Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Angelika Siwek

Nr albumu: 1164080

Badanie grafów bez trójek asteroidalnych z językiem Python

Praca magisterska na kierunku Informatyka stosowana

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

Kraków 2024

Oświadczenie autora pracy

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

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

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

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

Kraków, dnia Podpis kierującego pracą

Serdecznie dziękuję Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za poświęcony czas, merytoryczne wsparcie, cenne wskazówki, ogromne zaangażowanie i nieocenioną pomoc przy tworzeniu niniejszej pracy magisterskiej.

Streszczenie

W pracy przedstawiono wybrane zagadnienia związane z grafami bez trójek asteroidalnych (bez AT). Trójkę asteroidalną tworzą trzy niesąsiednie wierzchołki grafu takie, że każde dwa z nich są połączone ścieżką omijającą sąsiedztwo trzeciego wierzchołka. Grafy bez AT nie są grafami doskonałymi i zawierają m.in. grafy przedziałowe, grafy permutacji i grafy trapezoidalne. Wszystkie te rodziny reprezentują grafy o liniowej strukturze, a pojęcie trójki asteroidalnej pozwala jednolicie opisać własności tych grafów. W pracy pokazano za pomocą dowodów graficznych, że każdy graf permutacji i każdy graf przedziałowy jest grafem bez AT.

Do implementacji algorytmów i struktur danych użyto języka Python ze względu na jego czytelną składnię i bogatą bibliotekę standardową. Dla wszystkich algorytmów przygotowano testy poprawności (unittest) i wydajności (timeit).

W części teoretycznej pracy omówiono wybrane problemy z teorii grafów w odniesieniu do grafów bez AT: problem zbioru niezależnego, zbioru dominującego, triangulacji grafu i znajdowania najkrótszych ścieżek. Przytoczono przy tym wyniki uzyskane dla grafów permutacji i grafów przedziałowych. W części praktycznej przedstawiono implementację algorytmu wykrywającego grafy bez AT oraz sposoby generowania tych grafów. Następnie zaimplementowano algorytmy znajdujące liczność największego zbioru niezależnego i cały zbiór niezależny dla grafu bez AT. Ponadto, przedstawiono algorytm wyznaczania najmniejszego zbioru dominującego dla grafów bez AT. Algorytmy są dość złożone ze względu na brak znanego geometrycznego modelu dla ogólnych grafów bez AT.

Słowa kluczowe: trójka asteroidalna, grafy przedziałowe, grafy permutacji, zbiór niezależny, zbiór dominujący

English title: Study of asteroidal triple-free graphs with Python

Abstract

In this work selected problems connected with asteroidal triple-free (AT-free) graphs are presented. Three non-adjacent vertices of a graph form an asteroidal triple if every two of them are connected by a path avoiding the neighborhood of the third one. AT-free graphs are not perfect graphs and they include, among others, interval graphs, permutation graphs, and trapezoid graphs. These families consist of graphs with a linear structure and the notion of the asteroidal triple unify the description of graphs. The graphical proofs were presented in order to show that every permutation graph and every interval graph is an AT-free graph.

All algorithms and data structures are implemented using the Python programming language beacause of its clear syntax and the extensive standard library. For all algorithms unit tests and efficiency tests are prepared with unittest and timeit Python modules.

In the theoretical part, selected problems from the graph theory are discussed with regard to AT-free graphs: the independent set problem, the dominating set problem, graph triangulation, and the shortest path problem. The results for permutation graphs and interval graphs are recalled. In the practical part, the algorithm recognizing an AT-free graph is implemented and several useful generators are shown. The paper also presents the algorithms for determining the independence number and a maximum independent set for AT-free graphs. In addition, the work presents the algorithm for finding a minimum dominating set for AT-free graphs. The algorithms are rather long and complex because there is no known geometric model for general AT-free graphs.

Keywords: asteroidal triple, interval graphs, permutation graphs, independent set, dominating set

Spis treści

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy są grafy bez trójek asteroidalnych, w skrócie grafy bez AT (ang. *asteroidal triple-free graphs, AT-free graphs*) [1]. Trójkę asteroidalną tworzą trzy niesąsiednie wierzchołki grafu takie, że każde dwa z nich są połączone ścieżką omijającą sąsiedztwo trzeciego wierzchołka. Do grafów bez AT należą grafy przedziałowe [2], grafy permutacji [3], grafy trapezoidalne [4] i *co-comparability graphs*. Wszystkie te grafy mają liniową strukturę, co oryginalnie było opisywane różnymi pojęciami, natomiast użycie pojęcia trójki asteroidalnej przynosi pewne ujednolicenie języka i lepsze zrozumienie struktury grafów.

Grafy te są szeroko wykorzystywane, m.in. w genetyce, bioinformatyce i informatyce stosowanej [2]. Mają zastosowania na przykład w rozwiązywaniu problemów związanych z alokacją zasobów w analizie operacyjnej, jak również problemów lokalizacji obiektów [2]. Wykorzystywane są również w teorii planowania, która jest dziedziną zajmującą się optymalizacją i organizacją sekwencji zadań lub działań w czasie, aby osiągnąć określone cele. Grafy bez AT mogą być również wykorzystywane w mapowaniu DNA [2].

W pracy rozważane są zbiory niezależne i zbiory dominujące w grafach. Zbiory dominujące mają zastosowanie w sieciach bezprzewodowych, m.in. do znajdowania wydajnych tras w mobilnych sieciach [5]. Mogą być również używane do projektowania bezpiecznych systemów dla sieci elektrycznych [5], a także do problemów związanych z lokalizacją obiektów [6]. Zbiory niezależne natomiast mogą być wykorzystane w praktyce m.in. do zagadnień biologii systemowej i genetyki [7].

1.1. Cel i zakres pracy

Praca ma na celu zgłębienie zagadnień związanych z grafami bez AT, ich praktyczne zastosowania, zbadanie relacji z innymi klasami grafów i implementację wybranych algorytmów. Wykorzystując zebraną teorię, opisane i zaimplementowane zostały algorytmy rozpoznawania grafów bez AT, znajdowania największego zbioru niezależnego i jego liczności, a także minimalnego zbioru dominującego dla grafów bez AT. Algorytmy zostały zaimplementowane w języku Python [8] w ramach rozwoju pakietu *graphtheory* [9].

1.2. Struktura pracy

Treść pracy składa się z czterech rozdziałów. W rozdziale 1 znajduje się wprowadzenie przedstawiające opis, cel i zakres pracy oraz jej strukturę. W rozdziale 2 przedstawione zostały niezbędne definicje z teorii grafów. Znaj-

dują się tam również opisy dotyczące relacji grafów bez AT z innymi klasami grafów. W rozdziale 3 znajduje się opis i implementacja poszczególnych algorytmów, w tym rozpoznawania grafów bez AT, znajdywania największego zbioru niezależnego i jego liczności, a także minimalnego zbioru dominującego. Dla każdego algorytmu dodane zostały szczegółowe opisy, teortyczne złożoności czasowe i implementacje. W rozdziale 4 znajduje się podsumowanie pracy. Dodatek A zawiera testy algorytmów, gdzie eksperymentalnie wyznaczono złożoności czasowe zaimplementowanych algorytmów, a wyniki zilustrowano wykresami.

2. Teoria grafów

W tym rozdziale podamy podstawowe definicje i twierdzenia, z których będziemy korzystać w dalszej części pracy. Definicje są standardowe i na ogół zgodne z książką Cormena [10].

2.1. Podstawowe definicje

Definicja: Graf nieskierowany to uporządkowana para $G = (V, E)$, gdzie $V(G)$ jest niepustym zbiorem elementów, które nazywamy wierzchołkami, a $E(G)$ jest zbiorem nieuporządkowanych par elementów z $V(G)$, które nazywamy krawędziami. Krawędzie oznacza się przez $\{u, v\}$ lub $uv = vu$.

Definicja: Graf skierowany to uporządkowana para $G = (V, E)$, gdzie $V(G)$ jest niepustym zbiorem elementów, które nazywamy wierzchołkami, a $E(G)$ jest zbiorem uporządkowanych par różnych elementów z $V(G)$, które nazywamy krawędziami skierowanymi. Krawędzie oznacza się przez (u, v) lub uv.

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

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

Definicja: Ścieżka w grafie G z wierzchołka p do q jest to ciąg wierzchołków i krawędzi je łączących, które należy przejść w grafie, aby z wierzchołka p dostać się do wierzchołka q. Długość ścieżki jest to liczba krawędzi znajdujących się na ścieżce. Ścieżka długości k to ciąg $k + 1$ wierzchołków, niekoniecznie różnych.

Definicja: Zbiór sąsiedztwa otwartego wierzchołka x grafu $G = (V, E)$ wyrażony jest $N(x) = \{y \in V : xy \in E\}$ [11]. Jest to zbiór wszystkich wierzchołków połączonych krawędzią z wierzchołkiem x. Zbiór sąsiedztwa zamkniętego wierzchołka x grafu $G = (V, E)$ to $N[x] = N(x) \cup \{x\}$. Jest to zbiór wszystkich wierzchołków połączonych krawędzią z wierzchołkiem x i wierzchołka x [11].

2.2. Wybrane rodziny grafów

2.2.1. Grafy bez trójek asteroidalnych

Definicja: Trójkę asteroidalną (ang. *asteroidal triple, AT*) tworzą trzy niesąsiednie wierzchołki grafu nieskierowanego takie, że każde dwa z nich są połączone ścieżką omijającą sąsiedztwo trzeciego wierzchołka. Grafy bez trójek asteroidalnych (ang. *asteroidal triple-free graphs, AT-free graphs*) mają liniową strukturę i zawierają kilka ważnych rodzin grafów, np. grafy przedziałowe, grafy permutacji, grafy trapezoidalne.

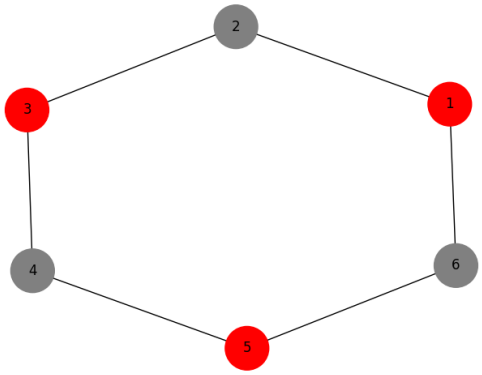
Grafy bez AT **nie** są grafami doskonałymi, o czym świadczy przykład grafu cyklicznego C_5. Jest to wskazówka, że teoria grafów bez AT może być trudna.

Pojęcie trójek asteroidalnych zdefiniowali Lekkerkerker i Boland w roku 1962 podczas badań nad grafami przedziałowymi [13]. Udowodnili twierdzenie, że grafy przedziałowe to grafy cięciwowe bez AT.

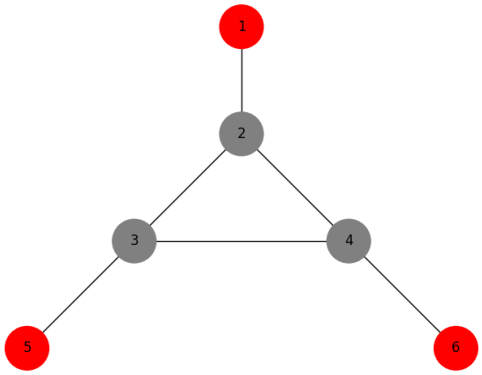
Przykłady grafów: Wszystkie grafy z liczbą wierzchołków co najwyżej $n = 5$ nie mają AT. Dla $n = 6$ mamy ogólnie 112 grafów spójnych, a wśród nich jest 5 grafów z trójkami asteroidalnymi. Warto zauważyć, że wszystkie te grafy z AT nie są grafami permutacji, są grafami kołowymi (ang. *circle graphs*), a dwa z nich są cięciwowe (nie są przedziałowe).

Pięć grafów z $n = 6$ posiadających AT przedstawiono na rysunkach 2.1 (graf C_6), 2.2 (graf cięciwowy), 2.3 (graf Hajósa, cięciwowy), 2.4 (graf Hajósa bez jednej cięciwy), 2.5 (graf Hajósa bez dwóch cięciw).

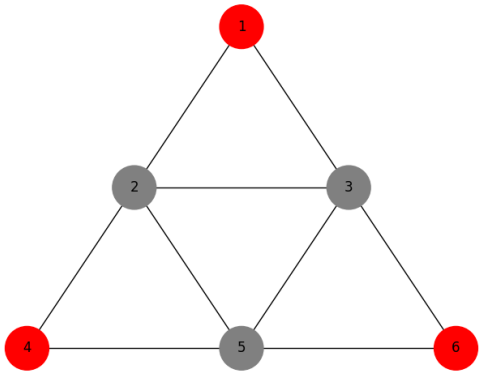
Wśród grafów bez AT z $n = 6$ warto wyróżnić dwa grafy, które nie są cięciwowe i nie są grafami permutacji. Są to graf 3-pryzma (graf Halina) i graf domek z dodatkową krawędzią na dachu. Grafy te wykorzystamy w testach algorytmów.



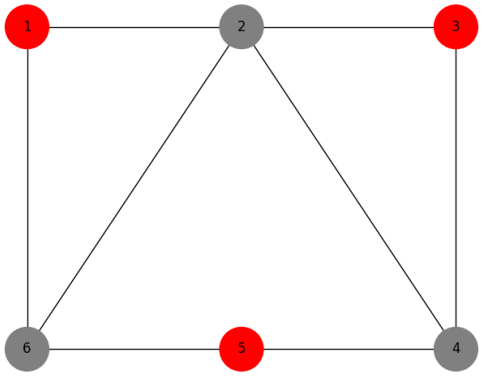
Rysunek 2.1. Graf cykliczny C_6 posiada AT. Można jako AT wybrać trzy czerwone wierzchołki lub trzy szare.



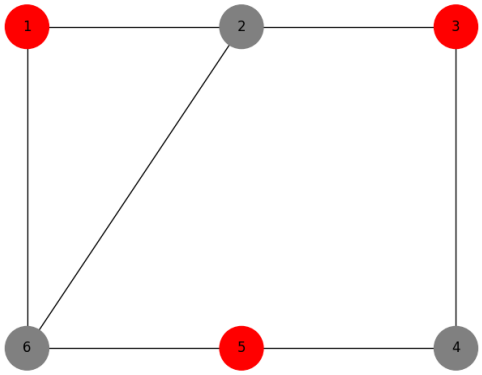
Rysunek 2.2. Graf cięciwowy z $n = 6$ posiadający AT (czerwone wierzchołki).



Rysunek 2.3. Graf Hajósa jest cięciwowy i posiada AT (czerwone wierzchołki).



Rysunek 2.4. Graf Hajósa bez jednej cięciwy posiada AT (czerwone wierzchołki). Zauważmy, że dodając cięciwę $(2, 5)$ otrzymamy graf przedziałowy bez AT.



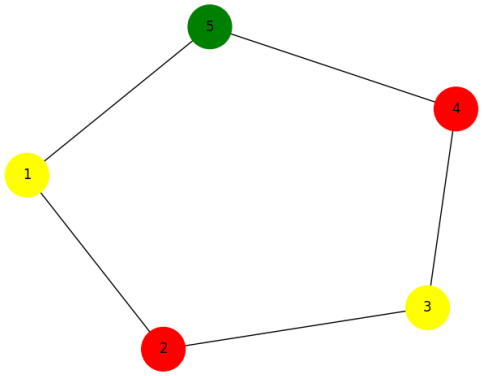
Rysunek 2.5. Graf Hajósa bez dwóch cięciw posiada AT (czerwone wierzchołki). Zauważmy, że dodając cięciwy $(2, 5)$ i $(3, 5)$ otrzymamy graf przedziałowy bez AT.

2.2.2. Grafy doskonałe

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

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

Grafy bez trójki astroidalnej nie zawsze są doskonałe, o czym świadczy przykład grafu cyklicznego C_5 (rys. 2.6). W grafie cyklicznym C_5 nie ma AT, jego liczba chromatyczna jest równa 3, natomiast największa klika ma dwa wierzchołki (końce dowolnej krawędzi).



Rysunek 2.6. Graf cykliczny C_5 z pokolorowanymi wierzchołkami. Wymagane są trzy kolory dla wierzchołków, a największa klika to końce dowolnej krawędzi grafu.

2.2.3. Grafy cięciwowe

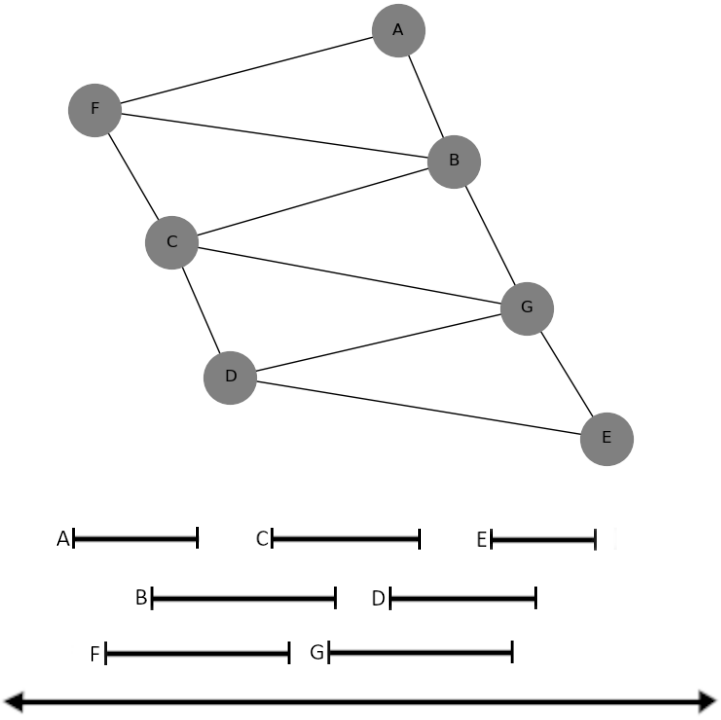
Definicja: Graf cięciwowy (ang. *chordal graph*) to graf nieskierowany, w którym wszystkie cykle indukowane o długości cztery lub więcej zawierają cięciwę. Cięciwa jest to krawędź nie należąca do cyklu, ale łącząca dwa wierzchołki

należące do cyklu [12]. Istnieje kilka innych równoważnych charakteryzacji grafów cięciwowych.

2.2.4. Grafy przedziałowe

Definicja: Grafy przedziałowe (ang. *interval graphs*) można opisać jako grafy przecięć przedziałów na osi liczbowej. Wierzchołki grafu odpowiadają przedziałom, a krawędzie łączą te pary wierzchołków, których odpowiednie przedziały mają niepuste przecięcie. Grafy przedziałowe były analizowane w pracy magisterskiej Macieja Mularskiego [14].

Grafy przedziałowe nie mają AT i są cięciwowe [13]. Przy pomocy reprezentacji przedziałowej łatwo można pokazać, że graf przedziałowy nie może mieć AT.

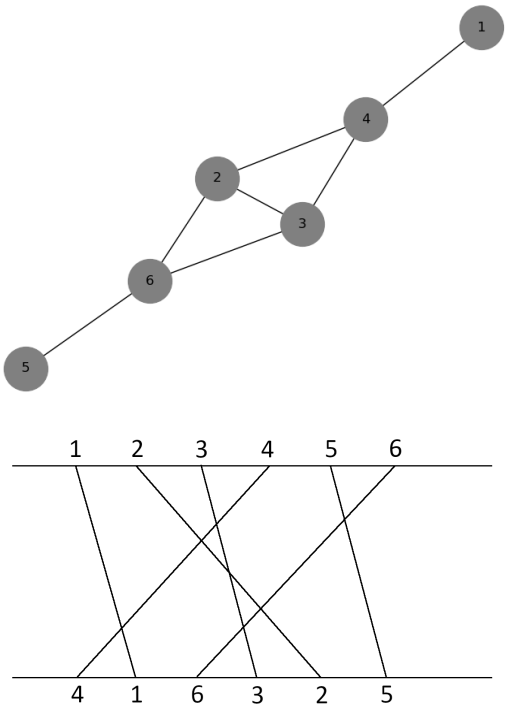


Rysunek 2.7. Graf i odpowiadająca mu reprezentacja przedziałowa.

Aby znaleźć wierzchołki, które nie są bezpośrednio połączone, wystarczy znaleźć niepokrywające się przedziały w reprezentacji przedziałowej. W przykładzie przedstawionym na rysunku 2.7, wierzchołki, które mogłyby tworzyć trójkę asteroidalną to A, C i E, ponieważ mają niepokrywające się przedziały. Jednak aby te wierzchołki tworzyły trójkę asteroidalną, musiałaby istnieć między każdą parą ścieżka omijająca najbliższe sąsiedztwo trzeciego. Można zauważyć, że ścieżka łącząca wierzchołki A i E (ciąg przekrywających się przedziałów) musi zawierać chociaż jeden przedział przekrywający się z przedziałem wierzchołka C.

2.2.5. Grafy permutacji

Definicja: Graf permutacji (ang. *permutation graph*) jest to graf nieskierowany, którego wierzchołki reprezentują elementy permutacji, a krawędzie reprezentują pary elementów tworzących inwersję w permutacji. Grafy permutacji były badane w pracy licencjackiej Alberta Surmacza [15].



Rysunek 2.8. Graf permutacji i jego model geometryczny.

Graf permutacji nie ma AT, co można pokazać przy pomocy modelu z prostymi równoległymi. Aby znaleźć wierzchołki, które nie są bezpośrednio połączone, należy znaleźć trzy takie odcinki, które się nie przecinają. W grafie permutacji między wierzchołkami istnieje krawędź, jeśli odpowiednie odcinki przecinają się. W przykładzie przedstawionym na rysunku 2.8, trzy wierzchołki 1, 3 i 5 nie są bezpośrednio połączone. Aby te wierzchołki tworzyły trójkę asteroidalną, między każdą parą z tych wierzchołków musiałaby istnieć ścieżka omijająca sąsiedztwo trzeciego. Łącząc wierzchołki 1 i 5, przy wykorzystaniu przecięcia odcinków, można zauważyć, że nie da się ominąć najbliższego sąsiedztwa wierzchołka 3.

2.3. Triangulacja grafów bez AT

Triangulacja grafu G to inaczej znajdowanie dopełnienia cięciwowego dla grafu G, czyli znajdowanie grafu cięciwowego H, dla którego $V(H) = V(G)$ oraz $E(G)$ zawiera się w $E(H)$. Zauważmy, że dodawanie nowych krawędzi do grafu bez AT może wytworzyć AT. Z drugiej strony, w roku 1996 Möhring pokazał, że dla grafów bez AT minimalna triangulacja prowadzi do grafów

przedziałowych (bez AT) [16]. Triangulacja minimalna to taka triangulacja, z której nie da się usunąć krawędzi tak, aby dostać inną triangulację.

2.4. Zbiory niezależne

Definicja: Zbiór niezależny (ang. *independent set*) w grafie nieskierowanym G to taki podzbiór S zbioru wierzchołków $V(G)$, że elementy zbioru S są parami rozłączne, czyli nie są połączone krawędzią. Maksymalny zbiór niezależny to zbiór niezależny, który nie jest podzbiorem właściwym innego zbioru niezależnego. Największy zbiór niezależny to zbiór niezależny o największej liczności. Liczba niezależności $\alpha(G)$ to liczba wierzchołków w największym zbiorze niezależnym grafu [11].

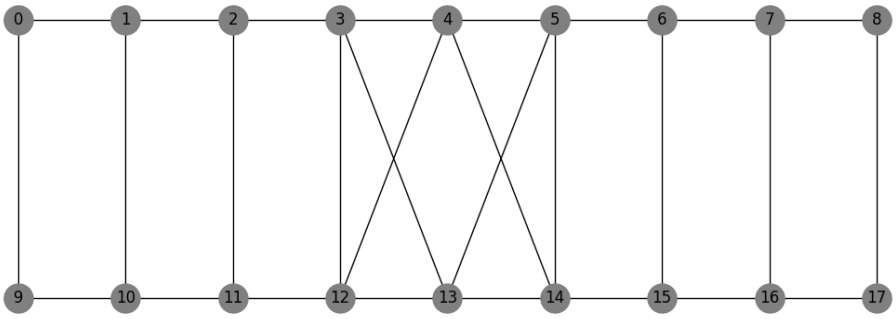
Definicja: Wierzchołek $z \in V \setminus \{x, y\}$ jest pomiędzy x i y, jeśli x i z zawierają się w jednym z komponentów $G \setminus N[y]$, oraz y i z zawierają się w jednym z komponentów $G \setminus N[x]$. $C^x(y)$ oznacza komponent $G \setminus N[x]$ zawierający y.

Definicja: Interwałem $I(x, y)$ grafu G jest zbiór wierzchołków z $V(G)$, które są pomiędzy x i y. Zachodzi związek $I(x, y) = C^x(y) \cap C^y(x)$ [11].

Istnieją komponenty C_s^1, C_s^2, ...,C_s^t wyznaczone przez $G \setminus N[s]$ takie, że

$$I(x, y) \setminus N[s] = I(x, s) \cup I(s, x) \cup \bigcup_{i=1}^{s} C_s^i. \tag{2.1}$$

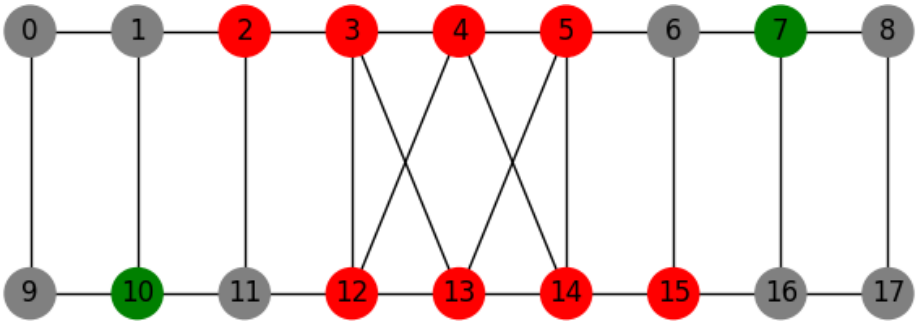
Powyższe twierdzenie wydaje się nieoczywiste, oczekiwalibyśmy rozdzielenia interwału $I(x, y)$ na dwa interwały $I(x, s)$ oraz $I(s, y)$. Aby pokazać, że mogą pojawić się dodatkowe komponenty, można rozważyć przykładowy graf bez AT przedstawiony na rysunku 2.9 i wierzchołki o etykietach 10, 7 i 4.



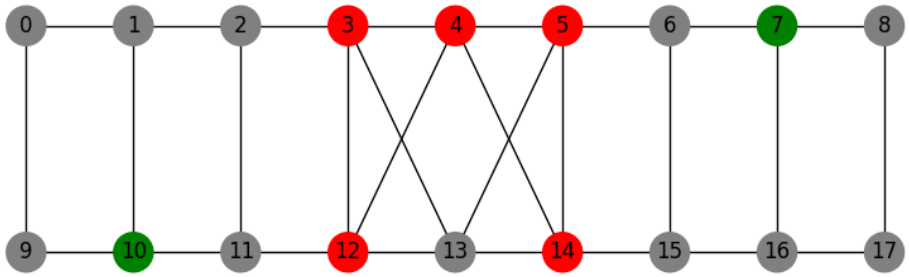
Rysunek 2.9. Przykładowy graf bez AT z $n = 18$.

Możemy wyznaczyć następujące interwały: $I(7, 4) = \{15\}$, $I(4, 10) = \{2\}$, $I(7, 10) = \{2, 3, 4, 5, 12, 13, 14, 15\}$. Rozważając zamknięty zbiór sąsiedztwa wierzchołka o etykiecie 4 otrzymamy $N[4] = \{3, 4, 5, 12, 14\}$.

Różnica zbiorów interwału wierzchołków 7 i 10, i zamkniętego zbioru sąsiedztwa wierzchołka 4 to $I(7, 10) \setminus N[4] = \{2, 13, 15\}$. Struktury komponentów wierzchołka 4 przedstawiają się następująco: $C_1^4 = \{0, 1, 2, 9, 10, 11\}$, $C_2^4 = \{13\}$, $C_3^4 = \{6, 7, 8, 15, 16, 17\}$.



Rysunek 2.10. Przykładowy graf bez AT z zaznaczonymi na czerwono wierzchołkami należącego do interwału $I(7, 10)$.



Rysunek 2.11. Przykładowy graf bez AT z zaznaczonymi na czerwono wierzchołkami należącego do zamkniętego zbioru sąsiedztwa wierzchołka 4 i na zielono wierzchołkami 7 i 10.

Na rysunku 2.10 można zauważyć, że jeśli odejmiemy od interwału $I(7,10)$ zamknięty zbiór sąsiedztwa wierzchołka 4, otrzymamy wierzchołki 2 i 15, które odpowiednio należą do $I(4,10)$ i $I(7,4)$. Jednak do tego zbioru również należeć będzie wierzchołek 13, który nie należy do $I(4,10)$ ani $I(7,4)$, lecz do komponentu C_2^4.

Liczba niezależności: W roku 1999 został zaprezentowany algorytm znajdowania liczby niezależności grafu bez AT o teoretycznej złożoności $O(n^4)$ [11]. Algorytm ten składa się z następujących kroków:

1. Dla każdego wierzchołka x w zbiorze wierzchołków $V(G)$ należy stworzyć strukturę danych, która zawiera informacje o spójnych składowych dla $G \setminus N[x]$.
2. Dla każdej pary *niesąsiednich* wierzchołków x i y w zbiorze wierzchołków $V(G)$ należy znaleźć interwał $I(x,y)$. Wystarczy skorzystać ze wzoru $I(x,y) = C^x(y) \cap C^y(x)$.
3. Należy posortować zbiory spójnych składowych i interwałów zgodnie z porządkiem niemalejącym wg liczności zbiorów.
4. Dla każdej spójnej składowej C i dla każdego interwału I należy obliczyć $\alpha(C)$ i $\alpha(I)$ wg ustalonego porządku.
5. Na podstawie wyników z poprzednich kroków należy obliczyć $\alpha(G)$.

Ogólny algorytm wyznaczający największy zbiór niezależny lub jego liczność dla grafów bez AT będziemy w ramach testów porównywać ze znanymi algorytmami dla szczególnych rodzin grafów, takich jak grafy permutacji i grafy przedziałowe.

Zbiory niezależne w grafach przedziałowych: Grafy przedziałowe są grafami cięciwowymi, więc znając PEO można znaleźć największy zbiór niezależny w czasie $O(n+m)$. Jeżeli jednak mamy dany graf przedziałowy w reprezentacji permutacyjnej ($2n$ etykiet), to wtedy największy zbiór niezależny można wyznaczyć w czasie $O(n)$. Algorytm został zaimplementowany w pracy magisterskiej Macieja Mularskiego [14].

Zbiory niezależne w grafach permutacji: Jeżeli mamy dany graf permutacji w reprezentacji permutacyjnej, to szukanie największego zbioru niezależnego jest równoważne szukaniu najdłuższej rosnącej podsekwencji. W pracy licencjackiej Alberta Surmacza przedstawiono dwa algorytmy rozwiązujące ten problem [15]. Pierwszy algorytm wykorzystujący programowanie dynamiczne działa w czasie $O(n^2)$. Drugi algorytm wykorzystujący metodę *dziel i zwyciężaj* działa w czasie $O(n \log n)$.

2.5. Zbiory dominujące

Definicja: Zbiór dominujący (ang. *dominating set*) w grafie G to taki podzbiór S zbioru wierzchołków $V(G)$, że każdy wierzchołek nie należący do S jest połączony krawędzią z przynajmniej jednym wierzchołkiem z S. Jeżeli

z wierzchołkami grafu są powiązane wagi (liczby rzeczywiste), to wtedy wagą zbioru dominującego jest suma wag jego wierzchołków.

W roku 2000 Kratsch przedstawił algorytm znajdujący najmniejszy zbiór dominujący grafu bez AT [6]. Algorytm na wejściu przyjmuje graf G i parametr w, przy czym dla grafów bez AT $w = 5$. Algorytm może być użyty dla innych typów grafów, a wtedy w ma inną wartość. W celu znalezienia najmniejszego zbioru dominującego grafu zostały użyte poziomy BFS z każdego wierzchołka. Jeśli graf G bez AT podany na wejściu algorytmu posiada wierzchołek x i zbiór dominujący o najmniejszej liczności D taki, że co najwyżej w wierzchołków D należy do dowolnych trzech kolejnych poziomów BFS, wówczas algorytm zwraca zbiór dominujący o najmniejszej liczności grafu G. Algorytm ten ma teoretyczną złożoność $O(n^{w+2})$, ale dla grafów bez AT złożoność można zredukować z $O(n^7)$ do $O(n^6)$. Algorytm ten składa się z następujących kroków:

1. Dla każdego wierzchołka x w zbiorze wierzchołków $V(G)$ należy stworzyć strukturę danych H, która zawiera informacje o poziomach BFS z danego wierzchołka (zbiory wierzchołków oddalonych od x o daną liczbę krawędzi): $H_0 = \{x\}, H_1 = N(x), ..., H_l = \{u \in V : d_G(x, u) = l\}$.
2. Należy ustawić zmienną $i = 1$.
3. Należy utworzyć kolejkę A_1, która zawiera trójki $(S, S, \mathrm{val}(S))$ dla każdego niepustego podzbioru S z $N[x]$ spełniającego warunek $\mathrm{val}(S) := |S| \leqslant w$. Pierwszy element trójki to zbiór wybranych wierzchołków z ostatnich trzech poziomów BFS. Drugi element trójki to częściowe rozwiązanie, które będzie powiększane aż do otrzymania końcowego zbioru dominującego. Trzeci element trójki to liczność częściowego rozwiązania.
4. Dopóki kolejka A_i jest niepusta i spełniony jest warunek $i < l$, należy wykonać następujące kroki:
   a. Należy ustawić zmienną $i = i + 1$.
   b. Dla każdej trójki $(S, S', \mathrm{val}(S))$ w kolejce A_{i-1} należy wykonać:
      i. Należy znaleźć wszystkie podzbiory U z H_i, dla których spełnione jest $|S \cup U| \leqslant w$.
      ii. Następnie dla każdego zbioru $S \cup U$, należy sprawdzić czy zamknięty zbiór sąsiedztwa wierzchołków z $S \cup U$ zawiera się w H_{i-1}. Jeśli warunek jest spełniony, tworzymy nową trójkę $(R, R', \mathrm{val}(R'))$ w następujący sposób: $R := (S \cup U) \setminus H_{i-2}$, $R' := S' \cup U$, $\mathrm{val}(R') = \mathrm{val}(S') + |U|$.
      iii. Jeśli zbiór R z powstałej trójki $(R, R', \mathrm{val}(R'))$ nie znajduje się w kolejce A_i, należy tą trójkę dodać do kolejki A_i.
      iv. Jeśli w kolejce A_i znajduje się taka trójka $(P, P', \mathrm{val}(P'))$, dla której $P = R$ i $\mathrm{val}(R') < \mathrm{val}(P')$, to należy zamienić $(P, P', \mathrm{val}(P'))$ na $(R, R', \mathrm{val}(R'))$.
   c. Pośród wszystkich trójek $(S, S', \mathrm{val}(S))$ w kolejce A_l (to jest ostatni poziom BFS), dla których H_l zawiera się w zamkniętym zbiorze najbliższego sąsiedztwa $N[S]$, należy znaleźć taką trójkę $(B, B', \mathrm{val}(B'))$, dla której $\mathrm{val}(S')$ jest najmniejsze. Jeśli spełniony jest warunek $\mathrm{val}(B') < |D|$, to zapisujemy $D := B'$.

5. Ostateczny wynik poszukiwania najmniejszego zbioru dominującego to zbiór D.

Zbiory dominujące w grafach przedziałowych: Do grafów bez AT należy rodzina grafów przedziałowych. W roku 1998 Chang przedstawił rozwiązania problemów dotyczących zbiorów dominujących dla grafów przedziałowych [17].

Zbiory dominujące w grafach permutacji: Do grafów bez AT należą także grafy permutacji. W roku 1985 Farber i Keil znaleźli algorytm znajdujący najmniejszy zbiór dominujący w grafie permutacji w czasie $O(n^2)$ [18]. Inny algorytm znajdujący zbiór dominujący o najmniejszej wadze działał w czasie $O(n^3)$. W roku 1996 Rhee i inni podali algorytm $O(n + m)$ znajdujący zbiór dominujący o najmniejszej wadze [19].

Ze względu na różne zastosowania rozważa się pewne odmiany zbiorów dominujących, np. niezależne zbiory dominujące, czyli zbiory dominujące będące jednocześnie zbiorami niezależnymi. W przypadku grafów permutacji niezależne zbiory dominujące muszą być maksymalnymi zbiorami niezależnymi, czyli maksymalnymi rosnącymi podsekwencjami w permutacji odpowiadającej danemu grafowi. W artykule przeglądowym Changa można znaleźć algorytm $O(n^2)$ rozwiązujący ten problem [20].

2.6. Najkrótsze ścieżki

Najkrótsza ścieżka między dwoma wierzchołkami w grafie ważonym to ścieżka o najmniejszej długości, przy czym przez długość ścieżki rozumiemy sumę wag krawędzi należących do ścieżki. Zwykle rozważa się graf ważony skierowany, aby uwzględnić możliwość różnych wag dla krawędzi przeciwnych [10]. Graf ważony nieskierowany można interpretować jako graf ważony skierowany, w którym każda krawędź nieskierowana odpowiada parze krawędzi przeciwnych o takiej samej wadze. Problem znalezienia najkrótszych ścieżek z jednego źródła polega na znalezieniu najkrótszych ścieżek od jednego, wybranego wierzchołka, do wszystkich pozostałych. Klasycznym rozwiązaniem jest algorytm Dijkstry, który może być zaimplementowany w czasie $O(m \log n)$ (lista sąsiedztwa) lub $O(n^2)$ (macierz sąsiedztwa). Drugi problem to znalezienie najkrótszych ścieżek między wszystkimi parami wierzchołków grafie. Tutaj dużym osiągnięciem jest algorytm Floyda-Warshalla działający w czasie $O(n^3)$.

Grafy bez AT opisują ogólnie grafy o liniowej strukturze. Intuicja podpowiada, że w takich grafach szukanie najkrótszych ścieżek może być łatwiejsze, niż w ogólnych grafach. Rzeczywiście, dla grafów permutacji i grafów przedziałowych można znaleźć w literaturze prace opisujące algorytmy szybsze niż dla ogólnych grafów. Poniżej przedstawimy wybrane wyniki znalezione w literaturze.

W roku 1995 Ibarra i Zheng przedstawili algorytm równoległy dla problemu znalezienia najkrótszych ścieżek z jednego źródła dla grafów permutacji w czasie $O(\log n)$ [21]. W celu rozwiązania problemu najkrótszych ścieżek

należało zbudować strukturę linków z danego wierzchołka grafu permutacji, za pomocą odpowiednich funkcji. Następnie, ze stworzonej struktury należało odnaleźć najkrótszą ścieżkę ze źródła do każdego wierzchołka.

Algorytm służący do rozwiązania problemu najkrótszych ścieżek dla wszystkich par dla grafów permutacji o złożoności $O(n^2)$ przedstawili Mondal, M. Pal i T. Pal w roku 2002 [22]. Główna idea algorytmu opiera się na utworzeniu struktury przechowującej dane o zamkniętym najbliższym sąsiedztwie dla każdego wierzchołka, wykonaniu obliczeń na podstawie odpowiednich funkcji i zapisaniu ich w tablicach. Następnie, za pomocą stosownej funkcji, znajdowany jest dystans pomiędzy parami wierzchołków.

W roku 2007 Sprague przedstawił algorytm służący do znajdywania najkrótszych ścieżek dla wszystkich par wierzchołków o całkowitej złożoności $O(n^2)$ [23]. Opiera się on na redukcji do dwudzielnych grafów permutacyjnych i dalszej redukcji do grafów przedziałowych jednostkowych. Następnie wykonywane są obliczenia dla stworzonych grafów przedziałowych jednostkowych i znajdowane są najkrótsze ścieżki.

Algorytm rozwiązujący problem najkrótszej ścieżki z jednego źródła w czasie $O(n)$ przedstawili Atallah, Chen i Lee w roku 1995 [24]. Za pomocą utworzonej struktury do przechowywania danych i odpowiednich obliczeń na przedziałach znajdowana jest najkrótsza ścieżka z jednego źródła.

Istnieje wiele przedstawionych algorytmów rozwiązujących problemy najkrótszych ścieżek dla grafów permutacji czy przedziałowych. Możliwe jest, że dla grafów bez AT również istnieją algorytmy szybsze niż dla ogólnych grafów, które czekają na odkrycie.

3. Algorytmy

Rozdział zawiera opisy implementacji wybranych algorytmów dla grafów bez AT, a także przykładowe obliczenia na grafach z użyciem pakietu graphtheory.

3.1. Generowanie grafów bez AT

W literaturze nie znaleźliśmy informacji o sposobach generowania grafów bez AT, więc do obliczeń wykorzystamy generatory grafów permutacji i grafów przedziałowych.

Graf przedziałowy można wygenerować używając jednego z generatorów dostępnych w pakiecie graphtheory. Dla n wymaganych wierzchołków grafu dostajemy w pierwszym kroku permutację $2n$ liczb (reprezentacja permutacyjna), a w drugim kroku tworzymy graf abstrakcyjny.

```
# Generating interval graphs.
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.chordality.intervaltools import make_path_interval
from graphtheory.chordality.intervaltools import make_tepee_interval
from graphtheory.chordality.intervaltools import make_2tree_interval
from graphtheory.chordality.intervaltools import make_star_interval
from graphtheory.chordality.intervaltools import make_ktree_interval
from graphtheory.chordality.intervaltools import make_abstract_interval_graph

# Creating a permutation representation.
n = 10
perm = make_random_interval(n)    # random interval graph
#perm = make_path_interval(n)    # P_n graph
#perm = make_tepee_interval(n)    # tepee graph
#perm = make_2tree_interval(n)    # 2-tree
#perm = make_star_interval(n)    # K_{1,n-1} graph
#perm = make_ktree_interval(n, k=n // 2)    # interval k-tree
assert len(perm) == 2*n

# Creating the abstract graph from the permutation.
G = make_abstract_interval_graph(perm)
assert isinstance(G, Graph)
```

Graf permutacji można wygenerować używając jednego z generatorów dostępnych w pakiecie graphtheory. Dla n wymaganych wierzchołków grafu dostajemy w pierwszym kroku permutację n liczb, a w drugim kroku tworzymy graf abstrakcyjny.

```
# Generating permutation graphs.
from graphtheory.structures.edges import Edge
```

```
from graphtheory.structures.graphs import Graph
from graphtheory.permutations.permtools import make_random_perm
from graphtheory.permutations.permtools import make_star_perm
from graphtheory.permutations.permtools import make_bipartite_perm
from graphtheory.permutations.permtools import make_path_perm
from graphtheory.permutations.permtools import make_ladder_perm
from graphtheory.permutations.permtools import make_abstract_perm_graph

# Create permutations of numbers from 0 to n-1.
n = 10
perm = make_random_perm(n)    # random perm graph
#perm = make_star_perm(n)    # bipartite graph K_{1,n-1}
#perm = make_bipartite_perm(p=3, q=4)    # bipartite graph K_{p,q}
#perm = make_path_perm(n)    # path graph P_n
#perm = make_ladder_perm(n)    # ladder graph (bipartite)
assert isinstance(perm, list)
assert len(perm) == n
assert sorted(perm) == list(range(n))

# Creating an abstract graph.
G = make_abstract_perm_graph(perm)
assert isinstance(G, Graph)
```

3.2. Rozpoznawanie grafów bez AT

Jeden z algorytmów rozpoznawania grafów bez AT podał Kohler w roku 2004 [25]. Jego implementacja znajduje się w bibliotece NetworkX [26]. Dalej przedstawimy analogiczną implementację korzystającą z narzędzi i konwencji pakietu graphtheory. Stworzono klasę ATFreeGraph, przez analogię do klas HalinGraph (grafy Halina) czy BipartiteGraphBFS (grafy dwudzielne rozpoznawane z użyciem BFS).

Dane wejściowe: Dowolny graf prosty nieskierowany G.

Problem: Sprawdzenie, czy graf G zawiera AT.

Dane wyjściowe: Znaleziona trójka asteroidalna (krotka) jest zapisywana w atrybucie asteroidal_triple. Jeżeli trójka nie zostanie znaleziona, to atrybut ma wartość None.

Opis algorytmu: Algorytm rozpoczynamy od utworzenia struktury danych, która zawiera informacje o spójnych składowych grafu dla każdego wierzchołka. Następnie stworzone zostaje dopełnienie grafu. Dla każdych sąsiadujących wierzchołków u i v dopełnienia grafu sprawdzamy, czy istnieje taki wierzchołek w, nienależący do najbliższego sąsiedztwa tych wierzchołków w oryginalnym grafie, dla którego spełniony jest następujący warunek: dla każdego z wierzchołków u, v, w sprawdzamy, czy pozostałe wierzchołki należą do tej samej spójnej składowej grafu danego wierzchołka, tj. czy są połączone. Jeśli znalezione zostały wierzchołki spełniające dane warunki,

wynik algorytmu zapisywany jest w atrybucie klasy. Jeśli atrybut klasy jest pusty, to graf nie posiada trójki asteroidalnej.

Złożoność: Złożoność czasowa algorytmu podawana w artykule Kohlera wynosi $O(n\bar{m}+nm)$, gdzie $\bar{m}$ oznacza liczbę krawędzi w dopełnieniu grafu wejściowego. Drugi wyraz to budowa struktury danych dla składowych spójnych. Upraszczając można określić złożoność czasową jako $O(n^3)$.

Listing 3.1. Moduł atfreegraphs, rozpoznawanie grafów bez AT.

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

from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.connectivity.connected import ConnectedComponentsBFS

class ATFreeGraph:
    """Check if graph contains any asteroidal triple.

    Attributes
    ----------
    graph : input graph

    Notes
    -----
    Based on description from:
    https://networkx.org/documentation/stale/reference/algorithms/asteroidal.html
    """
    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("The graph is directed")
        self.graph = graph
        self.asteroidal_triple = None
        self.component_structure = {}

    def run(self):
        """Executable pseudocode."""
        self.create_component_structure()
        self.find_asteroidal_triple()

    def is_at_free(self) -> bool:
        """Checking if the graph is AT-free."""
        if self.component_structure is None:
            raise ValueError("Run the algorithm first")
        return self.asteroidal_triple is None

    def create_component_structure(self):
        """Creating the component structure for the graph."""
        graph_nodes = set(self.graph.iternodes())
        for v in self.graph.iternodes():
            closed_neighbourhood = set(self.graph.iteradjacent(v)).union([v])
            row_dict = dict((u, 0) for u in closed_neighbourhood)
            G_reduced = self.graph.subgraph(
                graph_nodes - closed_neighbourhood)
            algorithm = ConnectedComponentsBFS(G_reduced)
            algorithm.run()
```

```
            for u in algorithm.cc:
                row_dict[u] = algorithm.cc[u] + 1
            self.component_structure[v] = row_dict
        return self.component_structure

    def find_asteroidal_triple(self):
        """Finding an asteroidal triple."""
        if self.graph.v() < 6:
            return None
        graph_nodes = set(self.graph.iternodes())
        graph_complement = self.graph.complement()

        for edge in graph_complement.iteredges():
            u, v = edge.source, edge.target
            u_neighbourhood = set(self.graph.iteradjacent(u)).union([u])
            v_neighbourhood = set(self.graph.iteradjacent(v)).union([v])
            uv_neighbourhoods_union = u_neighbourhood.union(v_neighbourhood)
            for w in (graph_nodes - uv_neighbourhoods_union):
                if (
                    self.component_structure[u][v] ==
                    self.component_structure[u][w]
                    and self.component_structure[v][u] ==
                    self.component_structure[v][w]
                    and self.component_structure[w][u] ==
                    self.component_structure[w][v]
                ):
                    self.asteroidal_triple = (u, v, w)
                    return None
```

3.3. Wyznaczanie liczności największego zbioru niezależnego dla grafów bez AT

Algorytm wyznaczania liczby niezależności został podany w pracy z roku 1999 [11], a jego listę kroków podaliśmy w rozdziale 2. W naszej implementacji wykorzystujemy inne struktury danych niż omawiane w oryginalnym artykule, ale testy pokazują, że osiągnęliśmy złożoność obliczeniową podawaną w literaturze.

Dane wejściowe: Dowolny spójny graf bez AT.

Problem: Znajdowanie liczby niezależności dla grafów bez AT.

Dane wyjściowe: Liczność największego zbioru niezależnego jest zapisywana w atrybucie cardinality, zgodnie z konwencją pakietu graphtheory. Atrybut independent_set ma wartość None i w tej implementacji nie jest wykorzystywany.

Opis algorytmu: Algorytm składa się z pięciu kroków opisanych w rozdziale 2 na bazie pracy [11].

Złożoność: Całkowita złożoność czasowa algorytmu wynosi $O(n^4)$. Omówimy dokładniej złożoność kolejnych kroków algorytmu. W kroku 1 wyznacza się strukturę danych przechowującą informację o składowych, analogicznie jak przy rozpoznawaniu grafu bez AT. W teorii złożoność wynosi $O(n(n+m))$, ponieważ dla każdego wierzchołka v należy wyznaczyć składowe $G \setminus N[v]$. Nasza implementacja również nie przekracza złożoności $O(n^3)$.

W kroku 2 obliczane są interwały $I(x, y)$ dla wszystkich par niesąsiednich wierzchołków x i y. Złożoność wynosi zgodnie z literaturą $O(n^3)$, przy czym w naszej implementacji wykorzystano operacje na zbiorach.

W kroku 3 następuje sortowanie bukietowe przedziałów i komponentów, których jest co najwyżej $O(n^2)$, więc całkowity czas to $O(n^3)$.

Najbardziej czasochłonny jest krok 4, w ktorym obliczane są wartości liczby niezależności dla coraz większych przedziałów i komponentów. Wg literatury obliczenia dla komponentów zajmują czas $O(n^3)$, a dla przedziałów $O(n^4)$. Nasza implementacja wykorzystuje zbiory i rozumowanie z literatury nie do końca można zastosować, ale ogólny schemat obliczeń jest zachowany.

W kroku 5 obliczana jest wartość $\alpha(G)$ w czasie $O(n^2)$.

Uwagi: Testy pokazują, że najtrudniejsze w obliczeniach są grafy wydłużone, takie jak graf ścieżka P_n, czy 2-drzewo. Wtedy czas obliczeń rzeczywiście skaluje się jak $O(n^4)$. Z kolei grafy bardziej gęste mają czas obliczeń rzędu $O(n^3)$. Prawdopodobnie wynika to z liczniejszych sąsiedztw wierzchołków, a co za tym idzie mniejszą liczbą komponentów i przedziałów do zbadania.

Listing 3.2. Moduł atfreeiset1 , wyznaczanie liczności największego zbioru niezależnego.

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

from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.connectivity.connected import ConnectedComponentsBFS

class ATFreeIndependentSet:
    """Find a maximum independent set for AT-free graphs."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.independent_set = None
        self.cardinality = 0
        self.component_structure = dict()
        self.component_lists = dict()
        self.intervals = dict()
        self.bucket_C = None
        self.bucket_I = None
        self.alpha_C = dict()
        self.alpha_I = dict()

    def run(self):
        """Executable pseudocode."""
```

```
        self.create_component_structure()
        self.find_intervals()
        self.sort_components_intervals()
        self.find_alpha_C_I()
        self.find_alpha_G()

    def create_component_structure(self):
        """Create component structure for G."""
        V = set(self.graph.iternodes())
        for v in V:
            closed_neighborhood = set(self.graph.iteradjacent(v)).union([v])
            G_reduced = self.graph.subgraph(V - closed_neighborhood)
            algorithm = ConnectedComponentsBFS(G_reduced)
            algorithm.run()
            component_list = [set() for x in range(algorithm.n_cc + 1)]
            component_list[0] = closed_neighborhood
            for u in closed_neighborhood:
                self.component_structure[v,u] = component_list[0]
            for u in algorithm.cc:
                idx = algorithm.cc[u] + 1
                component_list[idx].add(u)
                self.component_structure[v,u] = component_list[idx]
            self.component_lists[v] = [
                (v, next(iter(s))) for s in component_list[1:]]

    def find_intervals(self):
        """Finding intervals for nonadjacent pairs of vertices."""
        Gc = self.graph.complement()
        for edge in Gc.iteredges():
            v, u = edge.source, edge.target
            self.intervals[v,u] = (self.component_structure[v,u]
                & self.component_structure[u,v])
            self.intervals[u,v] = self.intervals[v,u]

    def sort_components_intervals(self):
        """Sorting components and intervals (bucket sort, O(n^2) time)."""
        self.bucket_C = [[] for i in range(self.graph.v())]
        self.bucket_I = [[] for i in range(self.graph.v())]
        for v in self.graph.iternodes():
            for key in self.component_lists[v]:
                self.bucket_C[len(self.component_structure[key])].append(key)
        for key in self.intervals:
            self.bucket_I[len(self.intervals[key])].append(key)

    def find_alpha_C_I(self):
        """Finding alpha(I) and alpha(C)."""
        for key in self.bucket_I[0]:
            self.alpha_I[key] = 0
        for key in self.bucket_I[1]:
            self.alpha_I[key] = 1
        for key in self.bucket_C[1]:
            self.alpha_C[key] = 1
        for key in self.bucket_I[2]:
            self.alpha_I[key] = self.find_alpha_I(key)
        for key in self.bucket_C[2]:
            self.alpha_C[key] = 1
        for i in range(3, self.graph.v()):
```

```
            for key in self.bucket_I[i]:
                self.alpha_I[key] = self.find_alpha_I(key)
            for key in self.bucket_C[i]:
                self.alpha_C[key] = self.find_alpha_C(key)

    def find_alpha_I(self, key):
        """Lemma 6.3 from [1999 Broersma]."""
        results = []
        x, y = key
        interval = self.intervals[key]
        for s in interval:
            res = self.alpha_I[x,s] + self.alpha_I[s,y]
            S = interval.difference(self.intervals[x,s],
                                    self.intervals[s,y],
                                    self.component_structure[s,s])
            for key in self.component_lists[s]:
                C = self.component_structure[key]
                if C.issubset(S):
                    S = S.difference(C)
                    res += self.alpha_C[key]
            assert len(S) == 0, S
            results.append(res)
        return 1 + max(results)

    def find_alpha_C(self, key):
        """Lemma 6.2 from [1999 Broersma]."""
        results = []
        x, y = key
        component = self.component_structure[key]
        for z in component:
            res = self.alpha_I[x,z]
            S = component.difference(self.component_structure[z,z],
                                     self.intervals[x,z])
            for key in self.component_lists[z]:
                C = self.component_structure[key]
                if C.issubset(S):
                    S = S.difference(C)
                    res += self.alpha_C[key]
            assert len(S) == 0, S
            results.append(res)
        return 1 + max(results)

    def find_alpha_G(self):
        """Final calculations of alpha(G)."""
        results = []
        for v in self.graph.iternodes():
            s = sum(self.alpha_C[key] for key in self.component_lists[v])
            results.append(s)
        self.cardinality = 1 + max(results)
```

3.4. Wyznaczanie największego zbioru niezależnego dla grafów bez AT

W celu wyznaczenia największego zbioru niezależnego dla grafów bez AT należało zmodyfikować niektóre części programu, który znajdował liczność największego zbioru niezależnego. Należało zmodyfikować metodę, która znajdowała $\alpha(I)$ oraz $\alpha(C)$. Zmienne, zamiast liczności zbiorów, przechowują wprost zbiory wierzchołków. Znaleziony największy zbiór niezależny jest zapisywany w atrybucie independent_set, a jego liczność w atrybucie cardinality.

Listing 3.3. Moduł atfreeiset2, wyznaczanie $\alpha(I)$ oraz $\alpha(C)$.

```
def find_alpha_C_I(self):
    """Finding alpha(I) and alpha(C)."""
    for key in self.bucket_I[0]:
        self.alpha_I[key] = set()
    for key in self.bucket_I[1]:
        self.alpha_I[key] = set(self.intervals[key])
    for key in self.bucket_C[1]:
        self.alpha_C[key] = set(self.component_structure[key])
    for key in self.bucket_I[2]:
        self.alpha_I[key] = self.find_alpha_I(key)
    for key in self.bucket_C[2]:
        S = set(self.component_structure[key])
        S.pop()
        self.alpha_C[key] = S
    for i in range(3, self.graph.v()):
        for key in self.bucket_I[i]:
            self.alpha_I[key] = self.find_alpha_I(key)
        for key in self.bucket_C[i]:
            self.alpha_C[key] = self.find_alpha_C(key)
```

Metoda znajdująca maksymalną liczność zbioru niezależnego również została zmodyfikowana tak, aby znalazła również największy niezależny zbiór z wyników obliczeń.

Listing 3.4. Moduł atfreeiset2, znajdowanie $\alpha(G)$.

```
def find_alpha_G(self):
    """Final calculations of alpha(G)."""
    results = []
    for v in self.graph.iternodes():
        S = set([v])
        for key in self.component_lists[v]:
            S.update(self.alpha_C[key])
        results.append(S)
    self.independent_set = max(results, key=len)
    self.cardinality = len(self.independent_set)
```

3.5. Wyznaczanie najmniejszego zbioru dominującego dla grafów bez AT

Dane wejściowe: Dowolny spójny graf bez AT.

Problem: Znajdowanie najmniejszego zbioru dominującego dla grafów bez AT.

Dane wyjściowe: Znaleziony najmniejszy zbiór dominujący jest zapisywany w atrybucie dominating_set, a jego liczność w atrybucie cardinality. Jest to zgodne z konwencją stosowaną w pakiecie graphtheory.

Opis algorytmu: Algorytm składa się z kroków opisanych w rozdziale 2 na bazie pracy [6].

Złożoność czasowa: Całkowita złożoność czasowa algorytmu wynosi $O(n^{w+2})$. Parametr w w przypadku grafów bez AT wynosi 5, dlatego złożność czasowa jest $O(n^7)$.

Na początku wyznacza się strukturę danych H, która zawiera informacje o poziomach BFS z danego wierzchołka. W teorii złożoność wynosi $O(n(n+m))$, ponieważ dla każdego wierzchołka należy wyznaczyć poziomy sąsiedztwa BFS w czasie liniowym $O(n+m)$.

W kolejnym kroku tworzona jest kolejka A_1 zawierająca trójki $(S, S, \mathrm{val}(S))$ dla każdego niepustego podzbioru S z $N[x]$ spełniającego $\mathrm{val}(S) := |S| \leqslant w$. Złożoność czasowa tego etapu wynosi $O(n^3)$.

Dopóki kolejka A_i jest niepusta oraz $i < l$, wykonywane są poniżej opisane obliczenia. Dla każdej trójki $(S, S', \mathrm{val}(S))$ należy znaleźć wszystkie podzbiory U z H_i, dla których spełnione jest $|S \cup U| \leqslant w$. Dla każdego takiego zbioru $|S \cup U|$ należy sprawdzić, czy zamknięty zbiór sąsiedztwa wierzchołków ze zbioru $|S \cup U|$ zawiera się w H_{i-1}. W teorii złożoność wynosi $O(n^{w+1})$, jest ona zdominowana przez czas obliczeń dla podbiorów $S \cup U$ spełniających $|S \cup U| \leqslant w$, które zawierają się w trzech następujących po sobie poziomach BFS z danego wierzchołka. Ilość czasu potrzebna na podzbiór $S \cup U$ wynosi $O(n)$ i w sumie rozważane jest $O(n^w)$ podzbiorów, ponieważ $|S \cup U| \leqslant w$. Całość tego etapu ma złożoność $O(n^{w+2})$.

Uwagi: Po wykonaniu testów widać, że dla dwudzielnych grafów permutacji i grafów k-drzewo złożoność czasowa rzeczywiście skaluje się jak $O(n^7)$. Natomiast dla grafów bardziej wydłużonych, w tym dla grafu ścieżka P_n, 3-drzewo czy grafu drabinowego, testy pokazują złożoność $O(n^3)$. Prawdopodobnie wynika to z liczby wierzchołków na poziomach BFS, im graf jest gęstszy tym więcej obliczeń należy wykonać.

Uwagi do implementacji: Poziomy BFS były wyznaczane za pomocą klasy BFSWithDepthTracker, dostępnej w pakiecie graphtheory. Każdy poziom BFS to po prostu zbiór wierzchołków grafu oddalonych o daną liczbę kroków/krawędzi.

W oryginalnym artykule obiektu A_i przechowujące trójki są nazywane kolejkami. W algorytmie potrzebne jest wyszukiwanie trójek ze względu na pierwszy element, dlatego zastosowaliśmy dla A_i słowniki, gdzie kluczem jest pierwszy element przechowywany jako **frozenset**.

W algorytmie pojawia się potrzeba sprawdzania podzbiorów U zbioru H_i o ograniczonej liczności $|U|$. Wykorzystano do tego funkcję itertools.combinations, a całość zamknięto w wygodnym generatorze iter_U.

Listing 3.5. Moduł atfreedset, wyznaczanie najmniejszego zbioru dominującego.

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

import itertools
import collections
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.traversing.bfs import BFSWithDepthTracker


class ATFreeDominatingSet:
    '''Find a minimum dominating set of AT-free graphs.

    Based on:

    D. Kratsch, Domination and total domination in asteroidal triple-free
    graphs, Discrete Appl. Math. 99 No.1-3, 111-123 (2000).
    '''
    def __init__(self, graph, w=5):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.w = w
        self.dominating_set = set(self.graph.iternodes())
        self.cardinality = self.graph.v()
        self._H = None
        self._max_level = 0
        self._A = None

    def run(self):
        """Executable pseudocode."""
        for source in self.graph.iternodes():
            self.find_bfs_levels(source)
            self.init_queue()
            i = 1
            while self._A[i] and i < self._max_level:
                i += 1
                for S1 in self._A[i-1]:
                    S1, S2, val_S2 = self._A[i-1][S1]
                    for U in self.iter_U(self._H[i], S1):
                        S1uU = S1.union(U)
                        N_S1uU = self.find_closed_neighborhood(S1uU)
                        if self._H[i-1].issubset(N_S1uU):
                            R1 = frozenset(S1uU - self._H[i-2])
                            R2 = S2.union(U)
                            val_R2 = val_S2 + len(U)

                            if R1 not in self._A[i]:
                                self._A[i][R1] = (R1, R2, val_R2)
                            else:
                                P1, P2, val_P2 = self._A[i][R1]
                                if val_R2 < val_P2:
                                    self._A[i][R1] = (R1, R2, val_R2)
            B1, B2, val_B2 = None, None, float('inf')
```

```
            for S1 in self._A[self._max_level]:
                S1, S2, val_S2 = self._A[self._max_level][S1]
                N_S1 = self.find_closed_neighborhood(S1)

                if (set(N_S1).issuperset(self._H[self._max_level]) and
                    val_S2 < val_B2):
                    B1, B2, val_B2 = S1, S2, val_S2

            if B2 and val_B2 < self.cardinality:
                self.dominating_set = B2
                self.cardinality = val_B2

    def find_bfs_levels(self, node):
        """Creating a structure with BFS levels."""
        order = []
        algorithm = BFSWithDepthTracker(self.graph)
        algorithm.run(node, pre_action=lambda pair: order.append(pair))
        self._max_level = max(level for (node, level) in order)
        self._H = [set() for i in range(self._max_level + 1)]
        for (node, level) in order:
            self._H[level].add(node)

    def init_queue(self):
        """Initialization of queue."""
        self._A = [dict() for i in range(self._max_level + 1)]
        for r in range(1, self.w + 1):
            for S in itertools.combinations(self._H[1].union(self._H[0]), r):
                S = frozenset(S)
                self._A[1][S] = (S, S, len(S))

    def iter_U(self, Hi, S):
        """Iterator for all combinations of vertices in Hi and S."""
        for r in range(0, len(Hi) + 1):
            for U in itertools.combinations(Hi, r):
                if len(S.union(U)) <= self.w:
                    yield U

    def find_closed_neighborhood(self, S):
        """Finding closed neighbourhood of vertices in S."""
        result = set(S)
        for node in S:
            result.update(self.graph.iteradjacent(node))
        return result
```

4. Podsumowanie

W ramach pracy zostały przedstawione wybrane zagadnienia związane z grafami bez AT. Grafy te są trudne do badania, ponieważ nie należą do grafów doskonałych, a także nie jest znany model geometryczny, który mógłby je reprezentować. Z tego powodu dużo uwagi zostało poświęcone rodzinom grafów należących do grafów bez AT, ze znanymi modelami geometrycznymi. Za pomocą dowodów graficznych pokazano, że grafy przedziałowe i grafy permutacyjne należą do grafów bez AT. Przedstawiono wybrane algorytmy z zebranej literatury dotyczące grafów bez AT, w tym grafów przedziałowych i permutacyjnych.

Zaimplementowano algorytm o złożoności obliczeniowej $O(n^3)$ sprawdzający, czy dany graf nieskierowany jest grafem bez AT. Wykorzystano tu bliźniaczą implementację z biblioteki *NetworkX*. Opisano problem dotyczący znalezienia liczności największego zbioru niezależnego i znalezienia samego zbioru niezależnego w grafie oraz zaprezentowano, na podstawie pracy [11], algorytm rozwiązujący ten problem dla grafów bez AT. Złożoność obliczeniowa algorytmów wynosi $O(n^4)$. Przedstawione zostało także rozwiązanie problemu znalezienia najmniejszego zbioru dominującego dla grafów bez AT na podstawie pracy [6]. Tutaj złożoność obliczeniowa wynosi $O(n^7)$. Zaprezentowano opisy wybranych algorytmów dotyczących problemu najkrótszych ścieżek dla dwóch rodzin grafów należących do grafów bez AT.

Wykonano testy wydajnościowe dla wszystkich zaimplementowanych algorytmów i porównano je z wynikami teoretycznymi. Zwracano uwagę na złożoności obliczeniową, wydajnościową i pamięciową. Testy poprawności zaprezentowanych modułów wykonywane były za pomocą modułu `unittest`. Do tych testów użyte zostały różne, wybrane klasy grafów należące do rodziny grafów bez AT.

Badania na grafami bez AT są aktywnie prowadzone w literaturze, m. in. w kontekście uporządkowania wierzchołków zwracanego przez różne sposoby przechodzenia przez graf. Obok klasycznych BFS i DFS, w pracy [27] analizowano ogólne przeszukiwanie (ang. *Generic Search*), LBFS, LDFS, MNS (ang. *Maximal Neighbor Search*). Wiadomo, że DFS może być użyte do rozpoznawania grafów planarnych, a LBFS pozwala znaleźć PEO w grafach cięciwowych. Okazuje się, że przeszukiwanie LBFS może być użyte do znalezienia pary dominującej w grafie bez AT. W pracy z roku 2014 Corneil i Stacho [28] udowodnili twierdzenie, że graf nie ma AT wtedy i tylko wtedy, gdy posiada specjalne uporządkowanie wierzchołków o nazwie *AT-free ORDER*. Okazuje się, że to uporządkowanie dla pewnych grafów bez AT jest istotnie różne od uporządkowań generowanych przez LBFS, LDFS, czy nawet zwykłego DFS.

A. Testy algorytmów

A.1. Testy rozpoznawania grafów bez AT

W celu znalezienia praktycznej wydajności algorytmu rozpoznawania grafów bez AT, zmierzono czas jego działania dla wybranych grafów bez AT. Wykorzystano do tego grafy przedziałowe o klikach maksymalnych równych 5, przy czym liczba wierzchołków była sukcesywnie powiększana.

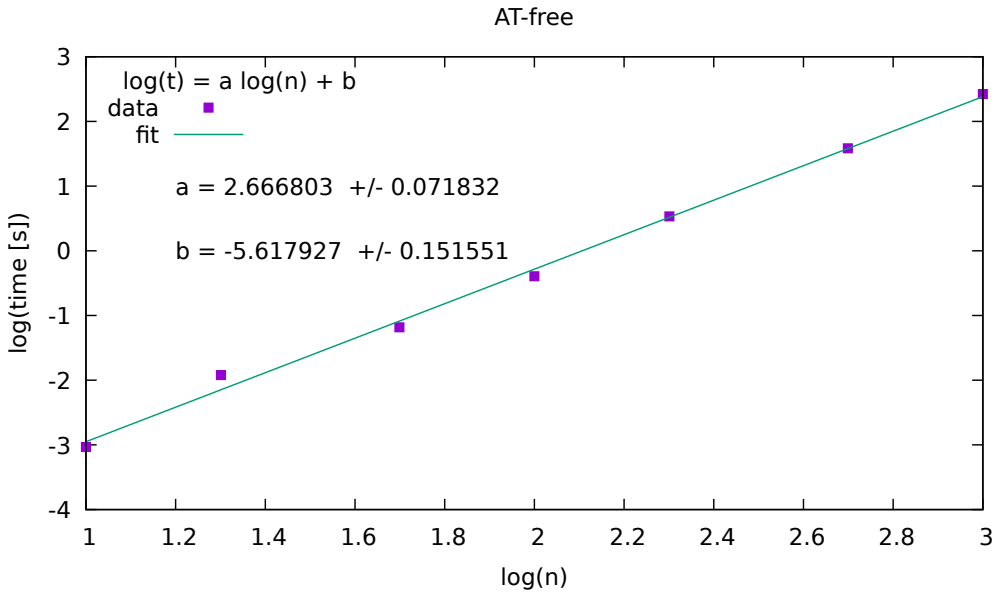
Teoretyczna złożoność algorytmu znajdowania trójek astroidalnych wynosi $O(n^3)$. Uzyskany współczynnik dopasowania, widoczny na wykresie A.1, wynosi $a = 2.667(72)$ i potwierdza założenia teoretyczne.

A.2. Testy wyznaczania największego zbioru niezależnego i jego liczności dla grafów bez AT

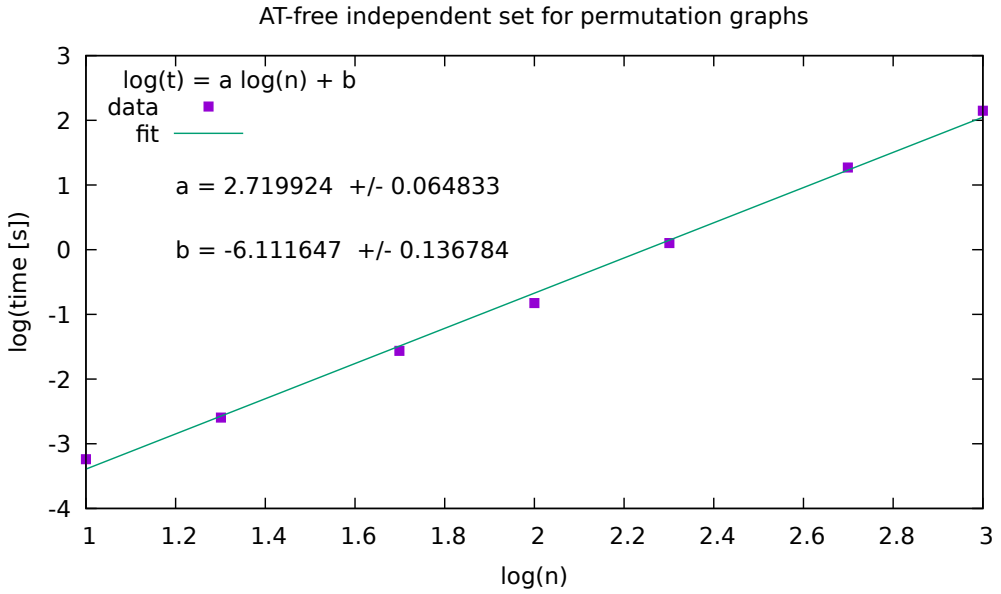
Aby znaleźć praktyczną wydajność algorytmu wyznaczania największego zbioru niezależnego, zmierzono czas jego działania dla wybranych rodzin grafów bez AT. Pomiary zostały sporządzone dla grafów permutacji, grafów przedziałowych ścieżka P_n oraz k-drzewo. Zostały również zmierzone czasy wykonywania algorytmu znajdywania liczności największego zbioru niezależnego, jednak wyniki nie różnią się znacząco od wyników algorytmu znajdywania największego zbioru niezależnego, co można zauważyć na zamieszczonych wykresach. Dla grafów wydłużonych, takich jak graf ścieżka P_n, czas obliczeń jest wydłużony i rzeczywiście skaluje się jak złożoność $O(n^4)$. Natomiast dla innych wykorzystanych grafów czas obliczeń mieści się w $O(n^3)$, czyli w mniejszej niż teoretyczna złożoności.

A.3. Testy wyznaczania najmniejszego zbioru dominującego dla grafów bez AT

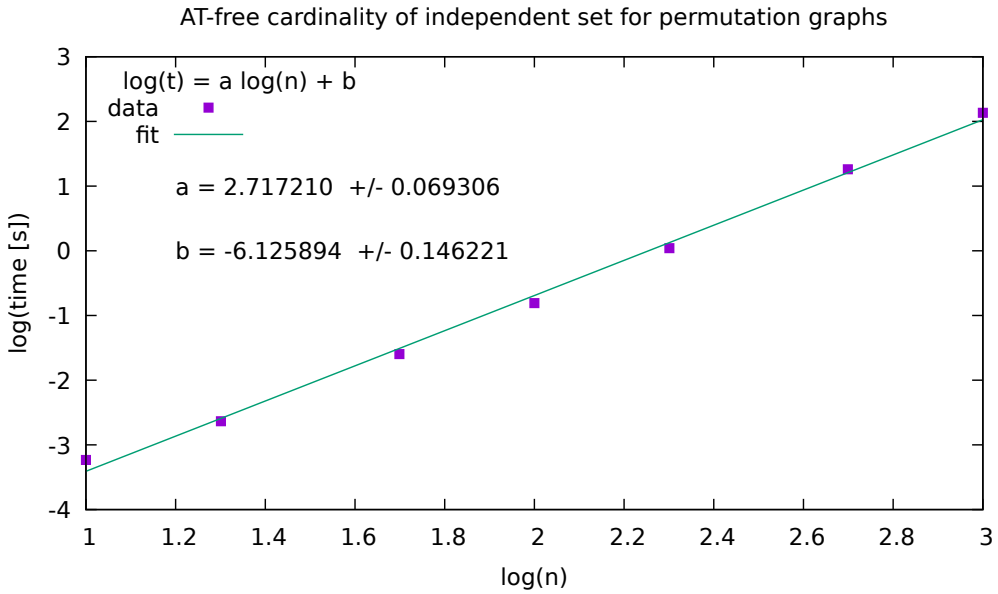
W celu znalezienia praktycznej wydajności algorytmu wyznaczania najmniejszego zbioru dominującego, zmierzono czasy działania algorytmu dla wybranych klas grafów bez AT. Pomiary zostały wykonane dla grafów permutacji (grafy dwudzielne pełne, grafy typu drabina) oraz grafów przedziałowych (graf ścieżka P_n, 3-drzewo, k-drzewo). Dla dwudzielnych grafów permutacji i grafów k-drzewo czas potrzebny do znalezienia najmniejszego zbioru dominującego jest dużo większy niż dla pozostałych rodzin grafów. Jest to spowodowane większą liczbą wierzchołków na poziomach BFS w tych grafach,



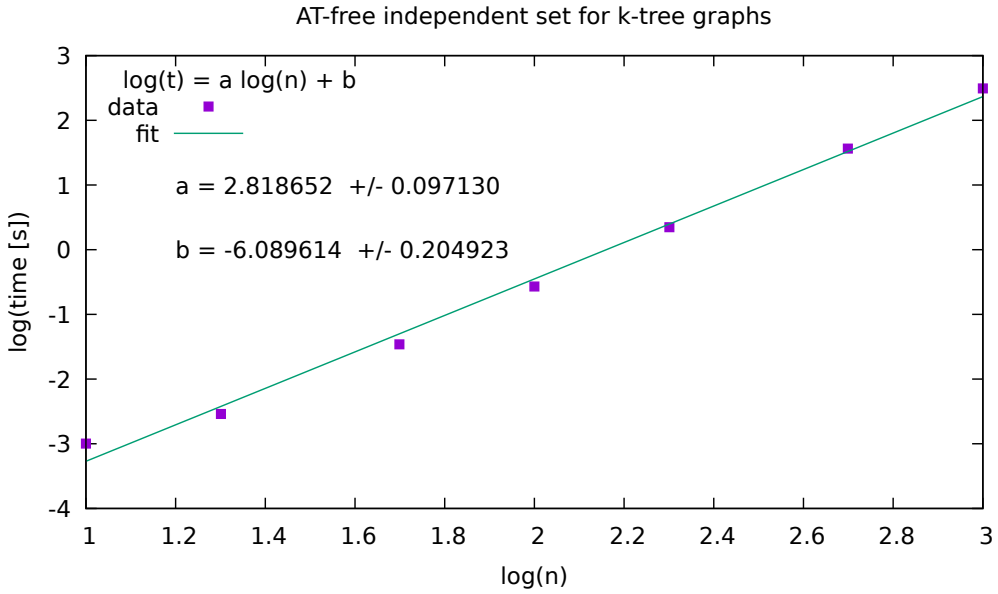
Rysunek A.1. Wyniki pomiarów algorytmu znajdującego trójki asteroidalne w grafie. Współczynnik dopasowania $a = 2.667(72)$ potwierdza teoretyczną złożoność obliczeniową $O(n^3)$.



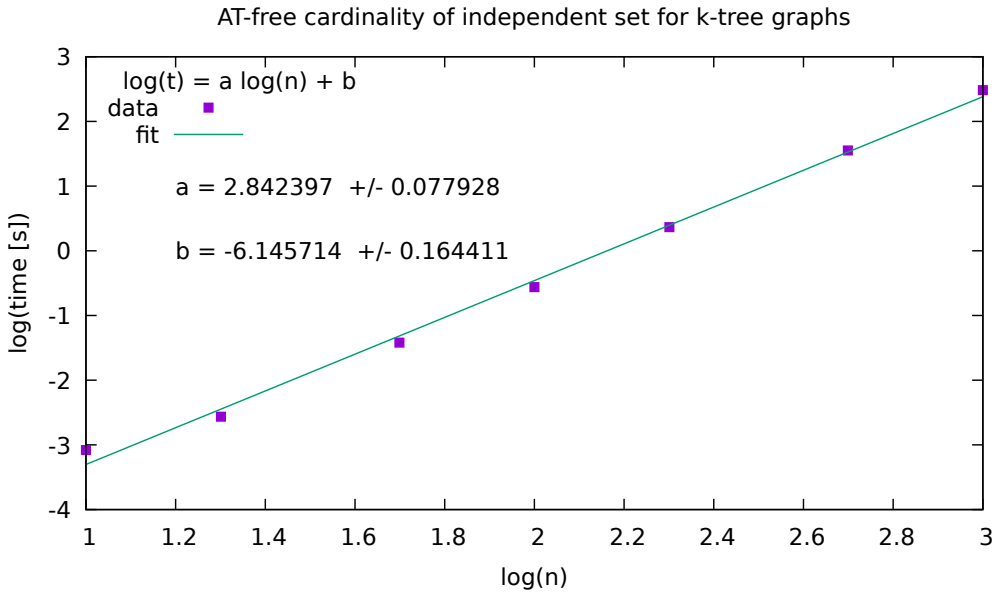
Rysunek A.2. Wyniki pomiarów algorytmu znajdującego największy zbiór niezależny w grafie permutacji. Współczynnik dopasowania $a = 2.720(65)$, co jest mniejsze od teoretycznej złożoności obliczeniowej $O(n^4)$.



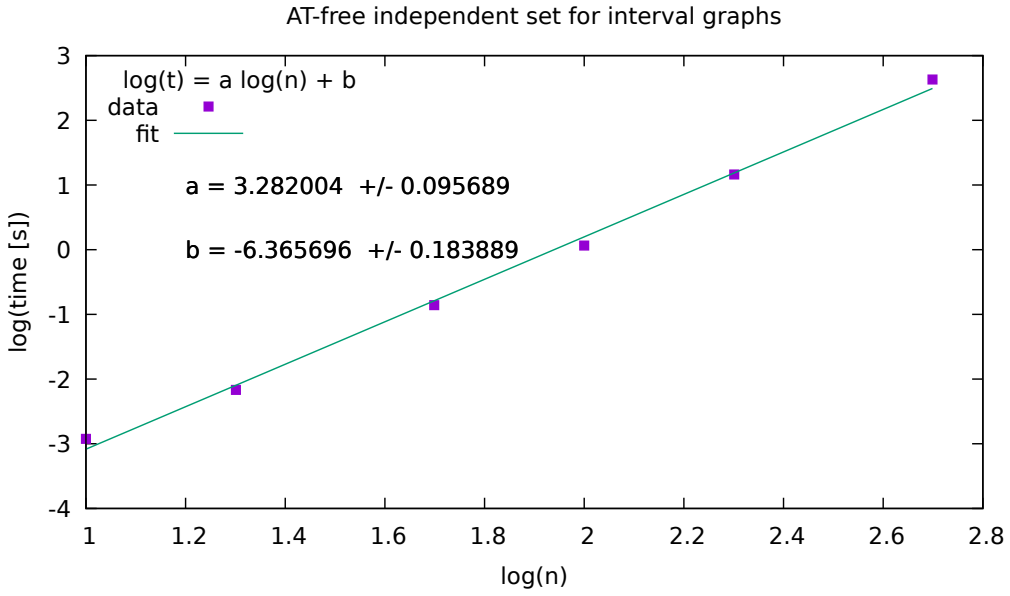
Rysunek A.3. Wyniki pomiarów algorytmu znajdującego liczność największego zbioru niezależnego w grafie permutacji. Współczynnik dopasowania $a = 2.717(70)$, co jest mniejsze od teoretycznej złożoności obliczeniowej $O(n^4)$.



Rysunek A.4. Wyniki pomiarów algorytmu znajdującego największy zbiór niezależny w grafie k-drzewo. Współczynnik dopasowania $a = 2.819(97)$, co jest mniejsze od teoretycznej złożoności obliczeniowej $O(n^4)$.

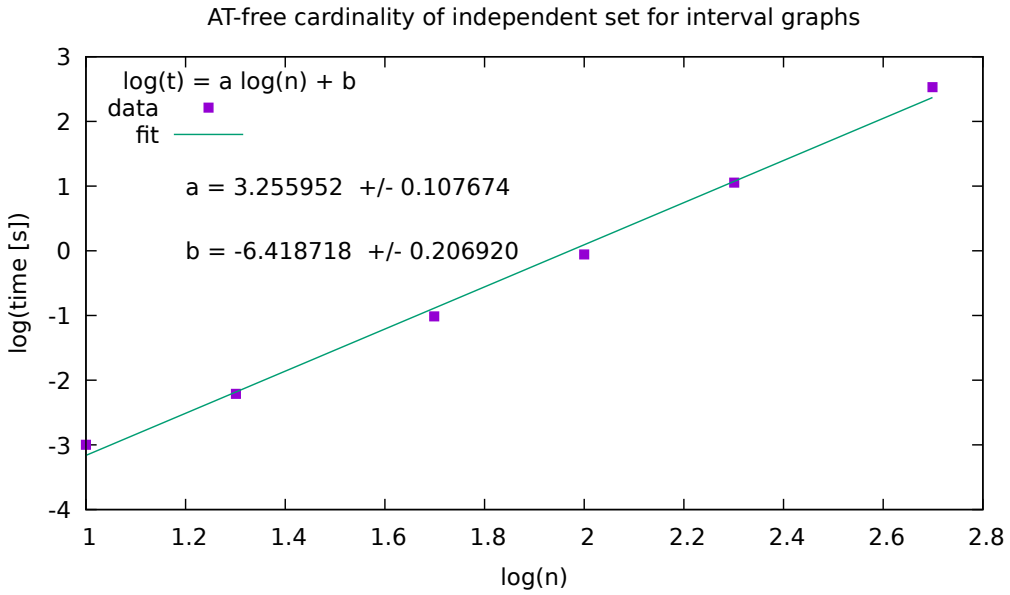


Rysunek A.5. Wyniki pomiarów algorytmu znajdującego liczność największego zbioru niezależnego w grafie k-drzewo. Współczynnik dopasowania $a = 2.717(69)$, co jest mniejsze od teoretycznej złożoności obliczeniowej $O(n^4)$.

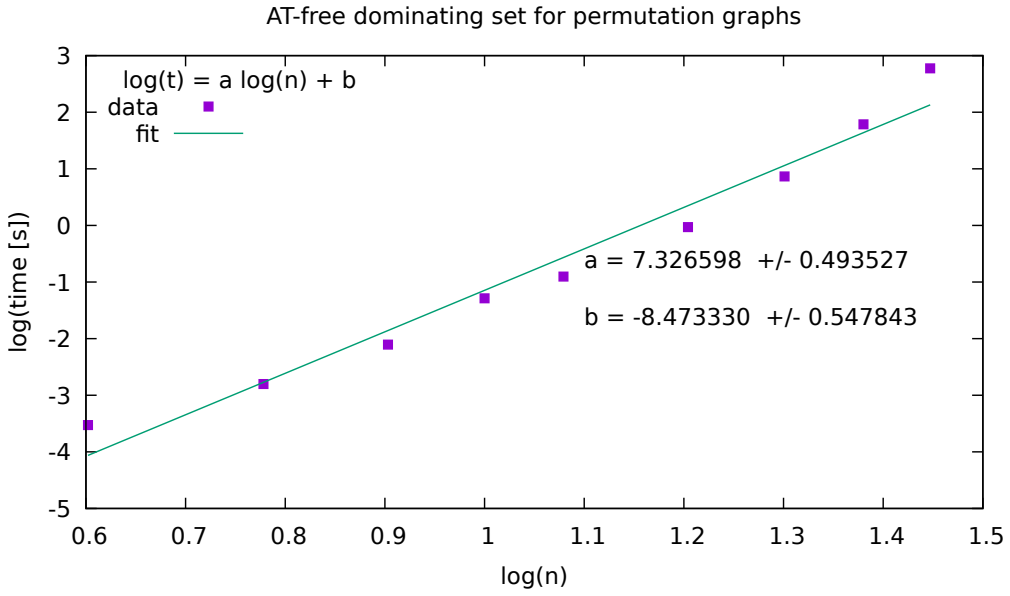


Rysunek A.6. Wyniki pomiarów algorytmu znajdującego największy zbiór niezależny w grafie ścieżka P_n. Współczynnik dopasowania $a = 3.282(96)$, co potwierdza teoretyczną złożoność obliczeniową $O(n^4)$.

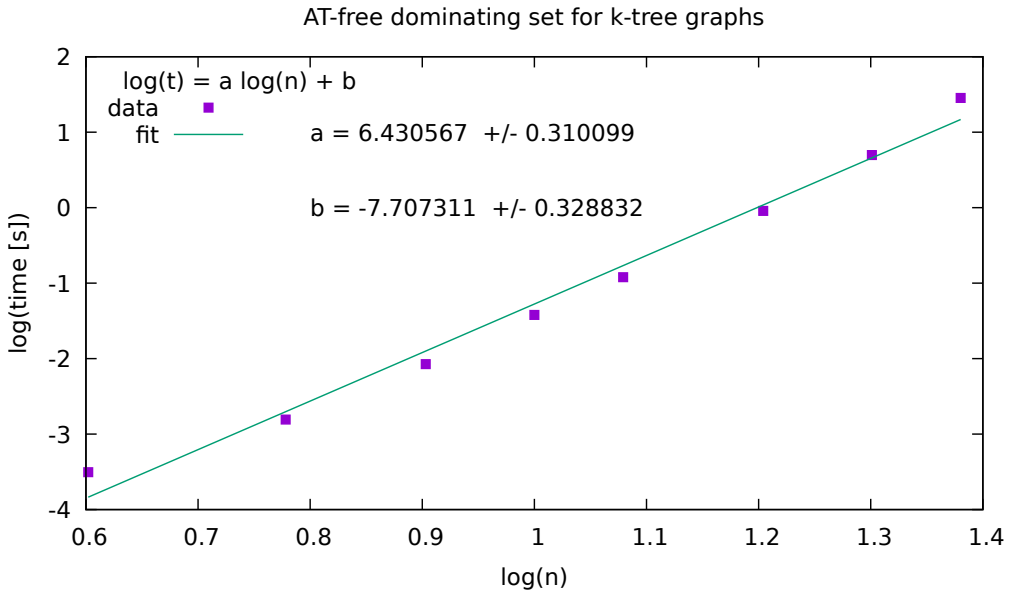
co daje większą liczbę podzbiorów do sprawdzenia. Wyniki dla grafów dwudzielnych pełnych i grafów k-drzewo pokrywają się z teoretyczną złożonością $O(n^7)$. Natomiast dla pozostałych wykorzystanych grafów czas obliczeń jest mniejszy od teoretycznej złożoności i skaluje się jak $O(n^3)$.



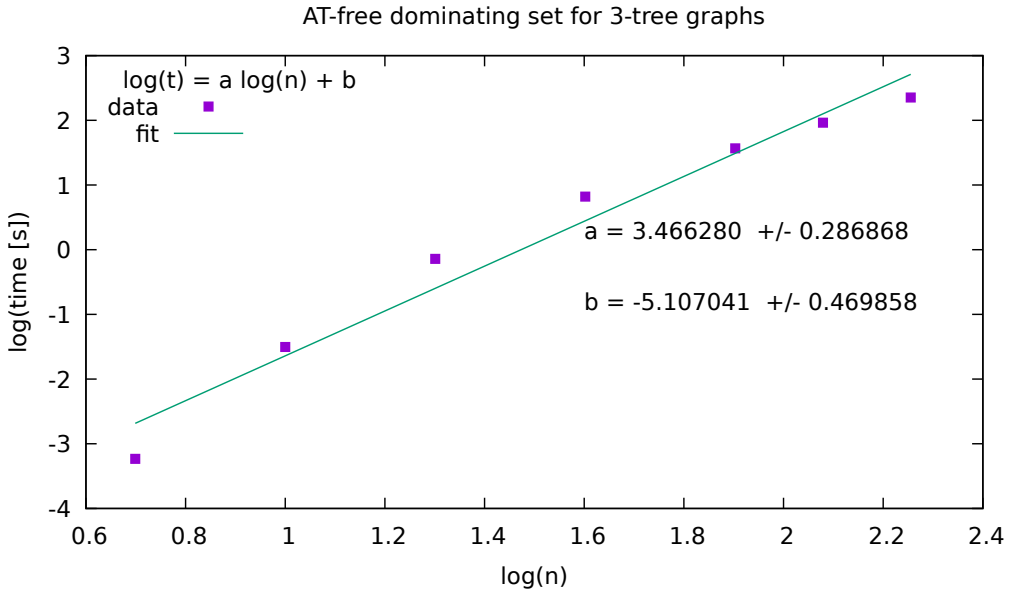
Rysunek A.7. Wyniki pomiarów algorytmu znajdującego liczność największego zbioru niezależnego w grafie ścieżka P_n. Współczynnik dopasowania $a = 3.26(11)$, co potwierdza teoretyczną złożoność obliczeniową $O(n^4)$.



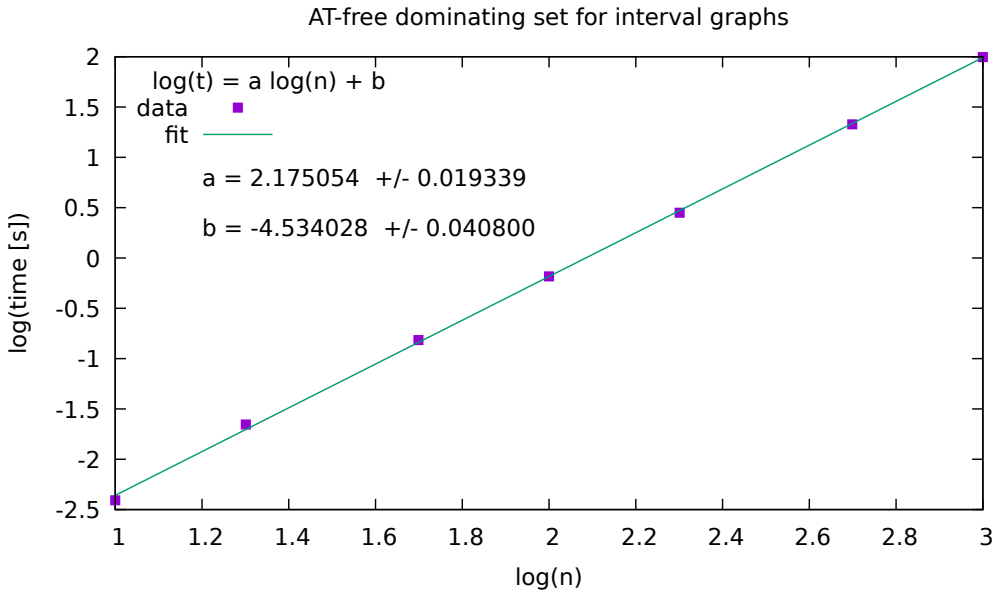
Rysunek A.8. Wyniki pomiarów algorytmu znajdującego liczność najmniejszego zbioru dominującego w grafie permutacji. Współczynnik dopasowania $a = 7.33(49)$, co potwierdza teoretyczną złożoność obliczeniową $O(n^7)$.



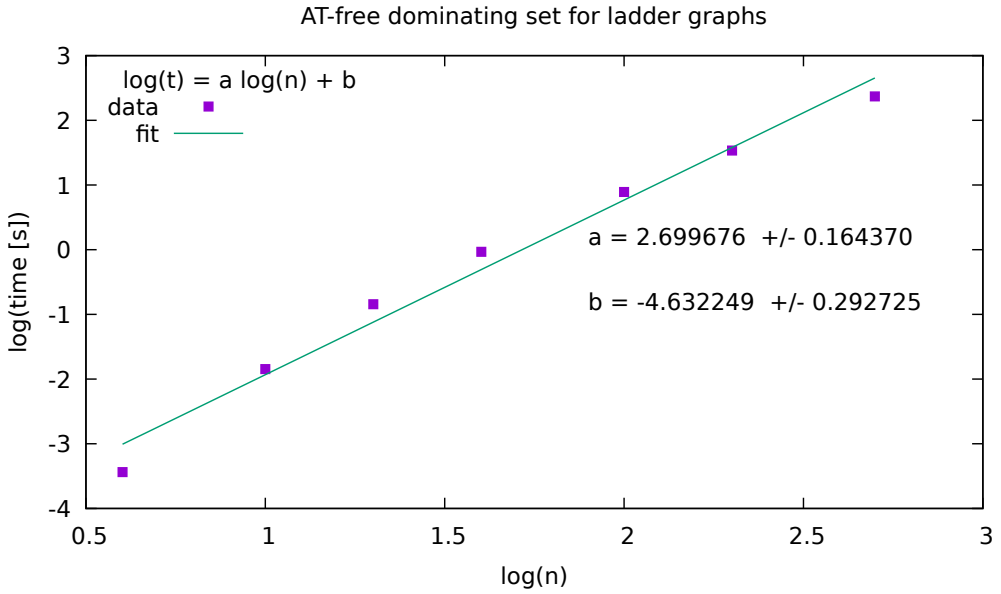
Rysunek A.9. Wyniki pomiarów algorytmu znajdującego liczność najmniejszego zbioru dominującego w grafie k-drzewo. Współczynnik dopasowania $a = 6.43(31)$, co potwierdza teoretyczną złożoność obliczeniową $O(n^7)$.



Rysunek A.10. Wyniki pomiarów algorytmu znajdującego liczność najmniejszego zbioru dominującego w grafie 3-drzewo. Współczynnik dopasowania $a = 3.47(29)$, co jest mniejsze od teoretycznej złożoności obliczeniowej $O(n^7)$.



Rysunek A.11. Wyniki pomiarów algorytmu znajdującego liczność najmniejszego zbioru dominującego w grafie przedziałowym ścieżka P_n. Współczynnik dopasowania $a = 2.18(2)$, co jest mniejsze od teoretycznej złożoności obliczeniowej $O(n^7)$.



Rysunek A.12. Wyniki pomiarów algorytmu znajdującego liczność najmniejszego zbioru dominującego w grafie drabinowym. Współczynnik dopasowania $a = 2.70(17)$, co jest mniejsze od teoretycznej złożoności obliczeniowej $O(n^7)$.

Bibliografia

[1] Information System on Graph Classes and their Inclusions, Graphclass: AT-free, 2024, `https://www.graphclasses.org/classes/gc_61.html`.

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

[3] Wikipedia, Permutation graph, 2024, `https://en.wikipedia.org/wiki/Permutation_graph`.

[4] Wikipedia, Trapezoid graph, 2024, `https://en.wikipedia.org/wiki/Trapezoid_graph`.

[5] Wikipedia, Dominating set, 2024, `https://en.wikipedia.org/wiki/Dominating_set`

[6] D. Kratsch, *Domination and total domination in asteroidal triple-free graphs*, Discrete Appl. Math. 99 No.1-3, 111-123 (2000).

[7] Wikipedia, Independent set, 2024, `https://en.wikipedia.org/wiki/Independent_set_(graph_theory)`

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

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

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

[11] H. Broersma, T. Kloks, D. Kratsch, H. Müller, *Independent sets in asteroidal triple-free graphs*, SIAM J. Discrete Math. 12, No.2, 276-287 (1999).

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

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

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

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

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

[17] Maw-Shang Chang, *Efficient algorithms for the domination problems on interval and circular-arc graphs*, SIAM J. Comput. 27(6), 1671-1694 (1998).

[18] M. Farber, J. M. Keil, *Domination in permutation graphs*, J. Algorithms 6, 309-321 (1985).

[19] C. Rhee, Y. D. Liang, S. K. Dhall, and S. Lakshmivarahan, *An $O(n+m)$-time algorithm for finding a minimum-weight dominating set in a permutation graph*, SIAM J. Comput. 25, 404-419 (1996).

[20] G. J. Chang, *Algorithmic Aspects of Domination in Graphs*. In: Pardalos, P., Du, DZ., Graham, R. (eds) Handbook of Combinatorial Optimization. Springer, New York, NY, 2013.

[21] O. H. Ibarra, Q. Zheng, *An Optimal Shortest Path Parallel Algorithm for Permutation Graphs*, Journal of Parallel and Distributed Computing 24(1), 94-99 (1995).

[22] S. Mondal, M. Pal, T. K. Pal, *An Optimal Algorithm to Solve the All-Pairs Shortest Paths Problem on Permutation Graphs*, Journal of Mathematical Modelling and Algorithms 2, 57-65 (2003).

[23] Alan P. Sprague, *O(1) query time algorithm for all pairs shortest distances on permutation graphs*, Discrete Applied Mathematics 155, 365-373 (2007).

[24] Mikhail J. Atallah, Danny Z. Chen, D. T. Lee, *An Optimal Algorithm for Shortest Paths on Weighted Interval and Circular-Arc Graphs, with Applications*, Algorithmica 14(5), 429-441 (1995).

[25] E. Kohler, *Recognizing graphs without asteroidal triples*, Journal of Discrete Algorithms 2(4), 439-452 (2004).

[26] NetworkX. Software for complex networks, 2024, `https://networkx.github.io/`.

[27] Derek G. Corneil, Richard M. Krueger, *A Unified View of Graph Searching*, SIAM Journal on Discrete Mathematics, 22(4), 1259-1276 (2008).

[28] D. G. Corneil, J. Stacho, *Vertex Ordering Characterizations of Graphs of Bounded Asteroidal Number*, Journal of Graph Theory, 78(1), 61-79 (2014).