**Uniwersytet Jagielloński w Krakowie**

Wydział Fizyki, Astronomii i Informatyki Stosowanej

**Oliwia Gil**

Nr albumu: 1165952

# Grafy łuków na okręgu

Praca magisterska na kierunku Informatyka stosowana

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

Kraków 2025

## 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

*Bardzo dziękuję Promotorowi Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za zaangażowanie, pomoc oraz rady, dzięki którym niniejsza praca powstała w tej postaci.*

**Streszczenie**

Grafy łuków na okręgu to szczególna klasa grafów zdefiniowana jako grafy przecięć dla łuków na okręgu. Oznacza to, że wierzchołki w grafie odpowiadają łukom na okręgu i są połączone krawędzią wtedy i tylko wtedy, gdy odpowiadające im łuki na okręgu przecinają się. Grafy łuków na okręgu stanowią uogólnienie grafów przedziałowych, które z kolei są grafami przecięć odcinków na osi liczbowej.

Wiele praktycznych problemów może zostać wyrażone jako abstrakcyjne problemy dotyczące grafów łuków na okręgu, a zatem algorytmy dotyczące tych grafów mogą zostać wykorzystane do rozwiązywania praktycznych zagadnień. Jako przykład można podać cykle sygnalizacji świetlnej, które można modelować jako grafy łuków na okręgu i w ten sposób poszukiwać najbardziej optymalnych cykli pod względem na przykład minimalizacji czasu oczekiwania na zielone światło przez uczestników ruchu drogowego. Podobnie wiele innych problemów o cyklicznej strukturze, gdzie grafy przedziałowe są niewystarczające, może być modelowanych za pomocą grafów łuków na okręgu.

Praca zajmuje się badaniem szeroko pojętej tematyki grafów łuków na okręgu. Głównym wynikiem są implementacje, w języku Python, algorytmów dotyczących grafów łuków na okręgu. W szczególności skupiono się na podstawowych problemach w teorii grafów takich jak wyznaczanie największego zbioru niezależnego, wyznaczanie najmniejszego zbioru dominującego, wyznaczenie najmniejszego pokrycia klikowego oraz wyznaczanie największej kliki. W ramach pracy powstały także algorytmy sprawdzające spójność danego grafu, generujące graficzną reprezentację modelu grafu, a także przekształcające model grafu w graf abstrakcyjny.

Każda ze stworzonych implementacji została przetestowana pod kątem poprawności jak i wydajności. Nie stwierdzono przy tym rozbieżności między teoretyczną złożonością obliczeniową, a złożonością uzyskaną w praktyce.

**Słowa kluczowe:** graf przecięć, graf łuków, zbiór niezależny, zbiór dominujący, największa klika, pokrycie klikowe

**English title:** Circular-arc graphs


## Abstract

Circular-arc graphs form a family of graphs defined as intersection graphs of a set of arcs on the circle. This means that vertices in a circular-arc graph correspond to arcs on the circle and are connected by an edge if and only if the corresponding arcs on the circle intersect. Circular-arc graphs are a generalization of interval graphs, which in turn are intersection graphs of intervals on the number line.

Many practical problems can be stated as abstract problems related to circular-arc graphs, and thus algorithms dealing with these graphs can be used to solve practical tasks. As an example, traffic light phasing can be modelled with circular-arc graphs and thus the most optimal phasing can be searched in terms of, for example, minimizing the waiting time for traffic participants for a green light. Similarly, many other problems with a cyclic structure, where interval graphs are insufficient, can be modelled using circular-arc graphs.

This thesis explores the broad topic of circular-arc graphs. The main results are implementations, in the Python language, of algorithms regarding circular-arc graphs. In particular, the focus has been on basic problems in graph theory such as determining the largest independent set, determining the smallest dominating set, determining the smallest clique cover and determining the largest clique. This thesis also includes algorithms for checking the connectivity of a given graph, generating a graphical representation of a graph model, and transforming a graph model into an abstract graph.

Each of the created implementations was tested for correctness as well as performance. In doing so, no discrepancies were found between the theoretical computational complexity and the complexity obtained in practice.



**Keywords:** intersection graph, circular-arc graph, independent set, dominating set, maximum clique, clique cover

# Spis treści

# Spis rysunków

# Listings

# 1. Wstęp

Tematem niniejszej pracy są grafy łuków na okręgu [1], które definiuje się jako grafy przecięć dla łuków na okręgu. Jest to naturalne uogólnienie grafów przedziałowych [2], które są grafami przecięć dla odcinków na osi liczbowej. Precyzyjne definicje tych rodzin grafów znajdują się w rozdziale 2. O popularności grafów łuków na okręgu świadczy między innymi fakt, że na ich temat powstały prace przeglądowe [3] i [4].

Przejście od grafów przedziałowych do grafów łuków na okręgu oznacza wyjście z rodziny grafów cięciwowych, ale także wyjście z rodziny grafów doskonałych [5]. Obrazują to na przykład grafy cykliczne z nieparzystą liczbą wierzchołków $C_5$, $C_7$, itp. Warto zwrócić uwagę, że chociaż cykle parzyste $C_4$, $C_6$, itp. nie są cięciwowe, to są doskonałe jako grafy dwudzielne.

Dodatkową konsekwencją opuszczenia rodziny grafów przedziałowych na rzecz grafów łuków na okręgu jest znaczący wzrost poziomu trudności pewnych problemów grafowych. Przykładowo problem kolorowania wierzchołków grafu przedziałowego ma rozwiązanie o złożoności liniowej $O(n)$ (na bazie reprezentacji permutacyjnej [6]). Z kolei dla grafów łuków na okręgu problem kolorowania wierzchołków jest NP-zupełny [7].

## 1.1. Cel pracy

Praca ma na celu bliższe poznanie i zbadanie tematyki grafów łuków na okręgu. Głównym celem jest stworzenie w języku Python implementacji algorytmów rozwiązujących dla tej podklasy grafów podstawowe problemy teorii grafów, takie jak sprawdzanie spójności grafu czy też wyznaczanie największego zbioru niezależnego albo najmniejszego zbioru dominującego. Powstałe implementacje mają być wydajne, dlatego poza testowaniem ich poprawności wykonane zostaną testy złożoności obliczeniowej w celu porównania otrzymanej złożoności z teoretyczną złożonością obliczeniową zaimplementowanych algorytmów. Docelowo stworzone implementacje uzupełnią bibliotekę graphtheory [8].

## 1.2. Organizacja pracy

Praca podzielona jest na pięć rozdziałów. Rozdział 1 stanowi wstęp do pracy magisterskiej zarysowujący jej tematykę i cel oraz opisujący w skrócie zastosowania badanej podklasy grafów, jak i podający organizację pracy. W rozdziale 2 znajdują się podstawowe definicje dotyczące dziedziny jaką jest teoria grafów. W dalszej części tego rozdziału opisanych jest pokrótce kilka klas grafów o szczególnyym znaczeniu dla tej pracy. Są to grafy

przecięć, grafy przedziałowe oraz grafy łuków na okręgu. Rozdział ten zwieńczony jest rysem historycznym problemu rozpoznawania grafów łuków na okręgu. Rozdział 3 przedstawia implementacje wykonane w ramach tej pracy. Poza podaniem problemu jaki jest rozwiązywany przez dany algorytm prezentowany jest także sposób użycia danej implementacji. W rozdziale 4 znajdują się opisy zaimplementowanych algorytmów. Omawiana jest zasada działania oraz teoretyczna złożoność obliczeniowa każdego z algorytmów. Całość uzupełniona jest kluczowymi fragmentami kodu źródłowego stworzonej implementacji. Rozdział 5 stanowi podsumowanie pracy. Na końcu pracy umieszczono dodatek A prezentujący opisy oraz wyniki wykonanych testów złożoności obliczeniowej implementacji.

## 1.3. Zastosowania

Grafy łuków na okręgu, jako uogólnienie grafów przedziałowych, znajdują zastosowania w wielu problemach gdzie wykorzystywane są grafy przedziałowe. Szczegółowy przegląd zastosowań grafów przedziałowych można znaleźć w pracy Macieja Mularskiego [9]. Konkretnym przykładem zastosowania grafów łuków na okręgu może być opracowywanie wydajnych cykli sygnalizacji świetlnej [6]. Okrąg reprezentuje wtedy jeden taki pełny cykl, a łuki oznaczają posiadanie zielonego światła i są przypisane do poszczególnych pasów ruchu oraz odpowiadającym im sygnalizatorom. Jak łatwo zauważyć, dwóm pasom ruchu, które nie mogą mieć w tym samym czasie zielonego światła, nie mogą być przypisane przecinające się łuki. Takie modelowanie za pomocą grafu łuku na okręgu skrzyżowań drogowych z sygnalizacją świetlną pozwala także na uwzględnienie dodatkowych warunków jak na przykład minimalizacja czasu oczekiwania na zielone światło [10]. Istnieje wiele analogicznych problemów o podobnej strukturze do problemu cykli sygnalizacji świetlnej, gdzie grafy łuków na okręgu mogą znaleźć zastosowanie. Jest tak wszędzie tam, gdzie mamy do czynienia z pewnym ograniczonym zasobem, który chcemy jak najefektywniej wykorzystać albo też jak najsprawiedliwiej podzielić dostęp do niego. Jeśli te problemy posiadają cykliczną strukturę, czyli zdarzenia powtarzają się co pewien interwał czasowy, jak na przykład co minutę, godzinę, dobę czy też tydzień, to do optymalnego opisu takiego problemu nie wystarczą grafy przedziałowe, tylko należy skorzystać z grafów łuków na okręgu. Zatem oprócz sygnalizacji świetlnej możemy mieć do czynienia z planowaniem przydziału pamięci co jest ważnym zagadnieniem przy projektowaniu kompilatorów [11]. Na koniec warto jeszcze zwrócić uwagę, że jeśli popatrzymy na sam problem planowania z szerszej perspektywy, nie tylko ograniczonej do urządzeń i sprzętu komputerowego, to okażę się, że wiele harmonogramów dotyczących organizacji społecznej ludzi także daje się w prosty sposób odzwierciedlić za pomocą grafów łuków na okręgu [12].

# 2. Teoria grafów

W tym rozdziale przedstawione zostaną podstawowe definicje i twierdzenia, które będą przydatne w dalszej części pracy.

## 2.1. Podstawowe definicje

**Definicja:** Graf nieskierowany $G$ jest uporządkowaną parą $(V, E)$, gdzie $V$ jest zbiorem wierzchołków, a $E$ jest zbiorem krawędzi. Krawędź $e$ jest dwuelementowym zbiorem różnych wierzchołków, które nazywamy końcami $e$. Będziemy stosować następującą notację na oznaczenie liczby wierzchołków i krawędzi w grafie $G = (V, E)$: $|V| = n$, $|E| = m$.

**Definicja:** Graf skierowany $G$ jest uporządkowaną parą $(V, E)$, gdzie $V$ jest zbiorem wierzchołków, a $E$ jest zbiorem krawędzi. Krawędź $e = (u, v)$ jest uporządkowaną parą różnych wierzchołków, przy czym $u$ jest początkiem $e$, a $v$ jest końcem $e$. Oznaczenia $n$ i $m$ są takie jak dla grafów nieskierowanych.

**Definicja:** Graf $G = (V, E)$ jest dwudzielny, jeśli istnieją rozłączne zbiory wierzchołków $V_1, V_2$ takie, że $V = V_1 \cup V_2$ oraz żadne dwa wierzchołki należące do $V_i$ nie są połączone krawędzią dla $i \in \{1, 2\}$.

**Definicja:** Podgraf $H = (V_2, E_2)$ grafu $G = (V_1, E_1)$ to graf, w którym $V_2 \subseteq V_1$ oraz $E_2 \subseteq E_1$, przy czym końce krawędzi z $E_2$ należą do $V_2$.

**Definicja:** Podgraf $H = (V_2, E_2)$ grafu $G = (V_1, E_1)$ indukowany wierzchołkowo to podgraf powstający z grafu $G$ poprzez usunięcie pewnych wierzchołków, przy czym krawędzie nie są usuwane o ile ich koniec nie jest usuniętym wierzchołkiem. Innymi słowy, zbiór wierzchołków $V_2$ jest ustalany jako pewien dowolny podzbiór zbioru $V_1$, ale już postać zbioru krawędzi $E_2$ jest narzucona przez konkretny wybór zbioru $V_2$. Zbiór $E_2$ powstaje ze zbioru $E_1$ poprzez usunięcie krawędzi, których przynajmniej jeden koniec nie należy do zbioru wierzchołków $V_2$.

**Definicja:** Podgraf $H = (V_2, E_2)$ grafu $G = (V_1, E_1)$ indukowany krawędziowo to podgraf powstający z grafu $G$ poprzez usunięcie pewnych krawędzi, przy czym wierzchołki nie są usuwane, o ile istnieje krawędź do której należą. Innymi słowy, zbiór krawędzi $E_2$ jest ustalany jako pewien dowolny podzbiór zbioru $E_1$, ale już postać zbioru wierzchołków $V_2$ jest narzucona przez konkretny wybór zbioru $E_2$. Zbiór $V_2$ powstaje ze zbioru $V_1$ poprzez usunięcie wierzchołków, które nie są końcem żadnej krawędzi w zbiorze $E_2$.

**Definicja:** Dopełnienie grafu $G$ jest grafem $G^c$ o takim samym zbiorze wierzchołków co $G$, przy czym między dwoma wierzchołkami w $G^c$ istnieje krawędź wtedy i tylko wtedy, gdy nie istnieje ona w $G$.

**Definicja:** Ścieżka $P$ w grafie $G$ jest ciągiem różnych krawędzi $e_1, e_2, \ldots, e_n$ grafu $G$ takich, że $e_i$ oraz $e_{i+1}$ dla $1 \leqslant i < n$ posiadają wspólny koniec, gdzie $e_1 - e_2$ nazywamy pierwszym wierzchołkiem ścieżki $P$, a $e_n - e_{n-1}$ ostatnim.

**Definicja:** Cykl w grafie $G$ jest ścieżką w $G$, w której pierwszy wierzchołek jest zarazem ostatnim, tzn. $e_1 - e_2 = e_n - e_{n-1}$. Graf nieskierowany, którego wszystkie $n$ wierzchołków tworzy cykl, oznaczamy jako $C_n$ i nazywamy grafem cyklicznym.

**Definicja:** Graf nieskierowany $G$ jest spójny, jeśli między każdymi dwoma wierzchołkami grafu $G$ istnieje ścieżka.

**Definicja:** Zbiór niezależny $I$ jest zbiorem takich wierzchołków grafu, że między dowolnymi dwoma wierzchołkami należącymi do $I$ nie istnieje krawędź.

**Definicja:** Największy zbiór niezależny grafu to zbiór niezależny grafu o największej liczbie wierzchołków. Innymi słowy, nie może istnieć zbiór niezależny składający się z większej liczby wierzchołków niż największy zbiór niezależny.

**Definicja:** Klika $K$ jest zbiorem takich wierzchołków grafu, że między każdymi dwoma wierzchołkami należącymi do $K$ istnieje krawędź. Graf nieskierowany, którego wszystkie $n$ wierzchołków wchodzi w skład kliki, oznaczamy jako $K_n$ i nazywamy grafem pełnym.

**Definicja:** Pokrycie klikowe grafu $G$ to zbiór takich klik, że ich suma to zbiór wszystkich wierzchołków grafu $G$. Innymi słowy, każdy wierzchołek grafu $G$ musi należeć do pewnej kliki należącej do pokrycia klikowego.

**Definicja:** Najmniejsze pokrycie klikowe grafu to pokrycie klikowe grafu o najmniejszej liczbie klik. Innymi słowy, nie może istnieć pokrycie klikowe składające się z mniejszej liczby klik niż najmniejsze pokrycie klikowe.

**Definicja:** Największa klika jest kliką taką, że nie istnieje w grafie klika o większej liczbie wierzchołków.

**Definicja:** Zbiór dominujący $D$ jest zbiorem takich wierzchołków grafu, że dla każdego wierzchołka grafu nienależącego do $D$ istnieje krawędź łącząca go z pewnym wierzchołkiem należącym do $D$.

**Definicja:** Najmniejszy zbiór dominujący grafu to zbiór dominujący grafu o najmniejszej liczbie wierzchołków. Innymi słowy, nie może istnieć zbiór dominujący składający się z mniejszej liczby wierzchołków niż najmniejszy zbiór dominujący.

## 2.2. Grafy przecięć

Graf przecięć (ang. *intersection graph*) jest to graf nieskierowany określony przez rodzinę zbiorów $\{S_i\}$. Dla każdego zbioru $S_i$ tworzy się jeden wierzchołek grafu $v_i$. Dwa wierzchołki $v_i$ oraz $v_j$ są połączone krawędzią wtedy i tylko wtedy, gdy zbiory $S_i$ oraz $S_j$ mają niepuste przecięcie [13].

Warto zauważyć, że każdy graf nieskierowany $G$ może być reprezentowany w postaci grafu przecięć. Dla każdego wierzchołka $v_i$ z $V(G)$ tworzymy zbiór $S_i$ zawierający krawędzie nieskierowane incydentne z wierzchołkiem $v_i$. Wtedy dwa zbiory $S_i$ oraz $S_j$ będą miały niepuste przecięcie tylko wtedy, gdy wierzchołki $v_i$ oraz $v_j$ są połączone krawędzią.

## 2.3. Grafy przedziałowe

Grafy przedziałowe (ang. *interval graphs*) definiuje się jako grafy przecięć odcinków na osi liczbowej [2]. Bez zmniejszenia ogólności przyjmuje się, że odcinki są domknięte, oraz że końce wszystkich odcinków są różne. Grafy przedziałowe były analizowane w pracy magisterskiej Macieja Mularskiego [9].

## 2.4. Grafy łuków na okręgu

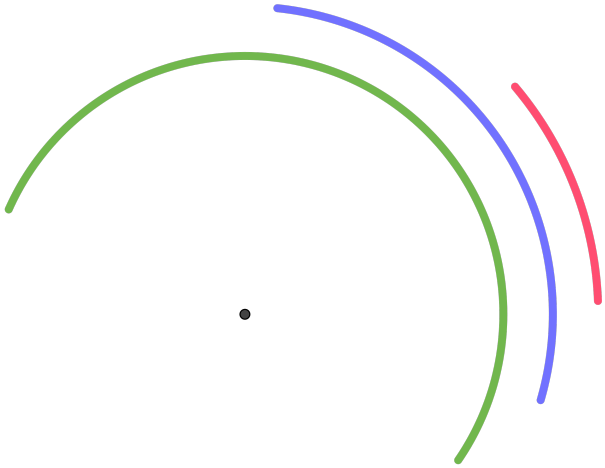
Grafy łuków na okręgu (ang. *circular-arc graphs*) definiuje się jako grafy przecięć łuków na okręgu [1]. Bez straty ogólności zakłada się, że łuki są domknięte (zawierają końce), oraz że końce wszystkich łuków są różne. Każdy graf przedziałowy jest grafem łuków na okręgu, wystarczy "nawinąć" odcinki na okrąg. Z drugiej strony, jeżeli łuki na okręgu nie pokrywają całego okręgu, to można rozciąć okrąg i dostać model odcinków na prostej.

Warto zauważyć, że dany abstrakcyjny graf łuków na okręgu może mieć kilka istotnie różnych modeli z łukami. Przykładem jest graf pełny $K_3$, który można przedstawić za pomocą dwóch różnych modeli z łukami. W pierwszym modelu, przedstawionym na rysunku 2.1, mamy trzy częściowo nachodzące na siebie łuki, przypominających stos odcinków na prostej. W drugim modelu, widocznym na rysunku 2.2, mamy trzy łuki całkowicie pokrywające okrąg, przy czym nie będzie punktu pokrytego jednocześnie przez trzy łuki.

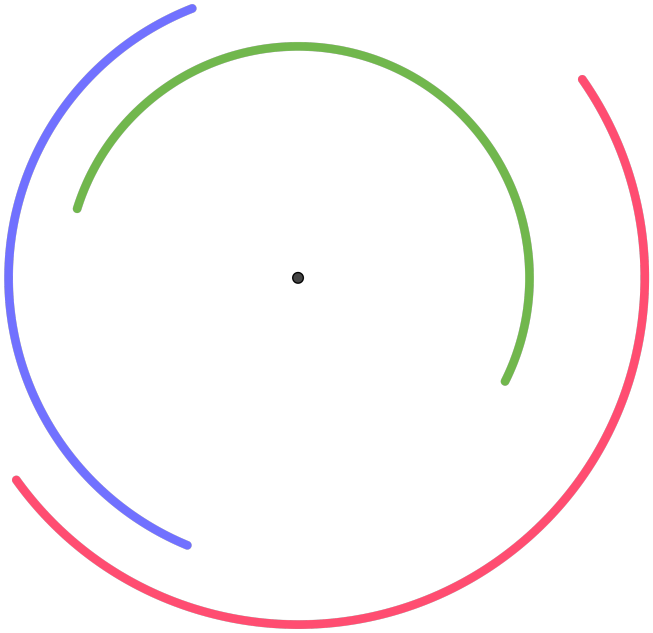
### 2.4.1. Rozpoznawanie grafów łuków na okręgu

O grafach łuków na okręgu w literaturze naukowej wspomniano pierwszy raz prawdopodobnie w 1964 roku [14]. Nie wykraczało to jednak poza stwierdzenie, że o ile grafy przedziałowe zostały już scharakteryzowane (na przykład w [15]), o tyle odnośnie grafów przecięć rodzin innych zbiorów, w tym także rodzin łuków na okręgu, nie uzyskano w tamtym czasie jeszcze żadnych wyników.

Na wstępie warto zaznaczyć, że problemy dotyczące grafów łuków na okręgu, w tym rozpoznawanie takich grafów, są problemami znacząco trudniejszymi od analogicznych problemów dotyczących grafów przedziałowych



Rysunek 2.1. Jeden z możliwych modeli grafu pełnego $K_3$. Trzy łuki posiadają wspólne punkty okręgu.



Rysunek 2.2. Jeden z możliwych modeli grafu pełnego $K_3$. Trzy łuki nie posiadają wspólnego punktu na okręgu.

[3]. Wynika to z faktu, że w przypadku grafów przedziałowych mamy do czynienia z osią liczbową, która jest ograniczona przez swój koniec zarówno z lewej jak i z prawej strony, tzn. można osiągnąć punkt, poza którym nie będzie przedziałów należących do grafu. Jednak na okręgu możemy podróżować w obu kierunkach cyklicznie bez końca. O ile w przypadku przedziałów na osi liczbowej oczywiste jest, że przedziały tworzące klikę muszą zawierać wspólny punkt, o tyle w przypadku łuków na okręgu wcale nie musi tak być (taką sytuację przedstawia rysunek 2.2).

W literaturze matematycznej używa się pojęcia *właściwości Hellego*. Przedziały na osi liczbowej mają właściwość Hellego, czyli dla $k$ przedziałów mających parami niepuste przecięcie, przecięcie wszystkich $k$ przedziałów również ma niepuste przecięcie. Zbiory łuków na okręgu zwykle nie mają właściwości Hellego. Czasem definiuje się grafy Hellego łuków na okręgu (ang. *Helly circular-arc graphs*), dla których istnieje model z łukami mający właściwość Hellego. W roku 1974 Gavril scharakteryzował te grafy, co doprowadziło do powstania algorytmu rozpoznawania grafów w czasie $O(n^3)$ [16].

Pierwszy wielomianowy algorytm rozpoznawania grafów łuków na okręgu został podany przez Alana Tuckera w roku 1980 [17]. Złożoność obliczeniowa algorytmu wynosiła $O(n^3)$, przy czym algorytm oraz dowód jego poprawności był, jak przyznał sam autor, dość skomplikowany. Mimo to, algorytm Tuckera okazał się przełomowy. Późniejsze algorytmy korzystały intensywnie z wprowadzonych w nim idei. Główna struktura algorytmu, polegająca na rozważaniu dwóch różnych przypadków, w zależności od tego czy dopełnienie wejściowego grafu jest grafem dwudzielnym czy też nie, oraz dalszy podział jednego z tych przypadków na dwa, stanowiła szkielet na którym bazowały późniejsze algorytmy.

Pierwszego rozwinięcia algorytmu Tuckera, całe 13 lat po jego opublikowaniu, dokonali Elaine Eschen i Jeremy Spinrad [18]. Utrzymują oni, że dzięki wprowadzeniu nowych technik oraz sprowadzeniu pewnych podproblemów do problemów dotyczących grafów cięciwowych przy zachowaniu pierwotnej struktury całego algorytmu, udało im się zredukować złożoność do $O(n^2)$.

W podobnym okresie pojawiła się też praca Wen-Lian Hsu opisująca algorytm rozpoznawania grafów łuków na okręgu o złożoności $O(mn)$ [19]. Pomysł, na którym opiera się ten algorytm, jest konceptualnie inny od pomysłu Tuckera i ma w zamyśle uprościć proces rozpoznawania grafów. Hsu opisał w tej samej pracy także algorytm służący stwierdzeniu czy dane grafy są izomorficzne. Ponad 15 lat po jego publikacji w pracy [20] zaprezentowano kontrprzykład dowodzący niepoprawności algorytmu dotyczącego izomorfizmu. W roku 2024 Tomasz Krawczyk pokazał w [21], że także algorytm rozpoznawania grafów łuków na okręgu autorstwa Hsu jest błędny.

Pierwszy algorytm rozpoznawania grafów łuków na okręgu działający w czasie liniowym $O(n + m)$ podał Ross McConnell w roku 2003 [22]. Jego zasada działania odbiega od tej zaproponownej przez Tuckera, gdyż wykorzystuje redukcję do problemu rozpoznawania grafów przedziałowych. Jednakże aby tego dokonać, konieczny jest preprocesing analogiczny do tego, jakiego dokonują Eschen i Spinrad w czasie kwadratowym. Niezbędne było więc pokazanie przez McConnella, że możliwe jest wykonanie wymaganego preprocesingu w czasie liniowym. Ciekawe jest także, że chociaż McConnell

w swojej pracy opisuje jak uzyskać algorytm działający w całości w czasie liniowym, to w przypadku kiedy konieczna jest praktyczna implementacja, sugeruje on użycie wariantu algorytmu, który także przedstawia, o złożoności $O(n + m \log n)$. Jest to motywowane tym, że otrzymanie liniowego czasu działania jest dosyć skomplikowane.

W roku 2011 zaprezentowany został kolejny liniowy algorytm, tym razem autorstwa Haima Kaplana i Yahava Nussbauma [23]. Pod względem podejścia do problemu ulepszyli oni kwadratowy algorytm przedstawiony przez Eschena i Spinrada, można więc powiedzieć, że stanowi on także rozwinięcie pierwotnego algorytmu Tuckera. Co ciekawe, autorzy zwracają uwagę na pewien błąd w algorytmie Eschena i Spinrada. Kaplan i Nussbaum nie tylko korygują dotychczas błędne fragmenty algorytmu Eschena i Spinrada, ale także dokładniej analizują jego złożoność. W ten sposób, przy minimalnych modyfikacjach w strukturze całego algorytmu, otrzymują algorytm o złożoności liniowej, który jest prostszy od zaprezentowanego przez McConnella.

Warto zauważyć, że wymienione powyżej algorytmy nie tylko stwierdzają czy dany graf jest grafem łuków na okręgu, ale także generują jego model jako rodzina łuków na okręgu. Należy pamiętać, że taki model powstaje także często w przypadkach, kiedy nie mamy do czynienia z grafem łuków na okręgu. Z tego powodu wiele algorytmów pod koniec sprawdza czy otrzymany model rzeczywiście odpowiada grafowi, ktory został podany na wejściu. McConnell podał w pracy [22], przy okazji swojego liniowego algorytmu rozpoznawania grafów łuków na okręgu, nieskomplikowany oraz działający w czasie liniowym sposób dokonania takiego sprawdzenia.

# 3. Implementacja grafów

W tym rozdziale przedstawione zostaną przykładowe sesje interaktywne w języku Python. Pozwalają one zobaczyć w jaki sposób można korzystać z zaimplemetowanych w ramach tej pracy algorytmów.

Warunkiem wstępnym wykonania poniższych algorytmów jest zainstalowany pakiet graphtheory. Można to zrobić za pomocą polecenia pip install graphtheory. Następnie należy zaimportować klasy Edge oraz Graph.

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

## 3.1. Konwencja dotycząca reprezentacji grafów

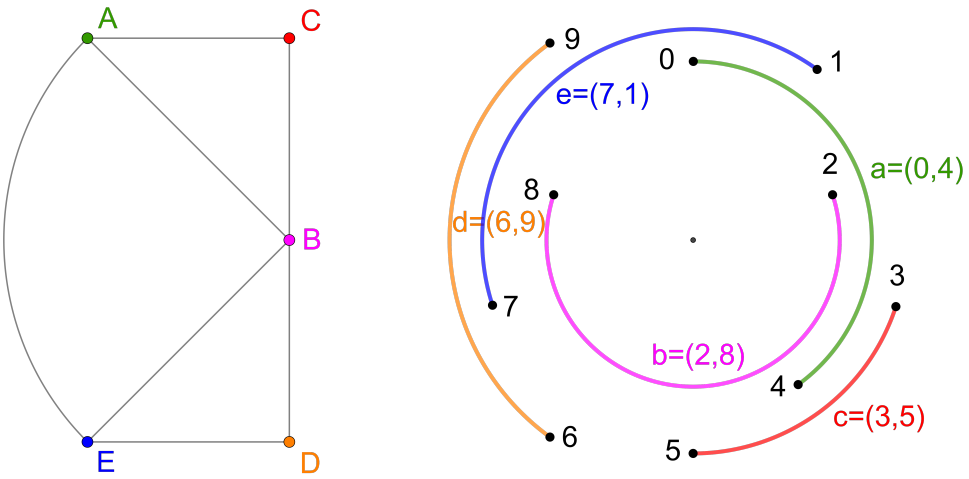
Jeśli nie zaznaczono inaczej, w całej pracy przyjmujemy reprezentację grafów łuków na okręgu za pomocą listy par zdarzeń. Dokładny opis tej reprezentacji można znaleźć w rozdziale pierwszym w pracy [24]. Przy $n$ łukach zdarzenia numerujemy od 0 do $2n-1$. Każdy łuk opisany jest za pomocą pary zdarzeń $(h, t)$. Pierwszy element pary informuje nas o numerze zdarzenia odpowiadającemu początkowi łuku (*head*). Z kolei drugi element jest numerem zdarzenia oznaczającego koniec danego łuku (*tail*). Przyjmujemy, że łuki zorientowane są zgodnie z kierunkiem ruchu wskazówek zegara. Na rysunku 3.1 zostało zaprezentowane zastosowanie tej konwencji.

Warto zauważyć, że $h$ może być większe od $t$. Taki łuk nazywać będziemy łukiem *wstecznym* czy też *backward*. Można myśleć o tym tak, że łuk *wsteczny* przechodzi przez "łączenie" okręgu, tzn. przez fragment okręgu pomiędzy zdarzeniem $2n-1$ oraz zdarzeniem 0. Łuk, którego koniec następuje po początku, czyli zachodzi $h < t$, nazywać będziemy łukiem *forward*. Lista wszystkich $n$ par opisujących łuki tworzy reprezentację grafu łuków na okręgu, która jest oczekiwana przez algorytmy opisane w dalszej części pracy.

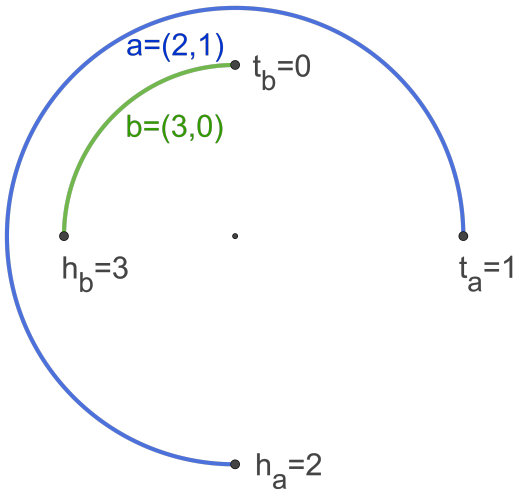
Dwa łuki, $a = (h_a, t_a)$ oraz $b = (h_b, t_b)$, mogą znajdować się w stosunku do siebie w różnych relacjach opisanych poniżej.

Łuk $a$ zawiera łuk $b$, czy też inaczej łuk $b$ jest zawierany przez łuk $a$, jeśli zaczynając w punkcie $h_a$ i idąc zgodnie z kierunkiem ruchu wskazówek zegara napotykamy końce tych łuków w następującej kolejności: $h_a$, $h_b$, $t_b$ oraz $t_a$. Należy zwrócić uwagę na fakt, że wcale nie musi zachodzić nierówność $h_a < h_b < t_b < t_a$, gdyż całość odbywa się na okręgu i, inaczej niż w przypadku osi liczbowej, umiejscowienie na okręgu zdarzenia o numerze 0 jest umowne. Taką sytuację przedstawia rysunek 3.2.

Łuk $a$ zawiera zdarzenie $c$, jeśli zaczynając w punkcie $h_a$ i idąc zgodnie z kierunkiem ruchu wskazówek zegara napotykamy zdarzenia w następują-




Rysunek 3.1. Przykładowy graf i jeden z jego modeli. Każdy łuk okręgu został opisany zgodnie z konwencją przedstawioną w podrozdziale 3.1. Czarne punkty to odpowiednio ponumerowane zdarzenia.




Rysunek 3.2. Przykładowy model grafu w którym łuk $a$ zawiera łuk $b$. Nierówność $h_a < h_b < t_b < t_a$ nie jest prawdziwa.

cej kolejności: $h_a$, $c$ oraz $t_a$. Nierówność $h_a < c < t_a$ nie musi być jednak prawdziwa.

Łuk $a$ przecina się z łukiem $b$, jeśli co najmniej jeden z tych łuków zawiera dowolny koniec drugiego łuku. W przeciwnym razie łuki te nie przecinają się.

## 3.2. Generowanie grafów łuków na okręgu

Wszystkie przedstawione w tym podrozdziale funkcje zwracają listę zawierającą pary zdarzeń reprezentujące wygenerowany graf. Zwracana lista jest posortowana rosnąco według początków łuków, a początek pierwszego łuku to zawsze zdarzenie o numerze 0. Złożoność obliczeniowa każdej z przedstawionych funkcji jest liniowa.

Losowy graf łuków na okręgu o $n$ wierzchołkach możemy wygenerować za pomocą funkcji make_random_arc.

```
>>> graph_model = make_random_arc(5)
>>> print(graph_model)
[(0, 4), (2, 3), (5, 9), (6, 1), (8, 7)]
```

Graf łuków na okręgu będący grafem cyklicznym o $n$ wierzchołkach możemy wygenerować za pomocą funkcji make_cycle_arc.

```
>>> graph_model = make_cycle_arc(5)
>>> print(graph_model)
[(0, 3), (2, 5), (4, 7), (6, 9), (8, 1)]
```

Losowy graf łuków na okręgu będący zarówno grafem pełnym o $n$ wierzchołkach jak i jednocześnie grafem przedziałowym możemy wygenerować za pomocą funkcji make_complete_random_interval.

```
>>> graph_model = make_complete_random_interval(5)
>>> print(graph_model)
[(0, 7), (1, 9), (2, 8), (3, 5), (4, 6)]
```

Losowy graf łuków na okręgu będący grafem pełnym o $n$ wierzchołkach, mający właściwość Hellego, ale niebędący grafem przedziałowym, możemy wygenerować za pomocą funkcji make_complete_helly_arc.

```
>>> graph_model = make_complete_helly_arc(5)
>>> print(graph_model)
[(0, 7), (1, 8), (2, 6), (3, 9), (5, 4)]
```

Losowy graf łuków na okręgu będący grafem pełnym o $n$ wierzchołkach, ale niemający właściwości Hellego oraz niebędący grafem przedziałowym, możemy wygenerować za pomocą funkcji make_complete_not_helly_arc.

```
>>> graph_model = make_complete_not_helly_arc(5)
>>> print(graph_model)
[(0, 5), (2, 9), (3, 7), (4, 8), (6, 1)]
```

## 3.3. Graficzna reprezentacja łuków grafu

Funkcja circular_drawing pozwala na wypisanie w terminalu postaci graficznej łuków. W zwracanej reprezentacji okrąg jest rozcięty. Należy sobie zatem wyobrazić, że koniec grafiki po prawej stronie przechodzi w początek grafiki po lewej stronie. W przypadku łuków wstecznych jest to niezbędna obserwacja.

Funkcja oferuje dwie opcje oznaczania łuków. Domyślna, niewymagająca podania dodatkowego parametru, wypisuje numery łuków. W przypadku podania jako drugiego argumentu wartości False, wypisane zostaną ponumerowane zdarzenia początku oraz końca każdego łuku. Poniżej przedstawiony jest przykładowy wynik wywołania funkcji w obu wariantach.

```
>>> graph_model = make_random_arc(10)
>>> print(graph_model)
[(5,9),(11,0),(18,6),(1,10),(13,3),(7,8),(4,14),(12,19),(16,15),
(2,17)]
>>> circular_drawing(graph_model)
-1  3-------------------------3  1----------------------
       9-------------------------------------------9
----------4      0----------0            4-----------------
             6-----------------------------6
------------------2  5--5                                2----
                                      7-------------------7
-----------------------------------------8  8----------
```

```
>>> graph_model = make_random_arc(7)
>>> print(graph_model)
[(11, 1), (13, 4), (3, 5), (9, 12), (2, 0), (7, 10), (6, 8)]
>>> circular_drawing(graph_model, False)
-0        2-------------------------------------------
-----1          3------5     6------8            11---------
-----------------4              7----------10           13-
                                    9----------12
```

## 3.4. Sprawdzanie spójności grafów łuków na okręgu

Za pomocą funkcji check_connectivity możemy sprawdzić czy dany graf łuków na okręgu w reprezentacji par zdarzeń jest grafem spójnym. Konieczne jest jednak podanie drugiego argumentu, którym jest wynik wykonania pomocniczej funkcji calculate_endpoints dla tego grafu.

```
>>> graph_model = make_random_arc(5)
>>> print(graph_model)
[(5, 0), (1, 6), (4, 2), (7, 8), (9, 3)]
>>> check_connectivity(graph_model, calculate_endpoints(graph_model))
True
```

## 3.5. Przekształcenie modelu par zdarzeń w graf abstrakcyjny

Mając dany graf łuków na okręgu w postaci listy par zdarzeń możemy taki model grafu przekształcić w graf abstrakcyjny klasy Graph, zdefiniowanej w pakiecie graphtheory, za pomocą funkcji make_abstract_arc_graph_efficient.

```
>>> graph_model = make_random_arc(5)
>>> print(graph_model)
[(8, 3), (7, 2), (1, 9), (4, 5), (6, 0)]
>>> G = make_abstract_arc_graph_efficient(graph_model)
>>> G.show()
0 : 1 2 4
1 : 0 2 4
2 : 0 1 3 4
3 : 2
4 : 0 1 2
```

## 3.6. Wyznaczanie największego zbioru niezależnego

Największy zbiór niezależny w grafie łuków na okręgów można wyznaczyć trzema różnymi funkcjami, GREEDY, FIND_MIAF oraz MIS_gavril. Należy pamiętać, że dla jednego grafu może istnieć kilka zbiorów będących największym zbiorem niezależnym, dlatego też wyniki wykonania tych funkcji dla danego grafu mogą się różnić. Złożoność funkcji GREEDY oraz FIND_MIAF jest liniowa, w przypadku MIS_gavril jest kwadratowa.

Poniżej przedstawione jest użycie wszystkich wymienionych funkcji. W przypadku GREEDY jako drugi argument należy podać "mis". W przypadku FIND_MIAF początek pierwszego łuku musi być zdarzeniem o numerze 0.

```
>>> graph_model = make_random_arc(5)
>>> print(graph_model)
[(0, 3), (4, 1), (7, 8), (5, 6), (9, 2)]
>>> GREEDY(graph_model, 'mis')
[3, 2, 4]
```

```
>>> graph_model = [(0, 3), (4, 1), (7, 8), (5, 6), (9, 2)]
>>> FIND_MIAF(graph_model)
[0, 3, 2]
```

```
>>> graph_model = [(0, 3), (4, 1), (7, 8), (5, 6), (9, 2)]
>>> MIS_gavril(graph_model)
{2, 3, 4}
```

## 3.7. Wyznaczanie najmniejszego pokrycia klikowego

W celu wyznaczenia najmniejszego pokrycia klikowego należy użyć funkcji GREEDY podając jako drugi argument "mqc".

```
>>> graph_model = make_random_arc(5)
>>> print(graph_model)
[(3, 4), (7, 6), (1, 0), (2, 5), (8, 9)]
>>> GREEDY(graph_model, 'mqc')
[{0, 1, 2, 3}, {1, 2, 4}]
```

## 3.8. Wyznaczanie najmniejszego zbioru dominującego

W celu wyznaczenia najmniejszego zbioru dominującego należy użyć funkcji GREEDY podając jako drugi argument "mds".

```
>>> graph_model = make_random_arc(5)
>>> print(graph_model)
[(8, 4), (7, 6), (0, 1), (3, 2), (5, 9)]
>>> GREEDY(graph_model, 'mds')
[3]
```

## 3.9. Wyznaczanie największej kliki

Za pomocą funkcji max_clique_gavril można wyznaczyć największą klikę w grafie.

```
>>> graph_model = make_random_arc(5)
>>> print(graph_model)
[(3, 1), (4, 5), (2, 8), (7, 9), (0, 6)]
>>> max_clique_gavril(graph_model)
[0, 2, 4, 1]
```

# 4. Algorytmy

W tym rozdziale zajmiemy się różnorodnymi problemami dotyczącymi grafów łuków na okręgu. Znajdziemy wśród nich zagadnienie wyznaczania największego zbioru niezależnego, najmniejszego pokrycia klikowego, czy też najmniejszego zbioru dominującego. Przedstawione zostaną implementacje wraz z opisami działania i złożoności obliczeniowej odpowiednich algorytmów.

## 4.1. Funkcja pomocnicza

Warto na początku wspomnieć o funkcji pomocniczej calculate_endpoints(), którą często należy wykonać przed wywołaniem docelowych algorytmów omówionych w tym rozdziale, ponieważ oczekują one w większości przypadków podania jako argumentu wyniku tej funkcji. Zaletą takiego podejścia jest to, że jeśli zamierzamy rozwiązać wiele problemów dla danego grafu, to wystarczy tylko raz wywołać calculate_endpoints() i podać wynik jako argument dla pozostałych funkcji. Nie musimy zatem wielkrotnie powtarzać obliczeń, a miałoby to niestety miejsce, gdyby ta funkcja wołana była bezpośrednio przez algorytmy.

Celem funkcji calculate_endpoints() jest stworzenie struktury, która pozwoli później w czasie stałym na identyfikację łuku, do którego należy dane zdarzenie oraz na stwierdzenie, czy dane zdarzenie jest początkiem czy też końcem łuku. Złożoność obliczeniowa funkcji, ze względu na jednorazowy przegląd łuków, wynosi $O(n)$.

Listing 4.1. Funkcja pomocnicza calculate_endpoints().

```
def calculate_endpoints(graph_model, size):    # O(n) time
    endpoints = [0]*size*2
    i = 0
    # endpoints[i] gives the arc the endpoint with position i
    # belongs to and the information if it is head or tail
    # endpoints[coordinate i]=(arc, head/tail)
    for (head, tail) in graph_model:
        endpoints[head] = {'arc': i, 'end': "head"}
        endpoints[tail] = {'arc': i, 'end': "tail"}
        i = i+1
    return endpoints
```

## 4.2. Graficzna reprezentacja łuków grafu

Przedstawiony tutaj algorytm jest uogólnieniem na grafy łuków na okręgu algorytmu Macieja Mularskiego, opisanego w pracy [9], dotyczącego gra-

fów przedziałowych. Oba algorytmy służą wygenerowaniu graficznej postaci danych przedziałów czy też łuków grafu. Możliwość pojawienia się łuków wstecznych, czyli takich dla których koniec następuje przed początkiem, w grafach łuków na okręgu stwarza konieczność dodania do algorytmu dla grafów przedziałowych sposobu postępowania z takimi łukami. Poza tym idea algorytmu pozostaje niezmieniona.

Warto zauważyć, że generowana przez algorytm graficzna reprezentacja prezentowana jest w postaci rozciętego okręgu. Należy więc pamiętać, że prawa strona grafiki kontynuowana jest z jej lewej strony, jak można zauważyć w przykładzie w podrozdziale 3.3.

Listing 4.2. Graficzna reprezentacja łuków grafu.

```
def circular_drawing(graph_model, print_arcs=True,
    empty_symbol=' ', full_symbol='-'):
    size = len(graph_model)
    if size == 0:
        print()
        return
    line_used = [False]*size
    line_to_backward_head = [math.inf]*size
    arc_to_line = [-1]*size
    if print_arcs:
        symbol_len = math.ceil(math.log(size, 10))
    else:
        symbol_len = math.ceil(math.log(2*size, 10))
    empty = empty_symbol*(symbol_len + 2)
    full = full_symbol*(symbol_len + 2)
    str_lines = []
    endpoints = calculate_endpoints(graph_model, size)
    max_used_index =- 1
    for number_of_endpoint, endpoint in enumerate(endpoints):
        curr_arc = endpoint['arc']
        if print_arcs:
            identifier = str(curr_arc)
        else:
            identifier = str(number_of_endpoint)
        (head, tail) = graph_model[curr_arc]
        padding = 2 + (symbol_len-len(identifier))
        padding_smaller = padding // 2
        padding_larger = padding-padding_smaller
        if head > tail:
            if endpoint['end'] == 'tail':
                str_lines.append('')
                line_index = max_used_index + 1
                max_used_index = line_index
                line_to_backward_head[line_index] = head
                arc_to_line[curr_arc] = line_index
                str_lines[line_index] = (full * number_of_endpoint
                    + full_symbol * padding_smaller + identifier
                    + empty_symbol * padding_larger)
            else:
                line_index = arc_to_line[curr_arc]
                line_used[line_index] = True
                str_lines[line_index] += (empty_symbol * padding_larger
                    + identifier + full_symbol * padding_smaller)
```

```
        else:
            if endpoint['end'] == 'head':
                line_index -= 1
                while True:
                    line_index = line_index + 1
                    if (not line_used[line_index] and
                        tail < line_to_backward_head[line_index]):
                        break
                if max_used_index < line_index:
                    str_lines.append('')
                    str_lines[line_index] = empty * number_of_endpoint
                    max_used_index = line_index
                arc_to_line[curr_arc] = line_index
                line_used[line_index] = True
                str_lines[line_index] += (empty_symbol * padding_larger
                    + identifier + full_symbol * padding_smaller)
            else:
                line_index = arc_to_line[curr_arc]
                str_lines[line_index] += (full_symbol * padding_smaller
                    + identifier + empty_symbol * padding_larger)
                line_used[line_index] = False
        for i in range(max_used_index+1):
            if i == line_index:
                continue
            if line_used[i]:
                str_lines[i] += full
            else:
                str_lines[i] += empty
    for line in str_lines:
        print(line)
```

## 4.3. Sprawdzanie spójności grafów łuków na okręgu

Graf przedziałowy jest niespójny, jeśli istnieje fragment osi liczbowej nienależący do żadnego przedziału i otoczony przedziałami. Inaczej wygląda to w przypadku grafu łuków na okręgu, który pozostaje w takiej sytuacji spójny, ze względu na swoją strukturę okręgu. Dopiero gdy istnieją co najmniej dwa takie fragmenty okręgu, które nie należą do żadnego łuku i są od siebie rozdzielone łukami, to mamy do czynienia z niespójnym grafem łuków na okręgu.

Na tej obserwacji opiera się istota działania funkcji check_connectivity(), która przechodząc po kolejnych zdarzeniach odpowiednio dodaje i usuwa łuk ze zbioru obecnie odwiedzanych łuków, w zależności od tego, czy napotkano początek, czy też koniec tego łuku. Celem jest policzenie ile razy zbiór odwiedzanych łuków stanie się pusty w czasie obejścia całego okręgu. Sprawę komplikuje nieco fakt, że poruszamy się po okręgu, a nie po osi liczbowej. Zbiór łuków może być pusty, a mimo wszystko może istnieć łuk, który zawiera obecnie rozważany fragment okręgu. Dzieje się tak, gdy ze względu na ciągłość okręgu, nie rozważaliśmy jeszcze początku tego okręgu. Przedstawiona tutaj funkcja radzi sobie z tym poprzez tworzenie listy fragmentów łuków,

co do których mamy zasadne przypuszczenie, że nie należą do żadnego łuku, oraz weryfikowanie tego przypuszczenia w odpowiednich momentach.

Złożoność prezentowanej funkcji wynosi $O(n)$. Zawiera ona dwie pętle **for**, przy czym jedna z tych pętli jest zagnieżdzona w drugiej. Z każdym obiegiem wewnętrznej pętli, poza obiegami przerwanymi od razu w pierwszej iteracji przez **break**, wiąże się usunięcie jednego elementu z listy gap_list. Zauważmy dodatkowo, że do listy gap_list dodanych może zostać maksymalnie $n$ elementów, jako że ma to miejsce tylko przy napotkaniu początku łuku. Oznacza to, że wewnętrzna pętla **for** wykona się łącznie nie więcej niż $O(n)$ razy.

Listing 4.3. Sprawdzanie spójności grafów łuków na okręgu.

```
def check_connectivity(graph_model, endpoints):
    current_arcs = set()
    # contains ascending sorted points where a gap is supposingly ending
    gap_list = []
    gap_active = True

    for number_of_endpoint, endpoint in enumerate(endpoints):
        curr_arc = endpoint['arc']
        if endpoint['end'] == 'head':
            current_arcs.add(curr_arc)
            if gap_active:
                gap_list.append(number_of_endpoint)
                gap_active = False

            (head, tail) = graph_model[curr_arc]
            stopped = False
            # arc is crossing the joint, so it can cover a gap discovered
            # earlier
            # we use the fact, that gap_list is sorted ascending
            if tail < head:
                for index, gap in enumerate(gap_list):
                    if gap > tail:
                        stopped = True
                        break
                if not stopped:
                    gap_list = []
                else:
                    gap_list = gap_list[index:]
        else:
            current_arcs.discard(curr_arc)
            if len(current_arcs) == 0:
                gap_active = True

    # with one gap the circular-arc graph is still connected
    if len(gap_list) > 1:
        return False
    return True
```

## 4.4. Przekształcanie modelu par zdarzeń w graf abstrakcyjny

Przedstawiony tutaj algorytm jest uogólnieniem na grafy łuków na okręgu algorytmu Macieja Mularskiego, opisanego w pracy [9], dotyczącego grafów przedziałowych. Oba algorytmy służą wygenerowaniu grafu abstrakcyjnego klasy Graph na bazie danego modelu grafu. W przypadku grafu łuków na okręgu model ten to reprezentacja w postaci listy par zdarzeń.

Możliwość pojawienia się łuków wstecznych, czyli takich, dla których koniec następuje przed początkiem, w grafach łuków na okręgu stwarza konieczność dodania do algorytmu dla grafów przedziałowych sposobu postępowania z takimi łukami. Jest to realizowane za pomocą drugiej pętli **for** iterującej po wszystkich końcach łuków. Dodaje ona do grafu krawędzie, których pierwsza pętla **for**, ze względu na łuki wstecznie, nie mogła wykryć. Poza tym idea algorytmu pozostaje niezmieniona.

Złożoność algorytmu wynosi $O(n+m)$. Dwie główne pętle **for** przeglądają wszystkie końce łuków. Dodatkowo zagnieżdżone są w nich kolejne pętle **for**, których zadaniem jest dodawanie krawędzi do grafu. Ponieważ w trakcie każdego obiegu wewnętrznej pętli do grafu dodawana jest nieobecna w nim jeszcze krawędź, to łączna liczba obiegów każdej z tych pętli wynosić będzie dokładnie $m$. Warto zwrócić uwagę, że w przypadku naiwnych rozwiązań omawianego tutaj problemu otrzymalibyśmy algorytm o złożoności $O(n^2)$. Jednakże w przedstawionym tutaj algorytmie udało się uzyskać złożoność $O(n+m)$, czyli otrzymano liniową zależność od liczby wierzchołków i krawędzi.

Listing 4.4. Przekształcanie modelu par zdarzeń w graf abstrakcyjny.

```
def make_abstract_arc_graph_efficient(graph_model, size):
    abstract_graph = Graph(n=len(graph_model))
    endpoints = calculate_endpoints(graph_model, size)
    active = set()
    for endpoint in endpoints:
        if endpoint == 0:
            continue
        curr_arc = endpoint['arc']
        if endpoint['end'] == 'head':
            abstract_graph.add_node(curr_arc)
            for arc in active:
                abstract_graph.add_edge(Edge(curr_arc, arc))
            active.add(curr_arc)
        else:
            active.discard(curr_arc)
    for endpoint in endpoints:
        if endpoint == 0:
            continue
        curr_arc = endpoint['arc']
        if endpoint['end'] == 'head':
            (head, tail) = graph_model[curr_arc]
            if head < tail:
                for arc in active:
                    (head_backward, _) = graph_model[arc]
                    if tail < head_backward:
```

```
                    abstract_graph.add_edge(Edge(curr_arc, arc))
            else:
                active.discard(curr_arc)
            if len(active) == 0:
                break
        return abstract_graph
```

## 4.5. Wyznaczanie największego zbioru niezależnego wg Gavrila

Przedstawiony w tym podrozdziale algorytm opiera się na pomyśle Gavrila przedstawionym w roku 1974 [16].

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

**Problem:** Wyznaczenie największego zbioru niezależnego.

**Dane wyjściowe:** Zbiór zawierający łuki grafu należące do jego największego zbioru niezależnego.

**Opis algorytmu:** Podstawą działania algorytmu jest sprowadzenie problemu znajdywania największego zbioru niezależnego w grafie łuków na okręgu do problemu wyznaczania takiego zbioru w grafach przedziałowych.

Niech MIS to pewien największy zbiór niezależny w danym grafie. Zawsze istnieje taki fragment okręgu, że nie należy on do żadnego łuku należącego do MIS. Jeżeli usuniemy ten fragment, rozcinając w ten sposób okrąg, to dostaniemy graf przedziałowy. Jak łatwo zauważyć, największy zbiór niezależny tego grafu przedziałowego będzie też największym zbiorem niezależnym pierwotnego grafu łuków na okręgu.

To oznacza, że aby wyznaczyć największy zbiór niezależny, można przetestować wszystkie możliwości rozcięcia okręgu i znaleźć dla każdego otrzymanego w ten sposób grafu przedziałowego jego największy zbiór niezależny. Spośród wyznaczonych zbiorów, ten o największej liczebności jest szukanym zbiorem niezależnym dla danego grafu łuków na okręgu.

Funkcja get_interval_graph_K realizuje usunięcie z danego grafu łuków na okręgu określonego fragmentu i zwraca wyznaczony model powstałego grafu przedziałowego. Warto zauważyć, że skutkuje to usunięciem zarówno pewnych krawędzi jak i wierzchołków z grafu. Następnie model ten jest dostarczany jako argument do funkcji interval_maximum_iset, przedstawionej przez Macieja Mularskiego w pracy [9], oraz stanowiącej część pakietu graphtheory. Ta ostatnia funkcja znajduje największy zbiór niezależny danego grafu przedziałowego. Na samym końcu, spośród wszystkich wyznaczonych w ten sposób zbiorów niezależnych, główna funkcja MIS_gavril wybiera zbiór najliczniejszy. Jest to szukany największy zbiór niezależny danego grafu łuków.

**Złożoność:** Złożoność czasowa algorytmu wynosi $O(n^2)$. Wynika ona z wielokrotnego wyznaczania największego zbioru niezależnego dla różnych grafów przedziałowych.

Funkcja get_interval_graph_K ze względu na pętle **for** ma złożoność liniową. Jak pokazano w pracy Mularskiego [9], funkcja interval_maximum_iset ma liniową złożoność. Główna funkcja algorytmu, MIS_gavril, realizuje swoje zadanie w czasie kwadratowym, ponieważ pętla **for**, wykonująca się dokładnie $2n$ razy, wywołuje wspomniane wcześniej funkcje o liniowej złożoności.

**Uwagi:** W oryginalnej pracy Gavrila [16] jest mowa o złożoności $O(n^4)$, podczas gdy omówiona tutaj implementacja ma złożoność $O(n^2)$. Wynika to z przedstawienia w międzyczasie algorytmu, nieznanego jeszcze w 1974 roku, znajdującego największy zbiór niezależny w grafie przedziałowym w czasie liniowym pod warunkiem, że końce przedziałów są już posortowane [25].

Listing 4.5. Największy zbiór niezależny wg Gavrila.

```
def MIS_gavril(graph):
    size = len(graph)
    endpoints = calculate_endpoints(graph, size)
    mis_size = 0
    for i in range(2*size):
        K_perm = get_interval_graph_K(graph, i, size, endpoints) # O(n)
        independet_set = interval_maximum_iset(K_perm) # O(n)
        independet_set_size = len(independet_set)
        if independet_set_size > mis_size:
            mis_size = independet_set_size
            mis = independet_set
    return mis

def get_interval_graph_K(graph, point, size, endpoints):
    graph_without_fragm = [True]*size
    for i in range(size):
        (head, tail) = get_head_and_tail_arc(i, graph)
        if (head <= point < tail or
            point < tail < head or
            tail < head <= point):
            graph_without_fragm[i] = False
    perm = []
    for endpoint in range(point+1, 2*size):
        arc = endpoints[endpoint]['arc']
        if graph_without_fragm[arc]:
            perm.append(arc)
    for endpoint in range(point+1):
        arc = endpoints[endpoint]['arc']
        if graph_without_fragm[arc]:
            perm.append(arc)
    return perm
```

## 4.6. Wyznaczanie największego zbioru niezależnego wg Masudy i Nakajimy

Przedstawiony w tym podrozdziale algorytm opiera się na pomyśle Masudy i Nakajimy, przedstawionym w roku 1988 [26].

**Dane wejściowe:** Dowolny graf łuków na okręgu w reprezentacji w postaci listy par zdarzeń. Lista par powinna być w postaci kanonicznej, tzn. dla $n$ łuków będą zdarzenia od 0 do $2n-1$, lewe końce par będą posortowane rosnąco, oraz lewy koniec pierwszej pary to zdarzenie 0 (pierwsza para to łuk *forward*).

**Problem:** Wyznaczenie największego zbioru niezależnego.

**Dane wyjściowe:** Lista zawierająca łuki grafu należące do jego największego zbioru niezależnego.

**Opis algorytmu:** Algorytm dzieli łuki na tak zwane łuki *forward* oraz łuki *backward*. Łuki *backward* charakteryzują się tym, że ich koniec ma mniejszy numer niż ich początek, czyli przechodzą przez fragment okręgu między zdarzeniem $2n-1$ oraz zdarzeniem 0. Podział na te dwie, rozłączne grupy łuków dokonywany jest przez funkcję calculate_all_forward_and_backward_arcs.

Łatwo stwierdzić, że szukając największego zbioru niezależnego nie trzeba wcale rozpatrywać wszystkich łuków, a wystarczą tylko te, które nie zawierają w sobie żadnego innego łuku. Dlatego też funkcja calculate_minimal_forward_arcs spośród wszystkich łuków *forward* wybiera te, które nie zawierają w sobie żadnego łuku i to właśnie z tym zbiorem będziemy pracować w dalszej części algorytmu. Z powodów, które staną się jasne później, algorytm operuje na wszystkich łukach *backward*.

Kluczową ideą algorytmu jest to, że w celu znalezienia największego zbioru niezależnego grafu sprawdzamy, czy istnieje taki największy zbiór niezależny wśród krawędzi odpowiadającym łukom *forward* oraz taka krawędź odpowiadająca łukowi *backward*, że razem tworzą zbiór niezależny. Jest to wtedy największy zbiór niezależny dla danego grafu. W przeciwnym razie dowolny największy zbiór niezależny wśród krawędzi odpowiadającym łukom *forward* stanowi największy zbiór niezależny danego grafu.

Sam pomysł, mimo że prosty, zrealizowany bezpośrednio prowadzi do bardzo dużej złożoności obliczeniowej takiego algorytmu, jako że liczba zbiorów niezależnych może być wykładnicza w zależności od liczby łuków *forward*.

Można jednak pokazać, że wcale nie musimy rozważać wszystkich możliwych największych zbiorów niezależnych wśród krawędzi odpowiadającym łukom *forward*, a wystarczy pewna, odpowiednio dobrana rodzina takich zbiórów essential_DMIAF_set. Za znalezienie tej rodziny odpowiada funkcja find_EDS, która realizuje to zadanie w czasie liniowym.

Ostatnim krokiem jest przegląd wszystkich łuków *backward* w poszukiwaniu łuku $a$, dla którego istnieje zbiór DMIAF w essential_DMIAF_set taki, że po dodaniu $a$ do zbioru DMIAF otrzymujemy zbiór niezależny. Jeśli taki łuk i taki zbiór istnieją, to razem tworzą szukany największy zbiór niezależny

grafu. W przeciwnym razie największym zbiorem niezależnym grafu będzie dowolny zbiór należący do essential_DMIAF_set.

**Złożoność:** Złożoność czasowa algorytmu wynosi $O(n)$, jako że przedstawiona implementacja oczekuje na wejściu reprezentacji grafu, w której lewe końce łuków są już posortowane. Jeśli zmodyfikujemy algorytm tak, żeby mógł operować także na rodzinie łuków z nieposortowanymi końcami, to złożoność czasowa wzrośnie do $O(n \log n)$. Jest to najlepszy czas jaki da się osiągnąć w przypadku grafów łuków na okręgu, gdyż jak pokazano w [25], w najgorszym wypadku potrzebujemy $\Omega(n \log n)$ czasu na znalezienie największego zbioru niezależnego w przypadku grafu przedziałowego, a zatem także grafu łuków na okręgu.

Warto w tym miejscu wspomnieć, że w pozostałych algorytmach przedstawionych w tym rozdziale także często mamy do czynienia ze złożonością liniową, o ile otrzymana na wejściu reprezentacja jest już posortowana. Złożoność obliczeniowa wzrasta jednak do $O(n \log n)$, jeśli to algorytm będzie musiał przeprowadzić odpowiednie sortowanie.

Liniowa złożoność algorytmu wynika z tego, że główna funkcja FIND_MIAF wywołuje tylko funkcje o liniowej złożoności, oraz dokonuje jeden raz przeglądu łuków. Większość pozostałych funkcji także albo dokonuje przeglądu łuków albo ich końców.

Bardziej skomplikowanie przedstawia się sytuacja funkcji find_EDS. Wywołuje ona mianowicie wielokrotnie rekurencyjną funkcję find_ec_and_sizes_of_Y. Złożoność tutaj jednakże także jest liniowa. Wynika to z faktu, że funkcja find_ec_and_sizes_of_Y zostanie wywołana nie więcej niż $n$ razy, gdyż jej zadaniem jest wypełnienie, i to w dodatku nie całkowite, list ec_Y oraz sizes_Y o rozmiarze $n$, a w trakcie każdego wywołania listy są uzupełniane o jedną, nieznaną dotychczas wartość.

**Uwagi:** O ile samo algorytmiczne znalezienie rodziny essential_DMIAF_set nie jest trudne, o tyle udowodnienie, że na prawdę wystarczy rozważać tylko zbiory należące do tej rodziny należy już do wymagających zadań. Zainteresowani znajdą szczegóły w pracy [26].

Listing 4.6. Największy zbiór niezależny wg Masudy i Nakajimy.

```
def FIND_MIAF(graph):
    size = len(graph)
    if graph[0][0] != 0:
        raise ValueError("First arc should start at coordinate 0")
    graph_endpoints = [i for arc in graph for i in arc]
    if set(graph_endpoints) != set(range(0, 2*size)):
        raise ValueError("Arc endpoints need to be distinct, consecutive numbers")
    endpoints = calculate_endpoints(graph, size)
    (S_f_all_array, S_b) = calculate_all_forward_and_backward_arcs(graph, size)
    (S_f_array, S_f) = calculate_minimal_forward_arcs(
        graph, S_f_all_array, endpoints, size)
    (essential_DMIAF_set, starting_arc_to_DMIAF) = find_EDS(
        graph, endpoints, S_f, S_f_array, size)
    NEXT_backward = calculate_next_for_backward_arcs(
        endpoints, S_f_all_array, starting_arc_to_DMIAF, size)
```

```
    for backward_arc in S_b:
        (head, tail) = graph[backward_arc]
        next_DMIAF = NEXT_backward[backward_arc]
        starting_arc = essential_DMIAF_set[next_DMIAF][0]
        starting_coordinate = graph[starting_arc][0]
        ending_arc = essential_DMIAF_set[next_DMIAF][-1]
        ending_coordinate = graph[ending_arc][1]
        if head > ending_coordinate and tail < starting_coordinate:
            return essential_DMIAF_set[next_DMIAF] + [backward_arc]
    return essential_DMIAF_set[0]

def calculate_all_forward_and_backward_arcs(graph, size):
    S_f_all_array = [False]*size
    S_b = []
    for index, arc in enumerate(graph):
        if arc[0] < arc[1]:
            S_f_all_array[index] = True
        else:
            S_b.append(index)
    return (S_f_all_array, S_b)

def calculate_minimal_forward_arcs(graph, S_f_all_array, endpoints, size):
    S_f_array = [False]*size
    S_f = []
    queue = []
    for endpoint in endpoints:
        curr_arc = endpoint['arc']
        if S_f_all_array[curr_arc]:
            if endpoint['end'] == 'head':
                queue.append(curr_arc)
            else:
                if curr_arc in queue:
                    index_of_curr_arc_in_queue = queue.index(curr_arc)
                    queue = queue[ (index_of_curr_arc_in_queue+1) :]
                    S_f_array[curr_arc] = True
                    S_f.append(curr_arc)
    return (S_f_array, S_f)

def find_EDS(graph, endpoints, S_f, S_f_array, size):
    Z = find_Z(graph, S_f)
    m = len(Z)
    ec_Y = [-1]*size
    sizes_Y = [-1]*size
    NEXT_forward = calculate_next_for_forward_arcs(
        endpoints, S_f_array, size)
    for i in range(m):
        (ec_Y, sizes_Y) = find_ec_and_sizes_of_Y(
            graph, Z[i], ec_Y, sizes_Y, NEXT_forward)
    R = []
    starting_arc_to_R = [-1]*size
    size_MIAF = sizes_Y[Z[0]]
    aborted = 0
    for i in range(m-1):
        curr_arc = Z[i]
        if sizes_Y[curr_arc] != size_MIAF:
            aborted = 1
            break
```

```
            if ec_Y[curr_arc] < ec_Y[Z[i+1]] or sizes_Y[Z[i+1]] < size_MIAF:
                starting_arc_to_R[curr_arc] = len(R)
                R.append(calculate_Y(NEXT_forward, curr_arc))
        if not aborted:
            if sizes_Y[Z[m-1]] == size_MIAF:
                starting_arc_to_R[Z[m-1]] = len(R)
                R.append(calculate_Y(NEXT_forward, Z[m-1]))
        return (R, starting_arc_to_R)

    def calculate_Y(NEXT, arc):
        solution = []
        solution.append(arc)
        while NEXT[arc] != -1:
            arc = NEXT[arc]
            solution.append(arc)
        return solution

    def find_ec_and_sizes_of_Y(graph, arc, ec_Y, sizes_Y, NEXT):
        target = NEXT[arc]
        if target == -1:
            sizes_Y[arc] = 1
            ec_Y[arc] = graph[arc][1]
        else:
            if sizes_Y[target] == -1:
                find_ec_and_sizes_of_Y(graph, target, ec_Y, sizes_Y, NEXT)
            sizes_Y[arc] = sizes_Y[target] + 1
            ec_Y[arc] = ec_Y[target]
        return (ec_Y, sizes_Y)

    def find_Z(graph, S_f):
        Z = []
        t_1 = graph[S_f[0]][1]
        for arc in S_f:
            if graph[arc][0] < t_1:
                Z.append(arc)
        return Z

    def calculate_next_for_forward_arcs(endpoints, S_f_array, size):
        NEXT = [-1]*size
        P = []
        for endpoint in endpoints:
            curr_arc = endpoint['arc']
            if S_f_array[curr_arc]:
                if endpoint['end'] == 'head':
                    if P:
                        for p in P:
                            NEXT[p] = curr_arc
                        P.clear()
                else:
                    P.append(curr_arc)
        return NEXT

    def calculate_next_for_backward_arcs(endpoints, S_f_all_array,
        starting_arc_to_DMIAF, size):
        NEXT = [-1]*size
        P = []
        for endpoint in endpoints:
```

```
        curr_arc = endpoint['arc']
        end = endpoint['end']
        if ((S_f_all_array[curr_arc] == False and end == 'tail') or
            (starting_arc_to_DMIAF[curr_arc] != -1 and end == 'head')):
            if endpoint['end'] == 'head':
                if P:
                    for p in P:
                        NEXT[p] = starting_arc_to_DMIAF[curr_arc]
                    P.clear()
            else:
                P.append(curr_arc)
    return NEXT
```

## 4.7. Wyznaczanie największego zbioru niezależnego wg Hsu i Tsai

Przedstawiony w tym podrozdziale algorytm opiera się na pomyśle Wen-Lian Hsu oraz Kuo-Hui Tsai przedstawionym w 1991 roku w [24]. Rozwiązuje on problem wyznaczania największego zbioru niezależnego, podobnie jak algorytm zaprezentowany w poprzednim rozdziale 4.6. Jego zaletą jest jednak prostota oraz, po pewnych modyfikacjach, także możliwość zastosowania do dwóch innych problemów, mianowicie wyznaczania najmniejszego pokrycia klikowego oraz wyznaczania najmniejszego zbioru dominującego. Algorytmy przeznaczone do rozwiązania tychże problemów omówione zostaną w następnych dwóch podrozdziałach 4.8 oraz 4.9.

Zamieszczony w tym podrozdziale kod źródłowy algorytmu jest pełną wersją rozwiązującą wszystkie trzy wspomniane zagadnienia. Na potrzeby tego podrozdziału będą nas interesować jednak tylko funkje GREEDY, calculate_next oraz GD służące rozwiązaniu problemu największego zbioru niezależnego. W celu znalezienia największego zbioru niezależnego należy wywołać główną funkcję GREEDY podając jako drugi argument 'mis'.

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

**Problem:** Wyznaczenie największego zbioru niezależnego.

**Dane wyjściowe:** Lista zawierająca łuki grafu należące do jego największego zbioru niezależnego.

**Opis algorytmu:** Kluczowym pomysłem w tym algorytmie jest zdefiniowanie pewnego zbioru GD, znajdywanego przez funkcję GD, dla każdego łuku grafu. Jeśli znajdziemy łuk spełniający pewne warunki, zwany *dobrym* łukiem, to zbiór GD wyznaczony dla takiego *dobrego* łuku jest szukanym największym zbiorem niezależnym grafu.

Zbiór GD dla łuku $i$ jest niczym innym niż tworzonym w zachłanny sposób największym zbiorem niezależnym zawierającym łuk $i$. Jego wyznaczanie zaczynamy od dodania do niego łuku $i$. Następnie istotna jest obserwacja,

że najbardziej optymalnie jest dodać taki łuk do zbioru GD, którego koniec następuje najwcześniej, zgodnie z kierunkiem ruchu wskazówek zegara, po końcu ostatnio dodanego do zbioru GD łuku. Oczywiście kolejny łuk nie może przecinać się z poprzednim, bo tworzony zbiór ma być zbiorem niezależnym. Dodawanie łuków kończymy, gdy dodanie kolejnego oznaczałoby, że będzie się on przecinał z łukiem $i$ dodanym jako pierwszy. Ponieważ w każdym kroku dokonywaliśmy lokalnie najbardziej korzystnego wyboru dotyczącego dodania kolejnego łuku, to otrzymany w ten sposób zbiór GD jest największym zbiorem niezależnym zawierającym łuk $i$ .

Łatwo zauważyć, że w rodzinie zbiorów GD, wyznaczonych dla wszystkich łuków, musi znajdować się taki zbiór niezależny, który jest największym zbiorem niezależnym dla całego grafu. Wynika to z tego, że dla każdego łuku należącego do największego zbioru niezależnego grafu, jego zbiór GD także będzie takim największym zbiorem niezależnym grafu.

W celu efektywnego wyznaczania zbioru GD dla dowolnego łuku, funkcja calculate_next oblicza dla każdego łuku $a$ grafu taki łuk $b$, który, w przypadku dodania łuku $a$ do pewnego zbioru GD, byłby dodany jako następny do GD. Oznacza to, że łuk $b$, zwany NEXT($a$), zdefiniowany jest jako łuk nieprzecinający się z $a$, którego koniec napotkany jest jako pierwszy po końcu $a$ w czasie obiegu okręgu zgodnie z kierunkiem ruchu wskazówek zegara.

Ze względu na złożoność obliczeniową, nie chcemy wyznaczać zbiorów GD dla każdego łuku, aby wybrać spośród nich największy. Będziemy za to szukać *dobrego* łuku, czyli takiego, że jego zbiór GD jest też największym zbiorem niezależnym dla całego grafu. Można pokazać, że każdy łuk będący częścią skierowanego cyklu w grafie skierowanym $H$, w którym wierzchołki odpowiadają łukom, a krawędzie zdefiniowane są poprzez relację NEXT, jest *dobrym* łukiem.

Główna funkcja algorytmu, GREEDY, skupia się właśnie dlatego na znalezieniu w grafie skierowanym $H$ cyklu. Jest to o tyle łatwe, że każdy wierzchołek w grafie $H$ ma stopnień wyjściowy równy jeden, co gwarantuje istnienie co najmniej jednego cyklu. Aby go znaleźć, wystarczy, zaczynając od dowolnego wierzchołka, podążać za krawędziami, aż odwiedzimy pewien wierzchołek $i$ dwukrotnie. Ostatnim krokiem algorytmu jest wyznaczenie zbioru GD dla znalezionego *dobrego* wierzchłka $i$, który jest szukanym największym zbiorem niezależnym danego grafu.

**Złożoność:** Złożoność zaprezentowanego algorytmu wynosi $O(n)$.

Funkcja GD tworzy zbiór niezależny, zatem wykonywana przez nią pętla **while**, dodająca w każdym obiegu łuk do szukanego zbioru, nie może wykonać się więcej niż $n$ razy.

W przypadku funkcji calculate_next mamy dwie pętle, iterujące się po końcach wszystkich łuków, w których zagnieżdzone są pętle **while**. Warto jednak zauważyć, że pętle **while** nie wykonają się więcej niż $O(n)$ razy. Wynika to z faktu, że w trakcie ich wykonywania usuwany jest jeden łuk ze zbioru $P$, do którego trafiają łuki dla których nie znaleziono jeszcze NEXT. Co prawda, w wyjątkowych sytuacjach, łuk usunięty z $P$ może chwilę później zostać ponownie dodany do $P$, ale powoduje to natychmiastowe przerwanie pętli **while** i nie zwiększa przez to złożoności obliczeniowej.

Podsumowując, główna funkcja GREEDY woła tylko funkcje o złożoności liniowej, a pętla **while** wykonywana w ramach funkcji GREEDY, odpowiedzialna za znalezienie *dobrego* łuku, odwiedzi każdy łuk maksymalnie jeden raz.

**Uwagi:** Funkcja calculate_next ma specyficzną postać, gdyż dwukrotnie iteruje się po wszystkich końcach łuków wykonując przy tym zbliżone obliczenia. Jest to skutek struktury grafu łuków na okręgu, gdzie, inaczej niż w przypadku grafów przedziałowych, łuki mogą przechodzić przez "łączenie" okręgu. Konsekwencją tego jest, że przy jednokrotnym przeglądzie końców łuków nie jesteśmy w stanie uchwycić wszystkich zależności. Dlatego wymagany jest drugi przegląd, gdzie co prawda nie odkrywamy już nowych łuków, ale możemy przypisać NEXT tym łukom, dla których nie potrafiliśmy tego zrobić w pierwszym przeglądzie.

Listing 4.7. Największy zbiór niezależny, najmniejsze pokrycie klikowe oraz najmniejszy zbiór dominujący wg Hsu i Tsai.

```
import random

def GREEDY(graph, problem=None):
    size = len(graph)
    graph_endpoints = [i for arc in graph for i in arc]
    if set(graph_endpoints) != set(range(0, 2*size)):
        raise ValueError(
        "Arc endpoints need to be distinct, consecutive numbers")
    endpoints = calculate_endpoints(graph, size)
    (heads, tails) = calculate_heads_and_tails(size, endpoints)
    if problem in ["mds", 2]:
        max_arcs = calculate_maximal_arcs(graph, heads, endpoints)
        size_max_arcs = len(max_arcs)
        tail_sorted_arcs = [endpoints[tail]['arc'] for tail in tails]
        (result, NEXT) = calculate_next_d(graph, max_arcs,
            tail_sorted_arcs, size, size_max_arcs)
        if result == "MDS":
            return NEXT
        arc = random.choice(max_arcs)
    else:
        NEXT = calculate_next(graph, heads, tails, endpoints)
        arc = random.randint(0,size-1)
    L = [0]*size
    while L[arc] == 0:
        L[arc] = 1
        arc = NEXT[arc]
    if problem in ["mis", 0, "mds", 2]:
        return GD(NEXT, graph, arc)
    elif problem in ["mqc", 1]:
        return GD_q(NEXT, graph, arc, endpoints)
    else:
        return (GD(NEXT, graph, arc), GD_q(NEXT, graph, arc, endpoints))

def calculate_heads_and_tails(size, endpoints):
    heads = []
    tails = []
    i = 0
```

```
    for i in range(0, 2*size):
        if endpoints[i]['end'] == "head" :
            heads.append(i)
        else:
            tails.append(i)
    return (heads, tails)

def calculate_maximal_arcs(graph, heads, endpoints):
    maximal_arcs = []
    maximum_tail = -1
    join_crossed = False
    for head in heads:
        curr_arc = endpoints[head]['arc']
        curr_tail = graph[curr_arc][1]
        if join_crossed:
            if head > curr_tail and curr_tail > maximum_tail:
                maximum_tail = curr_tail
                maximal_arcs.append(curr_arc)
        else:
            if head > curr_tail:
                join_crossed = True
                maximum_tail = curr_tail
                maximal_arcs.append(curr_arc)
            else:
                if maximum_tail < curr_tail:
                    maximum_tail = curr_tail
                    maximal_arcs.append(curr_arc)
    if join_crossed:
        i = -1
        for arc in maximal_arcs:
            (head, tail) = get_head_and_tail_arc(arc, graph)
            i = i+1
            if head > tail or maximum_tail < tail:
                break
        maximal_arcs = maximal_arcs[i:]
    return maximal_arcs

def calculate_next(graph, heads, tails, endpoints):
    size = len(graph)
    NEXT = [-1]*size
    P = []
    for tail in tails:
        curr_arc = endpoints[tail]['arc']
        if P:
            ih = graph[curr_arc][0]
            it = tail
            if ih < it:
                while P:
                    p = P.pop(0)
                    pt = graph[p][1]
                    if ih < pt :
                        P.insert(0, p)
                        break
                    NEXT[p] = curr_arc
        P.append(curr_arc)
    for tail in tails:
        if not P:
```

```
                break
            curr_arc = endpoints[tail]['arc']
            ih = graph[curr_arc][0]
            it = tail
            if ih < it:
                for p in P:
                    NEXT[p] = curr_arc
                break
            while P:
                p = P.pop(0)
                pt = graph[p][1]
                if ih < pt:
                    P.insert(0, p)
                    break
                NEXT[p] = curr_arc
        return NEXT

    def calculate_next_d(graph, max_arcs, tail_sorted_arcs, size_graph,
        size_max_arcs):
        if size_max_arcs == 1:
            return ("MDS", max_arcs)
        NEXT_d = {}
        j = None
        u = tail_sorted_arcs.index(max_arcs[0])
        (uh, ut) = get_head_and_tail_index(u, tail_sorted_arcs, graph)
        for i in max_arcs:
            (ih, it) = get_head_and_tail_arc(i, graph)
            i_visited = 0
            while (tail_sorted_arcs[u] == i or
                (ih < uh < it or uh < it < ih or it < ih < uh) or
                (ih < ut < it or ut < it < ih or it < ih < ut)):
                if tail_sorted_arcs[u] == i:
                    i_visited = i_visited + 1
                    if i_visited == 2:
                        return ("MDS", [i])
                u = (u+1) % size_graph
                (uh, ut) = get_head_and_tail_index(u, tail_sorted_arcs, graph)
            j = find_next_d(j, graph, max_arcs, size_max_arcs,
                tail_sorted_arcs[u], i)
            NEXT_d[i] = max_arcs[j]
        return ("NEXT_d", NEXT_d)

    def find_next_d(j, graph, max_arcs, size_max_arcs, u_arc, i):
        ut = graph[u_arc][1]
        if j == None and u_arc in max_arcs:
            j = max_arcs.index(u_arc)
        else:
            if j == None:
                j = max_arcs.index(i)
            (jh, jt) = get_head_and_tail_index(j, max_arcs, graph)
            while not (jh < ut <= jt or ut <= jt < jh or jt < jh <ut):
                j = (j+1) % size_max_arcs
                (jh, jt) = get_head_and_tail_index(j, max_arcs, graph)
        j_next = (j+1) % size_max_arcs
        (jnh, jnt) = get_head_and_tail_index(j_next, max_arcs, graph)
        while (jnh < ut < jnt or ut < jnt < jnh or jnt < jnh < ut):
            j = j_next
```

```
        j_next = (j+1) % size_max_arcs
        (jnh, jnt) = get_head_and_tail_index(j_next, max_arcs, graph)
    return j

def get_head_and_tail_index(arc_index, arc_list, graph):
    arc = arc_list[arc_index]
    head = graph[arc][0]
    tail = graph[arc][1]
    return (head, tail)

def get_head_and_tail_arc(arc, graph):
    head = graph[arc][0]
    tail = graph[arc][1]
    return (head, tail)

def GD(NEXT, graph, start_arc):
    solution = []
    solution.append(start_arc)
    next_arc = NEXT[start_arc]
    (nh, nt) = get_head_and_tail_arc(next_arc, graph)
    (sh, st) = get_head_and_tail_arc(start_arc, graph)

    while (nh < nt < sh < st or
        st < nh < nt < sh or
        sh < st < nh < nt or
        nt < sh < st < nh):
        solution.append(next_arc)
        next_arc = NEXT[next_arc]
        (nh, nt) = get_head_and_tail_arc(next_arc, graph)
    return solution

def GD_q(NEXT, graph, start_arc, endpoints):
    gd = GD(NEXT, graph, start_arc)
    LAST_start_arc = NEXT[gd[-1]]
    if LAST_start_arc != start_arc:
        gd.append(LAST_start_arc)
    gd_tails = [graph[arc][1] for arc in gd]
    min_tail_idx = gd_tails.index(min(gd_tails))
    gd_tails_sorted = gd_tails[min_tail_idx:] + gd_tails[:min_tail_idx]
    return compute_clique_cover(gd_tails_sorted, endpoints)

def compute_clique_cover(tails_sorted, endpoints):
    clique_cover = []
    tails_sorted_copy = tails_sorted.copy()
    A = set()
    for number_of_endpoint, endpoint in enumerate(endpoints):
        curr_arc = endpoint['arc']
        if endpoint['end'] == 'head':
            A.add(curr_arc)
        else:
            if tails_sorted and number_of_endpoint == tails_sorted[0]:
                clique_cover.append(A.copy())
                tails_sorted.pop(0)
            A.discard(curr_arc)
    index = 0
    for number_of_endpoint, endpoint in enumerate(endpoints):
        if not A or not tails_sorted_copy :
```

```
            break
        curr_arc = endpoint['arc']
        if endpoint['end'] == 'tail':
            if number_of_endpoint == tails_sorted_copy[0]:
                clique_cover[index].update(A)
                index = index+1
                tails_sorted_copy.pop(0)
            A.discard(curr_arc)
    return clique_cover
```

## 4.8. Wyznaczanie najmniejszego pokrycia klikowego wg Hsu i Tsai

Przedstawiony w tym podrozdziale algorytm opiera się na pomyśle Wen-Lian Hsu oraz Kuo-Hui Tsai przedstawionym w [24]. Rozwiązuje on problem wyznaczania najmniejszego pokrycia klikowego.

Kod źródłowy, rozwiązujący zarówno ten jak i dwa inne problemy, można znaleźć w poprzednim podrozdziale 4.7. Tutaj będą interesować nas funkcje GREEDY, calculate_next, GD_q oraz compute_clique_cover służące rozwiązaniu problemu wyznaczania najmniejszego pokrycia klikowego. W celu znalezienia najmniejszego pokrycia klikowego należy wywołać główną funkcję GREEDY podając jako drugi argument 'mqc'.

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

**Problem:** Wyznaczenie najmniejszego pokrycia klikowego.

**Dane wyjściowe:** Lista zawierająca zbiory łuków grafu tworzące jego najmniejsze pokrycie klikowe.

**Opis algorytmu:** Ogólna struktura algorytmu jest bardzo podobna do algorytmu, znajdującego największy zbiór niezależny, omówionego w poprzednim rozdziale 4.7. Oba algorytmy opierają swoje działanie na zachłannym podejściu, a kluczowe w znalezieniu rozwiązania jest znalezienie cyklu skierowanego w grafie $H$ zdefiniowanym w 4.7.

Tak jak poprzednio, w głównej funkcji algorytmu GREEDY poszukujemy *dobrego* łuku, dla którego wyznaczymy za pomocą funkcji compute_clique_cover pewną rodzinę klik $GD_q$, będącą w rzeczywistości najmniejszym pokryciem klikowym danego grafu. Funkcja GD_q ma tutaj tylko pomocnicze znaczenie, przekształcając wejście na odpowiedni format dla wykonującej kluczowe obliczenia funkcji compute_clique_cover.

Algorytm, po znalezieniu *dobrego* łuku, wyznacza największy zbiór niezależny MIS metodą opisaną w poprzednim podrozdziale 4.7. Przy okazji dla każdego łuku wyznaczany jest także łuk NEXT zdefiniowany w 4.7. Warto zauważyć, że każdy łuk należacy do największego zbióru niezależnego musi znaleźć się w innej klice.

W następnym kroku funkcja compute_clique_cover przystępuje do znalezienia dla każdego łuku $i$, należącego do MIS, kliki zawierającej łuk $i$ oraz wszystkie takie łuki, które zawierają koniec łuku $i$. Ponadto, jeśli NEXT dla ostatniego dodanego do MIS łuku $b$, nazwijmy go LAST, nie jest tożsamy z pierwszym dodanym do MIS łukiem $a$, wyznaczana jest jeszcze dodatkowo taka klika dla łuku LAST, mimo, że LAST nie jest częścią MIS.

Rodzina wyznaczonych w ten sposób klik tworzy minimalne pokrycie klikowe grafu. Wynika to wprost z definicji NEXT używanego do wyznaczenia MIS. Gwarantuje ona, że każdy łuk $a$ nienależący do MIS, którego koniec leży pomiędzy końcami dwóch łuków dodanych bezpośrednio po sobie do MIS, będzie zawierał koniec jednego z tych łuków. Gdyby tak nie było to, zgodnie z definicją, sam $a$ musiałby należeć do MIS. Wyjątkiem jest jedynie łuk LAST, gdyż może on nie należeć do MIS i jednocześnie nie zawierać końca żadnego łuku z MIS. Komplikacja ta wynika z natury grafów łuków na okręgu, gdyż taki łuk LAST przecina się z pierwszym dodanym do MIS łukiem, dlatego w takim przypadku konieczne jest obliczenie odpowiedniej kliki także dla LAST.

**Złożoność:** Złożoność zaprezentowanego algorytmu wynosi $O(n)$ jeśli ograniczymy się do znalezienia rozmiaru minimalnego pokrycia klikowego. Wynika to z faktu, że aby poznać rozmiar musimy jedynie wyznaczyć, w czasie liniowym, największy zbiór niezależny oraz przeprowadzić kilka sprawdzeń w czasie liniowym.

W momencie w którym interesuje nas także konkretna postać minimalnego pokrycia klikowego, zmuszeni jesteśmy do jawnego wyznaczenia każdej kliki wchodzącej w jej skład. Spowoduje to wzrost złożoności, gdyż jeden łuk może być elementem wielu klik. W najgorszym przypadku będziemy musieli dodawać prawie każdy łuk do każdej kliki. Dochodzi zatem czynnik odpowiadający sumie rozmiarów wszystkich klik.

**Uwagi:** Warto zauważyć, że rozwiązanie tego problemu wykorzystuje jako swój punkt wyjścia rozwiązanie problemu znajdywania największego zbioru niezależnego opisanego w poprzednim podrozdziale 4.7. Prezentowany tu algorytm jest zatem tylko rozbudowaniem poprzedniego o ściśle zdefiniowany krok dotyczący znajdowania rodziny klik tworzących minimalne pokrycie klikowe na bazie wyznaczonego największego zbioru niezależnego.

## 4.9. Wyznaczanie najmniejszego zbioru dominującego wg Hsu i Tsai

Przedstawiony w tym podrozdziale algorytm jest trzecim i ostatnim algorytmem przedstawionym w [24]. Rozwiązuje on problem wyznaczania najmniejszego zbioru dominującego w czasie liniowym.

Kod źródłowy, rozwiązujący zarówno ten jak i dwa problemy omówione wcześniej, można znaleźć w podrozdziale 4.7. Tutaj interesować będą nas funkcje GREEDY, calculate_maximal_arcs, calculate_next_d oraz GD służące rozwiązaniu problemu wyznaczania najmniejszego zbioru dominującego. W celu

znalezienia najmniejszego zbioru dominującego należy wywołać główną funkcję GREEDY podając jako drugi argument 'mds'.

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

**Problem:** Wyznaczenie najmniejszego zbioru dominującego.

**Dane wyjściowe:** Lista zawierająca łuki grafu tworzące jego najmniejszy zbiór dominujący.

**Opis algorytmu:** Ogólna struktura algorytmu pozostaje bardzo podobna do tej wykorzystanej w algorytmie znajdującym największy zbiór niezależny omówionym w 4.7. Konieczne są jednak pewne modyfikacje, jak na przykład zamienienie funkcji calculate_next na calculate_next_d, gdyż definicja $\mathrm{NEXT_d}$ znacząco odbiega od definicji NEXT, chociaż sama idea działania pozostaje analogiczna.

Podobnie jak wcześniej, główna funkcja algorytmu GREEDY koncentruje się na znalezieniu *dobrego* łuku. Tym razem jednak nie będziemy rozważać wszystkich łuków. Z naszych poszukiwań możemy wykluczyć wszystkie takie łuki, które są zawarte w innych łukach. Zamiast łuku $a$, który wykluczyliśmy, zawsze możemy w zamian wykorzystać łuk, w którym zawarty jest wykluczony łuk $a$.

Funkcja calculate_maximal_arcs służy znalezieniu podzbioru łuków maksymalnych, to znaczy takich, które nie są zawarte w żadnym innym łuku. Możliwość istnienia łuków wstecznych, tzn. takich, które przechodzą przez "łączenie" okręgu, prowadzi do widocznego zwiększenia stopnia skomplikowania funkcji w stosunku do sytuacji z jaką mielibyśmy do czynienia w grafie przedziałowym.

Następnym krokiem jest policzenie przez funkcję calculate_next_d, dla każdego łuku maksymalnego, ideowego odpowiednika łuku NEXT używanego w trakcie poszukiwań największego zbioru niezależnego przez algorytm w 4.7. Ten krok jest bardzo złożony i stanowi znaczne zwiększenie stopnia trudności tego algorytmu.

Analogicznie jak w przypadku algorytmu znajdującego największy zbiór niezależny, dla każdego maksymalnego łuku $i$ definiujemy zbiór $\mathrm{GD_d}$. $\mathrm{GD_d}(i)$ to tworzony w zachłanny sposób najmniejszy zbiór dominujący, którego pierwszym elementem jest łuk $i$, kolejny element to $\mathrm{NEXT_d}(i)$, następny element to $\mathrm{NEXT_d}(\mathrm{NEXT_d}(i))$, itd., aż dojdziemy do łuku przecinającego się z $i$.

Szukanym najmniejszym zbiorem dominującym dla całego grafu jest $\mathrm{GD_d}(i)$ wyznaczony dla *dobrego* łuku $i$.

**Złożoność:** Złożoność zaprezentowanego algorytmu wynosi $O(n)$.

Funkcje GREEDY oraz GD, występujące w tej samej postaci w algorytmie 4.7 znajdującym największy zbiór dominujący, mają złożoność liniową.

Dwie pętle **for** iterujące się po łukach skutkują liniową złożonością funkcji calculate_maximal_arcs.

Ocenienie złożoności funkcji calculate_next_d wymaga przyjrzenia się zachowaniu wskaźników i, j oraz u wykorzystywanych przez tę funkcję. Mianowicie nie cofają one się nigdy, a tylko przesuwają do przodu po okręgu, co skutkuje złożonością $O(n)$.

## 4.10. Wyznaczanie największej kliki

Przedstawiony w tym podrozdziale algorytm, służący do wyznaczania największej kliki grafu, opiera się na pomyśle Gavrila przedstawionym w roku 1974 [16].

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

**Problem:** Wyznaczenie największej kliki.

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

**Opis algorytmu:** Algorytm w swojej głównej funkcji max_clique_gavril przegląda za pomocą pętli **for** wszystkie łuki grafu i wyznacza dla nich, za pomocą funkcji get_X_Y, dwa zbiory wierzchołków. Do zbioru $X_i$ trafiają wszystkie łuki, które zawierają początek łuku $i$. Analogicznie, do zbioru $Y_i$ trafiają wszystkie łuki, które zawierają koniec łuku $i$, o ile nie zostały dodane do zbioru $X_i$. Sam łuk $i$ zostaje zaliczony do zbioru $X_i$. Zbiory $X_i$ oraz $Y_i$ indukują kliki.

Następnym krokiem jest znalezienie największej kliki w podgrafie danego grafu indukowanym wierzchołkowo o zbiorze wierzchołków $X_i \cup Y_i$. Łatwo zauważyć, że problem wyznaczenia największej kliki odpowiada problemowi wyznaczenia największego zbioru niezależnego w dopełnieniu grafu, a będzie to graf dwudzielny. Po obliczeniu dopełnienia podgrafu wyznaczny jest zatem największy zbiór niezależny w celu otrzymania zbioru wierzchołków należących do największej kliki podgrafu.

Jedna ze znalezionych w ten sposób klik jest szukaną największą kliką danego grafu. Wynika to z faktu, że w skład największej kliki $K$ grafu musi wchodzić co najmniej jeden łuk $a$ taki, że nie jest zawierany przez żaden inny łuk grafu. Oznacza to, że wszystkie inne łuki wchodzące w skład $K$ muszą zawierać co najmniej jeden koniec łuku $a$. Zatem istnieje łuk $a$ taki, że $K \subset X_a \cup Y_a$. Ponieważ algorytm przegląda wszystkie łuki oraz dla każdego z nich wyznacza największą klikę w podgrafie indukowanym wierzchołkowo o wierzchołkach $X_i \cup Y_i$, to największa spośród tych klik jest zarazem szukaną największą kliką grafu.

**Złożoność:** Teoretyczna złożoność czasowa algorytmu podana przez Gavrila wynosi $O(n^3)$. Oznacza to, że odpowiednie zbiory niezależne w algorytmie powinny być wyznaczone w czasie $O(n^2)$. Gavril powołuje się na pracę Deslera i Hakimi roku 1970, która niestety nie jest dostępna [27].

Złożoność podana przez Gavrila była przedmiotem dyskusji w literaturze. W pracy z roku 1982 [25] autorzy sugerują, że znalezienie największego zbioru niezależnego w grafie dwudzielnym, krok niezbędny w analizowanym tutaj algorytmie, zajmuje czas $O(n^{2.5})$, ponieważ użyty tam algorytm Deslera i Hakimi musi najpierw znaleźć największe skojarzenie, a można to zrobić najszybciej właśnie w tym czasie za pomocą algorytmu Hopcrofta-Karpa. To oznacza, że cały algorytm Gavrila będzie zajmował czas $O(n^{3.5})$.

W roku 1987 Apostolico i Hambrusch [28] przedstawili algorytm znajdujący największą klikę w grafie łuków na okręgu w czasie $O(n^2 \log\log n)$.

W naszej implementacji funkcja get_X_Y zawiera pętlę **for** iterującą po wszystkich łukach grafu, co skutkuje liniową złożonością tej funkcji.

W przypadku głównej funkcji algorytmu, max_clique_gavril, także mamy do czynienia z pętlą **for** przeglądającą wszystkie łuki grafu. Dodatkowo w każdym jej obiegu wołana jest funkcja get_X_Y, wykonywane są operacje takie jak przekształcenie modelu par zdarzeń w graf abstrakcyjny, czy też obliczenie dopełnienia tego grafu w czasie kwadratowym, oraz wyznaczenie największego zbioru niezależnego w utworzonym grafie abstrakcyjnym dwudzielnym.

W naszej implementacji największy zbiór niezależny jest wyznaczany przez ogólny algorytm z powrotami (klasa BacktrackingIndependentSet). Niestety jego złożoność jest wykładnicza i to psuje złożoność algorytmu Gavrila. W przyszłości będzie można łatwo zamienić ten fragment kodu na bardziej wydajny algorytm, wykorzystujący dwudzielność odpowiedniego grafu.

Listing 4.8. Największa klika grafu wg Gavrila.

```
def max_clique_gavril(graph):
    size = len(graph)
    max_clique = []
    for i in range(size):
        X_i, Y_i = get_X_Y(graph, size, i)
        M_i_arcs = X_i + Y_i
        M_i_model = [graph[i] for i in M_i_arcs]
        M_i = make_abstract_arc_graph_efficient(M_i_model, size)
        M_i_complement = M_i.complement()
        BIS = BacktrackingIndependentSet(M_i_complement)
        BIS.run()
        clique_i = BIS.independent_set
        clique_i_old_numbering = [M_i_arcs[i] for i in clique_i]
        if len(clique_i_old_numbering) > len(max_clique):
            max_clique = clique_i_old_numbering
    return max_clique

def get_X_Y(graph, size, index):
    X = []
    Y = []
    head_curr, tail_curr = graph[index]
    for i in range(size):
        head, tail = graph[i]
        if (head < head_curr < tail or
            head_curr < tail < head or
            tail < head < head_curr):
            X.append(i)
        elif (head < tail_curr < tail or
            tail_curr < tail < head or
```

```
            tail < head < tail_curr):
            Y.append(i)
    X.append(index)
    return (X, Y)
```

# 5. Podsumowanie

W ramach niniejszej pracy przyjrzano się bliżej zagadnieniu grafów łuków na okręgu. Zaczęto od zarysowania tematyki dotyczącej tej szczególnej klasy grafów oraz opisano możliwe praktyczne zastosowania. Następnie w rozdziale 2 stworzono podstawy teoretyczne konieczne do bliżej analizy zagadnień związanych z teorią grafów. W dalszej części przedstawiono sposób praktycznego wykorzystania przygotowanych w ramach tej pracy implementacji algorytmów. W rozdziale 4 zamieszczono opisy zaimplementowanych algorytmów. W ramach opisu każdej implementacji zwrócono uwagę na szczegóły implementacji jak i fragmenty kodu konieczne do prawidłowego funkcjonowania algorytmu, ale nieobecne w pseudokodzie. Następnie omówiono teoretyczną złożoność algorytmu i przedstawiono interesujące fragmenty przygotowanego kodu źródłowego. Poza algorytmami rozwiązującymi problemy teorii grafów, takimi jak wyznaczanie najmniejszego pokrycia klikowego, stworzono także implementacje pomagające w pracy z grafami łuków na okręgu. Za przykład niech służy algorytm pozwalający na stworzenie graficznej reprezentacji dowolnego grafu łuków na okręgu.

Wszystkie implementacje powstałe w ramach tej pracy zostały sprawdzone pod kątem poprawności. Wykorzystano w tym celu moduł unittest przy pomocy którego stworzono testy jednostkowe. Do eksperymentalnego przetestowania złożoności obliczeniowej stworzonych implementacji wykorzystano moduł timeit. Nie stwierdzono przy tym większych rozbieżności między teoretyczną złożonością algorytmów, a uzyskaną w testach złożonością implementacji. Dokładniejsze omówienie otrzymanych wyników znajduje się w dodatku A.

W ramach implementowania algorytmu wyznaczania największego zbioru niezależnego na postawie pracy Gavrila [16] odkryto, że możliwe jest osiągnięcie złożoności obliczeniowej $O(n^2)$. Jest to ciekawy wynik, ponieważ w pracy Gavrila stwierdzono, że złożoność wynosi $O(n^4)$. Otrzymane przyspieszenie wynika z zamiany algorytmu wyznaczającego największy zbiór niezależny w grafie przedziałowym użytego przez Gavrila na algorytm o złożoności liniowej.

Powstałe implementacje ze względu na funkcjonalny kod źródłowy w Pythonie mogą być wykorzystane do rozwiązywania praktycznych problemów dotyczących grafów łuków na okręgu. Implementacje te uzupełnią bibliotekę graphtheory skupiającą się na różnorakich problemach teorii grafów i udostępniającą implementacje algorytmów pozwalające na wykonanie odpowiednich obliczeń w ramach danych problemów grafowych.

# A. Testy algorytmów

W rozdziale 4 zaprezentowano implementacje wybranych algorytmów dotyczących grafów łuków na okręgu. Przetestowano poprawność oraz złożoność obliczeniową każdej z nich.

Poprawność została sprawdzona za pomocą testów jednostkowych korzystających z modułu unittest [29]. Złożoność obliczeniową sprawdzono eksperymentalnie z wykorzystaniem modułu timeit [30] przy użyciu procesora Intel Core i5. Zamieszczone poniżej rysunki stanowią opracowanie wyników tych testów. W każdym przypadku sporządzono wykres zależności czasu wykonania algorytmu od liczby wierzchołków grafu, przy czym dla obu tych wartości obliczono najpierw wartość logarytmu dziesiętnego. Dzięki temu wyznaczony współczynnik kierunkowy prostej dopasowanej do punktów pomiarowych pozwala stwierdzić z jaką złożonością obliczeniową mamy w danym przypadku do czynienia. Współczynnik kierunkowy bliski liczbie całkowitej $a$ wskazuje na złożoność $O(n^a)$, gdzie $n$ to liczba wierzchołków grafu.

Pomiary wykonywano dla spójnych, ale poza tym losowych grafów łuków na okręgu. Wyjątek stanowi pomiar czasu wykonania funkcji check_connectivity sprawdzającej czy dany graf jest spójny, gdzie wykorzystane zostały losowe grafy bez względu na to czy były spójne, czy też nie.

## A.1. Testy sprawdzania spójności

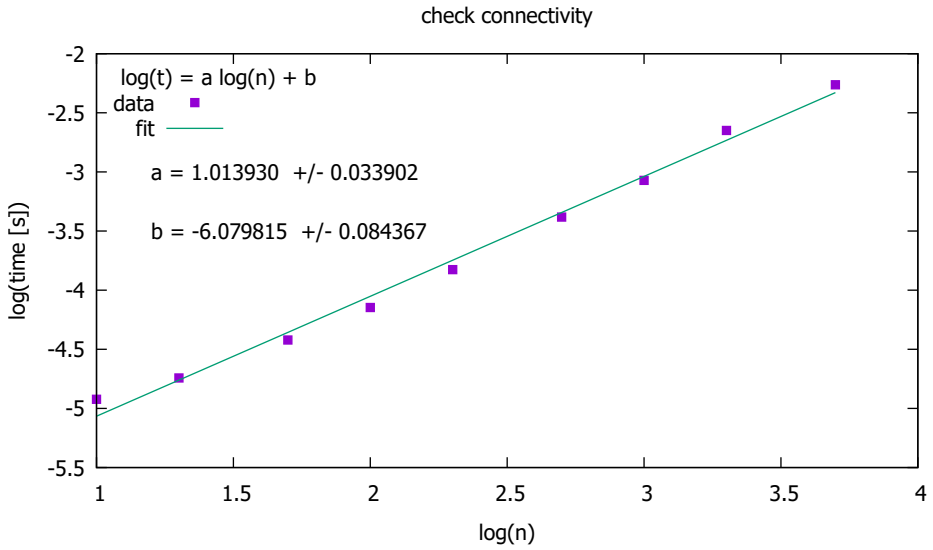
Teoretyczna złożość sprawdzania czy dany graf jest spójny za pomocą funkcji check_connectivity, opisanej w podrozdziale 4.3, wynosi $O(n)$. Potwierdzają to wyniki testów przedstawione na rysunku A.1. Wyznaczony współczynnik kierunkowy prostej wynosi $a = 1.014(34)$.

## A.2. Testy wyznaczania największego zbioru niezależnego

W pracy przedstawiono trzy różne algorytmy służące rozwiązaniu problemu wyznaczania największego zbioru niezależnego.

Teoretyczna złożność najprostszego ideowo spośród nich, MIS_gavril, opisanego w rozdziale 4.5, wynosi $O(n^2)$. Potwierdzają to przeprowadzone testy, których wyniki widoczne są na rysunku A.2. Wyznaczony współczynnik kierunkowy wynosi $a = 1.962(28)$.

W przypadku pozostałych dwóch algorytmów, FIND_MIAF oraz GREEDY, opisanych odpowiednio w rozdziałach 4.6 oraz 4.7, teoretyczna złożoność obliczeniowa wynosi $O(n)$. Na rysunku A.3 przedstawione są wyniki testów dla




Rysunek A.1. Wyniki testów wydajności algorytmu check_connectivity (4.3) sprawdzającego czy graf jest spójny. Współczynnik $a = 1.014(34)$ wskazuje na liniową złożoność obliczeniową $O(n)$.

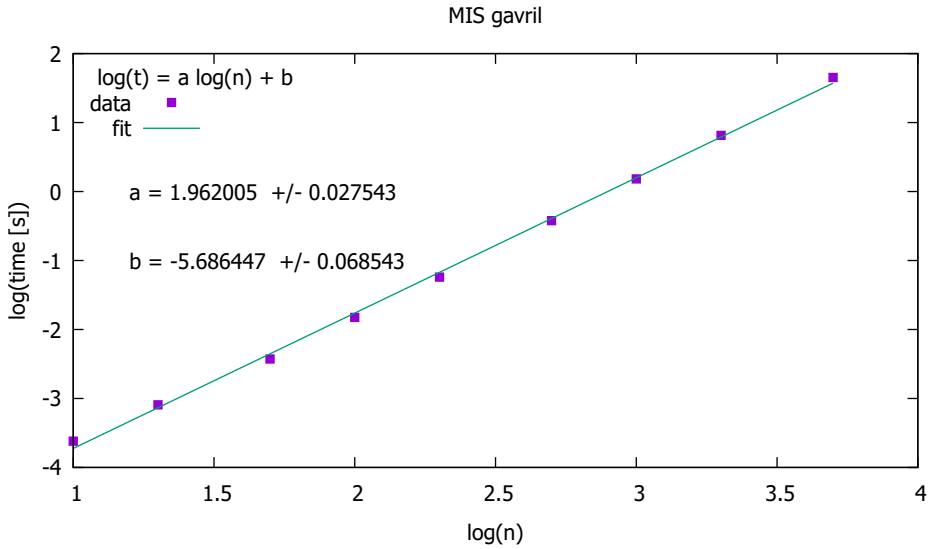
FIND_MIAF, a na rysunku A.4 dla GREEDY. W pierwszym przypadku otrzymany współczynnik kierunkowy prostej wynosi $a = 0.953(47)$, a w drugim $a = 0.969(39)$. Wyniki potwierdzają zatem liniową złożoność obliczeniową obu algorytmów.

## A.3. Testy wyznaczania najmniejszego pokrycia klikowego

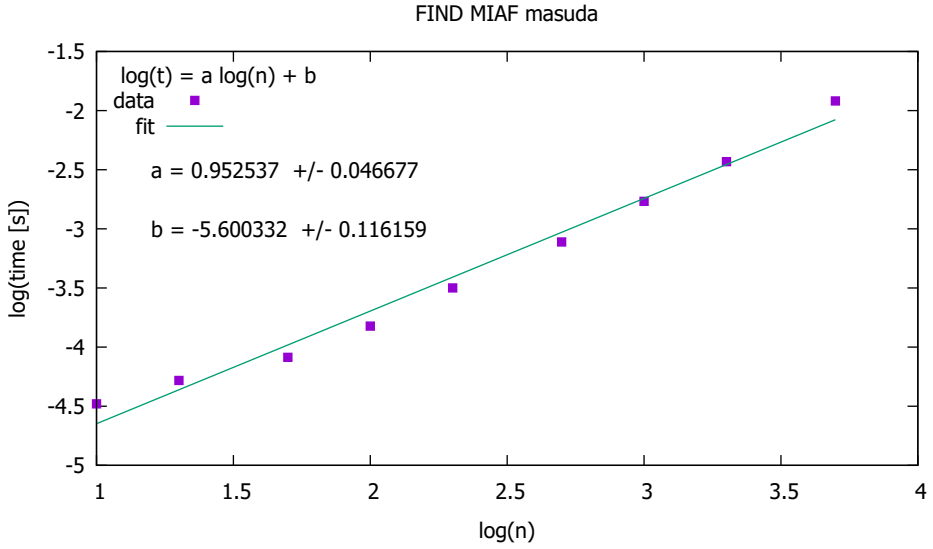
Teoretyczna złożość wyznaczania najmniejszego pokrycia klikowego za pomocą funkcji GREEDY, opisanej w podrodziale 4.8, wynosi $O(n)$. Potwierdzają to wyniki testów przedstawione na rysunku A.5. Wyznaczony współczynnik kierunkowy prostej wynosi $a = 0.981(44)$.

## A.4. Testy wyznaczania najmniejszego zbioru dominującego

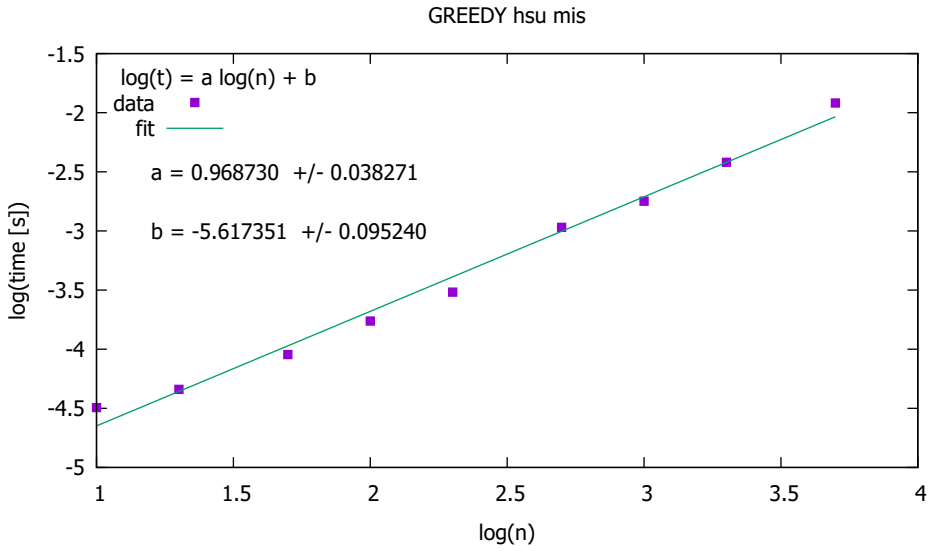
Teoretyczna złożość wyznaczania najmniejszego zbioru dominującego za pomocą funkcji GREEDY, opisanej w podrozdziale 4.9, wynosi $O(n)$. Potwierdzają to wyniki testów przedstawione na rysunku A.6. Wyznaczony współczynnik kierunkowy prostej wynosi $a = 0.968(29)$.



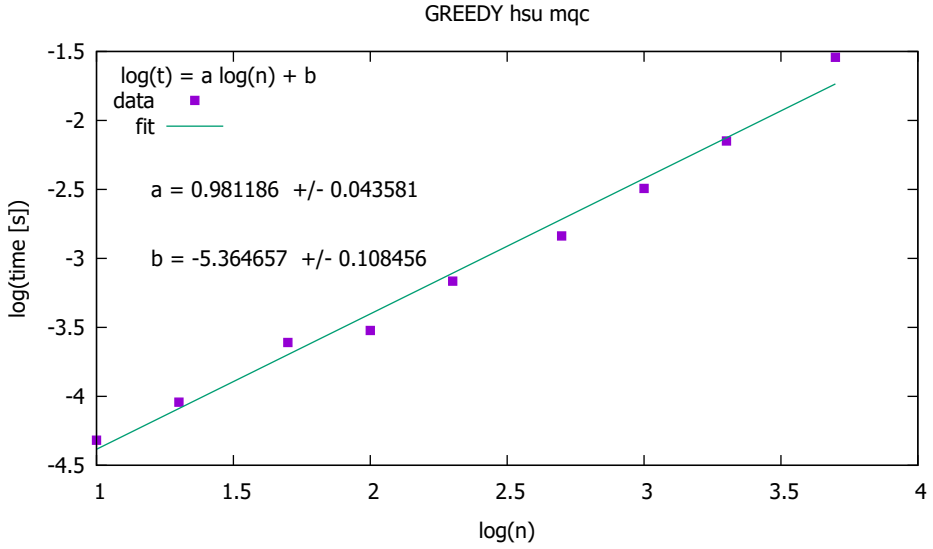

Rysunek A.2. Wyniki testów wydajności algorytmu MIS_gavril (4.5) wyznaczającego największy zbiór niezależny. Współczynnik $a = 1.962(28)$ wskazuje na kwadratową złożoność obliczeniową $O(n^2)$.



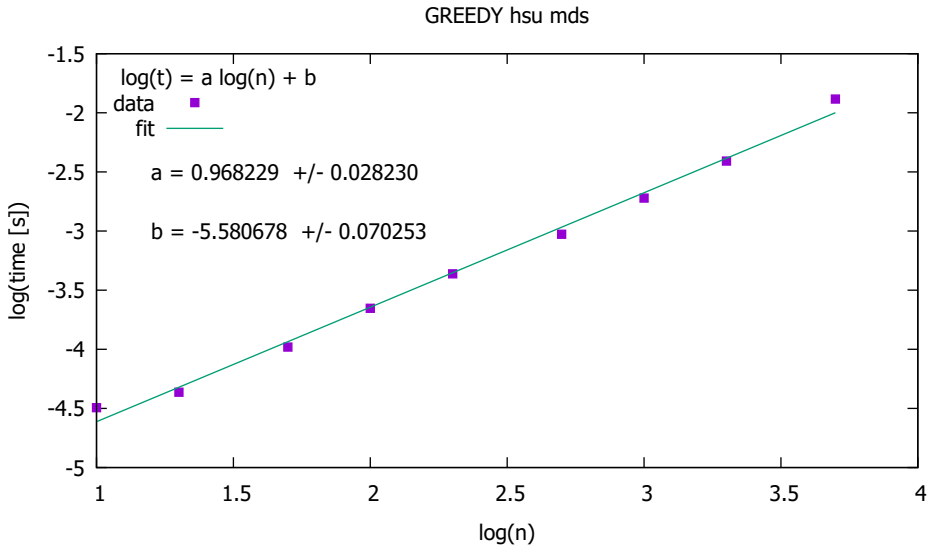

Rysunek A.3. Wyniki testów wydajności algorytmu FIND_MIAF (4.6) wyznaczającego największy zbiór niezależny. Współczynnik $a = 0.953(47)$ wskazuje na liniową złożoność obliczeniową $O(n)$.




Rysunek A.4. Wyniki testów wydajności algorytmu GREEDY (4.7) wyznaczającego największy zbiór niezależny. Współczynnik $a = 0.969(39)$ wskazuje na liniową złożoność obliczeniową $O(n)$.



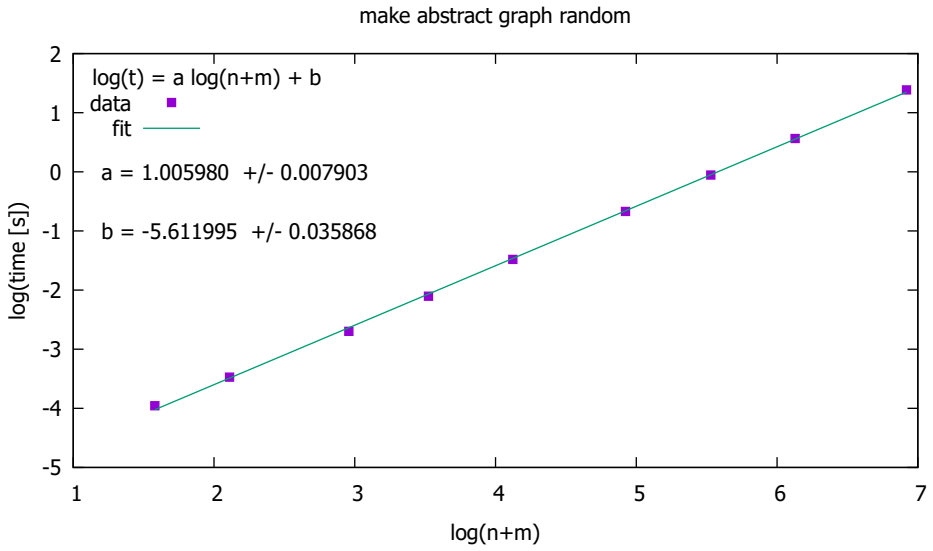

Rysunek A.5. Wyniki testów wydajności algorytmu GREEDY (4.8) wyznaczającego najmniejsze pokrycie klikowe. Współczynnik $a = 0.981(44)$ wskazuje na liniową złożoność obliczeniową $O(n)$.



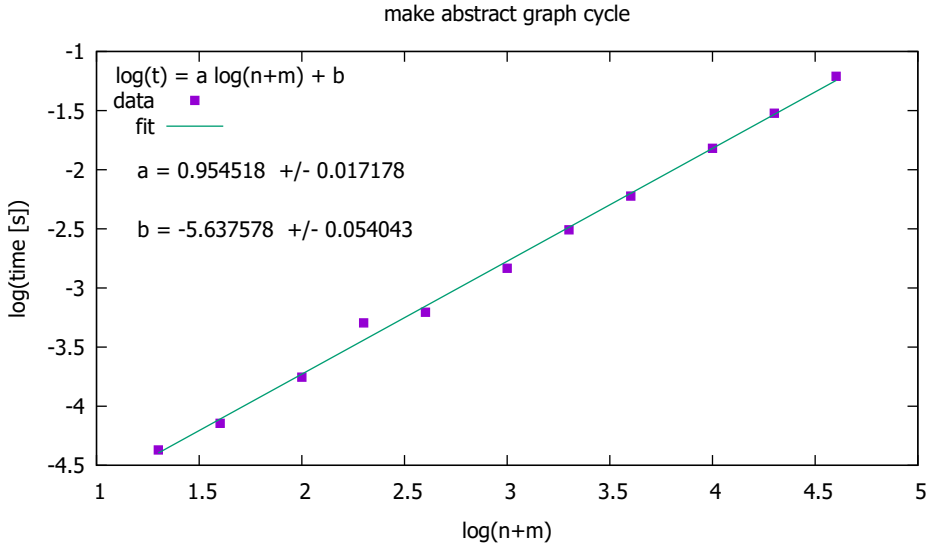

Rysunek A.6. Wyniki testów wydajności algorytmu GREEDY (4.9) wyznaczającego najmniejszy zbiór dominujący. Współczynnik $a = 0.968(29)$ wskazuje na liniową złożoność obliczeniową $O(n)$.

## A.5. Testy przekształcania modelu par zdarzeń w graf abstrakcyjny

Teoretyczna złożość przekształcania modelu par zdarzeń w graf abstrakcyjny za pomocą funkcji make_abstract_arc_graph_efficient, opisanej w podrozdziale 4.4, wynosi $O(n + m)$. Potwierdzają to wyniki testów przedstawione na rysunkach A.7 oraz A.8. Wyznaczony współczynnik kierunkowy prostej w przypadku pomiarów dokonanych dla spójnych grafów losowych wynosi $a = 1.006(8)$, a w przypadku pomiarów dokonanych dla grafów cyklicznych $a = 0.955(17)$.




Rysunek A.7. Wyniki testów wydajności algorytmu przekształcającego model par zdarzeń w graf abstrakcyjny (4.4) dla spójnych grafów losowych. Współczynnik $a = 1.006(8)$ wskazuje na liniową złożoność obliczeniową $O(n + m)$.




Rysunek A.8. Wyniki testów wydajności algorytmu przekształcającego model par zdarzeń w graf abstrakcyjny (4.4) dla grafów cyklicznych. Współczynnik $a = 0.955(17)$ wskazuje na liniową złożoność obliczeniową $O(n + m)$.

# Bibliografia


[1] Wikipedia, Circular-arc graph, 2025, `https://en.wikipedia.org/wiki/Circular-arc_graph`.

[2] Wikipedia, Interval graph, 2025, `https://en.wikipedia.org/wiki/Interval_graph`.

[3] Min Chih Lin and Jayme L. Szwarcfiter, *Characterizations and recognition of circular-arc graphs and subclasses: A survey*, Discrete Mathematics 309, 5618-5635 (2009).

[4] Guillermo Durán, Luciano N. Grippo, Martín D. Safe, *Structural results on circular-arc graphs and circle graphs: A survey and the main open problems*, Discrete Applied Mathematics 164, 427-443 (2014).

[5] Wikipedia, Perfect graph, 2025, `https://en.wikipedia.org/wiki/Perfect_graph`.

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

[7] Jeremy P. Spinrad, *Efficient Graph Representations*, A co-publication of the AMS and Fields Institute, 2003.

[8] Andrzej Kapanowski, graphtheory, GitHub repository, 2025, `https://github.com/ufkapano/graphtheory/`.

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

[10] Karl E. Stoffers, *Scheduling of traffic lights - a new approach*, Transportation Research 2, 199-234 (1968).

[11] A. Tucker, *Circular-arc graphs: New uses and a new algorithm*, Theory and Application of Graphs, Lecture Notes in Mathematics 642, 580-589 (1978).

[12] A. Tucker, *Coloring a family of circular-arc graphs*, SIAM J. on Appl. Math. 29, 493-502 (1975).

[13] Wikipedia, Intersection graph, 2025, `https://en.wikipedia.org/wiki/Intersection_graph`.

[14] Hugo Hadwiger, Hans Debrunner, and Victor Klee, *Combinatorial Geometry in the Plane*, p. 54, Holt, Rinehart and Winston, New York 1964.

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

[16] F. Gavril, *Algorithms on Circular-Arc Graphs*, Networks 4, 357–369 (1974).

[17] A. Tucker, *An efficient test for circular-arc graphs*, SIAM J. on Comput. 9, 1-24 (1980).

[18] Elaine M. Eschen and Jeremy P. Spinrad, *An $O(n^2)$ Algorithm for Circular-Arc Graph Recognition*, Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, 128-137 (1993).

[19] Wen-Lian Hsu, *$O(mn)$ Algorithms for the Recognition and Isomorphism Problems on Circular-Arc Graphs*, SIAM J. on Comput. 24, 411-439 (1995).

[20] Andrew R. Curtis, Min Chih Lin, Ross M. McConnell, Yahav Nussbaum, Francisco J. Soulignac, Jeremy P. Spinrad, and Jayme L. Szwarcfiter, *Isomorphism*

*of graph classes related to the circular-ones property*, Discrete Mathematics and Theoretical Computer Science 15(1), 157-182(2013).

[21] Tomasz Krawczyk, *Comments on "O(mn) Algorithms for the Recognition and Isomorphism Problems on Circular-Arc Graphs"*, arXiv:2411.13708 (2024).

[22] R. M. McConnell, *Linear-time recognition of circular-arc graphs*, Algorithmica 37(2), 93-147 (2003).

[23] Haim Kaplan and Yahav Nussbaum, *A Simpler Linear-Time Recognition of Circular-Arc Graphs*, Algorithmica 61, 694–737 (2011).

[24] Wen-Lian Hsu and Kuo-Hui Tsai, *Linear time algorithms on circular-arc graphs*, Information Processing Letters 40, 123–129 (1991).

[25] U. I. Gupta, D. T. Lee, and J. Y.-T. Leung, *Efficient Algorithms for Interval Graphs and Circular-Arc Graphs*, Networks 12, 459-467 (1982).

[26] Sumio Masuda and Kazuo Nakajima, *An optimal algorithm for finding a maximum independent set of a circular-arc graph*, SIAM J. Comput. 17, 41–52 (1988).

[27] J. F. Desler and S. L. Hakimi, *On finding a maximum internally stable set of a graph*, Proc. of Fourth Annual Princeton Conference on Information Sciences and Systems, Princeton, New Jersey, pp. 459-462, 1970.

[28] Alberto Apostolico, Susanne E. Hambrusch, *Finding maximum cliques on circular-arc graphs*, Information Processing Letters 26(4), 209-215 (1987).

[29] Python Programming Language - official documentation for unittest module, `https://docs.python.org/3/library/unittest.html`.

[30] Python Programming Language - official documentation for timeit module, `https://docs.python.org/3/library/timeit.html`.

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

[32] GeoGebra mathematics software, `https://www.geogebra.org`.

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

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