**Uniwersytet Jagielloński w Krakowie**

Wydział Fizyki, Astronomii i Informatyki Stosowanej

**Honorata Zych**

Nr albumu: 1113456

# Triangulacje grafów

Praca magisterska na kierunku Informatyka stosowana

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

Kraków 2024

## Oświadczenie autora pracy

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

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

Kraków, dnia Podpis autora pracy

## Oświadczenie kierującego pracą

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

Kraków, dnia Podpis kierującego pracą

*Składam serdeczne podziękowania Promotorowi Panu dr hab. Andrzejowi Kapanowskiemu za poświęcony czas, ogromne zaangażowanie, wsparcie oraz pomoc i uwagi merytoryczne, bez których ta praca nie powstałaby.*

**Streszczenie**

W pracy przedstawiono implementację w języku Python wybranych algorytmów związanych z triangulacją grafów. Triangulacja grafu G to inaczej znajdowanie dopełnienia cięciwowego dla G, czyli grafu cięciwowego będącego nadgrafem grafu G, z takim samym zbiorem wierzchołków jak graf G. Istnieją różne rodzaje triangulacji, jedne minimalizują liczbę dodawanych nowych krawędzi (problem najmniejszego wypełnienia), inne minimalizują rozmiar największej kliki w końcowym grafie cięciwowym (problem szerokości drzewowej). Rozważa się również triangulacje minimalne, czyli niekoniecznie najmniejsze, ale nie będące podzbiorem innej triangulacji.

Zaimplementowane algorytmy można podzielić na dwa rodzaje: algorytmy wyznaczania doskonałego uporządkowania eliminacji (ang. perfect elimination ordering, PEO) dla wybranych klas grafów oraz algorytm służący do znajdowania minimalnej triangulacji dla grafu wejściowego dowolnej klasy. Znajomość PEO wyznacza jednoznacznie zbiór nowych krawędzi dodanych do początkowego grafu.

W ramach pracy zaimplementowano pięć algorytmów do wyznaczania PEO dla klas grafów, dla których tą operację można zrealizować w czasie wielomianowym z podaniem dokładnego rozwiązania. Są to drzewa, grafy zewnętrznie planarne i grafy maksymalnie zewnętrznie planarne, grafy permutacji. Przedstawiono nową implementację algorytmu MCS do wyznaczania PEO dla grafów cięciwowych, który jest podstawą dla wydajnego algorytmu MCS-M wyznaczającego triangulacje minimalne. Algorytm MCS-M jest obecnie najlepszym znanym w literaturze.

Wszystkie algorytmy znajdujące się w pracy zostały przetestowane pod kątem poprawności oraz złożoności obliczeniowej z wykorzystaniem modułów unittest oraz timeit dostępych w Bibliotece Standardowej języka Python.

**Słowa kluczowe:** triangulacja grafu, triangulacja minimalna, grafy cięciwowe, uzupełnienie cięciwowe, grafy przedziałowe, grafy permutacji, dekompozycja drzewowa

**English title:** Triangulations of graphs


## Abstract

Python implementation of selected graph algorithms connected with triangulations of graphs is presented. A triangulation of a graph G is finding a chordal completion of G (or minimum fill-in) which is a chordal supergraph of G, on the same vertex set. There are different types of triangulations. The number of new edges added can be minimised (the minimum fill-in problem) or the size of the maximum clique in the resulting chordal graph can be minimised (the treewidth problem).

Implemented algorithms can be divided into two groups: algorithms for finding a perfect elimination ordering (PEO) for selected classes of graphs and algorithm for computing a minimal triangulation for general graphs. PEO determines a set of new edges added to the input graph.

For this thesis, we implemented five exact algorithms determing PEO for graph classes, where it can be done in polynomial time. The types of graphs are as follows: trees, outerplanar graphs, maximal outerplanar graphs, and permutation graphs. New implementation of the MCS algorithm was created. It is used to find PEO for chordal graphs and it is the starting point for later developed MCS-M algorithm for determining minimal triangulations. The MCS-M algorithm is currently the best algorithm known in the literature.

For all algorithms in this thesis corectness and real computational complexity were tested using modules unittest and timeit available in The Python Standard Library.



**Keywords:** graph triangulation, minimal triangulation, chordal graphs, chordal completion, interval graphs, permutation graphs, tree decomposition

# Spis treści

# Spis rysunków

# Listings

# 1. Wstęp

Tematem niniejszej pracy jest triangulacja grafów, nazywana inaczej znajdowaniem dopełnienia cięciwowego grafów (ang. *chordal completion*) [6]. Jest to takie dodawanie nowych krawędzi do grafu nieskierowanego, aby utworzyć graf cięciwowy [7]. Można jeszcze wymagać, aby końcowy graf cięciwowy był grafem przedziałowym, a wtedy mówimy o dopełnieniu przedziałowym (ang. *interval completion*) [8].

Istnieją dwa podstawowe rodzaje triangulacji grafów. W pierwszym przypadku minimalizuje się liczbę dodawanych nowych krawędzi, jest to problem najmniejszego wypełnienia (ang. *minimum fill-in problem*). W drugim przypadku minimalizuje się rozmiar największej kliki w końcowym grafie cięciwowym, jest to problem wyznaczania szerokości drzewowej (ang. *treewidth problem*). Można jeszcze minimalizować rozmiar największej kliki w końcowym grafie przedziałowym, wtedy jest to problem wyznaczania szerokości ścieżkowej (ang. *pathwidth problem*).

W roku 1981 Yannakakis pokazał, że problem najmniejszego wypełnienia jest NP-zupełny [9]. Podobnie w roku 1987 wykazano, że problem określenia, czy dany graf ma szerokość drzewową co najwyżej $k$, jest NP-zupełny [10]. W takiej sytuacji zwykle są trzy kierunki działań. (1) Można szukać algorytmów dokładnych, które są najmniej "wybuchowe", aby w rozsądnym czasie rozwiązywać coraz to większe instancje problemu. (2) Można szukać szybkich heurystyk znajdujących rozwiązania przybliżone, przy czym pożądane jest posiadanie oszacowań, jak daleko rozwiązanie przybliżone może być oddalone od rozwiązania dokładnego. (3) Można szukać rodzin grafów, dla których istnieją algorytmy dokładne działające w czasie wielomianowym. W niniejszej pracy zostaną przedstawione wybrane rodziny grafów i odpowiednie dla nich algorytmy.

Znajdowanie najmniejszych triangulacji (ang. *minimum triangulations*) w sensie liczby krawędzi lub szerokości drzewowej jest trudne, dlatego bada się także triangulacje minimalne (ang. *minimal triangulations*) [11]. Triangulacja najmniejsza jest minimalna, ale stwierdzenie odwrotne nie zawsze jest prawdziwe. Triangulacja minimalna nie zawiera w sobie innej triangulacji z mniejszą liczbą krawędzi.

W problemach, które wynikają z praktycznych zastosowań, zazwyczaj dąży się właśnie do zminimalizowania różnych parametrów triangulacji grafu. Na przykład problem znajdowania najmniejszej triangulacji, czyli problem najmniejszego wypełnienia, znajduje użytek w wielu dziedzinach: obliczenia na macierzach rzadkich [13], [17], zarządzanie bazami danych [14], systemy z bazą wiedzy (ang. *knowledge-based systems*) [15], czy rozpoznawanie obrazów [16].

## 1.1. Cele pracy

Głównym celem pracy jest przedstawienie problemu triangulacji grafów oraz implementacja wybranych algorytmów w języku Python. Są to algorytmy wyznaczania PEO dla różnych klas grafów: drzew, grafów zewnętrznie planarnych, czy grafów permutacji. Podanie PEO pozwala rozpoznać graf cięciwowy, a dla ogólnego grafu pozwala na wyznaczenie nowych krawędzi potrzebnych do utworzenia dopełnienia cięciwowego początkowego grafu. Stworzone na potrzeby pracy algorytmy posłużą także jako uzupełnienie biblioteki graphtheory [1].

## 1.2. Organizacja pracy

Rodział 2 przedstawia podstawowe pojęcia z teorii grafów, definicje grafów cięciwowych i przedziałowych, a także twierdzenia powiązane z problemem triangulacji omawianym w ninjeszej pracy magisterskiej, jak dekompozycja drzewowa, dekompozycja ścieżkowa, minimalne separatory wierzchołków, itp. Znajduje się tam także sekcja dotycząca problemu znajdowania dopełnienia cięciwowego dla wybranych rodzin grafów wraz z przedstawieniem funkcji istniejących w pakiecie graphtheory, a także tych stworzonych podczas pisania tej pracy algorytmów. W celu ilustracji niektórych problemów wykonywano rysunki za pomocą pakietu *NetworkX* [3]. W rozdziale 3 znajduje się implementacja algorytmów rozwiązujących problem znajdowania PEO dla wybranych rodzin grafów. Dla każdego algorytmu podano wyjaśnienie jego działania, kod źródłowy, oraz informację dotyczącą złożoności obliczeniowej. W podsumowaniu 4 przedstawiono wyniki pracy oraz końcowe wnioski. W pracy zamieszczono także dodatek A zawierający opisy sposobu testowania zaimplementowanych algorytmów oraz wykresy ich wydajności.

# 2. Teoria grafów

Przedstawimy podstawowe definicje i twierdzenia z teorii grafów, które będą wykorzystywane w całej pracy.

## 2.1. Podstawowe definicje

**Definicja:** Graf nieskierowany $G = (V, E)$ jest to para uporządkowana składająca się z niepustego zbioru wierzchołków $V(G)$ i zbioru krawędzi nieskierowanych $E(G)$. Krawędź nieskierowaną można zdefiniować jako zbiór dwóch wierzchołków ze zbioru $V(G)$. Zwyczajowo oznaczamy liczbę wierzchołków przez $n = |V(G)|$, a liczbę krawędzi przez $m = |E(G)|$. $N(v)$ to zbiór sąsiadów wierzchołka $v$. $N[v]$ to domknięte sąsiedztwo wierzchołka $v$, czyli suma $N(v) \cup \{v\}$.

**Definicja:** Podgraf $H$ grafu $G$ to graf, który zawiera wybrane wierzchołki grafu $G$, oraz może zawierać niektóre krawędzie łączące wybrane wierzchołki w grafie $G$. Formalnie zapisujemy, że $V(H) \subseteq V(g)$ oraz $E(H) \subseteq E(G)$. Inaczej można powiedzieć, że graf $H$ powstaje po usunięciu niektórych wierzchołków i krawędzi z grafu $G$. Usuwając wierzchołek należy pamiętać o usunięciu wszystkich krawędzi incydentnych z danym wierzchołkiem. Nie wymaga się, by wszystkie krawędzie grafu $G$ łączące wierzchołki znajdujące się również w podgrafie $H$ należały do niego [12].

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

**Definicja:** Klika to zbiór wierzchołków $C$ zawarty w $V(G)$, dla którego każda para wierzchołków jest połączona krawędzią. Czasem przez klikę rozumie się cały graf ze zbiorem wierzchołków $C$, a wtedy na oznaczenie kliki z $n$ wierzchołkami używa się oznaczenia jak dla grafów pełnych $K_n$. Inaczej można powiedzieć, że klika jest to podzbiór zbioru wierzchołków $V(G)$, który indukuje podgraf pełny. *Rozmiarem kliki* nazywamy liczbę wierzchołków zawartych w klice. *Klika maksymalna* nie może być rozszerzona przez dodanie kolejnego wierzchołka, w celu utworzenia większej kliki. *Klika największa* jest to klika najliczniejsza, czyli w grafie nie istnieją kliki o większej liczbie wierzchołków. Rozmiar największej kliki grafu $G$ oznaczmy $\omega(G)$.

## 2.2. Grafy cięciwowe

Graf nieskierowany jest *cięciwowy* (ang. *chordal*), jeżeli każdy cykl indukowany o długości większej lub równej cztery posiada cięciwę. Cięciwa to krawędź łącząca dwa niesąsiednie wierzchołki cyklu [7].

Wierzchołek simplicjalny $v$ to wierzchołek, dla którego graf indukowany przez $N[v]$ jest kliką. Każdy graf cięciwowy posiada wierzchołek simplicjalny, a po usunięciu tego wierzchołka otrzymujemy mniejszy graf cięciwowy. W ten sposób usuwając kolejne wierzchołki simplicjalne w coraz mniejszych grafach cięciwowych zredukujemy początkowy graf do zbioru pustego. Ciąg kolejno usuwanych wierzchołków simplicjalnych tworzy doskonałe uporządkowanie eliminacji (ang. *perfect elimination ordering, PEO*). Graf jest cięciwowy wtedy i tylko wtedy, gdy istnieje dla niego PEO. Warto zauważyć, że dla danego grafu cięciwowego może istnieć kilka różnych PEO. Druga uwaga to fakt, że bycie grafem cięciwowym jest własnością dziedziczną (ang. *hereditary property*), czyli przenosi się na podgrafy indukowane. Grafy cięciwowe były badane w pracy magisterskiej Małgorzaty Olak [23].

## 2.3. Grafy przedziałowe

**Definicja:** Graf przedziałowy to graf nieskierowany, którego wierzchołki mogą być reprezentowane przez przedziały na osi liczbowej w taki sposób, że dwa wierzchołki są sąsiednie wtedy i tylko wtedy, gdy odpowiednie przedziały na osi liczbowej mają niepuste przecięcie. Grafy przedziałowe mogą być scharakteryzowane na kilka sposobów. Jedną z pierwszych charakteryzacji podali w roku 1962 Lekkerkerker i Boland: grafy przedziałowe to grafy cięciwowe, które nie zawierają trójek asteroidalnych [25]. Oznacza to, że wybierając dowolne trzy wierzchołki grafu przedziałowego możemy je uporządkować w taki sposób, że każda ścieżka od pierwszego do trzeciego wierzchołka będzie przechodzić przez sąsiedztwo drugiego wierzchołka. Grafy przedziałowe były badane w pracy magisterskiej Macieja Mularskiego [26].

## 2.4. Dekompozycja drzewowa

**Definicja:** Mając dany graf nieskierowany $G$, dekompozycja drzewowa (ang. *tree decomposition*, TD) jest to para $(X, T)$, gdzie $X = \{X_1, ..., X_n\}$ jest rodziną podzbiorów zbioru wierzchołków $V$ oraz $T$ jest drzewem, którego węzły są workami $X_i$ spełniającymi następujące właściwości:

1. Suma wszystkich worków $X_i$ jest równa $V$, to znaczy, że każdy wierzchołek grafu jest związany z co najmniej jednym węzłem tego drzewa.
2. Dla każdej krawędzi grafu $vw$, istnieje worek $X_i$ zawierający oba wierzchołki $v$ oraz $w$. To oznacza, że wierzchołki sąsiadują ze sobą tylko wtedy, gdy odpowiednie poddrzewa mają wspólny węzeł.
3. Jeżeli dwa worki $X_i$ oraz $X_j$ zawierają wierzchołek $v$, wtedy wszystkie worki $X_k$ drzewa na ścieżce pomiędzy $X_i$ oraz $X_j$ zawierają także ten

wierzchołek (w drzewie istnieje dokładnie jedna ścieżka między $X_i$ oraz $X_j$). Ta zależność znana jest również pod pojęciem koherencji.

Szerokością danej dekompozycji drzewowej nazywamy liczność największego worka minus jeden. Dla grafu $G$ szerokość drzewową (ang. *treewidth*) oznacza się przez $\mathrm{tw}(G)$ i jest to najmniejsza szerokość ze wszystkich możliwych dekompozycji drzewowych. Dokładne omówienie tematu dekompozycji drzewowej wraz z przykładami i zastosowaniami znajduje się w pracy magisterskiej Macieja Niezabitowskiego [24].

## 2.5. Dekompozycja ścieżkowa

**Definicja:** Dla danego grafu nieskierowanego $G$ sekwencja $(X_i, ..., X_r)$ podzbiorów $V(G)$ jest dekompozycją ścieżkową (ang. *path decomposition*, PD) grafu $G$, jeżeli spełnione są następujące warunki:

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

**Definicja:** Szerokością ścieżkową grafu nieskierowanego $G$ (ang. *pathwidth*) jest najmniejsza liczba $k \geqslant 0$ taka, że $G$ posiada dekompozycję ścieżkową $(X_1, ..., X_r)$, przy czym $|X_i| \leqslant k + 1$ dla każdego $1 \leqslant i \leqslant r$. Szerokość ścieżkową oznaczamy przez $\mathrm{pw}(G) = k$.

## 2.6. Minimalne separatory wierzchołków

W badaniach nad wyznaczaniem szerokości drzewowej grafów i wieloma innymi problemami dużą rolę odgrywają minimalne separatory wierzchołków (ang. *minimal vertex separators*). Istnienie wydajnych algorytmów jest powiązane z istnieniem separatorów wierzchołków o ograniczonym rozmiarze lub wielomianową liczbą tych separatorów. W 1993 roku pokazano nawet, że jeżeli dla pewnej rodziny grafów mamy algorytm wielomianowy wyznaczający zbiór wszystkich minimalnych separatorów, to istnieją algorytmy wielomianowe rozwiązujące problem szerokości drzewowej i problem najmniejszego wypełnienia [27]. Poniżej przytoczymy definicje i najważniejsze fakty dotyczące separatorów.

**Definicja:** Mamy dany graf nieskierowany $G$. Podzbiór $S$ zawarty w $V(G)$ jest $(u, v)$-separatorem dla niesąsiednich wierzchołków $u$ i $v$, jeżeli usunięcie $S$ z grafu $G$ rozdziela $u$ i $v$, czyli wierzchołki znajdą się w różnych składowych spójnych grafu. Jeżeli nie istnieje podzbiór właściwy zbioru $S$, który byłby również $(u, v)$-separatorem, to wtedy $S$ nazywamy minimalnym

$(u, v)$-separatorem. Podzbiór $S$ nazywamy minimalnym separatorem, jeżeli jest $(u, v)$-separatorem dla pewnych niesąsiednich wierzchołków $u$ i $v$.

**Drzewa:** W drzewach minimalne separatory to pojedyncze wierzchołki, które nie są liśćmi. Drzewo ma $O(n)$ minimalnych separatorów.

**Grafy cykliczne:** W grafie cyklicznym $C_n$ każde dwa niesąsiednie wierzchołki tworzą minimalny separator. Można wybrać $n(n-3)/2$ takich par, więc będzie $O(n^2)$ minimalnych separatorów.

**Grafy cięciwowe i grafy przedziałowe:** W grafach cięciwowych każdy minimalny separator jest kliką [28], a liczba minimalnych separatorów jest rzędu $O(n)$. Wiadomo, że w grafach cięciwowych liczba klik maksymalnych jest rzędu $O(n)$. W drzewie dekompozycji kliki maksymalne stają się wierzchołkami drzewa (workami), a minimalne separatory będą częścią wspólną sąsiednich worków.

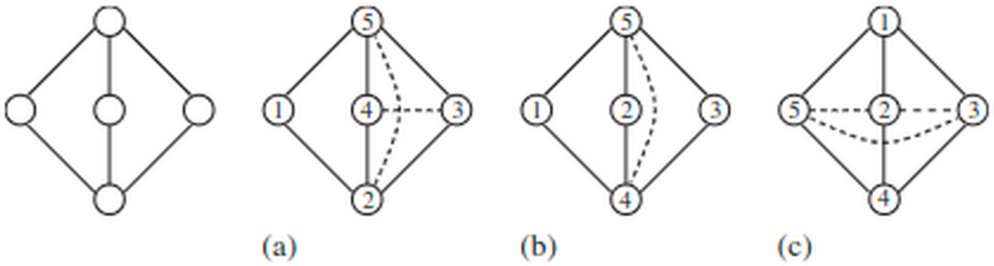
**Grafy permutacji:** Rozważmy model grafu permutacji z dwoma prostymi równoległymi i odcinkami łączącymi punkty z taką samą etykietą, leżące na obu prostych. Dwa wierzchołki niesąsiednie odpowiadają dwóm odcinkom, które nie przecinają się. Wobec tego prosta przechodząca pomiędzy tymi odcinkami przetnie odcinki tworzące pewien separator. W całym grafie można zrobić co najwyżej $O(n^2)$ różnych cięć, a więc liczba minimalnych separatorów musi być ograniczona przez $O(n^2)$.

**Grafy kołowe (ang. circle graphs):** Rozważmy model grafu kołowego z cięciwami na pewnym okręgu. Dwa niesąsiednie wierzchołki odpowiadają dwóm cięciwom, które nie przecinają się. Wobec tego prosta przechodząca pomiędzy tymi cięciwami przetnie cięciwy tworzące pewien separator. W całym grafie można zrobić co najwyżej $O(n^2)$ różnych cięć, a więc liczba minimalnych separatorów musi być ograniczona przez $O(n^2)$.

## 2.7. Minimalna i najmniejsza triangulacja

**Definicja:** Triangulacja najmniejsza (ang. *minimum triangulation*), nazywana również najmniejszym wypełnieniem (ang. *minimum fill-in*), to taka triangulacja, w której w celu uzyskania grafu cięciwowego dodajemy jak najmniejszą liczbę nowych krawędzi. Znajdowanie najmniejszej triangulacji jest problemem NP-trudnym [18]. Z drugiej strony, istnieje wiele ważnych zastosowań najmniejszej triangulacji, więc w odpowiedzi na zapotrzebowanie na rozwiązanie tego problemu rozważa się jego alternatywę, dla której istnieją algorytmy działające w czasie wielomianowym - jest to triangulacja minimalna. Różnicę między triangulacją najmniejszą a minimalną przedstawia rysunek 2.1.

**Definicja:** Triangulacja minimalna (ang. *minimal triangulation*) jest rezultatem dodania minimalnego zbioru krawędzi przy tworzeniu triangulacji danego grafu [11]. Wiadomo, że problem minimalnej triangulacji jest ściśle powiązany z minimalnymi separatorami grafu wejściowego. Minimalna triangulacja jest uzyskiwana przez dodawanie krawędzi tylko do minimalnych separatorów, dopełniając je w kliki. Minimalne separatory mogą charakteryzować klasy grafów: jak pokazał Dirac [28], graf jest cięciwowy wtedy i tylko wtedy, gdy jego minimalne separatory tworza klikę [41].




Rysunek 2.1. Różne rodzaje triangulacji na przykładzie danego grafu [11]: a) triangulacja nie-minimalna [zbędna krawędź $(3, 4)$], b) triangulacja najmniejsza, c) triangulacja minimalna. Linie przerywane reprezentują krawędzie dodawane przy triangulacji.

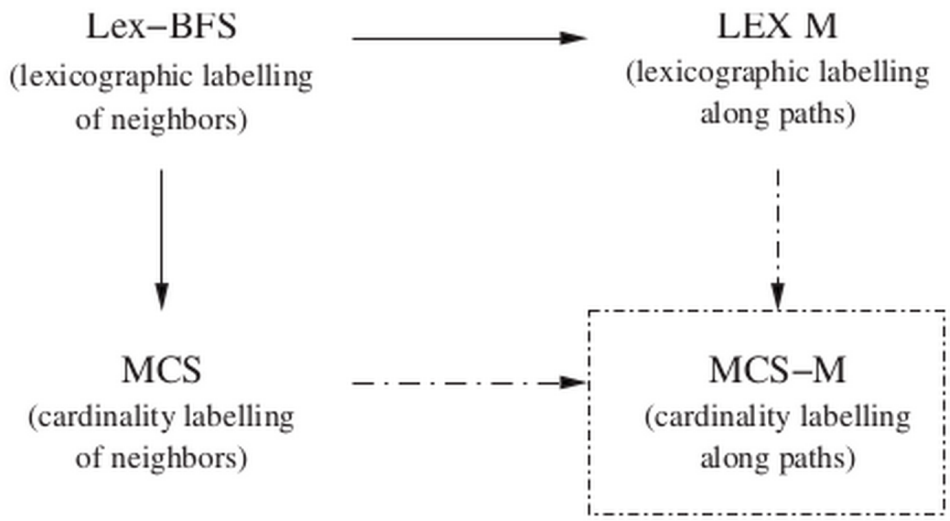
Rozpatrywanymi w literaturze algorytmami wyznaczającymi minimalną triangulację są Lex M i MCS-M [19]. Algorytm Lex M [22], to rozbudowana wersja algorytmu Lex-BFS - leksykograficzne przeszukiwanie wszerz (ang. *lexicographical breadth first search*) służącego do rozpoznawania grafów cięciowowych. Zarówno Lex-BFS jaki i Lex M używają leksykograficznych oznaczeń dla wierzchołków jeszcze nieprzetworzonych. W miarę kontynowania przetwarzania etykiety wierzchołków rosną, a każda z nich potencjalnie osiąga długość proporcjonalną do liczby wierzchołków w grafie. Lex-BFS dodaje do etykiet sąsiadów przetwarzanego wierzchołka, podczas gdy Lex M dodaje do etykiet zarówno sąsiadów przetwarzanego wierzchołka jak i inne wierzchołki osiągalne po specjalnych rodzajach ścieżek z przetwarzanego wierzchołka. Te specjalne rodzaje ścieżek są dokładnie tymi ścieżkami, które tworzą wypełnienie, a etykiety Lex M są dokładnie sąsiadami o wyższych numerach w bieżącym wypełnionym grafie. Proste rozszerzenie dodawania do etykiet na podstawie osiągalności wzdłuż takich ścieżek wypełnienia, a nie tylko wzdłuż pojedynczych krawędzi skutkuje algorytmem dającym właśnie minimalne triangulacje.

Koncepcje etykietowania przyległego (ang. *adjacency labelling concepts*) opracowane dla Lex-BFS okazały się kluczowe w zrozumieniu grafów cięciwowych i triangulacji. Tarjan i Yannakakis doszli później do zaskakującego wniosku, że w przypadków rozpoznawania grafów cięciwowych znajomość konkretnych przetworzonych sąsiadów (ich etykiet) nie jest konieczna, wystarczy zachować i porównać liczbę przetworzonych sąsiadów [21]. Było to ważne osiągnięcie skutkujące znacznie prostszą implementacją Lex-BFS, zna-

ną jako algorytm MCS - przeszukiwanie największej liczności (ang. *maximum cardinality search*), którego implementacja i opis znajduje się w rozdziale 3. Listing ukazuje sposób korzystania z funkcji find_peo_mcs.

```
# Wyznaczanie PEO dla grafu cieciwowego za pomoca MCS.
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from mcs import find_peo_mcs

G = Graph()
edge_list = [Edge("A","B"), Edge("B","C"), Edge("C","D"),
    Edge("A","D"), Edge("A","C")]
for edge in edge_list:
    G.add_edge(edge)
G.show()
order = find_peo_mcs(G)
print(order)
```

Odkrycie to pozwoliło również na znalezienie uproszczonego odpowiednika algorytmu Lex M do wyznaczania minimalnych triangulacji, również wykorzystującego przeszukiwanie największej liczności [18]. Implementacja oraz opis algorytmu MCS-M znajduje się w rozdziale 3. Zależności pomiędzy algorytmami Lex-BFS, Lex M, MCS, oraz MCS-M ukazuje rysunek 2.2.




Rysunek 2.2. Zależność pomiędzy algorytmami Lex-BFS, Lex M, MCS, oraz MSC-M. Algorytmy po lewej stronie rysunku służą do rozpoznawania grafów cięciwowych, algorytmy po prawej stronie służą to wyznaczanie minimalnej triangulacji dowolnego grafu [18].

Poniższy listing ukazuje sposób korzystania z klasy MCS_M do zwracania wyznaczonego PEO oraz krawędzi, które zostaly dodane do grafu w wyniku procedury minimalnej triangulacji.

```
# Wyznaczanie minimalnej triangulacji dla ogolnego grafu.
from graphtheory.structures.edges import Edge
```

```
from graphtheory.structures.graphs import Graph
from mcsmpeo import MCS_M

G = Graph()
edge_list = [Edge("A","B"), Edge("B","C"), Edge("C","D"),
    Edge("D","E"), Edge("A","E")]    # graf cykliczny C_5
for edge in edge_list:
    G.add_edge(edge)
G.show()
algorithm = MCS_M(G)
algorithm.run()
print(algorithm.order)    # PEO for minimal triangulation
print(algorithm.new_edges)
```

## 2.8. Znajdowanie dopełnienia cięciwowego

Strukturę grafu cięciwowego można opisać przez podanie PEO oraz drzewa dekompozycji zawierającego wszystkie kliki maksymalne. Dla grafu, który nie jest cięciwowy, można podać PEO dopełnienia cięciwowego, co jednoznacznie wyznacza zbiór krawędzi dodanych do początkowego grafu. Poniżej zestawimy narzędzia, którymi dysponujemy w pakiecie graphtheory do analizy problemu dopełnienia cięciwowego dla różnych rodzin grafów.

### 2.8.1. Ogólne grafy

Dla ogólnych grafów możemy podać ciąg wierzchołków, który wyznaczy jednoznacznie pewne dopełnienie cięciwowe za pomocą funkcji find_td_order(). W literaturze podano kilka heurystyk do znajdowania obiecujących ciągów wierzchołków i szacowania szerokości drzewowej. Jest to m.in. heurystyka najmniejszego stopnia (górne oszacowanie na szerokość drzewową) i heurystyka największego minimalnego stopnia (dolne oszacowanie na szerokość drzewową).

```
# Wyznaczanie drzewa dekompozycji (TD) dla ogolnych grafow.
from graphtheory.chordality.tdtools import find_td_order
from graphtheory.chordality.tdtools import find_treewidth_min_deg
from graphtheory.chordality.tdtools import find_treewidth_mmd

# G is a general connected undirected graph.
# 'order' is a selected sequence of nodes.

assert len(order) == G.v()
T = find_td_order(G, order)    # finding TD (heuristic)
treewidth = max(len(bag) for bag in T.iternodes()) - 1 # upper bound

# Finding 'order'.
treewidth, order = find_treewidth_min_deg(G) # upper bound for tw
treewidth, order = find_treewidth_mmd(G) # lower bound for tw
```

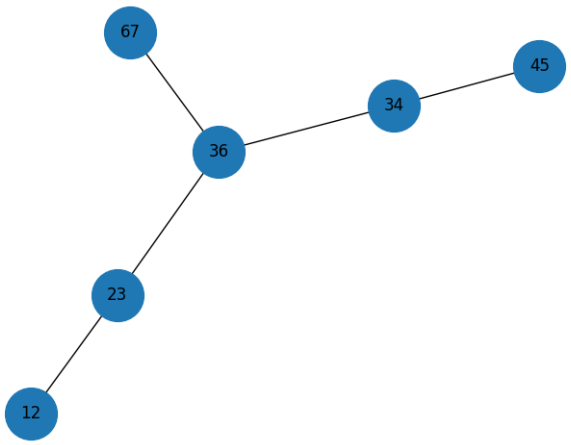
### 2.8.2. Drzewa

Drzewa są to grafy nieskierowane, które można zdefiniować na kilka sposobów: (1) drzewo to graf spójny acykliczny; (2) drzewo to graf, w którym każde dwa różne wierzchołki są połączone dokładnie jedną ścieżką prostą; (3) drzewo to graf spójny, który po usunięciu dowolnej krawędzi przestaje być spójny.

Drzewa są w trywialny sposób cięciwowe, ponieważ nie posiadają cykli. Każda krawędź drzewa stanowi klikę $K_2$ i to są wszystkie kliki maksymalne. Szerokość drzewowa dla drzew wynosi jeden. Wierzchołki simplicjalne to liście, czyli wierzchołki stopnia 1. PEO dla drzewa można wyznaczyć algorytmem odrywania liści, który jest także podstawą algorytmu wyznaczania największego zbioru niezależnego. W ramach tej pracy wyodrębniono algorytm odrywania liści do osobnej funkcji find_peo_tree(), zwracającej PEO dla drzewa. Listing przedstawia sposób korzystania z funkcji. Kod źródłowy funkcji jest opisany w rozdziale 3.

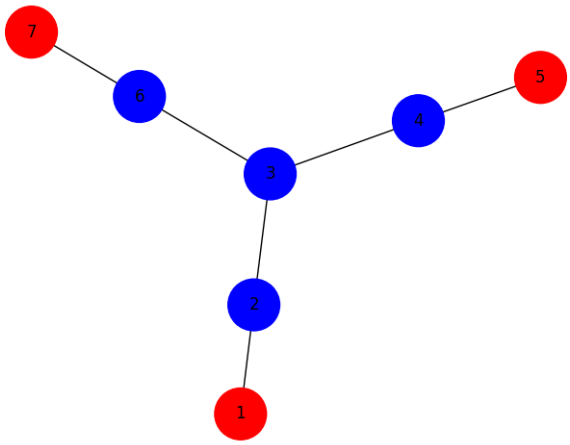
```
# Wyznaczanie PEO dla drzewa.
from graphtheory.structures.factory import GraphFactory
from treepeo import find_peo_tree

gf = GraphFactory(Graph)
G = gf.make_tree(n=10)    # random tree
assert G.v() == G.e() + 1
peo = find_peo_tree(G)    # list of nodes
assert len(peo) == G.v()
```

Wiele drzew jest grafami przedziałowymi, a wtedy ich szerokość ścieżkowa wynosi 1. Rysunek 2.3 przedstawia najmniejsze drzewo, które nie jest przedziałowe. Drzewo ma $n = 7$ wierzchołków, a jego szerokość ścieżkowa wynosi 2. Na rysunku zaznaczono trzy wierzchołki, które tworzą trójkę asteroidalną.

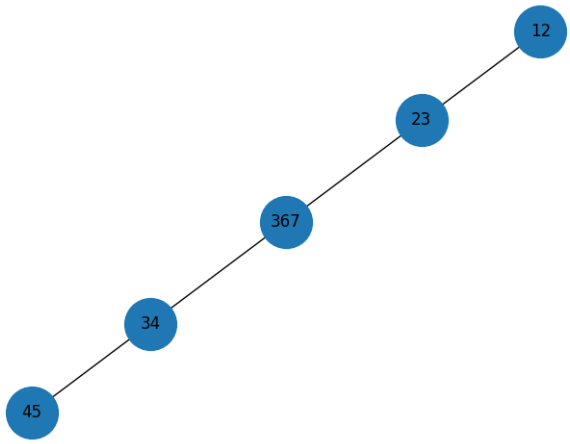



Rysunek 2.5. Szerokość drzewowa dla drzewa 2.3 wynosi 1.

Algorytm liniowy wyznaczania szerokości ścieżkowej dla drzew został podany przez Scheffler [29].




Rysunek 2.3. Najmniejsze drzewo, które nie jest przedziałowe, z zaznaczonymi wierzchołkami tworzącymi trójkę asteroidalną.




Rysunek 2.4. Szerokość ścieżkowa dla drzewa 2.3 wynosi 2.

### 2.8.3. Grafy cięciwowe

Znalezienie PEO dla grafu nieskierowanego jest rozpoznaniem, że dany graf jest cięciwowy. Istnieją co najmniej trzy metody wyznaczania PEO: (1) odrywanie wierzchołków simplicjalnych, (2) użycie MCS (ang. *maximum cardinality search*), (3) użycie LexBFS (ang. *lexicographic breadth-first search*). Należy zauważyć, że dwie ostatnie metody wyznaczają ciąg wierzchołków dla dowolnego grafu i należy osobno potwierdzić, że znaleziony ciąg jest rzeczywiście PEO, a co za tym idzie, że mamy graf cięciwowy.

```
# Wyznaczanie PEO i TD dla grafu cieciwowego.
from graphtheory.chordality.chordaltools import make_random_chordal
from graphtheory.chordality.peotools import find_peo_lex_bfs
from graphtheory.chordality.peotools import find_peo_mcs
from graphtheory.chordality.peotools import is_peo1
from graphtheory.chordality.tdtools import find_td_chordal

G = make_random_chordal(n=10)
peo = find_peo_lex_bfs(G)   # if G is not chordal, then 'peo' is not PEO
peo = find_peo_mcs(G)   # if G is not chordal, then 'peo' is not PEO
assert is_peo1(G, peo)   # testing PEO, O(V+E) time

T = find_td_chordal(G, peo)   # finding a tree decomposition (TD)
assert isinstance(T, Graph)
assert T.v() == T.e() + 1   # tree
treewidth = max(len(bag) for bag in T.iternodes()) - 1
```

Wyznaczenie szerokości ścieżkowej dla grafu cięciwowego jest problemem NP-trudnym [36].

### 2.8.4. Grafy przedziałowe

Grafy przedziałowe są cięciwowe, więc można wyznaczyć pewne PEO i drzewo dekompozycji metodami przygotowanymi dla grafów cięciwowych. Niestety nie ma wtedy gwarancji, że otrzymane drzewo dekompozycji będzie grafem ścieżką. Musimy więc postąpić inaczej. Z grafem przedziałowym jest związana jego reprezentacja przedziałowa, którą można zakodować jako podwójną permutację. Dla każdego wierzchołka grafu (przedziału) ustalamy etykietę, a następnie przechodząc wzdłuż prostej zapisujemy etykietę dla każdego początku lub końca przedziału. Dla $n$ przedziałów ciąg etykiet będzie miał długość $2n$.

Jeżeli dla grafu przedziałowego znana jest jego reprezentacja przedziałowa, to wtedy łatwo można wyznaczyć ciąg klik maksymalnych, które budują worki w dekompozycji ścieżkowej.

```
# Wyznaczanie PEO i TD dla grafu przedzialowego.
# Graf przedzialowy dany jako podwojna permutacja.
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.chordality.intervaltools import make_tepee_interval
from graphtheory.chordality.intervaltools import find_peo_cliques

perm = make_tepee_interval(n=10)   # generator grafu tepee
peo, cliques = find_peo_cliques(perm)   # PEO and ordered cliques
```

```
assert 2 * len(peo) == len(perm)
pathwidth = max(len(c) for c in cliques) - 1

# Wyznaczanie dekompozycji sciezkowej.
bags = [tuple(sorted(c)) for c in cliques]
T = Graph(n=len(bags))    # path decomposition
for bag in bags:
    T.add_node(bag)
for i in range(len(bags)-1):
    T.add_edge(Edge(bags[i], bags[i+1]))
assert T.v() == T.e() + 1    # tree
```

Jeżeli dla grafu przedziałowego nie znamy jego reprezentacji przedziałowej, to wtedy ogólnymi metodami dla grafów cięciwowych wyznaczamy pewne PEO, wyznaczamy zbiór klik maksymalnych, wyznaczamy uporządkowanie klik maksymalnych, a stąd można odczytać reprezentację przedziałową i dekompozycję ścieżkową.

```
# Wyznaczanie PEO i TD dla grafu przedzialowego.
# Graf przedzialowy bez reprezentacji przedzialowej.
from graphtheory.chordality.peotools import find_peo_lex_bfs
from graphtheory.chordality.peotools import find_peo_mcs
from graphtheory.chordality.peotools import is_peo1
from graphtheory.chordality.peotools import find_all_maximal_cliques

# G is an interval graph.
peo = find_peo_lex_bfs(G)
#peo = find_peo_mcs(G)
assert is_peo1(G, peo)
cliques = find_all_maximal_cliques(G, peo) # list of sets
pathwidth = max(len(c) for c in cliques) - 1

# Porzadkowanie klik maksymalnych wg pracy mgr Macieja Mularskiego.
```

### 2.8.5. Grafy zewnętrznie planarne

Grafy zewnętrznie planarne są podklasą grafów planarnych. Pierwsza wzmianka o tych grafach pojawia się w literaturze już w 1969 roku, w książce [35]. Graf planarny możemy nazwać zewnętrznie planarnym, jeśli można narysować go tak, by wszystkie jego wierzchołki leżały na tej samej ścianie, a ścianę tą nazywamy wtedy ścianą zewnętrzną [33] [34]. Do tej pory w literaturze nie pojawiały się algorytmy mające na celu konkretnie wyznaczania PEO dla tej klasy grafów. W ramach tej pracy, na podstawie algorytmu do rozpoznawania grafów zewnętrznie planarnych poprzez kolorowanie krawędzi zaproponowanego w pracy [32] przedstawiono taki algorytm. Listing przedstawia sposób korzystania z niego, a jego kod źródłowy oraz dokładny opis znajduje się w rozdziale 3.

```
# Wyznaczanie PEO dla grafu zewnetrznie planarnego.
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.planarity.outerplanarpeo import OuterplanarPEO

# tworzenie wybranego grafu zewnetrznie planarnego
```

```
N = 9
G = Graph(n=N, directed=False)
nodes = range(N)
edges = [Edge(0, 1), Edge(0, 5), Edge(1, 2), Edge(1, 3), Edge(1, 4),
         Edge(1, 5), Edge(2, 3), Edge(3, 4), Edge(4, 5)]
for node in nodes:
    G.add_node(node)
for edge in edges:
    G.add_edge(edge)
# wyznaczanie PEO dla danego grafu
algorithm = OuterplanarPEO(self.G)
peo = algorithm.run()
print(peo)
```

W klasie grafów zewnętrznie planarnych możemy wyznaczyć grafy maksymalnie zewnętrznie planarne. Są to grafy zewnętrznie planarne z maksymalną możliwą liczbą krawędzi. Podobnie jak w ogólniejszym przypadku grafów zewnętrznie planarnych, tak i tutaj, algorytmów do wyznaczanie PEO dla tej klasy grafów na próżno szukać w literaturze. Ze względu na fakt, że w przeciwieństwie do grafów zewnętrznie planarnych grafy te są w swej naturze cięciwowe [30], skorzystano ze znacznie mniej złożonego algorytmu, by wyznaczać ich PEO. Listing prezentuje przykład korzystania z wyżej wymienionego algorymu, a jego kod źródłowy oraz dokładny opis znajduje się w rozdziale 3.

```
# Wyznaczanie PEO dla grafu maksymalnie zewnetrznie planarnego.
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.planarity.outerplanarpeo import MaximalOuterplanarPEO

# tworzenie wybranego grafu zewnetrznie planarnego
N = 4
G = Graph(n=N)
E = [Edge(0, 1), Edge(0, 3), Edge(0, 2), Edge(1, 3), Edge(2, 3)]
for node in range(N):
    G.add_node(node)
for edge in E:
    G.add_edge(edge)
# wyznaczanie PEO dla danego grafu
algorithm = MaximalOuterplanarPEO(self.G)
peo = algorithm.run()
print(peo)
```

### 2.8.6. Grafy szeregowo-równoległe

Grafy szeregowo-równoległe (sp-grafy) to grafy nieskierowane planarne z wyróżnionymi dwoma wierzchołkami. Grafy te można wygenerować rekurencyjnie za pomocą dwóch operacji: szeregowej i równoległej. Czasem dodaje się jeszcze trzecią operację o nazwie *jackknife*. Analiza algorytmów dla sp-grafów znajduje się w pracy magisterskiej Konrada Gałuszki [37].

Każdy sp-graf jest częściowym 2-drzewem, a jego uzupełnienie cięciwowe to pełne 2-drzewo o szerokości drzewowej równej 2. Algorytm znajdowania PEO najpierw usuwa wszystkie wierzchołki stopnia 2. Jeżeli sąsiedzi usuwa-

nego wierzchołka nie są połączeni krawędzią, to dodawana jest nowa krawędź do uzupełnienia cięciwowego. Na końcu może zdażyć się jedna z trzech sytuacji. (1) Gdy pozostaje pojedyncza krawędź, to końce krawędzi są ostatnimi wierzchołkami w PEO. (2) Gdy pozostaje gwiazda, czyli graf $K_{1,p}$, $p > 2$, to oznacza, że sp-graf został zbudowany z użyciem dodatkowej operacji *jack-knife*. (3) Gdy pozostaje inny graf, to oznacza, że wejściowy graf nie był sp-grafem.

```
# Wyznaczanie PEO dla sp-grafu.
from graphtheory.seriesparallel.sptools import make_random_ktree
from graphtheory.seriesparallel.sptools import make_random_spgraph
from graphtheory.seriesparallel.sptools import find_peo_spgraph

# Generowanie sp-grafu.
G = make_random_spgraph(n=10)    # random partial 2-tree
#G = make_random_ktree(n=10, k=2)    # random 2-tree

# Wyznaczanie PEO.
peo = find_peo_spgraph(G)  # list of nodes
print(peo)
```

### 2.8.7. Grafy Halina

Grafy Halina to grafy planarne badane przez niemieckiego matematyka Rudolfa Halina w roku 1971. Od tego czasu utrzymuje się spore zainteresowanie tymi grafami. Analiza grafów Halina znajduje się w pracy magisterskiej Aleksandra Krawczyka [38]. Grafy Halina tworzy się na bazie płaskiego rysunku drzewa przez połączenie liści w cykl zewnętrzny, poruszając się zgodnie z ruchem wskazówek zegara. Drzewo musi zawierać co najmniej cztery wierzchołki i nie może zawierać wierzchołków stopnia dwa.

Rozpoznawanie grafu Halina polega m.in. na wyznaczeniu zbioru wierzchołków należących do cyklu zewnętrznego. Dalej korzystając z tej informacji można wyznaczyć PEO dopełnienia cięciwowego i drzewo dekompozycji. Szerokość drzewowa grafów Halina wynosi 3, co można obliczyć na bazie drzewa dekompozycji. Grafy Halina mają strukturę rekurencyjną, co wykorzystuje się przy wyznaczaniu drzewa dekompozycji. Bazowe grafy Halina to grafy koła $W_n$. Pozostałe grafy Halina posiadają dwa tzw. wachlarze. Redukując kolejne wachlarze można sprowadzić każdy graf Halina do pewnego grafu koła.

```
# Wyznaczanie PEO i TD dla grafu Halina.
from graphtheory.structures.graphs import Graph
from graphtheory.planarity.halintools import make_halin
from graphtheory.planarity.halintools import make_halin_cubic
from graphtheory.planarity.halintools import make_halin_outer
from graphtheory.planarity.halintools import make_halin_cubic_outer
from graphtheory.structures.factory import GraphFactory
from graphtheory.planarity.halinpeo import HalinGraphPEO
from graphtheory.planarity.halintd import HalinGraphTreeDecomposition

# Generowanie grafu Halina bez podawania cyklu zewnetrznego.
gf = GraphFactory(Graph)
#G = gf.make_wheel(n=10)        # wheel graph W_n
```

```
#G = gf.make_necklace(n=10)    # necklace graph, n even
#G = make_halin(n=7)
#G = make_halin_cubic(n=10)    # cubic Halin graph, n even

# Generowanie grafu Halina ze zbiorem wierzcholkow nalezacych do cyklu.
G, outer = make_halin_outer(n=10)
#G, outer = make_halin_cubic_outer(n=10)    # cubic Halin graph, n even

# Wyznaczanie PEO dla chordal completion.
algorithm = HalinGraphPEO(G, outer)
algorithm.run()
print(algorithm.parent)  # inner tree (dict)
print(algorithm.order)   # PEO (list)

# Wyznaczanie TD.
algorithm = HalinGraphTreeDecomposition(G, outer)
algorithm.run()
print(algorithm.parent)     # inner tree (dict)
print(algorithm.order)   # PEO (list)
print(algorithm.cliques)    # list of sets
print(algorithm.td)    # TD as a graph

assert algorithm.td.v() == algorithm.td.e() + 1    # tree
treewidth = max(len(bag) for bag in algorithm.td.iternodes()) - 1
assert treewidth == 3
```

### 2.8.8. Grafy permutacji

Grafy permutacji to grafy, których wierzchołki reprezentują elementy permutacji $n$ liczb. Krawędzie łączą te pary wierzchołków, dla których w permutacji występuje inwersja. Grafy permutacji były badane w pracy licencjackiej Alberta Surmacza [39]. Dla grafu permutacji można podać przydatny model geometryczny z dwoma poziomymi prostymi równoległymi. Na górnej prostej zaznacza się punkty z kolejnymi etykietami od 1 do $n$. Na dolnej prostej zaznacza się punkty z etykietami w kolejności wyznaczonej przez permutację. Punkty z tymi samymi etykietami łączy się odcinkiem. Wtedy liczby tworzące inwersję w permutacji odpowiadają przecinającym się odcinkom.

Bodlaender, Kloks i Kratsch pokazali, że w przypadku grafów permutacji szerokość ścieżkowa jest równa szerokości drzewowej, $\mathrm{tw}(G) = \mathrm{pw}(G)$ [40]. Podano algorytm wyznaczający te wielkości, który wykorzystuje minimalne separatory i linie skanujące. Złożoność obliczeniowa algorytmu wynosi $O(nk)$, gdzie $k = \mathrm{pw}(G)$.

W roku 2010 Meister podał algorytmy działające w czasie liniowym na wyznaczanie szerokości drzewowej i najmniejsze wypełnienie dla grafów permutacji [41].

W ramach pracy przygotowano algorytm zachłanny wyznaczający PEO dla grafu permutacji lub dla jego dopełnienia cięciwowego, przy czym graf musi być podany w postaci permutacji $n$ liczb. Algorytm zaimplementowano w postaci klasy PermGraphPEO. Listing przedstawia sposób korzystania z klasy. Kod źródłowy klasy jest opisany w rozdziale 3.

```
# Wyznaczanie PEO i TD dla grafu permutacji algorytmem zachlannym.
```

```
from graphtheory.chordality.peotools import is_peo1
from graphtheory.chordality.tdtools import find_td_chordal
from graphtheory.permutations.permtools import make_bipartite_perm
from graphtheory.permutations.permtools import make_star_perm
from graphtheory.permutations.permtools import make_path_perm
from permpeo2 import PermGraphPEO

# Generowanie grafu permutacji w postaci permutacji liczb od 0 do n-1.
#perm = make_star_perm(n=10)   # make perm for K_{1,n-1} graph, treewidth=1
#perm = make_path_perm(n=10)   # make perm for P_n graph, treewidth=1
perm = make_bipartite_perm(p=4, q=5)   # make perm for K_{p,q} graph
algorithm = PermGraphPEO(perm)
algorithm.run()
# algorithm.graph is a chordal completion
assert is_peo1(algorithm.graph, algorithm.order)   # testing PEO
print(algorithm.new_edges)
print(algorithm.order)   # PEO for chordal completion

T = find_td_chordal(algorithm.graph, algorithm.order)   # finding TD
assert T.v() == T.e() + 1   # tree
treewidth = max(len(bag) for bag in T.iternodes()) - 1
```

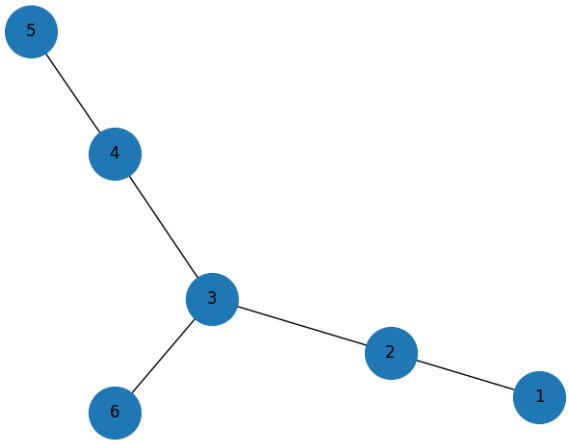
### 2.8.9. Grafy bez trójek asteroidalnych

Trójką asteroidalną (ang. *asteroidal triple, AT*) w grafie nazywamy trzy parami niezależne wierzchołki, gdzie pomiędzy dowolnymi dwoma istnieje ścieżka unikająca sąsiedztwa trzeciego wierzchołka [43]. Grafy bez trójek asteroidalnych (ang. *AT-free graphs*) zawierają ważne rodziny grafów, takie jak grafy przedziałowe, grafy permutacji, grafy trapezoidalne.

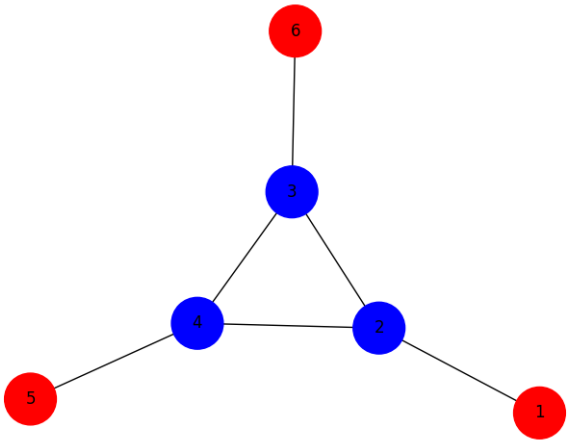
W roku 1996 Möhring pokazał, że dla grafów bez AT minimalna (pod względem inkluzji) triangulacja prowadzi do grafów przedziałowych [42]. Oznacza to, że dla tych grafów szerokość drzewowa (ang. *treewidth*) jest równa szerokości ścieżkowej (ang. *pathwidth*). Ponadto problem najmniejszego wypełnienia pokrywa się z problemem wyznaczania szerokości drzewowej i ścieżkowej [44]. W grafie bez AT jedyne możliwe cykle indukowane to $C_3$, $C_4$ i $C_5$, ponieważ cykl $C_6$ i większy posiada AT. Möhring zrobił dowód nie wprost, czyli dowód nie jest konstruktywny i nie podaje sposobu otrzymania triangulacji minimalnej. Warto zauważyć, że dodawanie nowych krawędzi do grafu bez AT może wytworzyć AT (rysunki 2.6 i 2.7). Podobnie usuwanie pewnych krawędzi z grafu bez AT może wytworzyć AT (rysunki 2.8 i 2.9).

### 2.8.10. Grafy kołowe

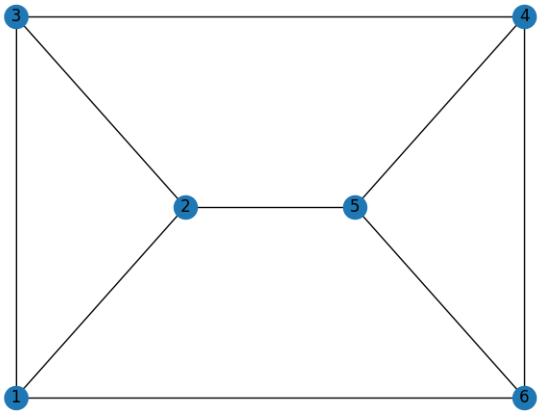
Uogólnieniem grafów permutacji są grafy kołowe (ang. *circle graphs*). Można je opisać jako grafy przecięć cięciw na okręgu. Analiza grafów kołowych znajduje się w pracy magisterskiej Alberta Surmacza [45]. W roku 1993 Kloks podał algorytm działający w czasie $O(n^3)$ na obliczanie szerokości drzewowej grafu kołowego [46]. Metoda obliczania szerokości drzewowej wykorzystuje triangulację pewnego wielokąta zbudowanego na bazie danego okręgu z cięciwami.



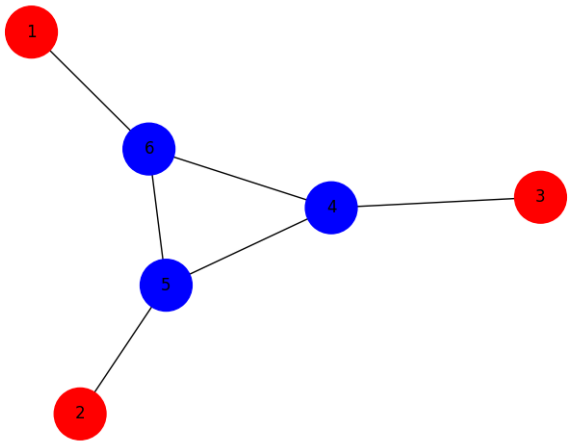

Rysunek 2.6. Graf (drzewo) bez trójki asteroidalnej.




Rysunek 2.7. Graf z rysunku 2.6 z wytworzoną trójką asteroidalną przez dodanie krawędzi $(2, 4)$. Dodanie nowej krawędzi do grafu bez AT może wytworzyć AT.




Rysunek 2.8. Graf bez trójki asteroidalnej (graf Halina, *3-prism*).




Rysunek 2.9. Graf z rysunku 2.8 z wytworzoną trójką asteroidalną przez usunięcie krawędzi $(1,2)$, $(1,3)$, $(2,3)$. Usunięcie krawędzi z grafu bez AT może wytworzyć AT.

# 3. Algorytmy

W tym rozdziale przedstawione zostały implementacje oraz opisy działania algorytmów wraz z wyszczególnionymi danymi wejściowymi, wyjściowymi, oraz uwagami o złożoności obliczeniowej. Są to algorytmy wyznaczania PEO dla wybranych klas grafów, takich jak drzewa, grafy zewnętrznie planarne, grafy maksymalnie zewnętrznie planarne, oraz grafy permutacji. Wyodrębiono również algorytm MCS-M wyznaczania PEO dla minimalnej triangulacji dla grafów dowolnej klasy, a także przedstawiono nową implementację algorytmu MCS, na którym bazuje algorytm MCS-M.

## 3.1. Wyznaczanie PEO dla drzew

W tym rozdziale pokażemy algorytm wyznaczania PEO dla drzew metodą odrywania liści, czyli węzłów o stopniu równym 1. Odrywanie liści jest stosowane w kilku innych algorytmach dla drzew, np. przy wyznaczaniu największego zbioru niezależnego, najmniejszego zbioru dominującego, czy przy wyznaczaniu najmniejszego pokrycia wierzchołkowego.

**Dane wejściowe:** Dowolne drzewo (graf nieskierowany spójny acykliczny).

**Problem:** Wyznaczanie PEO dla drzew metodą odrywania liści.

**Dane wyjściowe:** Lista wierzchołków tworząca PEO.

**Opis algorytmu:** Algorytm rozpoczynamy od sprawdzenia wszystkich wierzchołków danego grafu. Izolowane węzły grafu wejściowego - węzły o stopniu równym 0 (jeśli takie istnieją) są od razu dodawane do wyznaczanego PEO. Liście drzewa gromadzone są w osobnej kolejce, a następnie dodawane do wyznaczanego PEO, przy czym najpierw następuje redukcja stopnia wierzchołka, z którym dany liść był bezpośrednio połączony oraz usunięcie krawędzi pomiędzy tymi wierzchołkami.

**Złożoność:** Złożoność czasowa algorytmu wynosi $O(n)$, gdzie $n$ to liczba wierzchołków w drzewie podanym na wejściu. Złożoność pamięciowa algorytmu to też $O(n)$, ze względu na rozmiar kolejki i słownika degree_dict.

**Uwagi:** Jeśli podany na wejściu graf jest grafem skierowanym lub nie jest drzewem, funkcja rzuca wyjątek ValueError z odpowiednim komunikatem. Implementacja wykorzystuje deque z modułu collections do stworzenia szybkiej kolejki FIFO. Słownik degree_dict przechowuje informacje o stopniach wierzchołków drzewa, przy czym dane są aktualizowane po każdej operacji usunięcia liścia. Graf wejściowy nie jest modyfikowany.

Listing 3.1. Moduł treepeo.

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

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

def find_peo_tree(graph):
    """ Finding PEO for a tree using leaves detaching method."""
    if graph.is_directed():
        raise ValueError("the graph is directed")
    peo = []
    # a dictionary with node degrees, O(n) time.
    degree_dict = dict((node, graph.degree(node))
            for node in graph.iternodes())
    Q = collections.deque()   # for leaves
    # put leaves to the queue, O(n) time.
    for node in graph.iternodes():
        if degree_dict[node] == 0:   # isolated node from the beginning
            peo.append(node)
        if degree_dict[node] == 1:   # leaf
            Q.append(node)
    while len(Q) > 0:
        source = Q.popleft()
        if degree_dict[source] == 0:
            peo.append(source)
            continue
        peo.append(source)
        assert degree_dict[source] == 1
        for target in graph.iteradjacent(source):
            if degree_dict[target] > 0:
                # remove the edge from source to target.
                degree_dict[target] -= 1
                degree_dict[source] -= 1
                if degree_dict[target] == 1:   # parent is a new leaf
                    Q.append(target)
                break
    if len(peo) != graph.v():
        raise ValueError("the graph is not a tree")
    return peo
```

## 3.2. Wyznaczanie PEO dla grafów zewnętrznie planarnych

W celu znajdownia PEO dla grafów zewnętrznie planarnych zmodyfikowano algorytm rozpoznawania tych grafów zaprezentowany w pracy [32]. Wykorzystywana jest tam metoda kolorowania krawędzi grafów i za każdym razem odrywane są wierzchołki stopnia drugiego lub mniejszego. Fakt ten pozwala w łatwy sposób tworzyć listę wierzchołków składających się na PEO. Autor artykułu dzieli krawędzie na 3 kolory: "cross", "out" oraz "bridge". Krawędzie "cross" nie leżą na ścianie zewnętrznej, krawędzie "out" łączą ścianę wewnętrzną i zewnętrzną, a krawędzie "bridge" to wszystkie pozostałe krawędzie. Wykorzystywane jest również twierdzenie dotyczące grafów

zewnętrznie planarnych, mówiące o tym, że każdy podgraf grafu zewnętrznie planarnego jest zewnętrznie planarny.

**Dane wejściowe:** Dowolny graf nieskierowany spójny.

**Problem:** Wyznaczanie PEO dla grafów zewnętrznie planarnych.

**Dane wyjściowe:** Wartość logiczna (True/False) stwierdzająca, czy graf jest zewnętrznie planarny, oraz wyznaczone PEO.

**Opis algorytmu:** Algorytm rozpoczynamy od sprawdzenia, czy liczba krawędzi wynosi $|E| = 2|V| - 3$, gdzie $|V|$ to liczba wierzchołków, oraz dokonywane jest początkowe kolorowanie wszystkich krawędzi na kolor "cross". Następnie rekurencyjnie redukowane są wierzchołki stopnia co najwyżej 2 oraz za każdym razem aktualizowne są kolory krawędzi incydentnych z usuwaną. Algorytm działa do momentu, aż wszystkie wierzchołki zostaną zredukowane.

**Złożoność:** Złożoność czasowa algorytmu wynosi $O(n)$, gdzie $n$ to liczba wierzchołków w grafie podanym na wejściu. Złożoność pamięciowa algorytmu to też $O(n)$, ze względu na rozmiar kolejki przechowującej wierzchołki stopnia co najwyżej drugiego, oraz słownika przechowującego pary: krawędź i odpowiadający jej kolor.

**Uwagi:** Jeśli podany na wejściu graf jest grafem skierowanym funkcja rzuca wyjątek ValueError z odpowiednim komunikatem.

Listing 3.2. Moduł outerplanarpeo.

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

import sys
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.connectivity.connected import is_connected

class OuterplanarPEO:
    """Outerplanar graphs detection."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        if not is_connected(graph):
            raise ValueError("the graph is not connected")
        self.graph = graph
        self._graph_copy = self.graph.copy()
        self.color = None   # cross, out, bridge
        self.outerplanar = None
        self.peo = []

    def run(self):
        """Executable pseudocode."""
        nedges = self.graph.e()   # O(n) time
```

```
if nedges > 2 * self.graph.v() - 3:
    self.outerplanar = False
    return False
# color all edges.
self.color = dict((edge, "cross") for edge in self.graph.iteredges())
# find vertices of degree < 3.
M = set(v for v in self.graph.iternodes()
    if self.graph.degree(v) <= 2)
self.outerplanar = True

while len(M) > 0 and self.outerplanar:
    u = M.pop()
    self.peo.append(u)
    Nu = self._graph_copy.degree(u)

    if Nu == 0:
        pass
    elif Nu == 1:
        edge = next(self._graph_copy.iteroutedges(u))
        assert edge.source == u
        self._graph_copy.del_node(u)
        if self._graph_copy.degree(edge.target) == 2:
            M.add(edge.target)
        nedges -= 1
    elif Nu == 2:
        edge1, edge2 = list(self._graph_copy.iteroutedges(u))
        self._graph_copy.del_node(u)
        color1 = self.find_color(edge1)
        color2 = self.find_color(edge2)
        if edge1.target > edge2.target:
            edge3 = Edge(edge2.target, edge1.target)
        else:
            edge3 = Edge(edge1.target, edge2.target)

        if self._graph_copy.has_edge(edge3):
            for edge in self._graph_copy.iteroutedges(edge3.source):
                if edge.target == edge3.target:
                    edge3 = edge
                    break
            nedges -= 2
            if self._graph_copy.degree(edge3.source) == 2:
                M.add(edge3.source)
            if self._graph_copy.degree(edge3.target) == 2:
                M.add(edge3.target)
            if color1 in ("cross", "out") and color2 in ("cross", "out"):
                if self.color[edge3] == "cross":
                    self.color[edge3] = "out"
                elif self.color[edge3] == "out":
                    self.color[edge3] = "bridge"
                elif self.color[edge3] == "bridge":
                    self.outerplanar = False
            else:
                self.outerplanar = False
        else:
            self._graph_copy.add_edge(edge3)
            nedges -= 1
            if color1 in ("cross", "out") and color2 in ("cross", "out"):
```

```
                    self.color[edge3] = "out"
                else:
                    self.color[edge3] = "bridge"

        assert self._graph_copy.e() == nedges
        if self.outerplanar:
            self.outerplanar = (nedges == 0)
        return self.outerplanar, self.peo

    def find_color(self, edge):
        """Find color of edge."""
        if edge.source > edge.target:
            edge = ~edge
        return self.color[edge]
```

## 3.3. Wyznaczanie PEO dla grafów maksymalnie zewnętrznie planarnych

Grafy maksymalnie zewnętrznie planarne są podklasą grafów zewnętrznie planarnych. Ze względu na ich właściwości, wyznaczanie PEO dla tej podklasy grafów można przeprowadzić w sposób znacznie bardziej uproszczony w stosunków do ogólnych grafów zewnętrznie planarnych. Jak przedstawiono w pracy [30], graf maksymalnie zewnętrznie planarny jest cięciowy i można dla niego przeprowadzić eliminację wierzchołków simplicjalnych, która jest równoczesnie eliminacją wierzchołków drugiego stopnia. Eliminując taki wierzchołek rozbijamy trójkąt usuwając z niego wierzchołek, który nie ma sąsiadów na zewnątrz tego trójkąta. Taki rodzaj eliminacji wierzchołków nazwano trójkątnym uporządkowaniem eliminacji (ang. *triangle elimination ordering*). Poniżej przedstawiono modyfikację algorytmu zaproponowanego w pracy [31], rozpoznającego, czy graf jest maksymalnie zewnętrznie planarny, oraz dodano do niego wyznaczanie PEO.

**Dane wejściowe:** Dowolny graf nieskierowany spójny.

**Problem:** Wyznaczanie PEO dla grafów maksymalnie zewnętrznie planarnych.

**Dane wyjściowe:** Wyznaczone PEO dla danego grafu.

**Opis algorytmu:** Algorytm rozpoczynamy od sprawdzenia, czy liczba krawędzi wynosi $m = 2n-3$, gdzie $n$ to liczba wierzchołków. Następnie tworzymy początkową listę wierzchołków drugiego stopnia. Usuwamy kolejno wierzchołki drugiego stopnia, jednocześnie aktualizaując zawierającą je listę o nowo powstające wierzchołki tego typu. Podczas działania algorytmu korzystamy z faktu, że jeśli dana krawędź spina więcej niż dwa trójkąty, to albo graf nie może być planarny, albo wszystki wierzchołki nie mogą leżeć na jego zewnętrznej ścianie [31].

**Złożoność:** Złożoność czasowa algorytmu wynosi $O(n)$, gdzie $n$ to liczba wierzchołków w grafie podanym na wejściu. Złożoność pamięciowa algorytmu to też $O(n)$, ze względu na rozmiar kolejki przechowującej wierzchołki drugiego stopnia.

**Uwagi:** Jeśli podany na wejsciu graf jest grafem skierowanym, to funkcja rzuca wyjątek ValueError z odpowiednim komunikatem. Odpowiednie komunikaty rzucane są także, gdy podany na wejściu graf posiada nieodpowiednią liczbę krawędzi - nie spełnia definicji zewnętrznej planarności, oraz gdy w trakcie działania algorytmu zostanie dowiedziony fakt, że nie jest on grafem maksymalnie zewnętrznie planarnym.

Listing 3.3. Moduł maxouterplanar.

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

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

class MaximalOuterplanarPEO:
    """Maximal outerplanar graphs detection."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self._graph_copy = self.graph.copy()    # O(n) time
        self.outerplanar = None
        self.order = []

    def run(self):
        """Executable pseudocode."""
        if self.graph.e() != 2 * self.graph.v() - 3:    # O(n) time
            raise ValueError("incorrect edge count")
        deg2 = list(v for v in self.graph.iternodes()
            if self.graph.degree(v) == 2)
        if len(deg2) < 2:    # for single triangle there are 3 such veritces
            raise ValueError("not enough 2-vertices")

        # we create a set of edges that connect neighbours of vertex
        # form deg2 we will check if an egde is not a part of a few
        # triangles it can only be part of maximum two triangles
        pairs = set()
        self.order.extend(deg2)

        while self._graph_copy.v() > 2:
            v = deg2.pop()
            w1, w2 = list(self._graph_copy.iteradjacent(v))
            self._graph_copy.del_node(v)
            if self._graph_copy.degree(w1) == 2:
                deg2.append(w1)
                self.order.append(w1)
            if self._graph_copy.degree(w2) == 2:
                deg2.append(w2)
                self.order.append(w2)
```

```
            edge = Edge(w1, w2) if w1 < w2 else Edge(w2, w1)
            if edge in pairs:
                raise ValueError("three triangles at the edge")
            pairs.add(edge)
            if not self._graph_copy.has_edge(edge):
                raise ValueError("graph is not maximal outerplanar")
            if len(deg2) < 2:
                raise ValueError("removing node leaves non-maximal outerplanar")
        # we only have last edge remaining
        assert self._graph_copy.v() == 2
        self.outerplanar = True
        return self.order
```

## 3.4. Wyznaczanie PEO dla grafów permutacji algorytmem zachłannym

Algorytm bazuje na obserwacji, że grafy permutacji, które nie są cięciwowe, mogą posiadać tylko indukowane cykle o długości 4 (podgrafy $C_4$). Wobec tego wystarczy dla każdego takiego cyklu dodać cięciwę, aby graf uczynić cięciwowym. Może się jednak zdarzyć, że różne cykle mają wspólne wierzchołki, a co za tym idzie, jedna krawędź może być cięciwą wspólną dla wielu cykli. Chcemy minimalizować liczbę nowych krawędzi dodawanych do grafu, więc algorytm na każdym etapie wybiera krawędzie wspólne dla największej liczby cykli. Wymaga to dynamicznego aktualizowania danych o cyklach i cięciwach.

Problem na tym etapie przypomina kolorowanie wierzchołków grafu algorytmem SL (*Smallest Last*), gdzie usuwanie z grafu wierzchołka o najmniejszym stopniu wymagało aktualizacji danych o stopniach pozostałych wierzchołków grafu. Zastosujemy podobne grupowanie cięciw w bukiety ze względu na liczbę pokrytych cykli. Po wykorzystaniu cięciwy należy usunąć z danych pokryte cykle i odpowiednio przesunąć pewne cięciwy do bukietów z mniejszą liczbą pokrywanych cykli.

**Dane wejściowe:** Spójny graf permutacji $G$ dany jako permutacja $n$ liczb (lista Pythona).

**Problem:** Wyznaczanie PEO dla grafu permutacji $G$ lub dla jego dopełnienia cięciwowego.

**Dane wyjściowe:** PEO, lista dodanych krawędzi, dopełnienie cięciwowe grafu $G$ (graf abstrakcyjny).

**Opis algorytmu:** Algorytm rozpoczynamy od sprawdzenia spójności grafu w czasie $O(n)$ i od utworzenia grafu abstrakcyjnego w czasie $O(n^2)$. Sprawdzamy istnienie PEO i jeżeli graf jest cięciwowy, to algorytm kończy pracę.

W przeciwnym wypadku następuje sprawdzenie występowania indukowanych cykli $C_4$, wykorzystujemy wszystkie 4-elementowe kombinacje bez

powtórzeń ze zbioru $n$-elementowego. Dla każdego znalezionego cyklu $C_4$ zapisujemy jego dwie możliwe cięciwy, a z drugiej strony, każdej cięciwie przyporządkowujemy zbiór pokrywanych cykli.

W następnym etapie zachłannie wybieramy cięciwy pokrywające największą liczbę cykli i dodajemy je do powstającego dopełnienia cięciwowego. Na końcu wyznaczamy PEO dopełnienia cięciwowego.

**Złożoność:** Złożoność czasowa algorytmu jest zdeterminowana przez poszukiwanie cykli $C_4$, sprawdzenie wszystkich kombinacji zajmuje czas $O(n^4)$. Liczba znalezionych cykli zależy od rodzaju grafu i może być rzędu $O(n^4)$ dla grafów pełnych dwudzielnych $K_{p,q}$. Z kolei liczba cięciw jest najwyżej rzędu $O(n^2)$, bo tyle jest możliwych par wierzchołków.

**Uwagi:** Ze względu na złożoność czasową i pamięciową rzędu $O(n^4)$ algorytm zachłanny może być użyteczny jedynie dla niezbyt dużych grafów permutacji, rzędu kilkuset wierzchołków. Przykładowo dla grafu pełnego dwudzielnego $K_{50,50}$ obliczenia zajmują około $11s$, a zajmowana pamięć to $1.4GB$.

Listing 3.4. Moduł permpeo.

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

import itertools
from graphtheory.structures.edges import Edge
from graphtheory.permutations.permtools import perm_is_connected
from graphtheory.permutations.permtools import make_abstract_perm_graph
from graphtheory.permutations.permtools import make_bipartite_perm
from graphtheory.chordality.peotools import find_peo_lex_bfs
from graphtheory.chordality.peotools import is_peo1, is_peo2

class PermGraphPEO:
    """Finding a chordal completion for permutation graphs."""

    def __init__(self, perm):
        """The algorithm initialization."""
        self.perm = perm
        self.n = len(self.perm)
        if not perm_is_connected(self.perm):    # O(n) time
            raise ValueError("perm is not connected")
        self.graph = make_abstract_perm_graph(self.perm)
        self.new_edges = []
        self.order = None

    def run(self):
        """Executable pseudocode."""
        self.order = find_peo_lex_bfs(self.graph)
        if is_peo1(self.graph, self.order):    # skipping chordal graphs
            print("perm graph is chordal")
            return

        cycle2chord = dict()
        chord2cycle = dict()
        # looking for 4 numbers creating induced graph C_4
        for (i,j,k,r) in itertools.combinations(range(self.n), 4):
            if self.perm[k] < self.perm[r] < self.perm[i] < self.perm[j]:
```

```
                cycle = (self.perm[i], self.perm[k], self.perm[j], self.perm[r])
                chord1 = Edge(self.perm[i], self.perm[j])
                chord2 = Edge(self.perm[k], self.perm[r])
                cycle2chord[cycle] = {chord1, chord2}
                if chord1 in chord2cycle:
                    chord2cycle[chord1].add(cycle)
                else:
                    chord2cycle[chord1] = {cycle}
                if chord2 in chord2cycle:
                    chord2cycle[chord2].add(cycle)
                else:
                    chord2cycle[chord2] = {cycle}

        # taking chords that cover most cycles
        n_cycles = len(cycle2chord)
        bucket = list(set() for chord in range(n_cycles+1))
        for chord in chord2cycle:
            bucket[len(chord2cycle[chord])].add(chord)

        maxi = n_cycles
        while True:   # in one step we can delete a few cycles
            while not bucket[maxi]:   # looking for not empty bucket
                maxi -= 1
            chord1 = bucket[maxi].pop()   # taking the chord
            # giving back the chord for further usefullness
            bucket[maxi].add(chord1)
            self.new_edges.append(chord1)
            # deleting covered cycles
            for cycle in tuple(chord2cycle[chord1]):
                for chord2 in cycle2chord[cycle]:
                    k = len(chord2cycle[chord2])
                    chord2cycle[chord2].remove(cycle)
                    bucket[k].remove(chord2)
                    bucket[k-1].add(chord2)
                del cycle2chord[cycle]
            del chord2cycle[chord1]
            if not cycle2chord:
                assert all(len(chord2cycle[chord]) == 0
                    for chord in chord2cycle)
                break
        # creating a chordal completion.
        for edge in self.new_edges:
            self.graph.add_edge(edge)
        self.order = find_peo_lex_bfs(self.graph)
        assert is_peo1(self.graph, self.order)
```

## 3.5. Wyznaczanie PEO dla grafow cięciwowych algorytmem MCS

Algorytm MCS to ulepszona wersja algorytmu Lex-BFS, służąca do sprawdzania, czy graf jest cięciwowy, oraz do wyznaczania PEO dla grafów cięciwowych [18]. Stanowi on bazę do prezentowanego w następnej sekcji algorytmu MCS-M. Algorytm wykorzystuje ideę największej liczności.

**Dane wejściowe:** Dowolny graf nieskierowany.

**Problem:** Wyznaczenie PEO dla grafów cięciwowych.

**Dane wyjściowe:** Wyznaczone PEO dla grafu podanego na wejściu.

**Opis algorytmu:** Algorytm rozpoczynamy od zainicjalizowania etykiet dla wszystkich wierzchołków wartością 0. Na samym początku, z uwagi na fakt, że wszystkie wierzchołki mają tą samą wartość etykiety, możemy wybrać dowolny wierzchołek jako startowy. Wybrany wierzchołek dodajemy do PEO i usuwamy z grafu. Następnie etykiety wszystkich sąsiadów danego wierzchołka są zwiększane o 1. W kolejnych iteracjach za każdym razem wybierany jest wierzchołek, który w danym momencie ma największą etykietę. Jeśli kilka wierzchołków ma największą liczność, to możemy wybrać dowolny z nich. Z uwagi na fakt, że autorzy pracy [21] wybierają wierzchołki od końca, w wykonanej implementacji na końcu następuję odwrócenie elementów w liście przechowującej PEO.

**Złożoność:** Złożoność czasowa algorytmu wynosi $(n^2)$, gdzie $n$ to liczba wierzchołków grafu wejściowego. Mamy zewnętrzną pętlę wykonującą się $n$ razy, a wewnątrz pętli występuje liniowe wyszukiwanie wierzchołka o największej etykiecie. Złożoność pamięciowa algorytmu wynosi $(n+m)$ ze względu na stworzenie kopii grafu. Dodatkowo wykorzystywana jest pamięć $O(n)$ na słownik z etykietami i na PEO.

**Uwagi:** Nowa implementacja algorytmu MCS jest bliższa pseudokodowi prezentowanemu w literaturze. Wierzchołki nieprzetworzone znajdują się w kopii grafu $H$, która jest redukowana przez usuwanie kolejnych wierzchołków. W efekcie otrzymano kod działający szybciej niż poprzednia implementacja o takiej samej złożoności $O(n^2)$. Szczegółowy opis testów złożoności czasowej znajduje się w rodziale A.

Listing 3.5. Moduł mcspeo.

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

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

try:
    range = xrange
except NameError:    # Python 3
    pass

def find_peo_mcs3(graph):
    """Finding PEO in a chordal graph using MCS, O(n^2) time."""
    if graph.is_directed():
        raise ValueError("the graph is directed")
    order = list()    # PEO
    H = graph.copy()
    visited_degree = dict((node, 0) for node in graph.iternodes())
    for step in range(graph.v()):
```

```
        source = max(H.iternodes(), key=visited_degree.__getitem__)
        order.append(source)
        # update visited degree.
        for target in H.iteradjacent(source):
            visited_degree[target] += 1
        H.del_node(source)
    order.reverse()
    return order
```

## 3.6. Wyznaczanie triangulacji minimalnej grafów algorytmem MCS-M

Algorytm MCS-M to rozbudowana wersja algorytmu MCS, która rozwija ideę znajdowania PEO na podstawie największej liczności i dodaje do niej aspekt badania, oprócz sąsiadów przetwarzanego wierzchołka, innych wierzchołków osiągalnych po specjalnych rodzajach ścieżek z przetwarzanego wierzchołka [18].

**Dane wejściowe:** Dowolny graf nieskierowany.

**Problem:** Wyznaczenie triangulacji minimalnej dla dowolnych grafów.

**Dane wyjściowe:** Wyznaczone PEO triangulacji minimalnej grafu podanego na wejściu.

**Opis algorytmu:** Algorytm na starcie ustawia etykiety wszystkich wierzchołków na zero. Następnie w każdym obiegu pętli zewnętrznej przetwarzany jest jeden wierzchołek grafu startowego i zaliczany do PEO. Niech wybranym wierzchołkiem będzie $v$. W wewnętrznej pętli przetwarzane są wszystkie wierzchołki nie zaliczone do PEO i tworzony jest zbiór $S$ wierzchołków przeznaczonych do uaktualnienia etykiety. Do zbioru $S$ zaliczamy wszystkich nieprzetworzonych sąsiadów wierzchołka $v$ oraz takie wierzchołki $u$, do których prowadzi specjalna ścieżka. Ważny jest warunek, że wszystkie wierzchołki na ścieżce mają etykietę mniejszą niż $u$. Okazuje się, że wierzchołki $u$ połączone specjalną ścieżką z $v$, generują nowe krawędzie $(v, u)$, które budują dopełnienie cięciwowe startowego grafu.

**Złożoność:** Teoretyczna czasowa złożoność obliczeniowa algorytmu MCS-M wynosi $O(nm)$, ponieważ w każdym obiegu pętli zewnętrznej przetwarzane są wszystkie krawędzie grafu zbudowanego na wierzchołkach nie zaliczonych do PEO w czasie $O(m)$.

**Uwagi:** W naszej implementacji wyznaczanie zbioru $S$ realizowane jest w metodzie visit(). Metoda wykorzystuje schemat działania algorytmu BFS, jednak do kolejki wielokrotnie trafiają te wierzchołki, dla których zostaje znaleziona lepsza ścieżka, z wierzchołkami o mniejszych etykietach. Dla ścieżek definiujemy wagę równą największej etykiecie wierzchołka na ścieżce. Z tego

powodu złożoność obliczeniowa metody visit() jest gorsza niż $O(m)$, testy wskazują na $O(n^3)$.

Pierwszym sposobem na poprawę złożoności metody visit() jest zastosowanie kolejki priorytetowej, jak w algorytmie Dijkstry znajdowania najkrótszych ścieżek z jednego źródła. Wtedy z kolejki pobierane będą wierzchołki, do których prowadzi najlepsza ścieżka o najmniejszej wadze i nie będzie wielokrotnego przetwarzania wierzchołków. Spodziewana złożoność metody visit() wyniesie $O(m \log n)$ lub $O(n^2)$.

Drugi sposób na poprawę złożoności metody visit() bazuje na obserwacji, że wagi ścieżek nie są dowolne, ale są liczbami całkowitymi w przedziale od 0 do $n$. Wg literatury w takiej sytuacji można osiągnąć złożoność $O(m)$ dla algorytmu Dijkstry [47].

Listing 3.6. Moduł mcsmpeo.

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

import collections
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.chordality.peotools import is_peo1

try:
    range = xrange
except NameError:    # Python 3
    pass

class MCS_M:
    """Finding PEO of a chordal completion using MCS-M."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.H = self.graph.copy()    # O(V+E) time
        self.new_edges = []    # storing added edges
        self.order = []    # storing PEO
        self.visited_degree = dict((node, 0)
            for node in self.graph.iternodes())    # O(V) time

    def run(self):
        """Executable pseudocode."""
        for step in range(self.graph.v()):
            source = max(self.H.iternodes(),
                key=self.visited_degree.__getitem__) # O(V) time
            self.order.append(source)
            update = self.visit(source)
            # Update visited degree.
            for target in update:
                self.visited_degree[target] += 1
                edge = Edge(source, target)
                if not self.graph.has_edge(edge):
                    self.new_edges.append(edge)    # there is no need to add to H
            self.H.del_node(source)
        self.order.reverse()    # O(V) time
```

```
        # creating a chordal completion.
        for edge in self.new_edges:
            self.graph.add_edge(edge)
        # PEO already determined
        assert is_peo1(self.graph, self.order)   # O(V+E) time

    def visit(self, node):
        """Processing a vertex."""
        queue = collections.deque()   # local variable within method
        parent = dict()   # local variable within method
        parent[node] = None   # before queue.append
        queue.append(node)
        # dict for max weight in the path
        D = dict((target, 0) for target in self.H.iternodes())
        S = set()   # nodes to update

        # first loop, after it S is the same as in MCS
        for target in self.H.iteradjacent(node):
            queue.append(target)   # local variable within method
            S.add(target)   # every neighbour of source is added to S
            parent[target] = node
            D[target] = self.visited_degree[target]

        while len(queue) > 0:
            source = queue.popleft()
            for target in self.H.iteradjacent(source):
                if target == node:
                    continue
                if target not in parent:   # target was not met
                    parent[target] = source   # before queue.append
                    queue.append(target)
                    if self.visited_degree[target] > D[source]:
                        S.add(target)
                        D[target] = self.visited_degree[target]
                    else:
                        D[target] = D[source]
                else:   # target was met on a different path
                    if self.visited_degree[target] > D[source]: # better path
                        S.add(target)
                        D[target] = self.visited_degree[target]
                        # going further this path
                        queue.append(target)
                    elif D[target] > D[source]:
                        D[target] = D[source]   # correcting
                        queue.append(target)
        return S
```

# 4. Podsumowanie

W pracy zostały przedstawione zagadnienia dotyczące triangulacji grafów, czyli dodawania do danego grafu nowych krawędzi w celu otrzymania grafu cięciwowego. Opisano wyznaczanie doskonałego uporządkowania eliminacji dla wybranych klas grafów oraz przedstawiono sposób użycia funkcji wyznaczających PEO, zarówno tych uprzednio znajdujących się w pakiecie graphtheory [1], jak i tych zaimplementowanych na potrzeby tej pracy. Wyznaczone PEO określa strukturę grafów cięciwowych, a dla innych grafów określa strukturę dopełnienia cięciwowego. Zwrócono również uwagę na różnicę między triangulacją najmniejszą a triangulacją minimalną. Tylko triangulacja najmniejsza jest zawsze optymalnym rozwiązaniem problemu szerokości drzewowej czy problemu najmniejszego wypełnienia. Triangulacja minimalna może nie być optymalnym rozwiązaniem, ale zwykle daje się szybko wyznaczyć, co może być ważne w zastosowaniach.

W ramach pracy zaimplementowano pięć algorytmów wyznaczania PEO dla klas grafów, dla których tą operację można zrealizować w czasie wielomianowym z podaniem dokładnego rozwiązania. Są to drzewa, grafy zewnętrznie planarne, grafy maksymalnie zewnętrznie planarne, grafy permutacji. Podano nową implementację algorytmu MCS do wyznaczania PEO dla grafów cięciwowych, który jest podstawą algorytmu MCS-M do wyznaczania triangulacji minimalnych. Algorytm MCS-M jest chyba najszybszym spośród opisanych w literaturze.

Podczas implementacji wyżej wymienionych algorytmów zwracano szczególną uwagę na ich złożoność obliczeniową, czasową oraz pamięciową. Poprawność algorytmów testowano z pomocą modułu unittest, natomiast praktyczną złożoność obliczeniową testowano przy pomocy modułu timeit.

Wykonane implementacje algorytmów wzbogacą zbiór algorytmów dostępnych w pakiecie graphtheory [1]. Pakiet może być wykorzystywany do nauki algorytmów ze względu na czytelny kod w języku Python, bliski pseudokodowi. Z drugiej strony, implementacje czasem uzupełniają brakujące elementy z pseudokodu i pozwalają wykonać praktyczne obliczenia dla problemów średniej wielkości.

# A. Testy algorytmów

Implementacje algorytmów przedstawionych w rozdziale 3 niniejszej pracy zostały przetestowane zarówno pod kątem poprawności, jak i praktycznej wydajności, do czego użyto narzędzi dostępnych w języku Python. W celu sprawdzenia poprawności, dla każdego algorytmu napisano testy jednostkowe przy użycu modułu unittest [4]. Do wyznaczenia złożoności obliczeniowej posłużono się natomiast modułem timeit [5], przy czym dla każdego badanego przykładu wykonywano za każdym razem 3 próby, a następnie brano średnią z nich.

## A.1. Test wyznaczania PEO dla drzew metodą odrywania liści

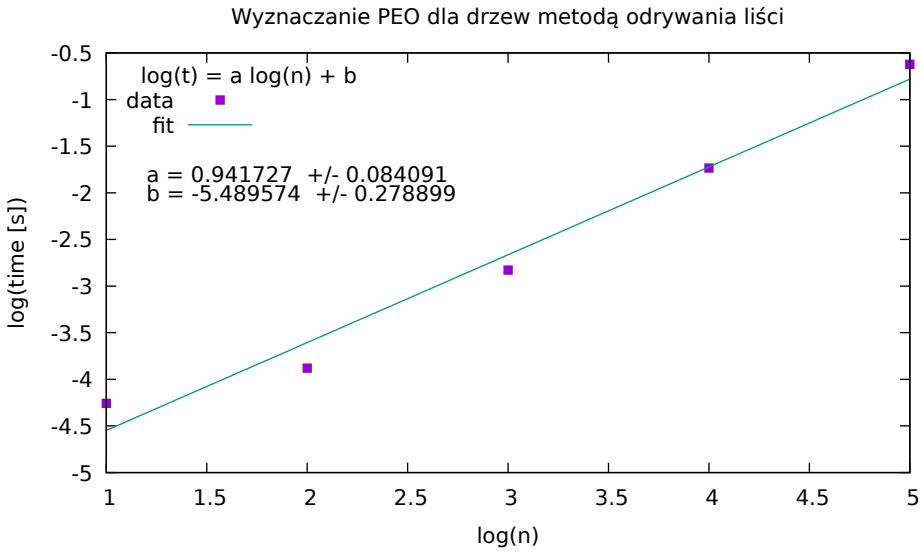
Algorytm przetestowano przy użyciu dostępnej w pakiecie graphtheory klasy GraphFactory, oraz należącej do niej funkcji make_tree(), która generuje losowe drzewa o zadanej liczbie wierzchołków. Współczynnik $a = 0.941(84)$ A.1 potwierdza liniową złożoność obliczeniową przedstawionego algorytmu.

## A.2. Test wyznaczania PEO dla grafów zewnętrznie planarnych

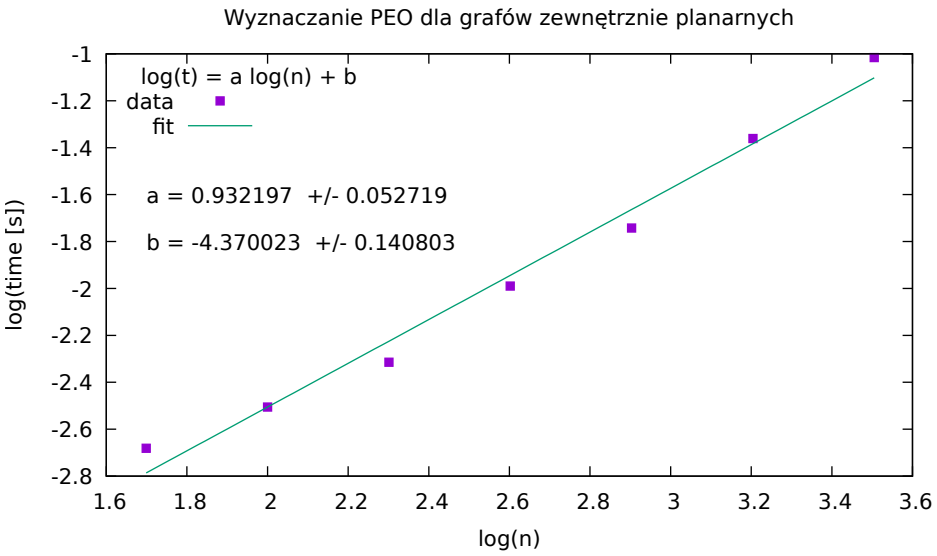
Algorytm przetestowano przy użyciu dostępnej w pakiecie graphtheory klasy GraphFactory, oraz należącej do niej funkcji make_cyclic(), która generuje grafy cykliczne o zadanej liczbie wierzchołków. Grafy cykliczne $C_n$ to najprostsze grafy zewnętrznie planarne o szerokości drzewowej 2. Współczynnik $a = 0.932(53)$ A.2 potwierdza liniową złożoność obliczeniową przedstawionego algorytmu.

## A.3. Test wyznaczania PEO dla grafów maksymalnie zewnętrznie planarnych

Algorytm przetestowano przy użyciu dostępnej w pakiecie graphtheory klasy MaximalOuterplanarGenerator, która generuje losowe grafy zewnętrznie planarne o zadanej liczbie wierzchołków. Współczynnik $a = 1.036(59)$ A.3 potwierdza liniową złożoność obliczeniową przedstawionego algorytmu.




Rysunek A.1. Wykres wydajności algorytmu wyznaczania PEO dla drzew metodą odrywania liści. Potwierdzona złożoność $O(n)$.




Rysunek A.2. Wykres wydajności algorytmu wyznaczania PEO dla grafów zewnętrznie planarnych. Potwierdzona złożoność $O(n)$.

## A.4. Test wyznaczania PEO dla grafów permutacji algorytmem zachłannym

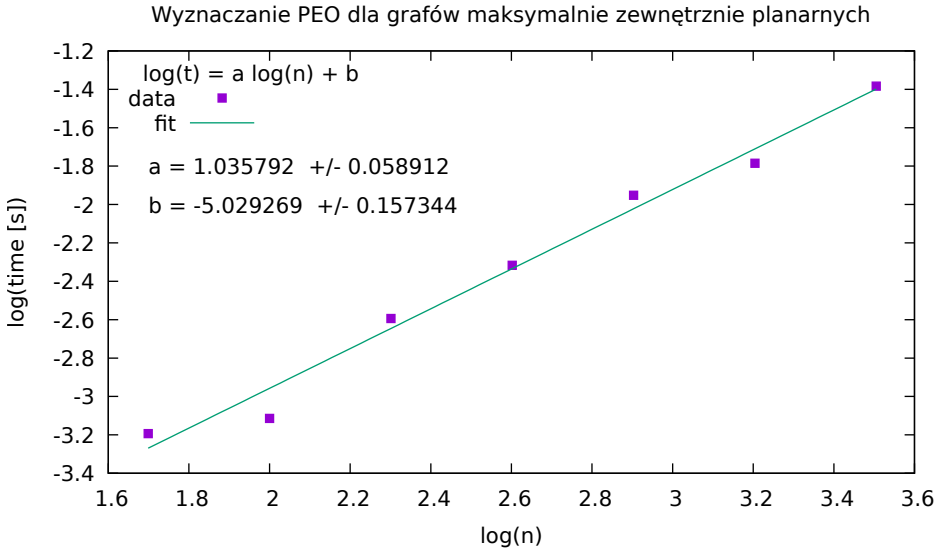
Algorytm przetestowano przy użyciu funkcji dostępnych w pakiecie graphtheory. Do wykonania każdego z testów wykorzystano najpierw funkcję generującą make_bipartite_perm(), która zwraca permutację o zadanej liczbie wierzchołków dla grafu pełnego dwudzielnego, a następnie funkcję make_abstract_perm_graph() w celu utworzenia abstrakcyjnego grafu permutacji, na podstawie zadanej permutacji. Współczynnik $a = 3.64(22)$ A.4 potwierdza złożoność obliczeniową na poziomie $O(n^4)$.

## A.5. Test wyznaczania PEO dla grafów cięciwowych algorytmem MCS

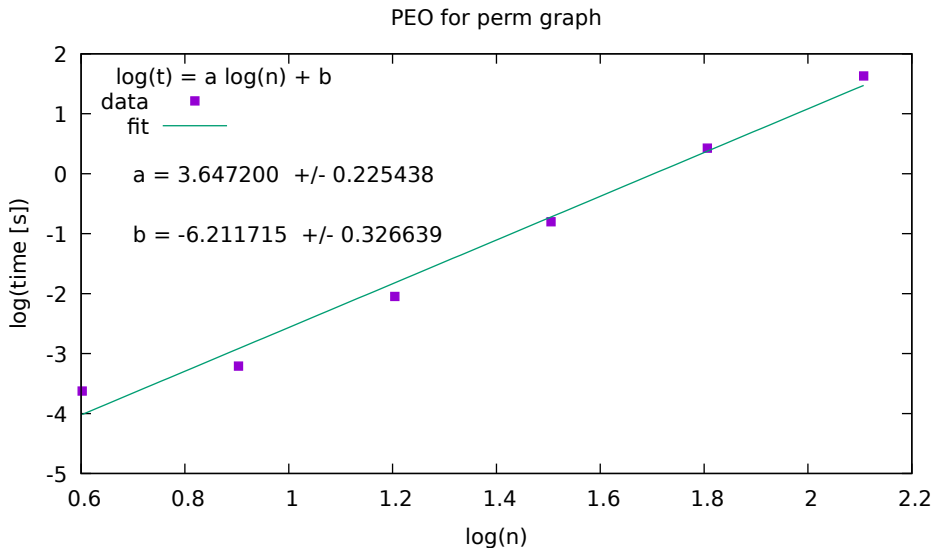
Algorytm przetestowano przy użyciu funkcji dostępnych w pakiecie graphtheory. Do wykonania pierwszego z testów użyto funkcji make_random_chordal(), generującej losowe grafy cięciwowe o zadanej liczbie wierzchołków. Do wykonania drugiego z testów użyto funkcji make_random_ktree(), generującej losowe k-drzewa o zadanej liczbie wierzchołków. Współczynnik $a = 1.123(98)$ dla losowych grafów cięciwowych A.5 oraz współczynnik $a = 1.992(59)$ dla losowych k-drzew A.6 potwierdzają złożoność obliczeniową na poziomie $O(n^2)$.

## A.6. Test wyznaczania minimalnej triangulacji grafów algorytmem MCS-M

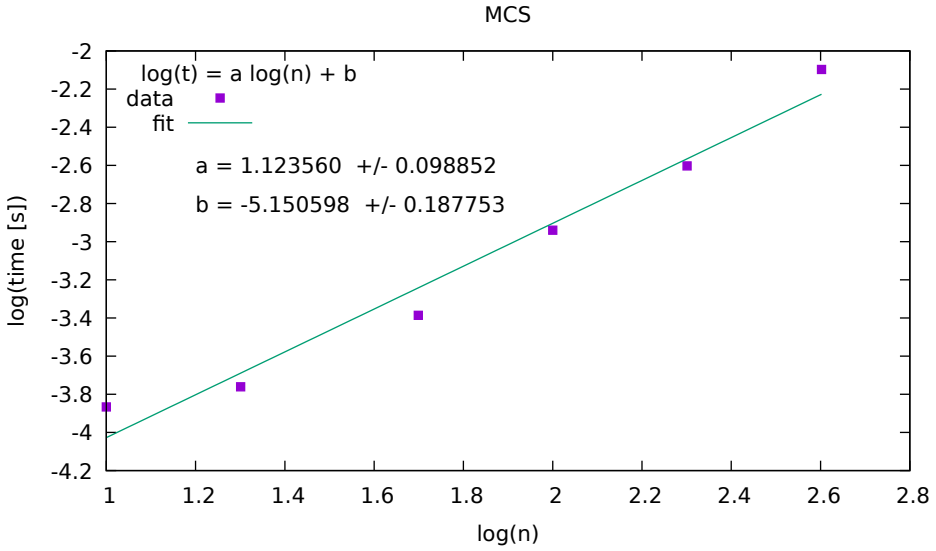
Algorytm przetestowano przy użyciu funkcji dostępnych w pakiecie graphtheory należących do klasy GraphFactory. Do wykonania pierwszego z testów użyto funkcji make_bipartite(), generującej losowe grafy dwudzielne. Do wykonania drugiego z testów użyto funkcji make_grid(), generującej losowe grafy typu krata (ang. *grid graph*). Współczynnik $a = 1.186(41)$ dla losowych grafów dwudzielnych A.7 oraz współczynnik $a = 0.763(80)$ dla losowych grafów krata A.8 pokazują złożoność obliczeniową zbliżoną do $O(nm)$, ale dla gęstych grafów dwudzielnych widać dodatkowy czynnik wynikający z niezbyt wydajnego wyszukiwania ścieżek.



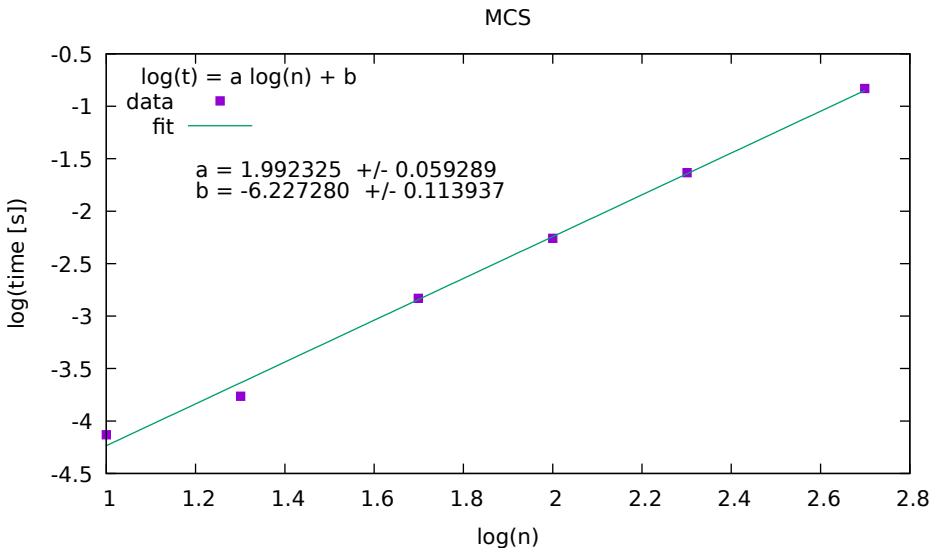

Rysunek A.3. Wykres wydajności algorytmu wyznaczania PEO dla grafów maksymalnie zewnętrznie planarnych. Potwierdzona złożoność $O(n)$.



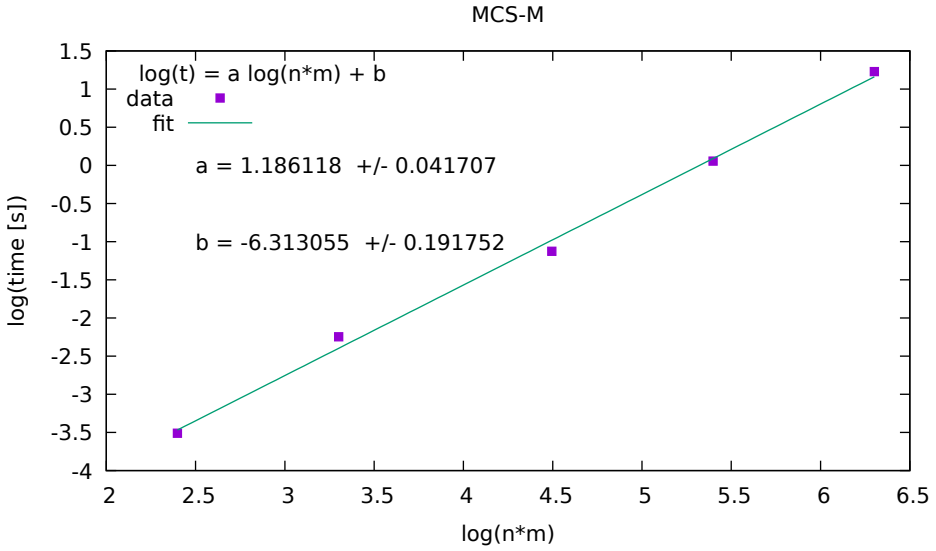

Rysunek A.4. Wykres wydajności algorytmu wyznaczania PEO dla grafów permutacji algorytmem zachłannym. Potwierdzona złożoność $O(n^4)$.



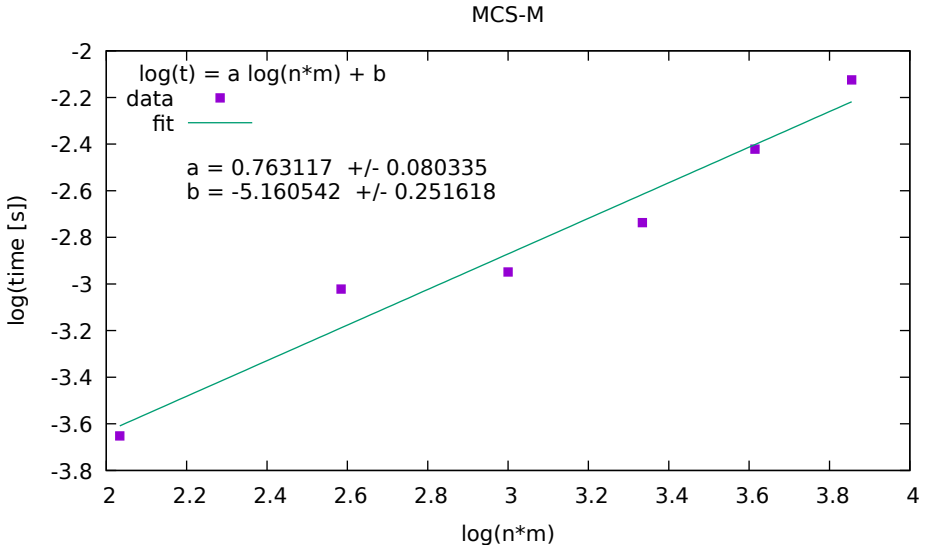

Rysunek A.5. Wykres wydajności algorytmu MCS dla losowych grafów cięciwowych. Potwierdzona złożoność $O(n^2)$.




Rysunek A.6. Wykres wydajności algorytmu MCS dla losowych k-drzew. Potwierdzona złożoność $O(n^2)$.




Rysunek A.7. Wykres wydajności algorytmu MCS-M dla grafów pełnych dwudzielnych. Złożoność zbliżona do $O(nm)$.




Rysunek A.8. Wykres wydajności algorytmu MCS-M dla grafów typu krata. Złożoność zbliżona do $O(nm)$.

# Bibliografia


[1] Andrzej Kapanowski, graphtheory, GitHub repository, 2024, `https://github.com/ufkapano/graphtheory/`.

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

[3] NetworkX Python package official documentation, 2024, `https://networkx.org/`.

[4] Python Programming Language - Official Documentation, `https://docs.python.org/3/library/unittest.html`.

[5] Python Programming Language - Official Documentation, `https://docs.python.org/3/library/timeit.html`.

[6] Wikipedia, Chordal completion, 2024, `https://en.wikipedia.org/wiki/Chordal_completion`.

[7] Wikipedia, Chordal graph, 2024, `https://en.wikipedia.org/wiki/Chordal_graph`.

[8] Wikipedia, Interval graph, 2024, `https://en.wikipedia.org/wiki/Interval_graph`.

[9] M. Yannakakis, *Computing the Minimum Fill-In is NP-Complete*, SIAM Journal on Algebraic Discrete Methods 2(1), 77-79 (1981).

[10] S. Arnborg, D. G. Corneil, A. Proskurowski, *Complexity of finding embeddings in a k-tree*, SIAM J. Alg. Disc. Meth. 8(2), 277-284 (1987).

[11] Pinar Heggernes, *Minimal triangulations of graphs: A survey*, Discrete Mathematics 306(3), 297-317 (2006).

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Wprowadzenie do algorytmów*, Wydawnictwo naukowe PWN, Warszawa 2012.

[13] D. J. Rose, *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations*, Graph Theory and Computing, 183-217 (1972).

[14] C. Beeri, R. Fagin, D. Maier, M. Yannakakis, *On the desirability of acyclic database systems*, Journal of the ACM (1983).

[15] S. L. Lauritzen, D. T. Spiegelhalter, *Local Computations with Probabilities on Graph Structures and Their Application to Expert Systems*, Journal of the Royal Statistical Society 50, 157-224 (1988).

[16] F. R. K. Chung, D. Mumford, *Chordal completions of planar graphs*, Journal of combinatorial theory 62, 96-106 (1994).

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

[18] A. Berry, J. R. S. Blair, P.Heggernes, B. W. Peyton, *Maximum Cardinality Search for Computing Minimal Triangulations of Graphs*, Algorithmica 39, 287-298 (2004).

[19] Y. Villanger, *Lex M versus MCS-M*, Discrete Mathematics 306, 393-400 (2006).

[20] A. Berry, P. Heggernes, G. Simonet, *The Minimum Degree Heuristic and the*

*Minimal Triangulation Process*, Graph-Theoretic Concepts in computer Science, 58-70 (2003).

[21] R. E. Tarjan, M. Yannakakis, *Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs*, SIAM Journal of Computing 13, 566-579 (1984).

[22] D. J. Rose, R. E. Tarjan, G. S. Lueker, *Algorithmic spects of vertex elimination on graphs*, SIAM Journal of Computing 5, 266-283 (1976).

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

[24] Maciej Niezabitowski, *Dekompozycja drzewowa w teorii grafów*, Uniwersytet Jagielloński, kKaków 2019

[25] C. G. Lekkerkerker, J. Ch. Boland, *Representation of a finite graph by a set of intervals on the real line*, Fundamenta Mathematicae 51, 45-64 (1962).

[26] Maciej Mularski, *Badanie grafów przedziałowych z językiem Python*, Uniwersytet Jagielloński, Kraków 2023.

[27] T.Kloks, H. Bodlaender, H. Müller, and D. Kratsch, *Computing treewidth and minimum fill-in: All you need are the minimal separators*. In: Lengauer, T. (eds) Algorithms-ESA '93. ESA 1993. Lecture Notes in Computer Science, vol. 726, 260-271 (1993).

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

[29] P. Scheffler, *A linear algorithm for the pathwidth of trees*, in R. Bodendiek, R. Henn (eds.), Topics in Combinatorics and Graph Theory, Physica-Verlag, pp. 613-620 (1990).

[30] R. C. Laskar, H. M. Mulder, B. Novick, *Maximal outerplanar graphs as chordal graphs, path-neighborhood graphs, and triangle graphs*, Australasian Journal of Combinations, Vol. 52, 185-195 (2012).

[31] Sandra L. Mitchell, *Linear Algorithms to recognize outerplanar and maximal outerplanar graphs*, Information Processing Letters 9(5), 229-232 (1979).

[32] Manfred Wiegers, *Recognizing Outerplanar Graphs in Linear Time*, WG, 165-176 (1986).

[33] Maciej M. Sysło, *Characterizations of outerplanar graphs*, Discrete Mathematics 26, 47-53 (1979).

[34] Joan P. Hutchinson, *On list-coloring outerplanar graphs*, Journal of Graph Theory, Vol. 59, 59-74 (2008).

[35] F. Harary, *Graph Theory*, Addison-Wesley, Reading Massachussets, 1969.

[36] J. Gustedt, *On the pathwidth of chordal graphs*, Disc. Appl. Math. 45, 233-248 (1993).

[37] Konrad Gałuszka, *Badanie grafów szeregowo-równoległych z językiem Python*, Uniwersytet Jagielloński, Kraków 2018.

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

[39] Albert Surmacz, *Badanie grafów permutacji z językiem Python*, Uniwersytet Jagielloński, Kraków 2021.

[40] H.L. Bodlaender, T. Kloks, D. Kratsch, *Treewidth and pathwidth of permutation graphs*, SIAM Journal on Discrete Mathematics 8, 606-616 (1995).

[41] Daniel Meister, *Treewidth and minimum fill-in on permutation graphs in linear time*, Theoretical Computer Science 411, 3685-3700 (2010).

[42] Rolf H. Möhring, *Triangulating graphs without asteroidal triples*, Discrete Applied Mathematics 64(3), 281-287 (1996).

[43] Ton Kloks, Dieter Kratsch, Jeremy Spinrad, *On treewidth and minimum fill-in*

*of asteroidal triple-free graphs*, Theoretical Computer Science 175, 309-335 (1997).

[44] H. J. Broersma, T. Kloks, D. Kratsch, and H. Müller, *A Generalization of AT-Free Graphs and a Generic Algorithm for Solving Triangulation Problems*, Algorithmica 32, 594-610 (2002).

[45] Albert Surmacz, *Badanie grafów kołowych z językiem Python*, Uniwersytet Jagielloński, Kraków 2023.

[46] T. Kloks, *Treewidth of circle graphs*, Lecture Notes in Comput. Sci., vol. 762, pp. 108-117 (1993).

[47] Mikkel Thorup, *Undirected single-source shortest paths with positive integer weights in linear time*, Journal of the ACM 46(3), 362-394 (1999).