3.6 oraz 3.9 zawierają reprezentacje grafu izomorficznego do przedstawionych, dla których mamy możliwość rozcięcia i uzyskania grafu permutacji. Dla grafu 3.10 nie ma możliwości utworzenia takiej reprezentacji, o czym możemy się przekonać zliczając uzyskane grafy izomorficzne lub po zmianach w kodzie, które wygenerują grafy, które jesteśmy w stanie rozciąć a następnie weryfikując je, że nie znajdują się w nich żadne grafy izomorficzne.

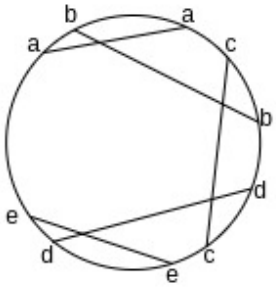
Listing 3.6. Moduł graph_tests.

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

import itertools
import networkx as nx
import matplotlib.pyplot as plt
from is_connected import is_connected
from create_graph_from_permutation import \
    create_graph_from_double_occurences_permutation
from detect_permutation_graph import is_permutation_graph

n = 5
letters = "abcdefghijklmnopqrstuvwxyz"
word = [letters[i // 2] for i in range(2 * n)]
last = word.pop()
perm_set = set()  # double perms
counter = 0

for perm in itertools.permutations(word):
    perm = perm + (last,)
    if perm in perm_set:
        continue
    else:
        perm_set.add(perm)
    if not is_connected(perm):
        continue
    try:
```

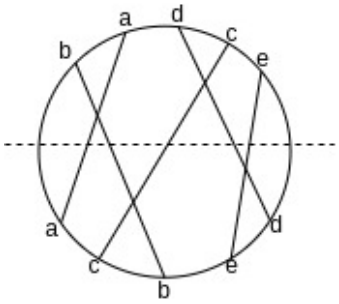
```
            if is_permutation_graph(perm):
                continue
        except:
            pass
        graph = create_graph_from_double_occurences_permutation(perm)
        print(perm)
        counter += 1

        filename = "".join(perm) + ".png"
        print(filename)
        G = nx.Graph()    # an empty undirected graph
        for node in graph.iternodes():
            G.add_node(node)
        for edge in graph.iteredges():
            G.add_edge(edge.source, edge.target)

        nx.draw(G, with_labels=True, font_weight='bold')
        plt.savefig(filename)
        plt.close(plt.gcf())

print("counter {}".format(counter))
```



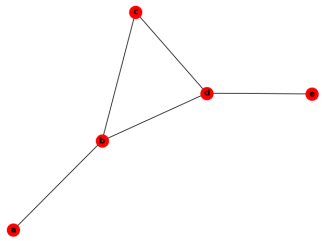
Rysunek 3.1. Graf P_5 (*abacbdcede*).



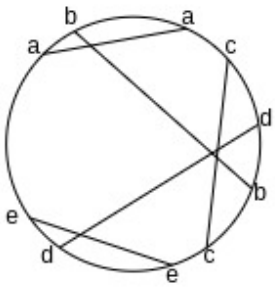
Rysunek 3.2. Graf P_5 (*abacbdcede*) na okręgu.



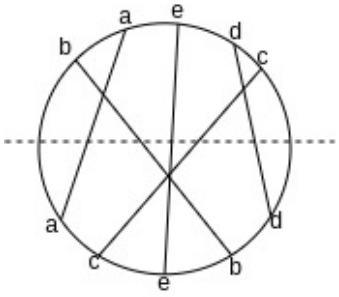
Rysunek 3.3. $(1, 3, 0, 4, 2)$ Izomorficzny graf permutacji P_5.



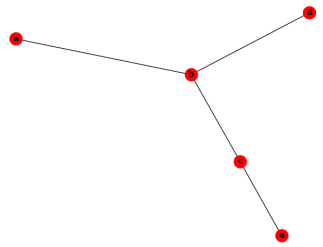
Rysunek 3.4. Graf *bull* (*abacdbcede*).



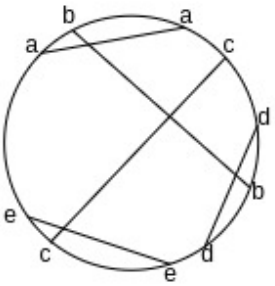
Rysunek 3.5. Graf *bull* (*abacdbcede*) na okręgu.



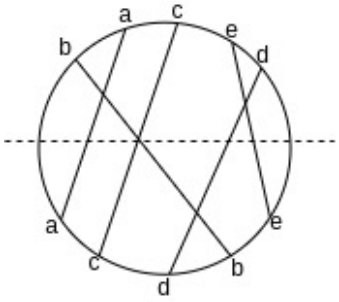
Rysunek 3.6. $(1, 4, 2, 0, 3)$ Izomorficzny graf permutacji *bull*.



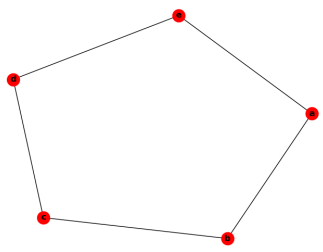
Rysunek 3.7. Drzewo (*abacdbdece*).



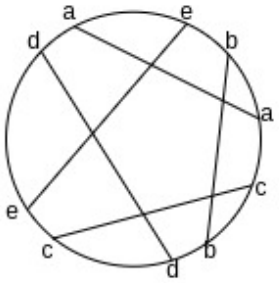
Rysunek 3.8. Drzewo (*abacdbdece*) na okręgu.



Rysunek 3.9. $(1, 2, 4, 0, 3)$ Izomorficzny do drzewa graf permutacji.



Rysunek 3.10. Graf C_5 (*daebacbdce*).



Rysunek 3.11. Graf C_5 (*daebacbdce*) na okręgu.

3.8. Obliczenia dla grafów permutacji

Chcemy pokazać obliczenia komputerowe w obszarze na styku grafów permutacji i grafów cięciwowych. Obie rodziny grafów mają unikalne własności, więc ich część wspólna może przynieść ciekawe odkrycia. Zaczniemy od wstępu teoretycznego.

Nie wszystkie grafy permutacji są cięciwowe, a najprostszy przykład to graf cykliczny C_4 opisany permutacją $(3, 4, 1, 2)$. Zauważmy, że łatwo możemy utworzyć dopełnienie cięciwowe przez dodanie krawędzi 1-2 [powstanie permutacja $(3, 4, 2, 1)$] lub krawędzi 3-4 [powstanie permutacja $(4, 3, 1, 2)$]. Powstały graf diament jest nie tylko grafem cięciwowym, ale dodatkowo grafem przedziałowym.

W grafach cięciwowych liczba klik maksymalnych jest rzędu $O(n)$, a kliki maksymalne wykorzystuje się do budowy drzewa dekompozycji. W grafach permutacji liczba klik maksymalnych może rosnąć wykładniczo z n, o czym świadczy przykład zaczerpnięty z serwisu *mathoverflow.net* [38]. W grafie G zbudowanym z k kopii grafu K_2 istnieje 2^k maksymalnych zbiorów niezależnych, a odpowiednia permutacja dla grafu ma postać $(2, 1, 4, 3, ..., 2k, 2k - 1)$, liczba wierzchołków $n = 2k$. Dopełnienie grafu G to graf zawierający 2^k klik maksymalnych, odpowiednia permutacja to permutacja odwrócona $(2k - 1, 2k, ..., 3, 4, 1, 2)$.

W grafach permutacji nie mogą istnieć indukowane grafy C_5 i większe, więc jedyne cykle indukowane bez cięciw to grafy $C_4 = K_{2,2}$. Daje to możliwość wykrycia cięciwowych grafów permutacji w czasie $O(n^4)$ przez wykluczenie występowania cykli C_4. Algorytm powinien sprawdzać wszystkie ciągi indeksów $(1 \leqslant i < j < k < r \leqslant n)$, aby wykryć uporządkowanie $(p[k] < p[r] < p[i] < p[j])$ tworzące cykl C_4.

Skrypt na listingu 3.7 generuje wszystkie permutacje n liczb i filtruje odpowiednie grafy spójne. Następnie tworzone są grafy abstrakcyjne, dla których możemy wykrywać grafy cięciwowe i ewentualnie grafy przedziałowe. W skrypcie filtrowane są grafy, które nie są cięciwowe, a następnie tworzone są odpowiednie obrazki. Dodatkowo na konsoli wypisywane są znalezione indukowane cykle C_4.

Listing 3.7. Moduł graph_tests2.

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

import itertools
import networkx as nx
import matplotlib.pyplot as plt
from graphtheory.permutations.permtools import perm_is_connected
from graphtheory.permutations.permtools import make_abstract_perm_graph
from graphtheory.chordality.peotools import find_peo_lex_bfs
from graphtheory.chordality.peotools import is_peo1, is_peo2

n = 5
counter = 0
for perm in itertools.permutations(range(n)):
    if not perm_is_connected(perm):    # O(n) time
        continue
    graph = make_abstract_perm_graph(perm)
```

```
        order = find_peo_lex_bfs(graph)
        if is_peo1(graph, order):    # pomijamy chordal graphs
            continue
        print(perm)
        # Szukam 4 liczb tworzacych indukowany graf C_4.
        for (i,j,k,r) in itertools.combinations(range(n),4):
            if perm[k] < perm[r] < perm[i] < perm[j]:
                print("idx",i,j,k,r,"perm",perm[i],perm[j],perm[k],perm[r])
        counter += 1

        filename = "".join(map(str,perm)) + ".png"
        print(filename)
        G = nx.Graph()    # an empty undirected graph
        for node in graph.iternodes():
            G.add_node(node)
        for edge in graph.iteredges():
            G.add_edge(edge.source, edge.target)
        nx.draw(G, with_labels=True, font_weight='bold')
        plt.savefig(filename)
        plt.close(plt.gcf())    # reset rysunku

print("counter {}".format(counter))
```

W tabeli 3.1 zestawiono porównanie, które obrazuje liczbę grafów różnych klas o n wierzchołkach. Tabela została utworzona na podstawie danych zebranych przy pomocy zaimplementowanych funkcji oraz częściowo przy wykorzystaniu zbiorów z internetowej encyklopedii [39].

Tabela 3.1. Porównanie liczby grafów różnych klas o n wierzchołkach.

n	n!	graf prosty	graf spójny	spójny graf permutacji	permutacja wyznaczająca graf spójny	permutacja dająca graf, który nie jest cięciwowy
1	1	1	1	1	1	0
2	2	2	1	1	1	0
3	6	4	2	2	3	0
4	24	11	6	6	13	1 (C_4)
5	120	34	21	20 (brak C_5)	71	15 (5 różnych)
6	720	156	112		461	172
7	5040	1044	853		3447	1844
8	40320	12346	11117		29093	19702
9	362880	274668	261080		273343	215906
10	3628800	12005168	11716571		2829325	2465552

W zaprezentowanej tabeli możemy dokonać ciekawych obserwacji. Przykładowo dla $n = 5$ mamy 21 spójnych grafów prostych, przy czym spójnych grafów permutacji jest 20. Jednym wyróżniającym się grafem jest graf cykliczny C_5, który nie jest grafem permutacji.

Rozważmy grafy permutacji, które nie są grafami cięciwowymi. Możemy zauważyć, że najmniejszy z nich to graf C_4, a przyjmując $n = 5$ możemy wyznaczyć już 15 grafów permutacji. Wśród nich znajduje się 5 istotnie różnych grafów, które zostały przedstawione na rysunkach 3.12, 3.13, 3.14, 3.15 i 3.16. Każdy z tych grafów ma indukowany cykl C_4. Co ciekawe, w każdym z nich wystarczy tylko jedna nowa krawędź do wykonania triangulacji, czyli utworzeniu grafu cięciwowego. W przypadku grafu $(1, 3, 4, 0, 2)$ 3.12 może to być krawędź 3-4 lub 0-2. W grafie $(2, 3, 4, 0, 1)$ 3.13 jest to krawędź 0-1. Dla grafu $(2, 4, 0, 3, 1)$ 3.14, również mamy dwie możliwości, krawędź 2-4 lub 0-1.

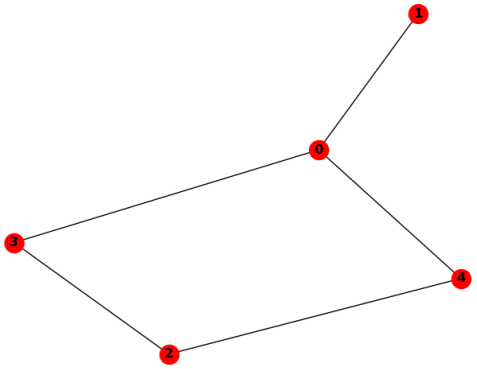
Graf $(3,4,0,2,1)$ 3.15 potrzebuje krawędzi 0-2, natomiast graf $(3,4,1,2,0)$ 3.16 krawędzi 1-2 lub 3-4.

Rozważmy grafy permutacji w powiązaniu z grafami cięciwowymi. Sformułowaliśmy trzy problemy, na które nie udało się znaleźć odpowiedzi w literaturze. Dwa pierwsze problemy mogą być rozwiązane przez podanie przykładów odpowiednich grafów, ale nie ma gwarancji, że takie grafy istnieją.

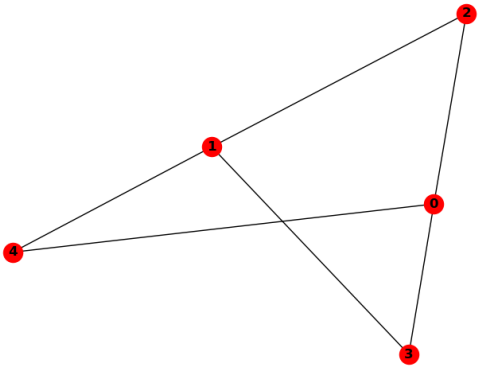
Problem 1. Czy każdy cięciwowy graf permutacji jest grafem przedziałowym? Jeżeli odpowiedź jest NIE, to jaki jest najmniejszy graf? Dla $n = 4$ i $n = 5$ wszystkie grafy cięciwowe są przedziałowe.

Problem 2. Czy istnieją grafy przedziałowe, które nie są grafami permutacji? Czy istnieją grafy cięciwowe, które nie są grafami permutacji? Jeżeli odpowiedź jest TAK, to podać przykłady.

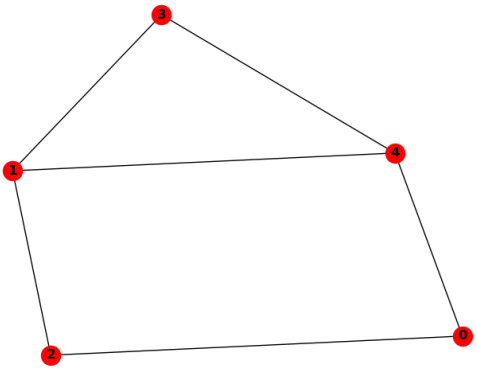
Problem 3. Optymalne uzupełnienie cięciwowe grafu permutacji jest grafem przedziałowym [40]. Czy gorsze niż optymalne uzupełnienie cięciwowe grafu permutacji może nie być grafem przedziałowym (jest związek z Problemem 1)? Czy optymalne uzupełnienie cięciwowe grafu permutacji zawsze jest grafem permutacji (jest związek z Problemem 2)?



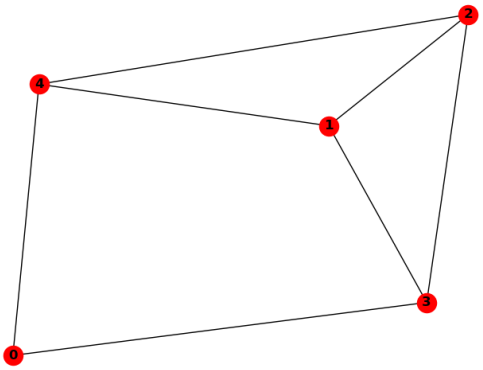
Rysunek 3.12. Graf permutacji $(1, 3, 4, 0, 2)$



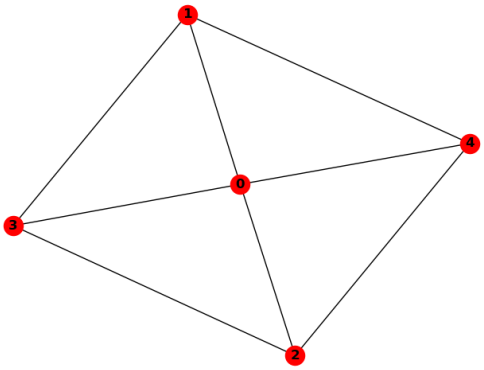
Rysunek 3.13. Graf permutacji $(2, 3, 4, 0, 1)$



Rysunek 3.14. Graf permutacji $(2, 4, 0, 3, 1)$



Rysunek 3.15. Graf permutacji $(3, 4, 0, 2, 1)$



Rysunek 3.16. Graf permutacji $(3, 4, 1, 2, 0)$

4. Algorytmy

W tej części omówione zostaną problemy z teorii grafów, które wzbudzają zainteresowanie ze względu na możliwość ich rozwiązania przy zastosowaniu reprezentacji grafu w postaci grafów kołowych. Przedstawione problemy to: badanie spójności grafu kołowego, znajdowanie maksymalnej kliki i maksymalnego zbioru niezależnego. Wykorzystanie ich właściwości pozwala na rozwiązanie tych problemów w sposób bardziej efektywny pod względem obliczeniowym.

4.1. Badanie spójności grafu algorytmem BFS

Spójność grafu to właściwość, która oznacza że istnieje w nim ścieżka między dowolnymi dwoma wierzchołkami. Dla grafu ogólnego problem ten rozwiązuje się w czasie liniowym $O(n+m)$ wykorzystując algorytm przeszukiwania grafu wszerz BFS (ang. *Breadth-First Search*) lub przeszukiwania grafu w głąb DFS (ang. *Depth-First Search*).

Dane wejściowe: Graf kołowy w reprezentacji permutacyjnej.

Problem: Badanie spójności grafu kołowego z wykorzystaniem BFS.

Opis algorytmu: Algorytm wykonuje przeszukiwanie BFS rozpoczynając od pierwszego wierzchołka, następnie sprawdza, czy liczba kolejno odwiedzonych wierzchołków jest równa ich liczbie w całym grafie. Jeżeli odwiedzono wszystkie wierzchołki, to graf jest spójny.

Złożoność: Złożoność czasowa algorytmu została zbadana w dodatku A i wynosi $O(n^2)$.

Uwagi: Listing 4.1 zawiera kod sprawdzający spójność grafu. W jego implementacji wykorzystano algorytm przeszukiwania grafu wszerz, którego opis znajduje się w sekcji 4.2.

Listing 4.1. Moduł is_connected.

```
from circlebfs import CircleBFS


def is_connected(double_perm):
    """Testing connectivity of the circle graph in O(n^2) time."""
    order = []
    algorithm = CircleBFS(double_perm)
    algorithm.run(double_perm[0], pre_action=lambda node: order.append(node))
    return len(order) * 2 == len(double_perm)
```

4.2. Przeszukiwanie wszerz grafu kołowego

Przeszukiwanie grafu wszerz BFS (ang. *Breadth-First Search*) to jeden z podstawowych algorytmów wykorzystywanych w analizie grafów. Jego implementacja różni się w zależności od przyjętej reprezentacji grafu. Poniżej przedstawiono opis algorytmu dla grafu kołowego w postaci permutacji etykiet cięciw okręgu.

Dane wejściowe: Graf kołowy w reprezentacji permutacyjnej.

Problem: Przeszukiwanie grafu wszerz.

Opis algorytmu: Algorytm rozpoczyna od wybranego wierzchołka startowego, następnie kontynuuje, przeszukując sąsiadów tego wierzchołka. Po odwiedzeniu sąsiadów pierwszego poziomu, przechodzi do sąsiadów drugiego poziomu, i tak dalej, aż przeszuka wszystkie wierzchołki grafu dostępne z danego wierzchołka startowego.

Implementacja została zorganizowana w klasie CircleBFS. Algorytm wykonuje się z poziomi metodu run, która wykorzystuje metodę _visit, w której przy wykorzystaniu kolejki Q wykonywany jest algorytm BFS rozpoczynając od danego wierzchołka. Wykorzystywana jest również metoda has_edge do sprawdzania, czy między dwoma wierzchołkami istnieje krawędź. Natomiast metoda path konstruuje ścieżkę od wierzchołka source do wierzchołka target, w drzewie BFS, o ile taka ścieżka istnieje.

Złożoność: Złożoność obliczeniowa algorytmu wynosi $O(n^2)$, co zostało potwierdzone w dodatku A.

Uwagi: Implementacja znajduje się na listingu 4.2.

Listing 4.2. Moduł CircleBFS.

```
import collections

class CircleBFS:
    """Breadth-First Search for circle graphs in O(n^2) time."""

    def __init__(self, double_perm):
        """The algorithm initialization."""
        self.perm = double_perm
        self.parent = dict()  # BFS tree
        self.pairs = dict((node, []) for node in set(self.perm))  # O(n) time
        for i, node in enumerate(self.perm):  # O(n) time
            self.pairs[node].append(i)

    def has_edge(self, source, target):
        s1, s2 = self.pairs[source]
        t1, t2 = self.pairs[target]
        return (s1 < t1 < s2 < t2) or (t1 < s1 < t2 < s2)

    def run(self, source=None, pre_action=None, post_action=None):
        """Executable pseudocode."""
        if source is not None:
```

```
            self._visit(source, pre_action, post_action)
        else:
            for node in self.perm:  # iternodes()
                if node not in self.parent:
                    self._visit(node, pre_action, post_action)

    def _visit(self, node, pre_action=None, post_action=None):
        """Explore the connected component."""
        Q = collections.deque()
        self.parent[node] = None  # before Q.put
        Q.appendleft(node)
        if pre_action:  # when Q.put
            pre_action(node)
        while Q:  # while not empty, O(n) time
            source = Q.pop()
            for target in self.pairs:  # iteroutedges(source), O(n) time
                if self.has_edge(source, target) and target not in self.parent:
                    self.parent[target] = source  # before Q.put
                    Q.appendleft(target)
                    if pre_action:  # when Q.put
                        pre_action(target)
            if post_action:
                post_action(source)

    def path(self, source, target):
        """Construct a path from source to target."""
        if source == target:
            return [source]
        elif self.parent[target] is None:
            raise ValueError("no path to target")
        else:
            return self.path(source, self.parent[target]) + [target]
```

4.3. Przeszukiwanie w głąb grafu kołowego

Dane wejściowe: Graf kołowy w reprezentacji permutacyjnej.

Problem: Przeszukiwanie grafu w głąb.

Opis algorytmu: Implementacja została zorganizowana w klasie CircleDFS w sposób analogiczny do opisanej wcześniej implementacji algorytmu BFS. Różnicą w stosunku do poprzedniego algorytmu jest to, że zamiast kolejki sąsiadów oczekujących na odwiedzenie wykorzystywana jest rekurencja. Algorytm eksploruje jak najdalej wzdłuż każdej gałęzi, po czym następuje nawrót (ang. *backtracing*) i przeszukiwane są kolejne gałęzie.

Złożoność: Złożoność obliczeniowa algorytmu wynosi $O(n^2)$, co zostało potwierdzone w dodatku A.

Uwagi: Algorytm zastosowany w implementacji jest rekurencyjny, co może stanowić problem w przypadku dużego grafu, gdzie może wystąpić zbyt wiele wywołań rekurencyjnych. Taka sytuacja może prowadzić do przekroczenia

limitu stosu pamięci, co w efekcie spowoduje błąd wewnętrzny interpretera Pythona RuntimeError. Jeśli zwiększymy limit stosu, możemy napotkać problem na poziomie jądra systemu operacyjnego.

Implementacja znajduje się na listingu 4.3.

Listing 4.3. Moduł CircleDFS.

```
import sys
import collections

class CircleDFS:
    """Depth-First Search for circle graphs in O(n^2) time."""

    def __init__(self, double_perm):
        """The algorithm initialization."""
        self.perm = double_perm
        self.parent = dict()  # BFS tree
        self.pairs = dict((node, []) for node in set(self.perm))  # O(n) time
        for i, node in enumerate(self.perm):  # O(n) time
            self.pairs[node].append(i)
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(len(self.perm) * 2, recursionlimit))

    def has_edge(self, source, target):
        s1, s2 = self.pairs[source]
        t1, t2 = self.pairs[target]
        return (s1 < t1 < s2 < t2) or (t1 < s1 < t2 < s2)

    def run(self, source=None, pre_action=None, post_action=None):
        """Executable pseudocode."""
        if source is not None:
            self.parent[source] = None  # before _visit
            self._visit(source, pre_action, post_action)
        else:
            for node in self.pairs:  # iternodes()
                if node not in self.parent:
                    self.parent[node] = None  # before _visit
                    self._visit(node, pre_action, post_action)

    def _visit(self, node, pre_action=None, post_action=None):
        """Explore recursively the connected component."""
        if pre_action:
            pre_action(node)
        for target in self.pairs:  # iteroutedges(node), O(n) time
            if self.has_edge(node, target) and target not in self.parent:
                self.parent[target] = node  # before _visit
                self._visit(target, pre_action, post_action)
        if post_action:
            post_action(node)

    def path(self, source, target):
        """Construct a path from source to target."""
        if source == target:
            return [source]
        elif self.parent[target] is None:
            raise ValueError("no path to target")
        else:
```

```
return self.path(source, self.parent[target]) + [target]
```

4.4. Wyznaczanie liczby kardynalnej największego zbioru niezależnego - algorytm naiwny

Zbiór wierzchołków w grafie nazywamy niezależnym, jeśli żadne dwa z jego elementów nie są połączone krawędzią. Największy zbiór niezależny to zbiór niezależny o największej liczności. Naiwny algorytm zaimplementowany został na podstawie artykułu, w którym znajduje się jego dokładny opis [33]. Implementacja została przedstawiona na listingu 4.4.

Dane wejściowe: Graf kołowy w postaci σ-reprezentacji.

Problem: Znalezienie liczności największego zbioru niezależnego.

Opis algorytmu: Algorytm implementuje technikę programowania dynamicznego, wykorzystując wzajemne zależności liczności największego zbioru niezależnego dla różnych podzbiorów w σ reprezentacji grafu. W rozwiązaniu zastosowano rekurencję obliczaną w tablicy M, zwiększając m od 1 do $2n$, a następnie zapisując wpisy tej rekurencji dla tej wartości m poprzez zmniejszanie q od $m-1$ do 1. W każdym obiegu pętli obliczane są wartości $M[q] = MIS[q, m]$, gdzie $MIS[q, m]$ definiujemy jako liczbę kardynalną największego zbioru niezależnego dla przedziału $[q, m]$ w σ-reprezentacji grafu kołowego. W ostatnim przebiegu pętli obliczana jest wartość $M[1] = MIS[1, 2n]$, co stanowi rozwiązanie problemu.

Złożoność: W wyniku działania algorytmu otrzymuje się liczbę kardynalną największego zbioru niezależnego w czasie $O(n^2)$ przy wykorzystaniu $O(n)$ pamięci. Złożoność czasowa została potwierdzona doświadczalnie w dodatku A.

Uwagi: Algorytm ten służy jako podstawa dla bardziej wydajnego algorytmu czułego na wyjście 4.5.

Listing 4.4. Moduł naive_maximum_independent_set.

```
def naiveMIS(sigma):
    '''
    :param sigma: sigma representation of a circle graph
    :return: cardinality of maximum independent set
    '''
    n = len(sigma) // 2
    M = [0] * (2 * n + 2)
    C = [0] * n

    idx = [0] * (2 * n + 1)
    for (i, m) in enumerate(sigma):
        idx[m] = i

    for m in range(1, 2 * n + 1):
```

```
        m_index = idx[m]
        if m_index % 2 == 1:  # m is right endpoint
            left = sigma[m_index - 1]  # left endpoint
            i = (m_index - 1) // 2
            C[i] = M[left + 1]
        for q in range(m - 1, 0, -1):
            M[q] = M[q + 1]  # right endpoint
            q_index = idx[q]
            if q_index % 2 == 0:  # left endpoint
                r = sigma[q_index + 1]  # right endpoint
                j = q_index // 2
                if r <= m:
                    M[q] = max((M[q + 1]), (1 + C[j] + M[r + 1]))
    return M[1]
```

4.5. Wyznaczanie liczby kardynalnej największego zbioru niezależnego - algorytm wrażliwy na wyjście

Algorytm wrażliwy na wyjście bierze pod uwagę liczby zmieniających się wartości w tablicy M, podczas zwiększania m. Implementacja znajduje się na listingu 4.5.

Dane wejściowe: Graf kołowy w postaci σ-reprezentacji.

Problem: Znalezienie liczności największego zbioru niezależnego.

Opis algorytmu: Algorytm jest usprawnieniem algorytmu naiwnego 4.4. Wyliczanie MIS nie jest konieczne dla wszystkich przedziałów, co ogranicza liczbę obliczeń. Zamiast iterować od $q = m - 1$ do 1 tworzymy stos, dzięki któremu badamy tylko te przedziały, w których mogło nastąpić zwiększenie wartości MIS.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n\alpha)$, gdzie n to liczba wierzchołków grafu kołowego, a α to rozmiar największego zbioru niezależnego. Złożoność czasowa została potwierdzona doświadczalnie w dodatku A.

Uwagi: W literaturze [33] opisana jest również modyfikacja zwiększająca wydajność tego rozwiązania.

Listing 4.5. Moduł sensitive_maximum_independent_set.

```
def update(sigma, m, M, C, idx):
    '''
    Input: The sigma-representation of an n vertex circle graph
        together with an integer m and the arrays M and C.
    Output: M such that M[q] = MIS[q,m] for 1 <= q <= m.
    '''
    stack_T = []
    m_index = idx[m]
    if m_index % 2 == 1:  # right endpoint
        left = sigma[m_index - 1]
```

```
            i = (m_index - 1) // 2
            M[left] = 1 + C[i]
            stack_T.append(left)  # PUSH(T, left)
            while stack_T:
                x = stack_T.pop()
                if x > 1 and M[x] > M[x - 1]:
                    M[x - 1] = M[x]
                    stack_T.append(x - 1)
                x_index = idx[x]
                if x_index % 2 == 1:  # right endpoint
                    p = sigma[x_index - 1]
                    j = (x_index - 1) // 2
                    if (1 + C[j] + M[x]) > M[p]:
                        M[p] = 1 + C[j] + M[x]
                        stack_T.append(p)  # PUSH(T, p)


def sensitiveMIS(sigma):
    '''
    Algorithm for computing the cardinality of a circle graph's
    maximum independent sets in O(n alpha) time.
    :param sigma: sigma representation of a graph
    :return: cardinality of maximum independent set
    '''
    n = len(sigma) // 2
    M = [0] * (2 * n + 1)
    C = [0] * n
    idx = [0] * (2 * n + 1)
    for (i, m) in enumerate(sigma):
        idx[m] = i
    for m in range(1, 2 * n + 1):
        m_index = idx[m]
        if m_index % 2 == 1:  # right endpoint
            left = sigma[m_index - 1]
            i = (m_index - 1) // 2
            C[i] = M[left + 1]
        update(sigma, m, M, C, idx)
    return M[1]
```

4.6. Wyznaczanie największej kliki w grafie kołowym

Załóżmy, że mamy graf kołowy $G = (V, E)$, gdzie V oznacza zbiór wierzchołków, a E oznacza zbiór krawędzi. Dla każdego wierzchołka v należącego do zbioru V, indukowany podgraf $G_v = (V_v, E_v)$ jest grafem permutacji [35]. Zatem problem szukania największej kliki sprowadzamy do znalezienia największych klik w n grafach permutacji, oraz wybranie z nich tej największej. Wyznaczanie największej kliki dla grafu permutacji wykonuje się poprzez szukanie największego malejącego podciągu (ang. *Longest Decreasing Subsequence, LDS*) w permutacji reprezentującej graf. Przygotowana implementacja wykorzystuje algorytm znajdowania największej kliki w grafach permutacji przedstawiony w mojej poprzedniej pracy [37] i została zaprezentowana na listingu 4.6.

Dane wejściowe: Graf kołowy jako instancja klasy Graph.

Problem: Wyznaczanie największej kliki w grafie kołowym.

Opis algorytmu: Podstawą algorytmu jest pętla, która iteruje po wszystkich wierzchołkach grafu i dla każdego z nich tworzy podgraf zawierający ten wierzchołek i wszystkich jego sąsiadów. Następnie przekształca ten podgraf w reprezentację grafu permutacji, by wywołać odpowiedni algorytm znajdowania największej kliki. W ostatnim kroku porównywany jest rozmiar znalezionej kliki z tymi sprawdzonymi do tej pory by wybrać największy.

Złożoność: Złożoność czasowa algorytmu wyznaczania największej kliki w grafie permutacji wynosi $O(n \log n)$ [37]. W zaprezentowanym algorytmie wywołujemy go n razy. Zatem teoretyczna całkowita złożoność czasowa przedstawionego algorytmu wynosi $O(n^2 \log n)$.

Złożoność obliczeniowa wyznaczona doświadczalnie w dodatku A wskazuje na złożoność $O(n^6)$. Jest to spowodowane wykonaniem niewydajnej implementacji algorytmu do rozpoznania grafu permutacji o złożoności $O(n^6)$.

Uwagi: W dalszej części pracy znajdziemy algorytm badający graf kołowy w reprezentacji permutacyjnej, dla którego udało się pominąć wykonywanie algorytmu rozpoznania grafu permutacji.

Listing 4.6. Moduł max_clique.

```
from PermutationGraph.recognition import PermutationGraph
from PermutationGraph.clique_and_iset import largest_clique


def max_clique(graph):
    '''
    :param G: Graph
    :return: maximum clique
    '''
    maxClique = set()
    for node in graph.iternodes():
        neighbors = set(graph.iteradjacent(node))
        neighbors.add(node)
        subgraph = graph.subgraph(neighbors)
        perm_graph = PermutationGraph(subgraph)
        perm_graph.run()
        perm = perm_graph.get_perm_repr()
        verticesMapping = perm_graph.perm_repr.vertice_to_number
        numberMapping = dict((i, v) for (v, i) in verticesMapping.items())
        clique = largest_clique(perm)
        clique2 = set(numberMapping[i] for i in clique)
        maxClique = max(maxClique, clique2, key=len)
    return maxClique
```

4.7. Wyznaczanie największej kliki w grafie kołowym w reprezentacji permutacyjnej

Ogólne założenia algorytmu są identyczne z poprzednim algorytmem, lecz wykorzystanie reprezentacji przez permutację z podwójnymi wystąpieniami pozwoliło na wykorzystanie jej właściwości oraz uzyskanie wydajniejszej złożoności obliczeniowej.

Dane wejściowe: Graf kołowy w reprezentacji permutacyjnej.

Problem: Wyznaczanie największej kliki w grafie kołowym

Opis algorytmu: Algorytm analogiczny do opisanego w 4.6, przy czym rozpoznanie grafu permutacji zastąpiono algorytmem badającym czy dany graf kołowy jest grafem permutacji, który działa w czasie liniowym. W implementacji zastąpiono również znajdowanie największej kliki dla grafu permutacji znajdowaniem największego zbioru niezależnego dla dopełnienia grafu. Ta zamiana wynika z właściwości grafów permutacji i pozwoliła na wykorzystanie wydajniejszego algorytmu.

Złożoność: Teoretyczna złożoność czasowa algorytmu wynosi $O(n^2 \log n)$. Złożoność obliczeniowa wyznaczona doświadczalnie w dodatku A wskazuje na złożoność bliską $O(n^2)$. Wynika to stąd, że mamy n indukowanych podgrafów permutacji o różnych liczbach wierzchołków k i zwykle $k < n$ (dla grafu C_n mamy $k = 3$). Na takim podgrafie szukamy największej kliki w czasie $O(k \log k)$, co po wyśredniowaniu dla przypadkowych grafów jest bliskie $O(n)$.

Listing 4.7. Moduł max_clique_from_double_perm.

```
from lis import find_maximum_iset
from detect_permutation_graph import is_permutation_graph


def max_clique_from_double_perm(double_perm):
    nodes = set(double_perm)
    pairs = dict((node, []) for node in nodes)
    for idx, node in enumerate(double_perm):
        pairs[node].append(idx)
    best_clique = set()  # max clique size
    for source in nodes:
        neighbors = set([source])
        for target in nodes:
            s1, s2 = pairs[source]
            t1, t2 = pairs[target]
            if (s1 < t1 < s2 < t2) or (t1 < s1 < t2 < s2):
                neighbors.add(target)
        subgraph = [node for node in double_perm if node in neighbors]
        perm, n2l, l2n = is_permutation_graph(subgraph)
        clique = find_maximum_iset(perm[::-1])  # O(n log n) time
        clique2 = set(n2l[i] for i in clique)
        best_clique = max(best_clique, clique2, key=len)
    return best_clique
```

5. Podsumowanie

Grafy kołowe są uogólnieniem grafów permutacji i grafów przedziałowych, ale nie są grafami doskonałymi. Z tego powodu są trudniejsze do analizy, a znalezione algorytmy są czasem dość złożone. W tej pracy zaimplementowano różne algorytmy wykorzystywane do obliczeń na grafach kołowych.

Przygotowano algorytmy do transformacji grafu kołowego pomiędzy różnymi reprezentacjami. Permutację z podwójnymi wystąpieniami można przekształcić do grafu abstrakcyjnego i do σ-reprezentacji.

Przygotowano generatory grafów kołowych w reprezentacji z podwójnymi wystąpieniami: graf przypadkowy spójny lub ogólny, graf ścieżka P_n, graf cykliczny C_n, graf k-tree, graf gwiazda $K_{1,n-1}$.

Przygotowano test czy dany graf kołowy jest grafem permutacji, a złożoność obliczeniowa testu jest liniowa $O(n)$. Przygotowano implementację dwóch wersji algorytmu wyznaczającego największą klikę w grafie kołowym.

Zaimplementowano dwa algorytmy wyznaczające liczność największego zbioru niezależnego α. Algorytm naiwny ma złożoność $O(n^2)$, algorytm czuły na wyjście ma złożoność $O(n\alpha)$. Dla grafów kołowych przypadkowych często obserwowaliśmy, że α było $O(\sqrt{n})$. Skrajne przypadki to $\alpha = 1$ dla K_n, $\alpha = n/2$ dla C_n, $\alpha = n$ dla grafu bez krawędzi.

Oprócz tego, warto zwrócić uwagę na perspektywy dalszych badań i rozwoju w dziedzinie grafów kołowych. Jednym z ciekawych obszarów badań jest poszukiwanie algorytmów, które mogą osiągnąć lepszą wydajność, mając informację, że analizujemy graf kołowy. Znając pewne cechy specyficzne dla grafów kołowych, można implementować algorytmy, które wykorzystują tę charakterystykę w celu optymalizacji ich działania.

Kolejnym interesującym aspektem, który warto zbadać, jest możliwość zastosowania σ-reprezentacji, która już udowodniła swoją użyteczność w algorytmie do wyznaczania liczności największego zbioru niezależnego, do innych problemów grafowych. W rezultacie, istotne jest sprawdzenie, czy to podejście mogłoby przyczynić się do poprawy efektywności rozwiązań tych problemów.

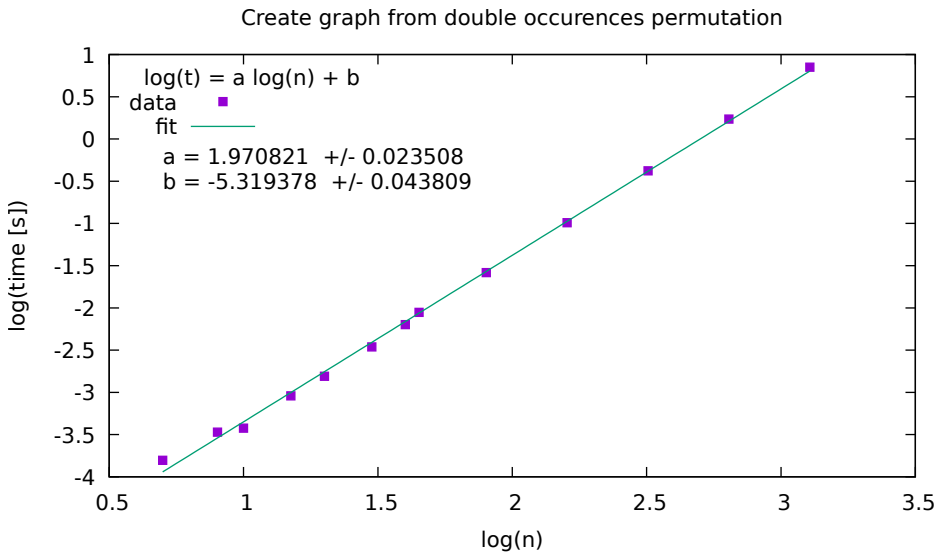
Na koniec musimy jeszcze raz wspomnieć o problemie rozpoznawania grafów kołowych, dla którego nie mamy działającej implementacji. W teorii problem jest rozwiązany (por. rozdział 1), podano kilka algorytmów. W praktyce okazuje się, że algorytmy albo mają dużą złożoność obliczeniową i nie nadają się do obliczeń na dużych grafach, albo w algorytmach stosowane są zaawansowane metody teoretyczne (np. teoria matroidów, *split decomposition*), przez co przygotowanie implementacji jest sprawą bardzo trudną. Z drugiej strony, w pewnych zastosowaniach (układy VLSI) na wejściu dana jest reprezentacja graficzna grafu kołowego, więc to ona może być wygodniejszym punktem startowym.

A. Testy algorytmów

Algorytmy zostały sprawdzone pod kątem poprawności przy użyciu testów jednostkowych za pomocą modułu *unittest* [41]. Przeprowadzono również testy wydajności, które zostały wykonane za pomocą modułu *timeit* [42], umożliwiającego pomiar czasu wykonania poszczególnych funkcji. Dla każdego zestawu danych przeprowadzono trzykrotne testy, a przedstawione w tym rozdziale wykresy prezentują wyniki najszybszego z tych testów.

A.1. Testy tworzenia grafu abstrakcyjnego na podstawie reprezentacji permutacyjnej

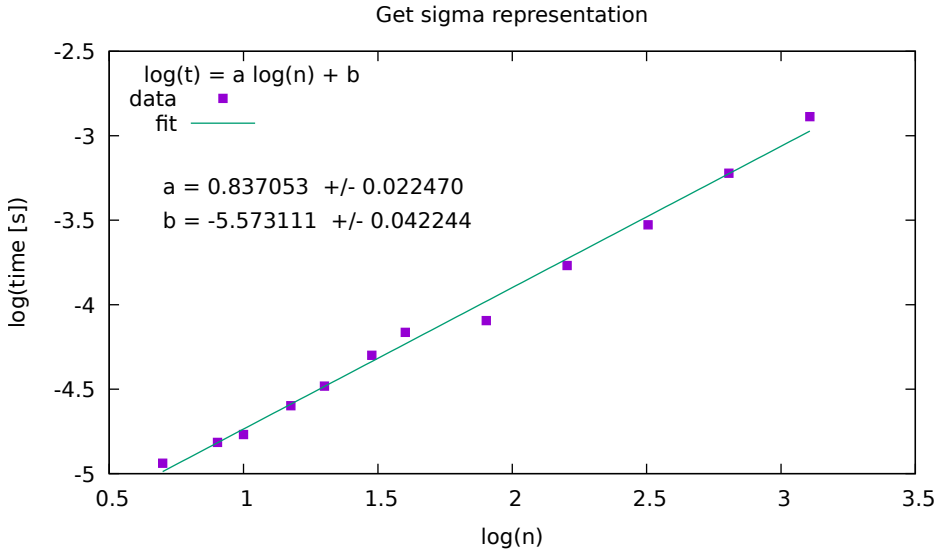
W celu oceny wydajności algorytmu przetestowano go na losowych permutacjach z podwójnymi wystąpieniami, które reprezentują spójne grafy kołowe. Czasy wykonania algorytmu zostały zmierzone i przedstawione na wykresie A.1. Dopasowanie krzywej o współczynniku $a = 1.971(24)$ potwierdza, że czas wykonania algorytmu jest kwadratowy, co jest zgodne z teoretyczną złożonością obliczeniową.



Rysunek A.1. Wykres wydajności algorytmu utworzenia grafu z pakietu *graphtheory* na podstawie reprezentacji permutacyjnej.

A.2. Testy tworzenia σ-reprezentacji grafu kołowego

Losowe permutacje z podwójnymi wystąpieniami, które reprezentują spójne grafy kołowe zostały również wykorzystane do przetestowania algorytmu generującego σ-reprezentację. Uzyskane wyniki znajdują się na wykresie A.2. Współczynnik dopasowania prostej $a = 0.837(23)$ potwierdza liniową złożoność obliczeniową $O(n)$ tego algorytmu.



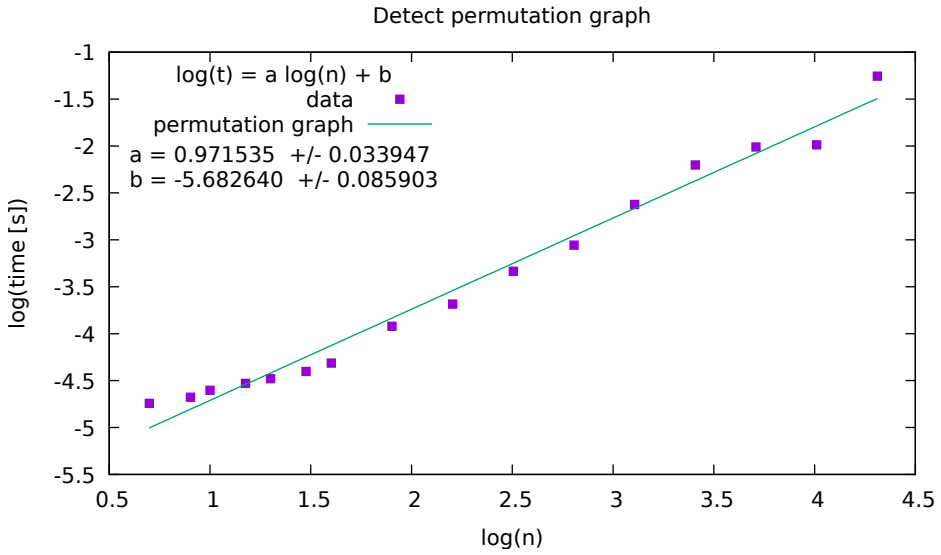
Rysunek A.2. Wykres wydajności algorytmu utworzenia σ-reprezentacji grafu na podstawie reprezentacji permutacyjnej.

A.3. Testy algorytmu do sprawdzania czy dany graf kołowy jest grafem permutacji

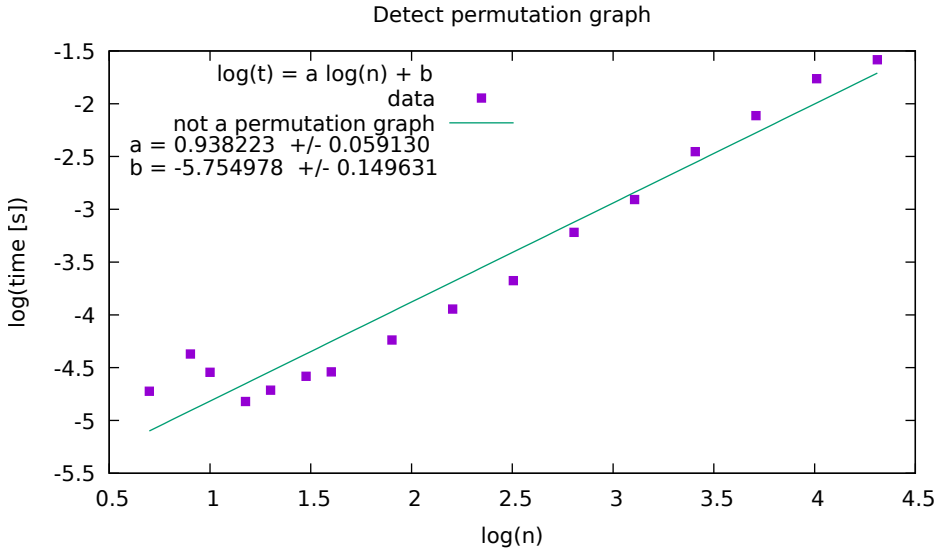
Przeprowadzono testy wydajności algorytmu sprawdzającego czy dany graf kołowy jest grafem permutacji. Badania zostały przeprowadzone na dwóch typach grafów. Pierwszy to losowe grafy permutacji, które zapewniły wykonanie się całego algorytmu. Współczynnik dopasowania prostej wyniósł $a = 0.972(34)$.

W drugim przypadku użyto grafów cyklicznych C_n, które nie są grafami permutacji dla $n > 4$. W ten sposób zbadano część algorytmu wystarczającą do sprawdzenia czy graf jest grafem permutacji, a pominięty został etap generowania permutacji i mapowań etykiet. W tym przypadku uzyskano współczynnik $a = 0.938(60)$.

Uzyskane wyniki potwierdzają liniową złożoność obliczeniową algorytmu co zostało zaprezentowane na wykresach A.3 i A.4.



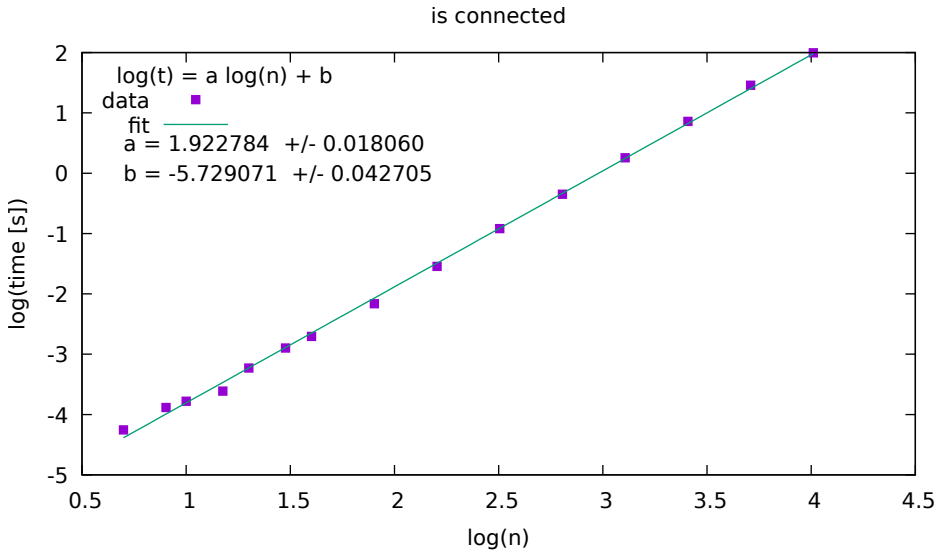
Rysunek A.3. Wykres wydajności algorytmu do sprawdzania czy dany graf kołowy jest grafem permutacji w przypadku, gdy faktycznie nim jest.



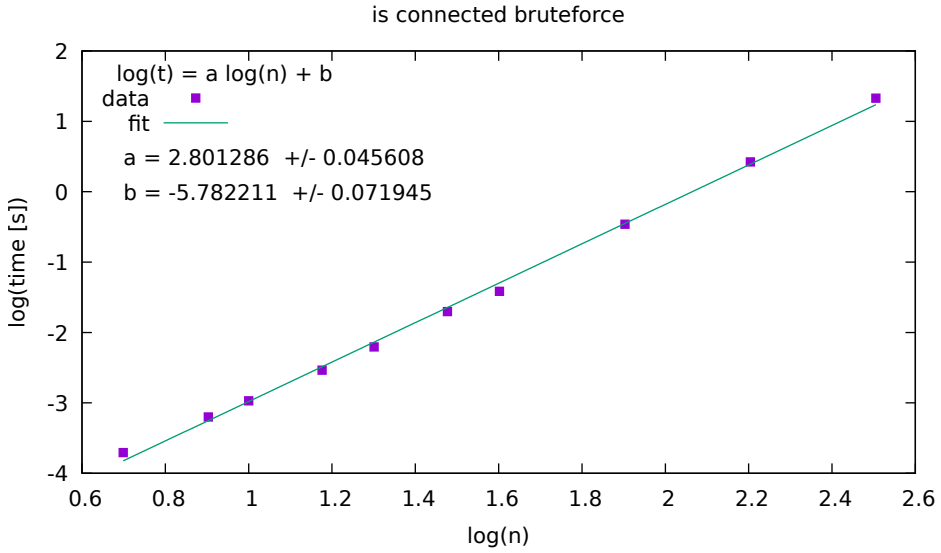
Rysunek A.4. Wykres wydajności algorytmu do sprawdzania czy dany graf kołowy jest grafem permutacji w przypadku, gdy nim nie jest.

A.4. Testy badania spójności grafu kołowego

Wydajność sprawdzania spójności grafu kołowego została wyznaczona na przypadkowych grafach. Wykres A.5 prezentuje rezultaty pomiarów. Wartość współczynnika a w dopasowanej krzywej wyniosła 1.923(18) i świadczy o złożoności $O(n^2)$. Na wykresie A.6 przedstawiono wyniki dla algorytmu siłowego, gdzie współczynnik a wyniósł 2.801(46) i świadczy o złożoności $O(n^3)$. Na wykresie A.13 przedstawiono również porównanie wydajności algorytmu wykorzystującego BFS dla szczególnych grafów.



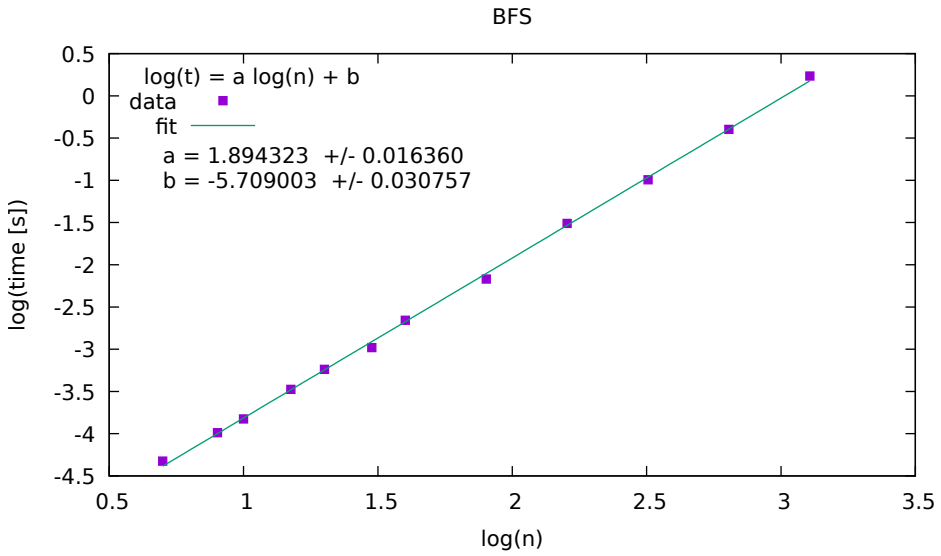
Rysunek A.5. Wykres wydajności algorytmu badania spójności grafu kołowego.



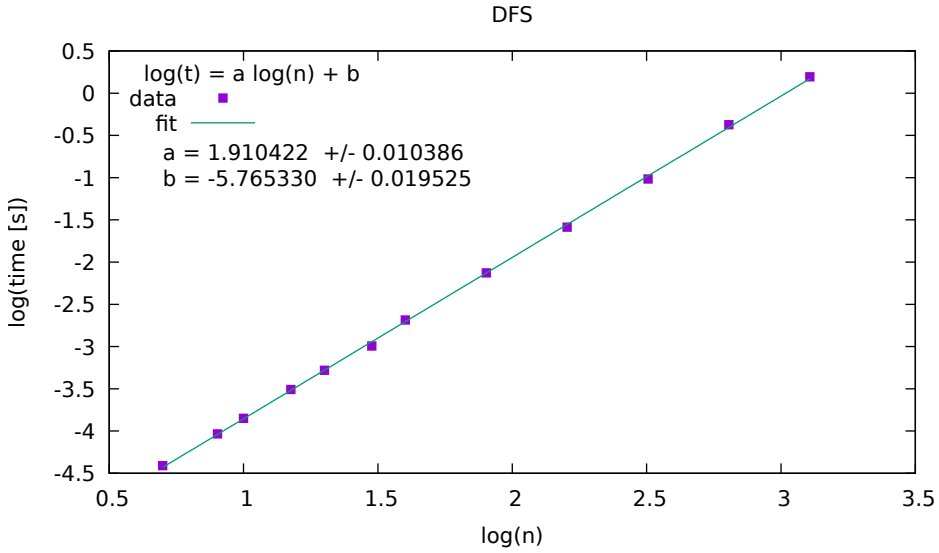
Rysunek A.6. Wykres wydajności algorytmu badania spójności grafu kołowego - metoda siłowa

A.5. Testy przeszukiwania grafów kołowych

Przeprowadzono testy algorytmów BFS i DFS, za każdym razem rozpoczynając przeszukiwanie od wierzchołka 0. Wyniki dla przeszukiwania wszerz zaprezentowano na wykresach A.7 i A.14, a dla przeszukiwania w głąb na wykresach A.8 i A.15. Otrzymane wartości współczynników dopasowania prostej a to odpowiednio 1.894(16) dla przeszukiwania wszerz, oraz 1.910(10) dla przeszukiwania wgłąb. W obu przypadkach współczynniki a świadczą o złożoności rzędu $O(n^2)$.



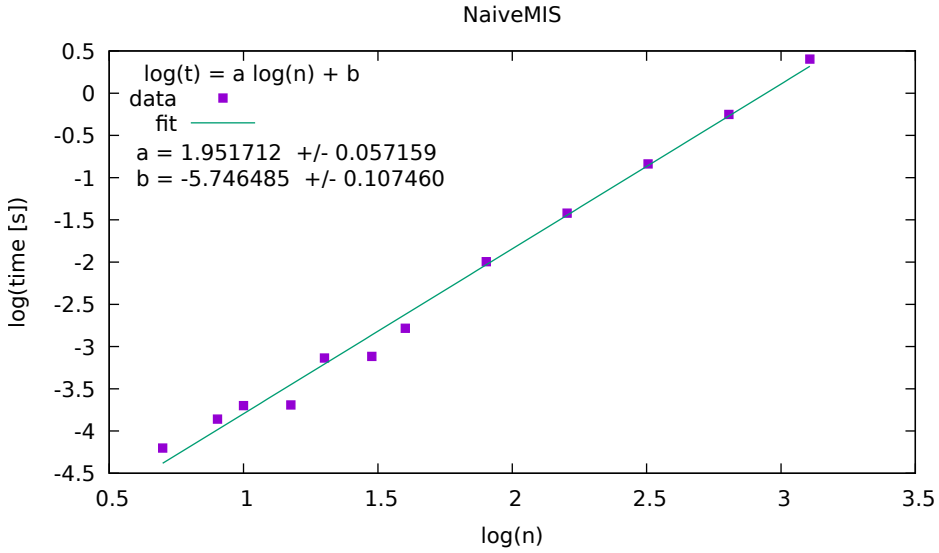
Rysunek A.7. Wykres wydajności algorytmu przechodzenia grafu wszerz.



Rysunek A.8. Wykres wydajności algorytmu przechodzenia grafu w głąb.

A.6. Testy algorytmów do wyznaczania liczby kardynalnej największego zbioru niezależnego

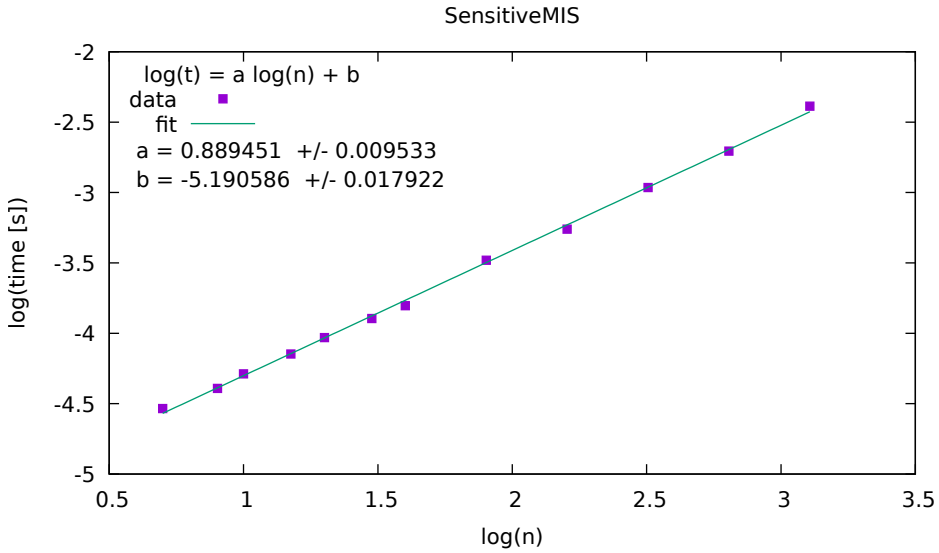
Zostały przetestowane dwa algorytmy do wyznaczania liczby kardynalnej największego zbioru niezależnego. Wyniki pomiarów zostały zobrazowane na wykresach dla algorytmu naiwnego A.9 i algorytmu wrażliwego A.10. Wartość współczynnika a wskazuje na różnicę w ich złożoności obliczeniowej. Dla algorytmu naiwnego $a = 1.952(57)$ oznacza czas kwadratowy, a dla algorytmu wrażliwego $a = 0.89(1)$, świadczy o czasie liniowym. Na wykresach A.16 i A.17 przedstawiono również porównanie szybkości tych algorytmów dla grafów, gdzie każdy wierzchołek jest osobny, grafu liniowego i grafu przypadkowego. Dla algorytmu wrażliwego możemy zauważyć sporą różnicę między wynikami dla różnych grafów. Potwierdza to zakładaną złożoność algorytmu $O(n\alpha)$. Współczynnik dopasowania w przypadku grafu liniowego wyniósł $a = 1.669(46)$ czyli lepiej niż $O(n^2)$. Wynika to z faktu, że dla grafu liniowego $\alpha = n/2$.



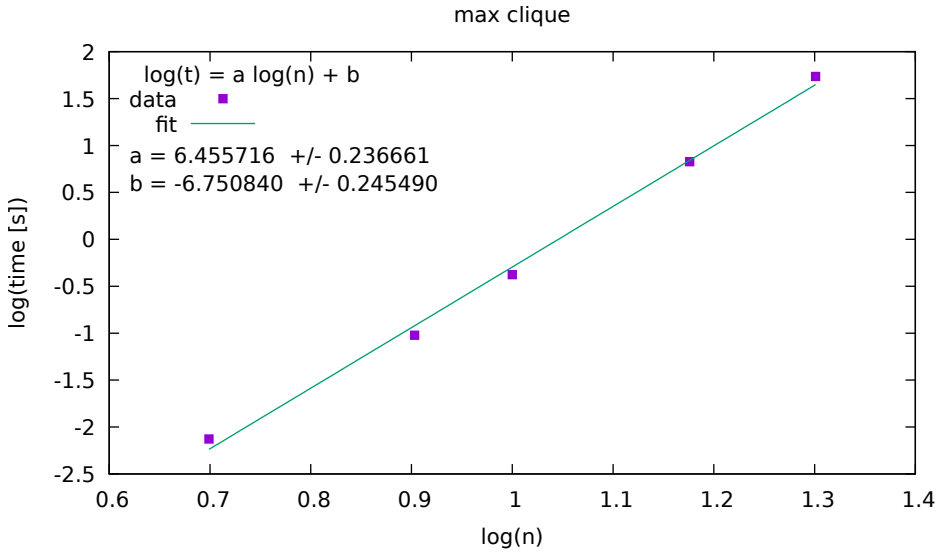
Rysunek A.9. Wykres wydajności algorytmu naiwnego do wyznaczanie liczby kardynalnej największego zbioru niezależnego.

A.7. Testy algorytmów do wyznaczania największej kliki w grafie kołowym

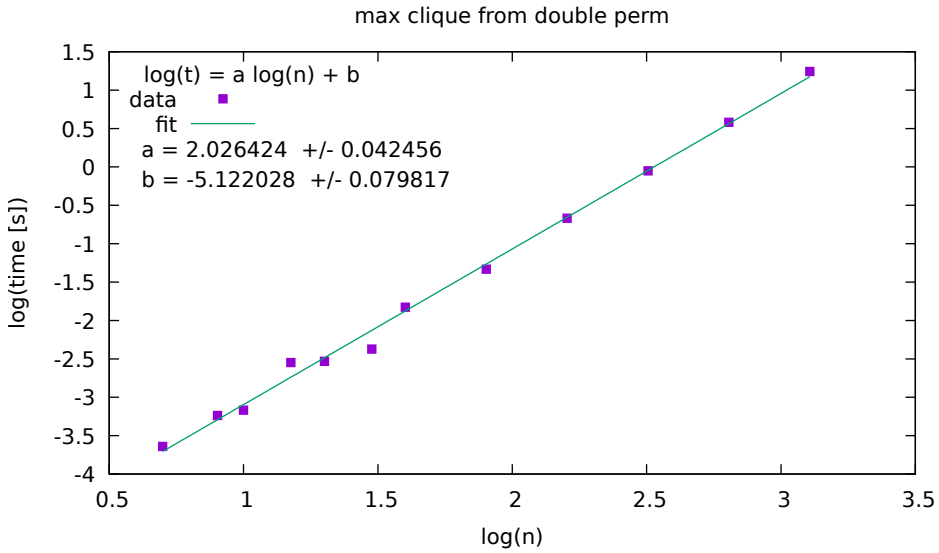
Zostały przetestowane dwa algorytmy do wyznaczania największej kliki w grafie kołowym. Pierwszy wykorzystujący algorytm rozpoznania grafu permutacji, dla którego współczynnik a wyniósł w przybliżeniu 6 co oznacza złożoność obliczeniową $O(n^6)$. W drugim algorytmie uzyskano współczynnik $a = 2.026(42)$, świadczący o złożoności bliskiej $O(n^2)$. Opisane wyniki znajdują się na wykresach A.11 oraz A.12.



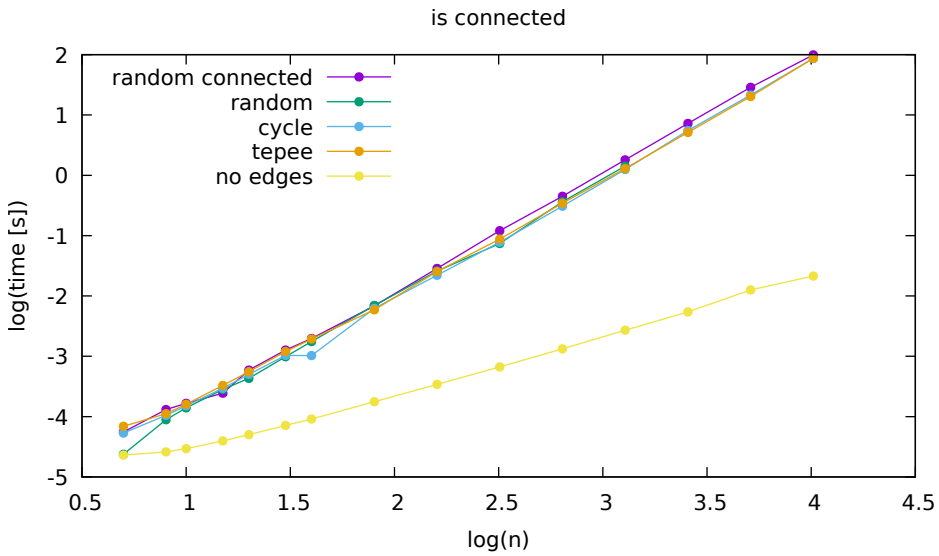
Rysunek A.10. Wykres wydajności algorytmu wrażliwego do wyznaczanie liczby kardynalnej największego zbioru niezależnego.



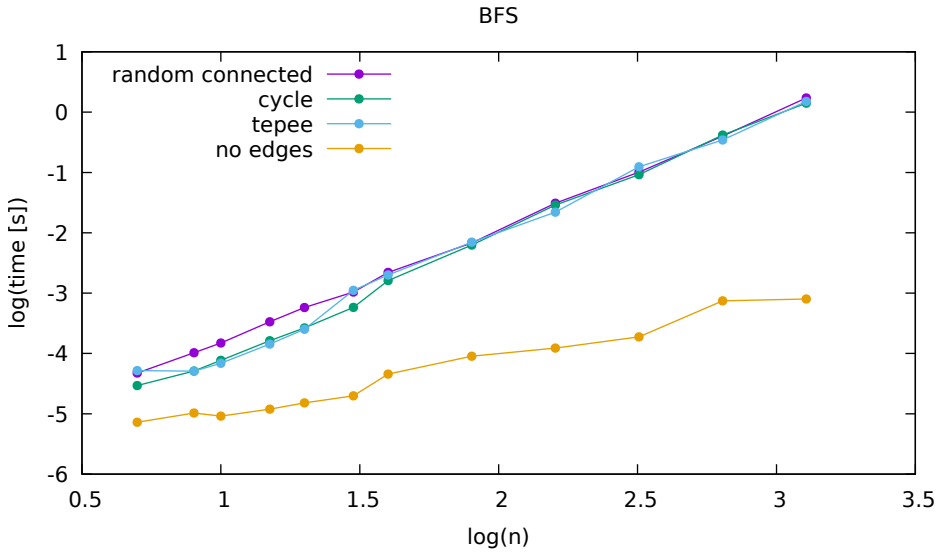
Rysunek A.11. Wykres wydajności algorytmu algorytmu do wyznaczania największej kliki w grafie kołowym.



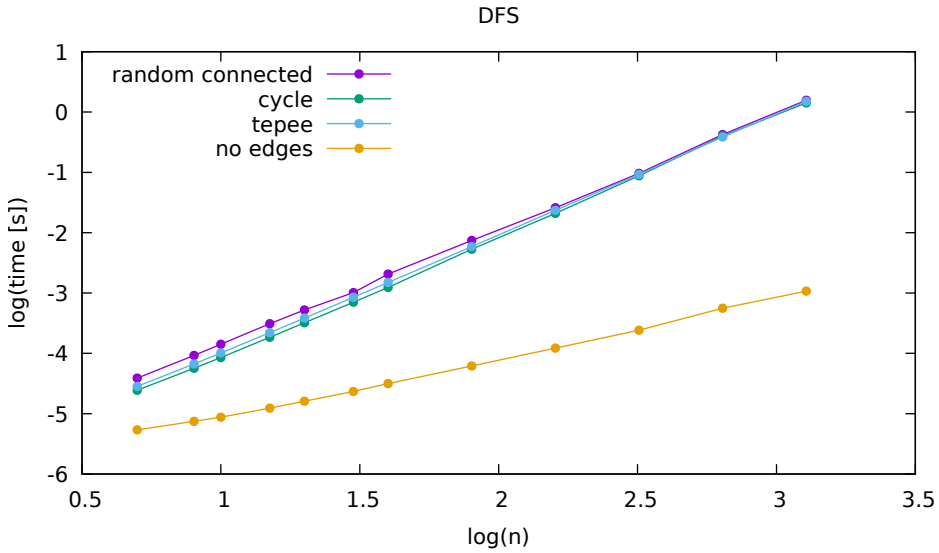
Rysunek A.12. Wykres wydajności algorytmu algorytmu do wyznaczania największej kliki w grafie kołowym w reprezentacji permutacyjnej.



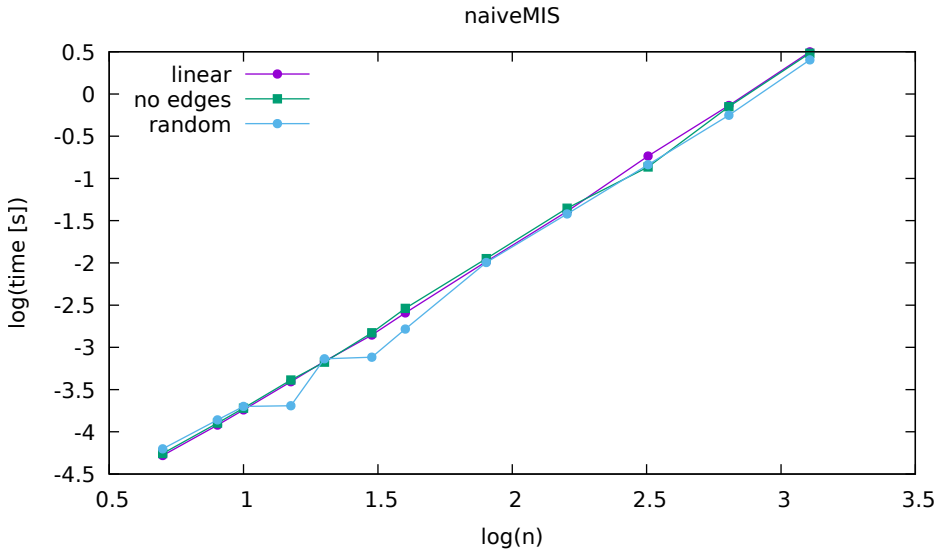
Rysunek A.13. Porównanie wydajności algorytmu badania spójności grafu kołowego dla różnych danych wejściowych.



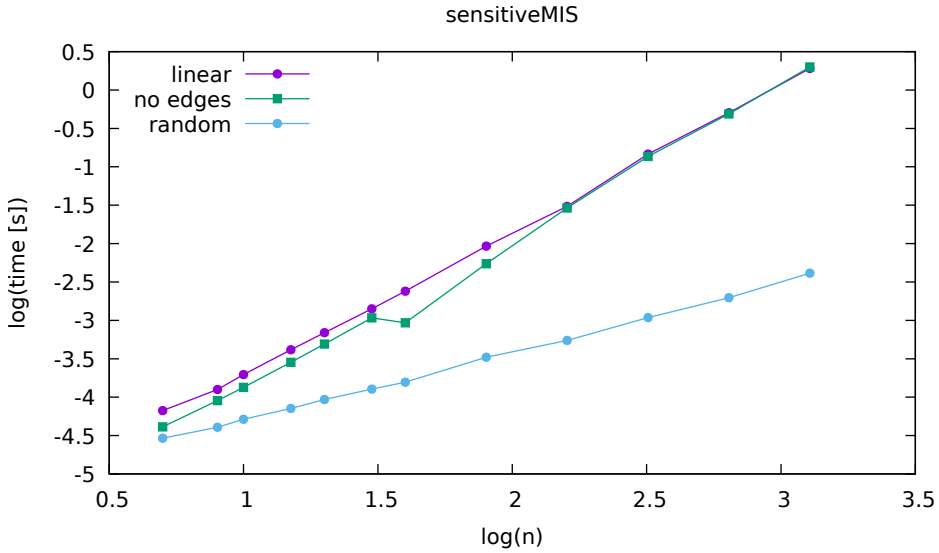
Rysunek A.14. Porównanie działania algorytmu BFS dla szczególnych grafów.



Rysunek A.15. Porównanie działania algorytmu DFS dla szczególnych grafów.



Rysunek A.16. Porównanie działania algorytmu *naiveMIS* dla grafu liniowego, grafu z izolowanych wierzchołków i grafu przypadkowego.



Rysunek A.17. Porównanie działania algorytmu *sensitiveMIS* dla grafu liniowego, grafu z izolowanych wierzchołków i grafu przypadkowego.

Bibliografia

[1] Wikipedia, Circle graph, 2023, https://en.wikipedia.org/wiki/Circle_graph.

[2] Even S., Itai A., *QUEUES, STACKS AND GRAPHS*, Theory of Machines and Computations 71–86, 1971.

[3] Gavril F., *Algorithms for a maximum clique and a maximum independent set of a circle graph*, Networks 3 (3): 261–273, 1973.

[4] Naji W., *Reconnaissance des graphes de cordes*, Discrete Mathematics 54 (3): 329–337, 1985.

[5] Bouchet A., *Reducing prime graphs and recognizing circle graphs*, Combinatorica 7 (3): 243–254, 1987.

[6] Gabor C.P., Supowit K.J., Hsu W.-L., *Recognizing circle graphs in polynomial time*, Journal of the ACM 36 (3): 435–473, 1989.

[7] Spinrad J., *Recognition of Circle Graphs*, Journal of Algorithms, 16(2), 264–282, 1994.

[8] J. Geelen, S. Oum, *Circle graph obstructions under pivoting*, Journal of Graph Theory 61, 1-11 (2009).

[9] Garey M.R., Johnson D.S., Miller G.L., Papadimitriou C.H., *The Complexity of Coloring Circular Arcs and Chords*, SIAM Journal on Algebraic Discrete Methods 1 (2): 216–227, 1980.

[10] Damaschke, P., *The Hamiltonian circuit problem for circle graphs is NP-complete*, Information Processing Letters 32(1), 1–2, 1989.

[11] Keil, J.M., *The complexity of domination problems in circle graphs*, Discrete Appl. Math. 42 (1), 51–63, 1993.

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

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

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

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

[16] Golumbic M.C., *Algorithmic Graph Theory and Perfect Graphs*, Annals of Discrete Mathematics Ser. North Holland. 2004.

[17] Wikipedia, Permutation graph, 2023, https://en.wikipedia.org/wiki/Permutation_graph.

[18] Bouchet A., *Circle Graph Obstructions*, J. Comb. Theory B 60(1), 107-144, 1994.

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

[20] Information System on Graph Classes and their Inclusions, circle graph, 2023 https://www.graphclasses.org/classes/gc_913.html.

[21] Wikipedia, Split (graph theory), 2023, https://en.wikipedia.org/wiki/Split_(graph_theory).

[22] Ma T.H., Spinrad J., *An $O(n^2)$ Algorithm for Undirected Split Decomposition*, Journal of Algorithms 16 (1): 145–160, 1994.
[23] Darty K., Denise A., Ponty Y., *VARNA: Interactive drawing and editing of the RNA secondary structure*, Bioinformatics 25 (15): 1974–1975, 2009.
[24] Nussinov R., Jacobson A.B., *Fast algorithm for predicting the secondary structure of single-stranded RNA*, Proceedings of the National Academy of Sciences 77 (11): 6309–6313, 1980.
[25] Nussinov R., Pieczenik G., Griggs J.R., Kleitman D.J., *Algorithms for Loop Matchings*, SIAM Journal on Applied Mathematics 35 (1): 68–82, 1978.
[26] Ward-Graham, Max Hector, *Algorithms for RNA Structure and Related Graphs*, Diss. The University of Western Australia, 2019.
[27] Blasum U., Bussieck M.R., Hochstättler W., Moll C., Scheel H.-H., Winter T., *Scheduling trams in the morning*, Mathematical Methods of Operations Research 49 (1): 137–148, 1999.
[28] Cornelsen S., Di Stefano G., *Track assignment*, Journal of Discrete Algorithms 5 (2): 250–261, 2007.
[29] Demange M., Di Stefano G., Leroy-Beaulieu B., *On the online track assignment problem*, Discrete Applied Mathematics 160 (7–8): 1072–1093, 2012.
[30] Gallo G., Miele F.D., *Dispatching Buses in Parking Depots*, Transportation Science 35 (3): 322–330, 2001.
[31] Avriel M., Penn M., Shpirer N., *Container ship stowage problem: complexity and connection to the coloring of circle graphs*, Discrete Applied Mathematics 103 (1–3): 271–279, 2000.
[32] Tanaka M., Matsui T., *New Formulation for Coloring Circle Graphs And its Application to Capacitated Stowage Stack Minimization*, SSRN Electronic Journal, 2022.
[33] Nash N., Gregg D., *An output sensitive algorithm for computing a maximum independent set of a circle graph*, Information Processing Letters 110 (16): 630–634, 2010.
[34] Asano Takao, Asano Tetsuo, Imai H., *Partitioning a polygonal region into trapezoids*, Journal of the ACM 33 (2): 290–312, 1986.
[35] Sherwani N.A., *Algorithms for VLSI Physical Design Automation*, 1993.
[36] Bonsma P., Breuer F., *Counting Hexagonal Patches and Independent Sets in Circle Graphs*, Algorithmica 63 (3): 645–671, 2011.
[37] Albert Surmacz, *Badanie grafów permutacji z językiem Python.*, Kraków 2021.
[38] The number of maximal cliques of the intersection graphs, Serwis *mathoverflow.net*, 2023.
`https://mathoverflow.net/questions/443170/the-number-of-maximal-cliques-of-the-intersection-graphs`
[39] The On-Line Encyclopedia of Integer Sequences, 2023,
`https://oeis.org/`.
[40] Daniel Meister, *Treewidth and minimum fill-in on permutation graphs in linear time*, Theoretical Computer Science 411, No. 40-42, 3685-3700 (2010).
[41] Python Docs, unittest, 2023,
`https://docs.python.org/3/library/unittest.html`.
[42] Python Docs, timeit, 2023,
`https://docs.python.org/3/library/timeit.html`.