Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Mikołaj Szymański

Nr albumu: 1178157

Zbiory dominujące w teorii grafów

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

Chciałbym bardzo podziękować Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za pomoc, poświęcony czas, ogromne zaangażowanie, rady oraz wsparcie, dzięki któremu powstała ta praca magisterska.

Streszczenie

W pracy przedstawiono analizę i implementację w języku Python wybranych algorytmów dla zbiorów dominujących w grafach nieskierowanych. Zbiór dominujący jest to taki podzbiór wierzchołków grafu, że każdy z wierzchołków grafu należy do tego zbioru lub sąsiaduje z co najmniej jednym wierzchołkiem należącym do tego zbioru. Dla grafów ogólnych problem znajdowania najmniejszego zbioru dominującego jest problemem NP-zupełnym, jednakże istnieją efektywne algorytmy dla poszczególnych klas grafów. Podobnie jest ze znajdowaniem zbioru dominującego o najmniejszej wadze w grafach z danymi wagami wierzchołków.

W części teoretycznej pracy omówione zostały podstawowe definicje związane ze zbiorami dominującymi oraz przedstawiono znane algorytmy znajdowania najmniejszych zbiorów dominujących dla różnych klas grafów. Przedstawiono przykłady zastosowania oraz pseudokody algorytmów dokładnych oraz heurystycznych dla różnych klas grafów.

W części praktycznej zaimplementowano wybrane algorytmy w języku Python oraz przeprowadzono dla każdego z nich odpowiednio testy poprawności oraz testy wydajnościowe. Zaimplementowane zostały algorytmy dla grafów typu drzewa lub lasy, grafów przedziałowych oraz grafów permutacji. Dla podanych typów grafów zaimplementowano algorytmy wyznaczające najmniejszy zbiór dominujący, zbiór dominujący o najmniejszej wadze, najmniejszy niezależny zbiór dominujący oraz niezależny zbiór dominujący o najmniejszej wadze. Wybrane algorytmy zaimplementowano w kilku wersjach.

Poprawność wyników każdego z algorytmów została przetestowana, a wydajność i złożoność obliczeniowa została oszacowana, sprawdzona eksperymentalnie i opisana.

Słowa kluczowe: grafy, grafy ważone, zbiór dominujący, zbiór niezależny, drzewa, grafy przedziałowe, grafy permutacji, programowanie dynamiczne

English title: Dominating sets in graph theory

Abstract

This paper presents Python implementation and an analysis of selected algorithms for dominating sets in undirected graphs. A dominating set is a subset of graph vertices such that each vertex in the graph belongs to this set or is adjacent to at least one vertex from this set. For general graphs, the problem of finding a minimum dominating set is an NP-complete problem, but there are efficient algorithms for specific graph classes. The same is true for the problem of finding a minimum weight dominating set in graphs with vertex weights.

The theoretical part of this paper discusses the basic definitions related to dominating sets and it presents algorithms for finding the minimum dominating sets for different graphs classes. Examplary applications and pseudocodes of exact and heuristic algorithms for different graph classes are presented.

In the practical part, selected algorithms were implemented in Python and for each of them correctness and performance tests were performed. Algorithms were implemented for trees or forest graphs, interval graphs and permutation graphs. For each graph type, algorithms were implemented to find a minimum dominating set, a minimum weight dominating set, a minimum independent dominating set and minimum weight independent dominating set. Some of the implemented algorithms have been developed in several versions.

The correctness of the results of each algorithm has been tested and the performance and computational complexity have been estimated, experimentally verified and described.

Keywords: graphs, weighted graphs, dominating set, independent set, trees, interval graphs, permutation graphs, dynamic programming

Spis treści

Spis tabel

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy są zbiory dominujące (ang. *dominating sets*) w grafach nieskierowanych [1]. Są to takie zbiory $D \subseteq V$ w grafie $G = (V, E)$, że każdy wierzchołek ze zbioru V należy do zbioru D albo sąsiaduje z co najmniej jednym wierzchołkiem należącym do zbioru D.

Zbiory dominujące są jednym z podstawowych pojęć w teorii grafów. Możemy wyróżnić kilka rodzajów zbiorów dominujących, które zostaną wymienione w kolejnym rozdziale 2, lecz najważniejsze z nich to klasyczne zbiory dominujące, najmniejsze zbiory dominujące, minimalne zbiory dominujące, oraz całkowite zbiory dominujące [2].

Zbiory dominujące znajdują praktyczne zastosowanie w wielu dziedzinach. Problem znajdowania najmniejszego zbioru dominującego jest bardzo często wykorzystywany w problemach analizy sieci społecznych, lokalizacyjnych - przykładowo do wyznaczenia optymalnej trasy autobusów szkolnych w mieście, rozmieszczenia drukarek w biurze [2], projektowania sieci bezprzewodowych, tak aby ograniczyć liczbę urządzeń potrzebnych do pokrycia całego obszaru [3], czy też znalezienia najmniejszej liczby obiektów, np. remiz straży pożarnych, które będą w stanie obsługiwać wszystkie miejscowości w powiecie [4]. Kolejnymi przypadkami użycia są modelowanie i analiza sieci transportowych oraz logistycznych [5], czy też algorytmy aproksymacyjne, gdzie wykorzystuje się je do szybkiego znajdowania rozwiązań *wystarczająco dobrych*, choć niekoniecznie najlepszych [6]. W obszarze biologii molekularnej zbiory dominujące pomagają w identyfikacji minimalnego zestawu kluczowych białek lub genów, które kontrolują cały system biologiczny [7]. Następnym przykładem zastosowania zbiorów dominujących jest tworzenie modeli rozprzestrzeniania się chorób zakaźnych w dużych sieciach miejskich, które wyznaczą zestaw lokalizacji jakie należy monitorować w celu wczesnego wykrycia epidemii i ograniczenia jej zasięgu [8].

Najczęściej w wymienionych powyżej praktycznych zastosowaniach zagadnienie sprowadza się do zbadania problemu znalezienia najmniejszego zbioru dominującego w grafie, a jest to problem NP-zupełny [9].

1.1. Cele pracy

Głównym celem pracy jest przedstawienie problemu zbiorów dominujących w grafach nieskierowanych. Zaprezentowane zostaną najważniejsze wyniki teoretyczne i wybrane algorytmy, które zostaną zaimplementowane w języku Python [10]. Python pozwala na zapisanie kodu programu w czytelnej formie bliskiej pseudokodowi z artykułów naukowych. Z drugiej strony, pełna implementacja danego algorytmu umożliwia praktyczne rozwiązywanie problemów średniej wielkości. Implementacje powstałe w ramach niniejszej

pracy uzupełnią bibliotekę algorytmów grafowych graphtheory [11], rozwijaną na Wydziale Fizyki, Astronomii i Informatyki Stosowanej Uniwersytetu Jagiellońskiego w Krakowie.

1.2. Struktura pracy

W niniejszej pracy zastosowano podział na cztery rozdziały oraz jeden moduł dodatkowy. Rozdział 1 zawiera krótkie wprowadzenie pokazujące cel oraz strukturę pracy. W rozdziale 2 przedstawiono podstawowe definicje związane z grafami, rodzaje zbiorów dominujących w grafach i sposoby znalezienia najmniejszych zbiorów dominujących dla różnych rodzajów grafów. Kolejny rozdział 3 zawiera implementacje i opisy algorytmów wykorzystanych do rozwoju biblioteki graphtheory wspomnianej w poprzednim akapicie. W rozdziale 4 znajduje się podsumowanie pracy zawierające wnioski końcowe. W dodatku A znajdują się testy algorytmów przeprowadzone dla implementacji przedstawionych w tej pracy.

2. Teoria grafów

W tym rozdziale przedstawione zostaną podstawowe definicje związane z grafami, rodzaje zbiorów dominujących i sposoby wyznaczania najmniejszych zbiorów dominujących dla różnych rodzin grafów.

2.1. Podstawowe definicje

Definicja: Graf prosty $G = (V, E)$ jest uporządkowaną parą składającą się z niepustego zbioru wierzchołów $V(G)$ i zbioru krawędzi $E(G)$, gdzie każda krawędź jest parą różnych wierzchołków [12]. Ilustrując grafy powszechnie przyjmuje się, że wierzchołki są reprezentowane przez punkty na płaszczyźnie, a krawędzie przez linie lub łuki łączące te punkty. Liczbę wierzchołków w grafie zwyczajowo oznacza się przez $n = |V(G)|$, a liczbę krawędzi przez $m = |E(G)|$.

Definicja: Graf nieskierowany prosty (ang. *undirected simple graph*) jest uporządkowaną parą składająca się z niepustego zbioru wierzchołków $V(G)$ i zbioru krawędzi $E(G)$, gdzie każda krawędź jest nieuporządkowaną parą (zbiorem) różnych wierzchołków, $E(G) \subseteq \{\{u, v\} \mid u, v \in V(G), u \neq v\}$ [13].

Definicja: Ścieżka w grafie G jest to ciąg wierzchołków, taki że każdy kolejny wierzchołek jest połączony krawędzią z poprzedzającym go wierzchołkiem [12]. Jeśli w ścieżce wierzchołki nie powtarzają się, to taką ścieżkę nazywamy ścieżką prostą. Jeżeli w ścieżce pierwszy i ostatni wierzchołek są takie same, to taką ścieżkę nazywamy cyklem. Cykl prosty nie zawiera powtarzających się wierzchołków, z wyjątkiem ostatniego.

Definicja: Graf spójny (ang. *connected graph*) jest to graf, w którym dla każdej pary wierzchołków istnieje ścieżka łącząca te wierzchołki [13].

Definicja: Stopień wierzchołka v w grafie G (ang. *degree of a vertex*), oznaczenie $\deg(v)$, jest liczbą krawędzi incydentnych z wierzchołkiem v [12], czyli takich, które mają swój początek lub koniec w wierzchołku v. Jeśli krawędź jest pętlą, czyli jej początek i koniec jest w tym samym wierzchołku, to taka krawędź jest liczona dwa razy w stopniu wierzchołka. Zauważmy, że pętle nie występują w grafach prostych.

Definicja: Graf regularny (ang. *regular graph*) jest to graf nieskierowany, w którym każdy wierzchołek ma taki sam stopień. Jeśli każdy wierzchołek w grafie G ma stopień r, to graf nazywamy grafem regularnym stopnia r, w skrócie grafem r-regularnym [12].

Definicja: Graf cykliczny (ang. *cyclic graph*) jest grafem spójnym i regularnym stopnia 2 [12].

Definicja: Graf acykliczny (ang. *acyclic graph*) jest grafem, który nie zawiera w sobie żadnych cykli [14].

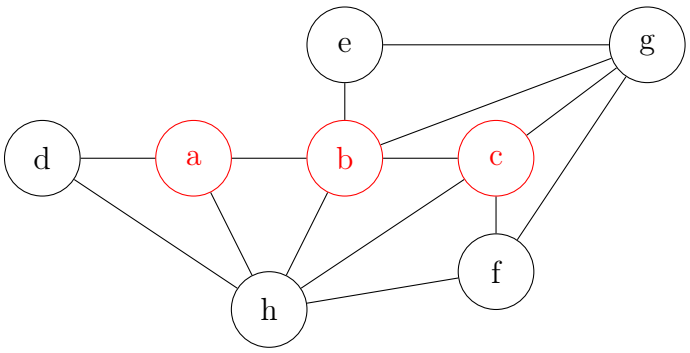
Definicja: Graf jest drzewem (ang. *tree*), jeśli jest acyklicznym grafem spójnym [15].

Definicja: Zbiór niezależny (ang. *independent set*) to taki podzbiór $S \subseteq V$ w grafie $G = (V, E)$, że każde dwa wierzchołki w S nie są połączone krawędzią. Maksymalny zbiór niezależny nie jest podzbiorem większego zbioru niezależnego. Największy zbiór niezależny to zbiór niezależny o największej liczności.

2.2. Rodzaje zbiorów dominujących

2.2.1. Klasyczny zbiór dominujący

Klasyczny zbiór dominujący (ang. *dominating set*) to taki podzbiór $D \subseteq V$ w grafie $G = (V, E)$, że każdy wierzchołek ze zbioru V należy do zbioru D albo sąsiaduje z co najmniej jednym wierzchołkiem należącym do zbioru D [1].



Rysunek 2.1. Graf z zaznaczonym zbiorem dominującym $D = \{a, b, c\}$ (czerwone wierzchołki). Każdy wierzchołek spoza zbioru D sąsiaduje z co najmniej jednym wierzchołkiem ze zbioru D.

Najmniejszy zbiór dominujący (ang. *minimum dominating set*) to taki zbiór $D \subseteq V$ w grafie $G = (V, E)$, który spełnia warunek dominacji i ma najmniejszą możliwą liczność spośród wszystkich zbiorów dominujących w danym grafie [16].

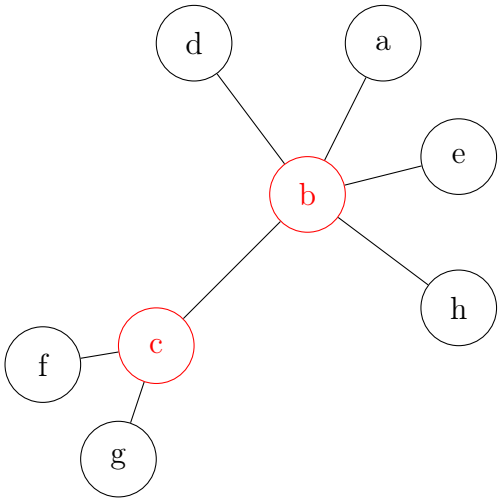
Minimalny zbiór dominujący (ang. *minimal dominating set*) to taki zbiór dominujący, że usunięcie z niego dowolnego wierzchołka powoduje, że warunek dominacji nie jest spełniony. Każdy najmniejszy zbiór dominujący jest minimalnym zbiorem dominującym, ale nie na odwrót.

Warunek dominacji: Każdy wierzchołek ze zbioru V należy do zbioru D albo sąsiaduje z co najmniej jednym wierzchołkiem należącym do zbioru D.

Liczba dominacji: Liczność najmniejszego zbioru dominującego w grafie G nazywa się liczbą dominacji $\gamma(G)$.

Stwierdzenie: Każdy maksymalny zbiór niezależny jest minimalnym zbiorem dominującym. Stwierdzenie odwrotne nie jest prawdziwe.

Grafy z dodatnimi wagami wierzchołków: W grafach z dodatnimi wagami wierzchołków definiuje się wagę zbioru dominującego jako sumę wag wierzchołków należących do zbioru dominującego. W tej sytuacji można poszukiwać zbioru dominującego o najmniejszej wadze. Zauważmy, że jeżeli wszystkie wagi wierzchołków są równe, to otrzymujemy zwykły problem bez wag. Problem z wagami jest więc co najmniej tak trudny, jak problem bez wag.



Rysunek 2.2. Graf z zaznaczonym najmniejszym zbiorem dominującym $D = \{b, c\}$ (czerwone wierzchołki), tak więc $\gamma(G) = 2$. Zbiór D jest także minimalnym zbiorem dominującym.

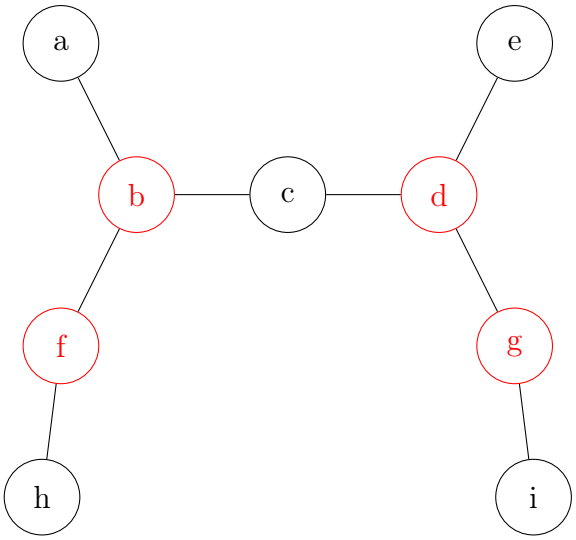
2.2.2. Niezależny zbiór dominujący

Niezależny zbiór dominujący (ang. *independent dominating set*) to zbiór dominujący, który jest jednocześnie zbiorem niezależnym. Berge pokazał, że niezależny zbiór dominujący to jest to samo co maksymalny zbiór niezależny. Czyli wystarczy znaleźć najmniejszy maksymalny zbiór niezależny, żeby dostać najmniejszy niezależny zbiór dominujący.

Zbiory niezależne łatwo znajduje się w grafach pemutacji jako rosnące podsekwencje elementów permutacji.

2.2.3. Całkowity zbiór dominujący

Całkowity zbiór dominujący (ang. *total dominating set*) to taki zbiór $D \subseteq V$ w grafie $G = (V, E)$, że każdy wierzchołek ze zbioru V należy do zbioru D albo sąsiaduje z co najmniej jednym wierzchołkiem należącym do zbioru D [2].

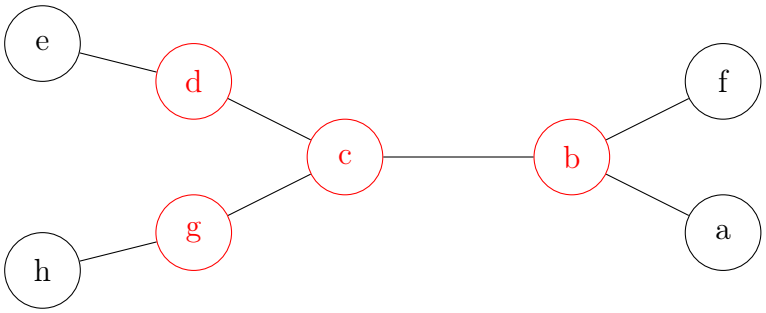


Rysunek 2.3. Graf z zaznaczonym całkowitym zbiorem dominującym $D = \{b, d, f, g\}$ (czerwone wierzchołki).

2.2.4. Spójny zbiór dominujący

Spójny zbiór dominujący (ang. *connected dominating set*) to taki zbiór $D \subseteq V$ w grafie $G = (V, E)$, który spełnia dwa warunki [17]:

— D jest zbiorem dominującym.
— podgraf utworzony przez wierzchołki zbioru D (tzn. podgraf indukowany przez D) jest spójny, czyli każde dwa wierzchołki w D można połączyć ścieżką, która składa się wyłącznie z wierzchołków należących do D.



Rysunek 2.4. Graf z zaznaczonym spójnym zbiorem dominującym $D = \{b, c, d, g\}$ (czerwone wierzchołki). Podgraf indukowany przez D jest spójny.

Warto się zastanowić nad algorytmami wyznaczania spójnego najmniejszego zbioru dominującego. W przypadku drzew wystarczy usunąć wszystkie liście (wierzchołki stopnia 1), a pozostałe wierzchołki utworzą optymalne rozwiązanie. Osobno trzeba potraktować izolowany wierzchołek i izolowaną krawędź K_2.

Podobny pomysł można zastosować w celu uzyskania heurystyki działającej dla ogólnych grafów [18]. Chcemy znaleźć drzewo rozpinające grafu, a potem wykorzystać algorytm dla drzewa. Zależy nam na znalezieniu drzewa

rozpinającego z jak największą liczbą liści, więc drzewo BFS będzie lepsze niż drzewo DFS. Algorytm można zapisać w trzech krokach:
(1) Wybrać wierzchołek, który będzie korzeniem drzewa BFS. Kandydatem może być wierzchołek o największym stopniu w grafie.
(2) Wyznaczyć drzewo BFS w czasie $O(n + m)$.
(3) Z drzewa BFS odrzucić wszystkie liście w czasie $O(n)$.

2.2.5. k-krotny zbiór dominujący

k-krotny zbiór dominujący (ang. *k-tuple domination set*) to rozszerzenie zbioru dominującego, gdzie stopień pokrycia każdego wierzchołka wynosi k. Oznacza to tyle, że w zwykłym zbiorze dominującym każdy wierzchołek musi mieć co najmniej jednego sąsiada należącego do zbioru dominującego, a w k-krotnym wariancie każdy wierzchołek musi mieć co najmniej k sąsiadów należących do tego zbioru [19].

— Dla $k = 1$ k-krotny zbiór dominujący jest równoważny klasycznemu zbiorowi dominującemu (każdy wierzchołek musi mieć sąsiada w zbiorze dominującym).
— Dla $k > 1$ każdy wierzchołek w grafie musi posiadać co najmniej k sąsiadów należących do zbioru dominującego.

2.2.6. R-krotny zbiór dominujący

Zbiór dominujący o zadanym promieniu (ang. *R-distance dominating set*) to rozszerzenie koncepcji zbioru dominującego, w którym każdy wierzchołek musi znajdować się w odległości maksymalnie R od co najmniej jednego wierzchołka należącego do zbioru dominującego [20].

— Dla $R = 1$ R-krotny zbiór dominujący jest równoważny klasycznemu zbiorowi dominującemu (każdy wierzchołek musi mieć sąsiada w zbiorze dominującym).
— Dla $R > 1$ wymóg dominacji rozciąga się na dalsze sąsiedztwo – dany wierzchołek może być zdominowany przez wierzchołek, który jest od niego oddalony o 2, 3 lub więcej krawędzi, w zależności od wartości R.

2.2.7. Ograniczone zbiory dominujące

Ograniczony zbiór dominujący (ang. *restrained dominating set*) to wariant zbioru dominującego, w którym oprócz spełnienia warunku zbioru dominującego spełnione musi być jeszcze pewne ograniczenie, a mianowicie każdy wierzchołek spoza zbioru dominującego musi mieć również co najmniej jednego sąsiada spoza tego zbioru [21]. Jako przykład zastosowania podaje się więźniów i strażników. Strażnicy tworzą zbiór dominujący, przy czym krawędzie reprezentują możliwość obserwowania więźnia przez strażnika. Dodatkowo, w celu zapewniania ochrony praw więźniów, każdy więzień powinien być obserwowany przez innego więźnia. Ze względu na koszty chcemy minimalizować liczbę strażników.

Całkowity ograniczony zbiór dominujący (ang. *total restrained dominating set*) to ograniczony zbiór dominujący, w którym dodatkowo wymaga

się, aby każdy wierzchołek należący do zbioru dominującego miał sąsiada w zbiorze dominującym.

Spójny ograniczony zbiór dominujący (ang. *connected restrained dominating set*) to ograniczony zbiór dominujący, w którym dodatkowo wymaga się, aby zbiór dominujący tworzył spójny podgraf.

2.3. Znajdowanie najmniejszego zbioru dominującego

W tym rozdziale opiszemy różne algorytmy wyznaczania najmniejszego zbioru dominującego obecne w bibliotece graphtheory oraz zaimplementowane w ramach tej pracy. Algorytmy na ogół mają postać klasy, gdzie konstruktor klasy dostaje jako argument graf nieskierowany. Metoda run uruchamia obliczenia, a rozwiązanie jest zapisywane w atrybutach dominating_set i cardinality.

2.3.1. Ogólne grafy

Algorytmy dokładne wyznaczające najmniejszy zbiór dominujący w ogólnych grafach mają wykładniczą złożoność obliczeniową. W bibliotece graphtheory mamy dwa algorytmy tego rodzaju. Algorytm w klasie BacktrackingDominatingSet to algorytm z powrotami, w którym po kolei zalicza się lub wyklucza wierzchołki ze zbioru dominującego. Ogólnie jest 2^n konfiguracji wierzchołków grafu do sprawdzenia, ale algorytm przerywa badanie danego poddrzewa możliwych rozwiązań, jeżeli liczba wierzchołków zaliczona do zbioru dominującego stanie się równa liczbie wierzchołków w najlepszym do tej pory znalezionym rozwiązaniu. Wtedy algorytm wycofuje się do poprzedniego kroku rekurencyjnego i zaczyna sprawdzać inne możliwości.

Algorytm w klasie HybridDominatingSet jest modyfikacją poprzedniego algorytmu dokładnego. W pierwszym etapie wykorzystywana jest heurystyka z klasy LargestFirstDominatingSet do szybkiego znalezienia małego zbioru dominującego, jako kandydata na najlepsze rozwiązanie. W drugim etapie czynności są takie jak w klasie BacktrackingDominatingSet. Testy pokazują pewne przyspieszenie w stosunku do algorytmu z klasy BacktrackingDominatingSet.

```
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.dominatingsets.dsetbt import BacktrackingDominatingSet
from graphtheory.dominatingsets.dsethb import HybridDominatingSet

G = Graph()
# Add nodes and edges here.
#algorithm = BacktrackingDominatingSet(G)
algorithm = HybridDominatingSet(G)
algorithm.run()
print(algorithm.dominating_set)
print(algorithm.cardinality)
```

Algorytmy heurystyczne działają szybko w czasie liniowym $O(n + m)$, ale znalezione rozwiązanie może nie być optymalne. W bibliotece graphtheory mamy trzy algorytmy tego rodzaju. W klasie UnorderedSequentialDominatingSet

wierzchołki są zaliczane do zbioru dominującego w kolejności wyznaczonej przez implementację. W klasie RandomSequentialDominatingSet wierzchołki wybierane są losowo. W klasie LargestFirstDominatingSet na każdym kroku wybierany jest wierzchołek o największym stopniu w bieżącym podgrafie. Wierzchołek wybrany do zbioru dominującego jest usuwany z grafu wraz ze swoimi sąsiadami. Zastosowano strukturę bukietów, aby dynamicznie uaktualniać stopnie wierzchołków w malejącym podgrafie.

```
from graphtheory.structures.edges import Edge
from graphtheory.structures.graphs import Graph
from graphtheory.dominatingsets.isetus import UnorderedSequentialDominatingSet
from graphtheory.dominatingsets.isetrs import RandomSequentialDominatingSet
from graphtheory.dominatingsets.isetll import LargestFirstDominatingSet

G = Graph()
# Add nodes and edges here.
# algorithm = UnorderedSequentialDominatingSet(G)
# algorithm = RandomSequentialDominatingSet(G)
algorithm = LargestFistDominatingSet(G)
algorithm.run()
print(algorithm.dominating_set)
print(algorithm.cardinality)
```

2.3.2. Drzewa

Najmniejszy zbiór dominujący dla drzew można wyznaczyć w czasie $O(n)$ na kilka sposobów. Najprostszy sposób polega na odrywaniu liści przy jednoczesnym zaliczaniu rodziców liści do zbioru dominującego. Dwa warianty implementacji tego algorytmu opisano w pracach magisterskich Aleksandra Krawczyka [22] i Małgorzaty Olak [23].

Tutaj podamy pseudokod innego algorytmu, który w czasie $O(n)$ wyznacza jednocześnie najmniejszy zbiór dominujący i największy zbiór 2-niezależny (ang. *maximum 2-stable set*), czyli zbiór niezależny z odległościami pomiędzy wierzchołkami większymi niż dwa. Zauważmy, że w zwykłych zbiorach niezależnych wystarczy zachować odległość między wierzchołkami większą niż jeden. Pseudokod pochodzi z książki [24], a zastosowane podejście jest nazywane *primal-dual approach*. Implementacja algorytmu w języku Python zostanie przedstawiona w rozdziale 3. Na wejście algorytmu należy podać graf (drzewo) oraz listę wierzchołków grafu, są to liście odrywane z malejącego podgrafu indukowanego. W praktyce ciąg liści można wyznaczyć osobno innym algorytmem lub wyznaczać na bieżąco.

```
# Primal-Dual Approach for Trees wg [2013 Chen] p.241.
Input: A tree T with a tree ordering (PEO) [v1,v2,...vn].
Output: A minimum dominating set D and a maximum 2-stable set S of T.
Algorithm DomTreePD(T, PEO):
    let D be an empty set
    let S be an empty set
    for i from 1 to n do
        let vj be the parent od vi (assume vj=vn for vi=vn)
        if the intersection of N[vi] and D is empty then
            add vj to D
            add vi to S
```

```
    return D and S
```

Inny algorytm znajdowania najmniejszego zbioru dominującego dla drzew wykorzystuje strukturę rekurencyjną drzew (metoda Borie). W korzeniu łączy się zwykle kilka poddrzew, więc w korzeniu łączy się częściowe rozwiązania dla poddrzew. Analogicznie w każdym poddrzewie łączy się rozwiązania znalezione dla jeszcze mniejszych poddrzew. Algorytm został szczegółowo opisany w pracy magisterskiej Aleksandra Krawczyka [22].

```
from graphtheory.forests.treedset import BorieDominatingSet
from graphtheory.forests.treedset import TreeDominatingSet

algorithm = BorieDominatingSet(G)    # Borie method
algorithm.run()
print(algorithm.dominating_set)
print(algorithm.cardinality)    # the size of min dset
print(algorithm.parent)         # DFS tree as a dict

algorithm = TreeDominatingSet(G)    # removing leaves
algorithm.run()
print(algorithm.dominating_set)
print(algorithm.cardinality)    # the size of min dset
```

W roku 2018 Jou i Lin przedstawili dwa algorytmy dla drzew z dodatnimi wagami wierzchołków [25]. W czasie liniowym $O(n)$ wyznaczali najmniejszą wagę zbioru dominującego i najmniejszą wagę niezależnego zbioru dominującego. Implementacja tych algorytmów w języku Python zostanie przedstawiona w rozdziale 3.

2.3.3. Grafy szeregowo-równoległe

Definicja: Grafy szeregowo-równoległe, inaczej nazywane sp-grafami (ang. *series-parallel graphs*) są to grafy planarne (definicja w rozdziale 2.3.10) z dwoma wyróżnionymi wierzchołkami. Grafy takie możemy utworzyć rekurencyjnie za pomocą trzech operacji: równoległej, szeregowej oraz *jackknife*. Analiza sp-grafów nieskierowanych znajduje się w pracy magisterskiej Konrada Gałuszki [26].

Algorytm wyznaczający najmniejszy zbiór dominujący dla sp-grafu wykorzystuje sp-drzewo, czyli drzewo binarne opisujące jednoznacznie strukturę sp-grafu. Optymalne rozwiązanie dla końcowego sp-grafu budowane jest z rozwiązań dla mniejszych sp-grafów (sp-grafy mają strukturę rekurencyjną). Dla mniejszych sp-grafów należy rozważyć dziewięć rodzajów rozwiązań, gdzie końce sp-grafów mogą być w różny sposób zdominowane.

```
# Finding a minimum dominating set for sp-graphs.
from graphtheory.seriesparallel.spdset import SPGraphDominatingSet
from graphtheory.seriesparallel.spdset import SPTreeDominatingSet

# G is an sp-graph, T is a binary sp-tree.
algorithm = SPGraphDominatingSet(G, T)
algorithm.run()
print(algorithm.dominating_set)
print(algorithm.cardinality)
```

```
algorithm = SPTreeDominatingSet(T)
algorithm.run()
print(algorithm.dominating_set)
print(algorithm.cardinality)
```

2.3.4. Grafy przedziałowe

Definicja: Graf przedziałowy (ang. *interval graph*) to graf nieskierowany, w którym każdemu wierzchołkowi przypisany jest przedział na osi liczbowej. Dwa wierzchołki są połączone krawędzią, jeżeli odpowiednie przedziały mają niepuste przecięcie. Zwykle przyjmuje się, że przedziały są domknięte, a wszystkie końce przedziałów są różne [27]. Każdy graf przedziałowy jest grafem cięciwowym.

Przedstawiony poniżej pseudokod algorytmu opiera się na podejściu przedstawionym w książce [24] do wyznaczenia jednocześnie najmniejszego zbioru dominującego i największego zbioru 2-niezależnego (*primal-dual approach*). Graf przedziałowy jest podany na wejście algorytmu w postaci listy przedziałów (pary liczb), przy czym prawe końce przedziałów muszą być posortowane rosnąco. Implementacja algorytmu w języku Python zostanie przedstawiona w rozdziale 3.

```
# Primal-Dual Approach for Interval Graphs wg [2013 Chen] p.245.
Input: An interval model for G with intervals {Ii=[ai,bi]},
right ends of intervals (bi) are sorted ascending.
Output: A minimum dominating set D and a maximum 2-stable set S of G.
Algorithm DomIntervalGraphPD(G, {Ii}):
    let D be an empty set
    let S be an empty set
    for i from 1 to n do
        let j be the the largest index such that vj belongs to N[vi]
        (assume vj=vn for vi=vn)
        if the intersection of N[vi] and D is empty then
            add vj to D
            add vi to S
    return D and S
```

Problem znajdowania zbioru dominującego o najmniejszej wadze w grafach przedziałowych został rozwiązany w roku 1988 przez Ramalingama i Rangana [28]. Autorzy rozważali kilka odmian dominacji w grafach przedziałowych w jednolity sposób. Implementacja ich algorytmu będzie przedstawiona w rozdziale 3.

2.3.5. Grafy cięciwowe

Definicja: Graf cięciwowy (ang. *chordal graph*) to taki graf nieskierowany, w którym każdy cykl indukowany o długości większej niż 3 ma cięciwę [29]. Cięciwa to krawędź nie należąca do cyklu, ale łącząca pewne dwa wierzchołki z tego cyklu. Każdy graf cięciwowy G posiada charakterystyczną dekompozycję drzewową, zwaną drzewem klik $TD(G)$. Każdy wierzchołek (worek)

w $TD(G)$ odpowiada klice maksymalnej grafu G [30]. Zauważmy, że w przypadku grafu przedziałowego dekompozycja drzewowa jest grafem ścieżką P_k, gdzie k to liczba klik maksymalnych.

Jak udowodnili Booth i Johnson, problem wyznaczania najmniejszego zbioru dominującego w grafach cięciwowych jest NP-zupełny. Jednakże, jak udało się pokazać w pracy magisterskiej Macieja Niezabitowskiego, dzięki istnieniu dekompozycji drzewowej możliwe jest podanie algorytmu dokładnego, opartego na dynamicznym programowaniu na drzewie kliki. Algorytm taki rozważa wszystkie możliwe kombinacje 3-kolorowania wierzchołków w klice i scala rozwiązania z poddrzew, minimalizując przy tym liczbę wierzchołków w zbiorze dominującym. Złożoność obliczeniowa algorytmu jest liniowa w liczbie wierzchołków, ale wykładnicza ze względu na rozmiar największej kliki. Drzewo dekompozycji dla grafu cięciwowego trzeba wyznaczyć innym algorytmem.

```
# Finding a minimum dominating set for chordal graphs.
from graphtheory.chordality.chordaldset import ChordalDominatingSet

# Input: G is a chordal graph, T is a tree decomposition of G.
algorithm = ChordalDominatingSet(G, T)
algorithm.run()
print(algorithm.dominating_set)
print(algorithm.cardinality)
```

2.3.6. Grafy łuków na okręgu

Definicja: Graf łuków na okręgu (ang. *circular-arc graph*) to graf nieskierowany, w którym wierzchołki odpowiadają łukom na okręgu, a krawędzie łączą dwa wierzchołki tylko wtedy, gdy odpowiadające im łuki mają niepuste przecięcie [31]. Grafy te, są naturalnym rozszerzeniem grafów przedziałowych i opisują pewne procesy odbywające się cyklicznie, np. rozkład lotów samolotów na lotnisku powtarzający się co 24 godziny.

Najmniejszy zbiór dominujący w grafie łuków na okręgu można wyznaczyć w czasie liniowym algorytmem podanym przez Hsu i Tsai w roku 1991 [32]. Implementację tego algorytmu można znaleźć w pracy magisterskiej Oliwii Gil [33].

2.3.7. Grafy permutacji

Definicja: Graf permutacji (ang. *permutation graph*) to graf nieskierowany, którego wierzchołki odpowiadają odcinkom łączącym dwa rzędy uporządkowanych liczb (nazywanych również diagramami permutacyjnymi), a krawędzie występują pomiędzy parami wierzchołków wtedy i tylko wtedy, gdy odpowiadające im odcinki przecinają się. Jest to interpretacja geometryczna grafu permutacji. Grafy takie można interpretować także jako grafy przecięć odcinków reprezentujących permutację, ponieważ kiedy wierzchołki są połączone, występują w odwrotnej kolejności na górnym i dolnym rzędzie, co odpowiada przecięciu linii w diagramie permutacyjnym [34].

Inna definicja grafu permutacji mówi, że wierzchołki grafu permutacji odpowiadają elementom permutacji, a krawędzie łączą te elementy, które tworzą inwersję w permutacji. Zakodowanie grafu przez permutację daje bardzo oszczędną reprezentację, wykorzystanie pamięci jest rzędu $O(n \log n)$, gdzie n to liczba wierzchołków grafu.

W literaturze można znaleźć kilka algorytmów znajdujących najmniejszy zbiór dominujący, działających w czasie wielomianowym. Podany zostanie krótki opis podejścia opartego na programowaniu dynamicznym, działającego bez budowania grafu, bezpośrednio na permutacji P i jej odwrotności P^{-1} [35]. Element $D[i][j]$ oznacza najmniejszy zbiór dominujący dla wierzchołków leżących w permutacji na pozycjach $\{1, \ldots, i\}$, przy założeniu, że największy „prawy koniec" przykryty dotąd na dolnym rzędzie wynosi j. Dla kolejnego wierzchołka i rozważamy: brak zmiany, domknięcie największego prawego końca przez dodanie odpowiedniego wierzchołka lub dodanie samego i. Algorytm wg autorów działa w czasie $O(n^2)$, choć mamy wątpliwości co do realizacji wyznaczania kolejnych elementów $D[i][j]$ w stałym czasie.

```
Input: A permutation P of numbers from 1 to n.
Output: A minimum dominating set D of the permutation graph.
Algorithm DomPermGraph(P):
    let Q be the inverse permutation of P
    let D[1..n][0..n] be an array of empty sets
    for j from 0 to n do
        if j >= P(1) then
            set D[1][j] to {1}
        else
            set D[1][j] to {}

    def max_P(S):
        return max({P(v) : v in S})

    for i from 2 to n do
        set p to max({P(1), ..., P(i)})
        set k to Q(p)
        for j from 0 to n do
            if j < P(i) then
                set D[i][j] to D[i-1][j]
            if j = P(i) then
                if max_P(D[i-1][j]) > P(i) then
                    set D[i][j] to D[i-1][j]
                else
                    set D[i][j] to D[i-1][j] union {k}
            if j > P(i) then
                if max_P(D[i-1][j]) > P(i) then
                    set A to D[i-1][j]
                else
                    set A to D[i-1][j] union {k}
                set B to D[i-1][P(i)-1] union {i}
                # choose better: fewer vertices; tie -> larger max_P()
                if |A| < |B| then
                    set D[i][j] to A
                else if |B| < |A| then
                    set D[i][j] to B
                else if max_P(A) > max_P(B) then
                    set D[i][j] to A
```

```
                else
                    set D[i][j] to B
    return D[n][n]
```

2.3.8. Grafy kołowe

Graf kołowy (ang. *circle graph*) jest grafem nieskierowanym, którego wierzchołki odpowiadają cięciwom okręgu. Dwa wierzchołki są połączone krawędzią tylko w sytuacji, gdy odpowiednie cięciwy przecinają się. Przyjmuje się bez zmniejszania ogólności, że cięciwy nie mają wspólnych końców. Każdy graf permutacji jest grafem kołowym, ponieważ diagram permutacyjny można skleić z obu końców otrzymując okrąg z cięciwami.

Z drugiej strony, grafy kołowe można odwzorować na zbiór odcinków na prostej, przy czym dwa odcinki będą tworzyć krawędź grafu, jeżeli częściowo na siebie zachodzą, a jeden nie zawiera się w całości w drugim [36]. Do takiej interpretacji odcinków na prostej stosuje się nazwę *overlap graphs*. Problem znajdowania najmniejszego zbioru dominującego w grafach kołowych jest NP-zupełny [37].

2.3.9. Grafy bez trójek asteroidalnych

Trzy niesąsiednie wierzchołki grafu tworzą trójkę asteroidalną (ang. *asteroidal triple, AT*), jeżeli każde dwa z tych trzech wierzchołków można połączyć krawędzią omijającą sąsiedztwo trzeciego wierzchołka. Grafy nie posiadające trójki asteroidalnej nazywamy grafami bez AT (ang. *AT-free graphs*). Grafy bez AT są uogólnieniem m.in. grafów przedziałowych i grafów permutacji. Grafy te były badane w pracy magisterskiej Angeliki Siwek [38].

Algorytm wyznaczania najmniejszego zbioru dominującego w grafach bez AT podał Kratsch w roku 2000 [39], a jego implementację opisano w pracy Siwek. Algorytm jest dość złożony i składa się z pięciu kroków. Teoretyczna złożoność obliczeniowa algorytmu wynosi $O(n^6)$, natomiast w praktyce udało się osiągnąć złożoność $O(n^7)$.

```
# Finding a minimum dominating set for AT-free graphs.
from graphtheory.asteroidal.atfreedset import ATFreeDominatingSet

G = Graph()
# Add nodes and edges here.
algorithm = ATFreeDominatingSet(G)
algorithm.run()
print(algorithm.dominating_set)
print(algorithm.cardinality)
```

2.3.10. Grafy planarne

Definicja: Graf planarny (ang. *planar graph*) to graf, który można narysować na płaszczyźnie w taki sposób, że jego krawędzie nie przecinają się w żadnych punktach, jedynie w wierzchołkach. Taki rysunek nazywamy rysunkiem planarnym grafu. W takim przypadku planarność oznacza istnienie jakiejś reprezentacji grafu w dwuwymiarowej przestrzeni, tak aby żadne

krawędzie się nie przecinały poza wierzchołkami [40]. Problem wyznaczania najmniejszego zbioru dominującego w grafach planarnych jest NP-trudny.

3. Algorytmy

W tym rozdziale przedstawimy implementacje w języku Python wybranych algorytmów wyznaczających zbiory dominujące dla drzew, grafów przedziałwoych i grafów permutacji.

3.1. Wyznaczanie najmniejszego zbioru dominującego dla drzew

Przedstawiona zostanie implementacja algorytmu podanego przez Chena [24].

Dane wejściowe: Graf nieskierowany będący drzewem lub lasem.

Problem: Wyznaczenie najmniejszego zbioru dominującego dla drzew lub lasów.

Dane wyjściowe: Wyznaczone jednocześnie najmniejszy zbiór dominujący i największy zbiór 2-niezależny.

Opis algorytmu: Algorytm rozpoczynamy od identyfikacji wyizolowanych wierzchołków, jeśli takie istnieją w lesie oraz liści w drzewie. Wierzchołki izolowane trafiają bezpośrednio do zbioru dominującego oraz do zbioru 2-niezależnego, a liście dodawane są do kolejki, gdzie następnie są przetwarzane (odrywane) iteracyjnie. Szukamy rodzica każdego z liści znajdujących się w kolejce i jeśli liść nie został jeszcze zdominowany, to rodzic trafia do zbioru dominującego, a liść do zbioru niezależnego. Po każdej takiej operacji aktualizujemy stopnie wierzchołków i dodajemy nowe liście do kolejki. Iterujemy po kolejce liści do momentu, aż całe drzewo będzie zdominowane.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n)$, gdzie n to liczba wierzchołków w podanym grafie. Każdy wierzchołek oraz krawędź odwiedzane są co najwyżej raz.

Listing 3.1. Moduł dset1.

```
import sys
import collections

class TreeDominatingSet3:
    """Find a minimum dominating set and a maximum 2-stable set for trees.
    A single tree is connected during computations."""
    def __init__(self, graph):
        """The algorithm initialization."""
```

```
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.dominating_set = set()
        self.cardinality = 0
        self.two_stable_set = set()

    def run(self):
        """Executable pseudocode."""
        used = set()   # for dset and neighbors
        # A dictionary with node degrees, O(n) time.
        degree_dict = dict((node, self.graph.degree(node))
            for node in self.graph.iternodes())
        Q = collections.deque()   # for leaves
        # Put leafs to the queue, O(n) time.
        for node in self.graph.iternodes():
            if degree_dict[node] == 0:   # isolated node from the beginning
                self.dominating_set.add(node)
                self.two_stable_set.add(node)
                used.add(node)
                self.cardinality += 1
            elif degree_dict[node] == 1:   # leaf
                Q.append(node)

        while len(Q) > 0:
            source = Q.popleft()
            # A leaf may become isolated.
            if degree_dict[source] == 0:
                if source not in used:   # parent is not in dset
                    self.dominating_set.add(source)
                    self.two_stable_set.add(source)
                    used.add(source)
                    self.cardinality += 1
                continue
            assert degree_dict[source] == 1

            for target in self.graph.iteradjacent(source):
                if degree_dict[target] > 0:   # this is parent
                    if source not in used:   # parent goes to dset
                        self.dominating_set.add(target)
                        self.two_stable_set.add(source)
                        used.add(target)
                        self.cardinality += 1
                        used.update(self.graph.iteradjacent(target))
                    # Remove the edge from source to target.
                    degree_dict[target] -= 1
                    degree_dict[source] -= 1
                    if degree_dict[target] == 1:   # parent is a new leaf
                        Q.append(target)
                    break
```

Uwagi: Na potrzeby implementacji tego algorytmu stworzone zostały dwie klasy, z dwoma różnymi podejściami do problemu. Działanie pierwszej klasy TreeDominatingSet4 zostało opisane powyżej. Lista wierzchołków (liści) nie jest dana z góry, tylko jest wyznaczana w czasie pracy algorytmu.

Z kolei druga klasa TreeDominatingSet4 daje takie same wyniki, lecz ma znacznie krótszy kod dzięki wykorzystaniu w niej klasy pomocniczej TreePEO, która odpowiada za przetworzenie grafu i wyznaczenie:

— PEO, uporządkowania eliminacji liści (ang. *Perfect Elimination Order*), tj. kolejności usuwania wierzchołków od liści do korzeni,
— słownika, w którym dla każdego wierzchołka zapisana jest informacja o jego rodzicu w grafie.

Z klasy TreePEO otrzymujemy od razu całą listę wierzchołków, pokazaną w pseudokodzie w rozdziale 2. Jednak do otrzymania wydajnej implementacji potrzebna jest znajomość rodziców liści, o czym nie mówi pseudokod. W naszej implementacji klasa TreePEO wylicza słownik z rodzicami liści.

Listing 3.2. Moduł dset2.

```
from treepeo1 import TreePEO

class TreeDominatingSet4:
    """Find a minimum dominating set and a maximum 2-stable set for trees."""
    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.dominating_set = set()
        self.cardinality = 0
        self.two_stable_set = set()

    def run(self):
        """Executable pseudocode."""
        algorithm = TreePEO(self.graph)
        algorithm.run()  # O(n) time
        used = set()  # for dset and neighbors
        for source in algorithm.peo:  # to beda kolejne liscie
            target = algorithm.parent[source]
            if target is None:  # to jest ostatni wierzcholek
                if source not in used:  # parent is not in dset
                    self.dominating_set.add(source)
                    self.two_stable_set.add(source)
                    used.add(source)
                    self.cardinality += 1
                continue
            if source not in used:  # parent goes to dset
                self.dominating_set.add(target)
                self.two_stable_set.add(source)
                used.add(target)
                self.cardinality += 1
                used.update(self.graph.iteradjacent(target))
```

3.2. Wyznaczanie najmniejszej wagi zbioru dominującego dla drzew

Algorytm jest rozwinięciem podejścia nazywanego składaniem drzew, które pojawiło się w pracy magisterskiej Aleksandra Krawczyka [22]. Analogiczne podejście do wyznaczania zbiorów dominujących z wagami w drzewach zostało podane przez Jou i Lina, którzy sformułowali liniowy algorytm [25]. Algorytm przedstawiony w tym rozdziale opiera się na tej samej technice, lecz jawnie przechowuje zbiory dominujące, podczas gdy w pracy Jou i Lina zbiory dominujące są przechowywane w postaci wag.

Dane wejściowe: Graf nieskierowany będący drzewem lub lasem oraz słownik zawierający wagi poszczególnych wierzchołków dla tego grafu.

Problem: Wyznaczenie zbioru dominującego o najmniejszej sumie wag wierzchołków dla drzew lub lasów.

Dane wyjściowe: Zbiór dominujący o najmniejszej sumie wag wierzchołków oraz wartość liczbowa tej sumy.

Opis algorytmu: Algorytm rozpoczynamy od zidentyfikowania spójnych składowych grafu. Dla każdej z tych składowych wyznaczamy wierzchołek źródłowy i uruchamiamy funkcje rekurencyjnego przetwarzania drzewa. W procedurze tej każdy rozważany wierzchołek może być w jednym z trzech stanów, to jest:
— wierzchołek należy do zbioru dominującego,
— wierzchołek nie należy do zbioru dominującego, ale jest zdominowany,
— wierzchołek nie jest zdominowany.
Rekurencyjnie przetwarzamy wszystkie poddrzewa, a następnie łączymy ich wyniki za pomocą funkcji składającej, która wybiera kombinacje o najmniejszej sumie wag. Dla każdego wierzchołka funkcja ta rozważa różne możliwości połączeń wyników z poddrzew i wybiera to połączenie, które daje wynik z najmniejszą wagą. Po przetworzeniu całego drzewa algorytm wybiera wynik z mniejszą wagą spomiędzy zbioru dominującego zawierającego korzeń, a zbioru, który go nie zawiera. W przypadku lasu, wyniki z poszczególnych drzew łączone są w jeden zbiór dominujący.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n)$, gdzie n to liczba wierzchołków w podanym grafie. Każdy wierzchołek jest odwiedzany dokładnie raz, a operacje wykonywane dla wszystkich wierzchołków mają stałą złożoność. Testy ujawniły pewien dodatkowy narzut związany z wielokrotnym użyciem metody `_calc_weight`. Narzut można zmniejszyć kosztem skomplikowania kodu. Wywołania rekurencyjne powinny zwracać nie tylko sam częściowy zbiór dominujący, ale także jego wagę.

Uwagi: Algorytm jest uogólnieniem standardowego algorytmu znajdowania najmniejszego zbioru dominującego dla drzew lub lasów. Różnica polega na tym, że zamiast minimalizować liczność zbioru, minimalizujemy sumę wag

wierzchołków należących do zbioru dominującego. Na potrzeby implementacji tego algorytmu stworzona została druga klasa TreeWeightedDominatingSet2, która nie wykorzystuje funkcji _calc_weight i ma nieznacznie lepszą złożoność czasową, lecz kod jest wtedy bardziej rozbudowany.

Listing 3.3. Moduł wdset1.

```
import sys
import collections

class TreeWeightedDominatingSet1:
    """Find a minimum weight dominating set for trees."""

    def __init__(self, graph, weight_dict):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.parent = dict()
        self.dominating_set = set()
        self.cardinality = 0
        self.weight_dict = weight_dict
        self.dominating_set_weight = 0
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self, source=None):
        """Executable pseudocode."""
        if source is not None:
            # A single connected component, a single tree.
            self.parent[source] = None  # before _visit
            a2_set, b2_set, c2_set = self._visit(source)
            self.dominating_set.update(min(a2_set, b2_set,
                key=self._calc_weight))
            self.cardinality = len(self.dominating_set)
            self.dominating_set_weight = self._calc_weight(self.dominating_set)
        else:
            # A forest is possible.
            for node in self.graph.iternodes():
                if node not in self.parent:
                    self.parent[node] = None  # before _visit
                    a2_set, b2_set, c2_set = self._visit(node)
                    self.dominating_set.update(min(a2_set, b2_set,
                        key=self._calc_weight))
            self.cardinality = len(self.dominating_set)
            self.dominating_set_weight = self._calc_weight(self.dominating_set)

    def _calc_weight(self, aset):
        """Find the weight of a given vertex set."""
        return sum(self.weight_dict[node] for node in aset)

    def _compose(self, arg1, arg2):
        """Compose results."""
        # a_set : min dset that includes root
        # b_set : min dset that excludes root
        # c_set : root undominated
        a1_set, b1_set, c1_set = arg1
```

```
        a2_set, b2_set, c2_set = arg2
        a_set = a1_set | min(arg2, key=self._calc_weight)
        b_set = min(b1_set | a2_set, b1_set | b2_set,
            c1_set | a2_set, key=self._calc_weight)
        c_set = c1_set | b2_set
        return (a_set, b_set, c_set)

    def _visit(self, root):
        """Explore recursively the connected component."""
        # Start from a single node.
        arg1 = (set([root]), set([root]), set())
        for target in self.graph.iteradjacent(root):
            if target not in self.parent:
                self.parent[target] = root    # before _visit
                arg2 = self._visit(target)
                arg1 = self._compose(arg1, arg2)
        return arg1
```

3.3. Wyznaczanie najmniejszego niezależnego zbioru dominującego dla drzew

W celu znalezienia najmniejszego niezależnego zbioru dominującego dla drzewa wystarczy nieznacznie zmodyfikować algorytm składania drzew dla zwykłego zbioru dominującego z pracy Krawczyka [22]. Jedyna zabroniona sytuacja to łączenie krawędzią dwóch poddrzew, w których oba korzenie należą do częściowego rozwiązania. Tak więc zmieniona tabelka składania rozwiązań z pracy Krawczyka będzie wyglądać tak:

Tabela 3.1. Tabliczka składania dla niezależnego zbioru dominującego

$\oplus$	$T_2.a$	$T_2.b$	$T_2.c$
$T_1.a$	—	$T.a$	$T.a$
$T_1.b$	$T.b$	$T.b$	—
$T_1.c$	$T.b$	$T.c$	—

Gdzie przyjęto następujące oznaczenia:
— $T.a$ - najmniejszy niezależny zbiór dominujący zawierający korzeń drzewa T,
— $T.b$ - najmniejszy niezależny zbiór dominujący niezawierający korzenia drzewa T,
— $T.c$ - najmniejszy niezależny zbiór dominujący, gdy tylko korzeń drzewa T nie jest zdominowany.
Algorytm został zaimplementowany jako klasa TreeIndependentDominatingSet1.

Dane wejściowe: Graf nieskierowany będący drzewem lub lasem.

Problem: Wyznaczenie najmniejszego niezależnego zbioru dominującego dla drzew lub lasów.

Dane wyjściowe: Niezależny zbiór dominujący oraz jego liczność.

Opis algorytmu: Algorytm rozpoczynamy od uruchomienia algorytmu DFS po drzewie lub każdej składowej lasu. W trakcie przetwarzania wierzchołków za pomocą DFS, dla każdego wierzchołka wyznaczamy rekurencyjnie trzy możliwe warianty rozwiązań:
— najmniejszy niezależny zbiór dominujący zawierający korzeń,
— najmniejszy niezależny zbiór dominujący niezawierający korzenia,
— najmniejszy niezależny zbiór (prawie) dominujący, gdzie korzeń nie jest zdominowany.
W przypadku liści, dwa pierwsze warianty redukują się do prostego wybrania liścia, a trzeci wariant jest zawsze pustym zbiorem. W następnym kroku łączymy wyniki poddrzew funkcją składającą: gdy korzeń jest obecny w rozwiązaniu, dołączamy te poddrzewa, które zachowują niezależność. Gdy korzeń nie jest obecny w rozwiązaniu, wybieramy te kombinacje poddrzew, które zdominują go minimalnym kosztem. Po przetworzeniu całego drzewa wybieramy ten wariant, z pierwszych dwóch, który jest lepszy. W przypadku lasów łączymy rozwiązania ze wszystkich drzew.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n)$, gdzie n to liczba wierzchołków w podanym grafie. Każdy wierzchołek jest odwiedzany dokładnie raz, a operacje wykonywane dla wszystkich wierzchołków mają stałą złożoność. Testy ujawniły pewien dodatkowy narzut związany z wielokrotnym użyciem metody _compose do łączenia zbiorów, które są zbiorami rosnącymi. W literaturze zwykle wyznacza się tylko liczność zbioru, a wtedy ten narzut nie występuje. Z drugiej strony, w praktyce chcemy znać także sam zbiór dominujący, dlatego nasza implementacja wyznacza ten zbiór.

Listing 3.4. Moduł idset1.

```
import sys
import collections

class TreeIndependentDominatingSet1:
    """Find a minimum cardinality independent dominating set for trees."""

    def __init__(self, graph):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.parent = dict()
        self.dominating_set = set()
        self.cardinality = 0
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self, source=None):
        """Executable pseudocode."""
        if source is not None:
            # A single connected component, a single tree.
            self.parent[source] = None   # before _visit
            a2_set, b2_set, c2_set = self._visit(source)
            self.dominating_set.update(min(a2_set, b2_set, key=len))
            self.cardinality = len(self.dominating_set)
        else:
            # A forest is possible.
            for node in self.graph.iternodes():
                if node not in self.parent:
                    self.parent[node] = None   # before _visit
                    a2_set, b2_set, c2_set = self._visit(node)
                    self.dominating_set.update(min(a2_set, b2_set, key=len))
            self.cardinality = len(self.dominating_set)

    def _compose(self, arg1, arg2):
        """Compose results."""
        # a_set : min dset that includes root
        # b_set : min dset that excludes root
        # c_set : root undominated
        a1_set, b1_set, c1_set = arg1
        a2_set, b2_set, c2_set = arg2
        a_set = a1_set | min(b2_set, c2_set, key=len)
        b_set = min(b1_set | a2_set, b1_set | b2_set,
            c1_set | a2_set, key=len)
        c_set = c1_set | b2_set
        return (a_set, b_set, c_set)

    def _visit(self, root):
        """Explore recursively the connected component."""
        arg1 = (set([root]), set([root]), set())
        for target in self.graph.iteradjacent(root):
            if target not in self.parent:
                self.parent[target] = root   # before _visit
                arg2 = self._visit(target)
                arg1 = self._compose(arg1, arg2)
        return arg1
```

3.4. Wyznaczanie najmniejszej wagi niezależnego zbioru dominującego dla drzew

Algorytm jest uogólnieniem wersji bez wag, analogicznie jak dla zwykłych zbiorów dominujących. Algorytm został zaimplementowany jako klasa TreeWeightedIndependentDominatingSet1.

Dane wejściowe: Graf nieskierowany będący drzewem lub lasem oraz słownik zawierający wagi poszczególnych wierzchołków dla tego grafu.

Problem: Wyznaczenie niezależnego zbioru dominującego o najmniejszej sumie wag wierzchołków dla drzew lub lasów.

Dane wyjściowe: Niezależny zbiór dominujący o najmniejszej sumie wag wierzchołków oraz wartość liczbowa tej sumy.

Opis algorytmu: Algorytm rozpoczynamy od zidentyfikowania spójnych składowych grafu. Dla każdej z tych składowych wyznaczamy wierzchołek źródłowy i uruchamiamy funkcję rekurencyjnego przetwarzania drzewa. W procedurze tej każdy wierzchołek rozważamy w jednym z czterech możliwych stanów:

— należy do niezależnego zbioru dominującego (oznaczenie a_set),
— nie należy do niezależnego zbioru dominującego, ale co najmniej jedno dziecko do niego należy (oznaczenie b_set),
— najlepszy wariant spośród wariantów a_set i b_set, czyli zbiór o najmniejszej wadze (oznaczenie c_set).
— zbiór rozwiązań dzieci, który jest wykorzystywany przez rodzica, gdy ten sam nie należy do niezależnego zbioru dominującego (oznaczenie lambda_set).

Dla liści przyjmujemy, że muszą należeć do niezależnego zbioru dominującego, więc a_set = c_set = {korzeń}, b_set = {}.

Dla każdego wierzchołka wyznaczamy dzieci i rekurencyjnie przetwarzamy każde z nich. Wyznaczamy wtedy:

— zbiór a_set, który zawiera bieżący wierzchołek oraz zbiory lambda_set wszystkich dzieci (które nie mogą być wybrane do rozwiązania, bo rodzic je dominuje),
— zbiór b_set, który zakłada sytuację odwrotną, więc nie zawiera bieżącego wierzchołka, ale zawiera co najmniej jedno dziecko. Dla każdego dziecka sprawdzamy wtedy wariant: a_set tego dziecka oraz c_set pozostałych dzieci i wybieramy ten z najmniejszą wagą,
— zbiór c_set, który wybiera lepszą opcję (tę z mniejszą wagą) z dwóch powyższych,
— zbiór lambda_set, który jest sumą zbiorów c_set wszystkich dzieci.

Po przetworzeniu całego drzewa wybieramy ten wariant c_set korzenia, który daje najlepszy wynik. W przypadku lasu wyniki poszczególnych drzew są łączone w jeden końcowy zbiór niezależny dominujący.

Złożoność: Złożoność obliczeniowa algorytmu wynosi $O(n)$, przy czym tu również występuje dodatkowy narzut związny z wielokrotnym użyciem metody _calc_weight, który można zmniejszyć kosztem skomplikowania kodu.

Uwagi: Algorytm został zaimplementowany w trzech wersjach. Pierwsza wersja TreeWeightedIndependentDominatingSet1 jest wersją zrobioną na bazie algorytmu TreeWeightedDominatingSet1 oraz algorytmu zaproponowanego przez Jou i Lina [25]. Druga i trzecia wersja algorytmu różnią się od siebie obecnością funkcji _calc_weight i posłużyły do porównania złożoności czasowej i ukazania, że można nieznacznie zniwelować narzut czasowy spowodowany wielokrotnym użyciem tej funkcji i zamiast niej kosztem skomplikowania kodu dla wywołań rekurencyjnych zwracać nie tylko częściowy zbiór dominujący, ale także jego wagę.

Listing 3.5. Moduł widset1.

```
import sys

class TreeWeightedIndependentDominatingSet1:
    """Find a minimum weight independent dominating set for trees."""

    def __init__(self, graph, weight_dict):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.weight_dict = weight_dict
        self.parent = dict()
        self.independent_dominating_set = set()
        self.independent_dominating_set_weight = 0
        self.cardinality = 0
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self, source=None):
        """Executable pseudocode."""
        if source is not None:
            # single tree
            self.parent[source] = None    # before _visit
            _, _, c2_set, _ = self._visit(source)
            self.independent_dominating_set.update(c2_set)
        else:
            # forest is possible
            for node in self.graph.iternodes():
                if node not in self.parent:
                    self.parent[node] = None  # before _visit
                    _, _, c2_set, _ = self._visit(node)
                    self.independent_dominating_set.update(c2_set)
        self.cardinality = len(self.independent_dominating_set)
        self.independent_dominating_set_weight = (
            self._calc_weight(self.independent_dominating_set)
        )

    def _calc_weight(self, vertex_set):
        """Find the weight of a given vertex set."""
        return sum(self.weight_dict[v] for v in vertex_set)

    def _visit(self, root):
        """Explore recursively the connected component."""
        children = (
```

```
        [v for v in self.graph.iteradjacent(root) if v not in self.parent]
    )

    if not children:
        # leaf node: root must be in the set to dominate itself
        a_set = {root}        # root in the set
        b_set = set()         # root out of set
        c_set = {root}        # best set
        lambda_set = set()    # children not in set
        return a_set, b_set, c_set, lambda_set

    results = []
    for child in children:
        self.parent[child] = root   # before _visit
        results.append(self._visit(child))

    # a_set: min idset that includes root
    # (children excluded, using their lambda sets)
    a_set = {root} | set().union(*(lmbda for _, _, _, lmbda in results))

    # b_set : min idset that excludes root (at least one child must be in)
    b_options = []
    for i, (a_i, _, _, _) in enumerate(results):
        candidate = a_i.copy()
        for j, (_, _, c_j, _) in enumerate(results):
            if j != i:
                candidate |= c_j
        b_options.append(candidate)
    b_set = min(b_options, key=self._calc_weight)

    # c_set : best overall (min of a_set, b_set)
    c_set = min([a_set, b_set], key=self._calc_weight)

    # lambda_set : solution when root excluded
    # and children handle domination
    lambda_set = set()
    for _, _, c, _ in results:
        lambda_set |= c

    return a_set, b_set, c_set, lambda_set
```

3.5. Wyznaczanie najmniejszego zbioru dominującego dla grafów przedziałowych

Pseudokod tego algorytmu został podany przez Chena [24].

Dane wejściowe: Permutacja etykiet o długości $2n$, gdzie n to liczba wierzchołków, w której etykieta dla danego wierzchołka reprezentuje początek lub koniec jego przedziału. Podwójna permutacja opisuje graf przedziałowy, w którym dwa wierzchołki są połączone krawędzią wtedy i tylko wtedy, gdy ich odpowiadające przedziały mają część wspólną. Na przykład permutacja:

$$[1, 2, 3, 4, 1, 3, 2, 4]$$

reprezentuje cztery przedziały, etykieta każdego z nich występuje dwukrotnie (jego początek i koniec). Oznacza to, że:
— wierzchołek 1 przecina się z 2, 3 i 4 (ponieważ ich przedziały nakładają się wzajemnie),
— wierzchołek 2 przecina się z 1, 3 i 4,
— wierzchołek 3 przecina się z 1 i 4,
— wierzchołek 4 przecina się z 1, 2 i 3.
Taka reprezentacja pomaga w sposób bardziej kompaktowy zapisać pełną strukturę grafu przedziałowego bez konieczności posiadania obiektu grafu lub jawnej listy krawędzi.

Problem: Wyznaczenie najmniejszego zbioru dominującego dla grafów przedziałowych.

Dane wyjściowe: Wyznaczone jednocześnie najmniejszy zbiór dominujący i największy zbiór 2-niezależny.

Opis algorytmu: Algorytm rozpoczynamy od przeanalizowania podwójnej permutacji, która reprezentuje nasz graf przedziałowy. Na tej podstawie identyfikowane są początki oraz końce przedziałów dla każdego z wierzchołków, a następnie dla każdego z nich są również tworzone zbiory sąsiadów, bez konieczności jawnego konstruowania grafu. W kolejnym kroku permutacja jest przetwarzania liniowo, a algorytm śledzi aktualnie aktywne przedziały. Kiedy napotykamy na koniec przedziału danego wierzchołka to sprawdzamy, czy został on już zdominowany. Jeśli nie, wtedy do zbioru dominującego dodajemy najbardziej oddalonego sąsiada, a sam wierzchołek dodajemy do zbioru 2-niezależnego. Proces taki powtarzamy do momentu, aż skończymy przetwarzać wszystkie wierzchołki, dzięki czemu pokrywamy każdą klikę. W rezultacie otrzymujemy najmniejszy zbiór dominujący oraz zbiór 2-niezależny, które budujemy równocześnie w trakcie pracy algorytmu.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n+m)$, gdzie n to liczba wierzchołków, a m to liczba krawędzi w podanym grafie. W pierwszej części algorytmu budowa słownika ze zbiorami sąsiadów wierzchołków zajmuje czas $O(n+m)$. W drugiej części algorytmu iterujemy po wierzchołkach grafu (po permutacji), a wewnątrz pętli przeszukujemy sąsiedztwo danego wierzchołka. Łączny rozmiar sąsiedztw to dwukrotność liczby krawędzi, więc ta część również zajmuje czas $O(n+m)$. Przeprowadzone testy potwierdzają tę złożoność, dla rzadkich grafów 2-drzewowych czas rośnie liniowo wraz ze wzrostem liczby wierzchołków, a dla gęstszych grafów k-drzewowych obserwujemy wzrost bliski $O(n^2)$ z powodu dużej liczby krawędzi. We wszystkich przypadkach potwierdzono optymalną złożoność czasową względem wielkości grafu.

Uwagi: Na potrzeby implementacji tego algorytmu stworzone zostały dwie funkcje, z dwoma różnymi podejściami do problemu.
Pierwsza funkcja interval_minimum_dset1 do wyznaczenia sąsiadów wierzchołka wymagała podania dodatkowo grafu abstrakcyjnego, odpowiadającego przedziałom. Z kolei druga funkcja interval_minimum_dset2, opisana powyżej, sama wyznacza sąsiadów wierzchołka na bazie podwójnej permutacji.

Listing 3.6. Moduł interval2.

```
def interval_minimum_dset2(perm):
    """Finding a minimum dominating set and a maximum 2-stable set
    of an interval graph (double perm)."""
    # Zapisuje indeksy koncow przedzialow.
    pairs = dict((node, []) for node in set(perm))    # O(n) time
    for idx, node in enumerate(perm):    # O(n) time
        pairs[node].append(idx)

    # Tworze zbiory sasiadow, graph[v] is closed neighborhood of v.
    graph = dict((node, set([node])) for node in set(perm))    # O(n) time
    used = set()    # current nodes
    for source in perm:    # O(n) time
        if source in used:    # bedzie usuwanie node
            used.remove(source)
        else:    # dodajemy nowy node i krawedzie
            for target in used:
                graph[source].add(target)
                graph[target].add(source)
            used.add(source)

    dset = set()
    stable2 = set()
    used = set()    # aktywne przedzialy
    for source in perm:    # idziemy wzdluz permutacji, O(n) time
        if source in used:    # bedzie usuwanie node, klika zmaleje
            target = max(used, key=lambda v: pairs[v][1])
            if len(graph[source].intersection(dset)) == 0:
                dset.add(target)
                stable2.add(source)
            used.remove(source)
        else:    # dodajemy nowy node, klika rosnie
            used.add(source)
    return dset, stable2
```

3.6. Wyznaczanie najmniejszej wagi zbioru dominującego dla grafów przedziałowych

Przedstawiony poniżej algorytm pochodzi z pracy Ramalingama i Rangana z roku 1988 [28]. W pracy tej przedstawiono bazowy algorytm do znajdowania najmniejszych zbiorów dominujących w grafach przedziałowych. Podano również proste modyfikacje algorytmu bazowego, pozwalające na wyznaczenie zbiorów dominujących o najmniejszej sumie wag czy też niezależnych zbiorów dominujących.

Dane wejściowe: Graf przedziałowy jest zapisany w postaci podwójnej permutacji o długości $2n$ z etykietami wierzchołków, gdzie n to liczba wierzchołków, a etykieta dla danego wierzchołka reprezentuje początek lub koniec jego przedziału.

Problem: Wyznaczenie zbioru dominującego o najmniejszej sumie wag dla grafów przedziałowych.

Dane wyjściowe: Wyznaczony zbiór dominujący o najmniejszej sumie wag wraz z jego całkowitą wagą.

Opis algorytmu: Algorytm rozpoczynamy od przeanalizowania podwójnej permutacji, która reprezentuje nasz graf przedziałowy. Na tej podstawie identyfikowane są początki i końce przedziałów dla każdego z wierzchołków, wyznaczamy także perfekcyjny schemat eliminacji wierzchołków (PEO) przez iteracyjne usuwanie wierzchołków o najmniejszych końcach przedziałów. Następnie tworzone są zbiory sąsiadów dla każdego z wierzchołków, pozwala to uniknąć jawnego konstruowania grafu. Kolejny krok to wyznaczenie funkcji pomocniczych $low(i)$, $maxlow(i)$ i zbiorów M_i, które zawierają sąsiadów wierzchołka i z większymi numerami. Algorytm wykorzystuje następnie programowanie dynamiczne, dla każdego wierzchołka i rozważamy wszystkich możliwych kandydatów do zbioru dominującego: wierzchołki z przedziału $[maxlow(i), i]$ oraz sąsiadów ze zbioru M_i. Każdemu z kandydatów obliczamy sumę jego wag oraz wagę optymalnego rozwiązania. Wyboru dokonujemy na podstawie najmniejszej sumy wag. W celu optymalizacji pamięci oraz czasu wykonania algorytmu, zamiast przechowywać pełne zbiory dominujące, zapamiętujemy tylko indeksy wybranych wierzchołków w tablicy *choice*. Na końcu odtwarzamy iteracyjnie nasze rozwiązanie poprzez cofanie się po wierzchołkach od ostatniego do pierwszego.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n+m)$, gdzie n to liczba wierzchołków, a m to liczba krawędzi w grafie.

Uwagi: Poprzez przechowywanie tylko indeksów wybranych wierzchołków udało się uniknąć kopiowania całych zbiorów dominujących w każdym kroku algorytmu co poskutkowało zmniejszeniem zużycia pamięci oraz przyszpieszeniem działania algorytmu.

Listing 3.7. Moduł interval_wdset.

```
def interval_minimum_weight_dset(perm, weight_dict):
    """Find a minimum weight dominating set for interval graphs."""
    peo = []
    used = set()    # current nodes
    for node in perm:    # O(n) time
        if node in used:
            used.remove(node)
            peo.append(node)
        else:
            used.add(node)

    n = len(peo)
    node_to_index = dict((node, i+1) for i, node in enumerate(peo))

    # Przepisujemy perm na indeksy z peo plus jeden, O(n) time.
    perm2 = [node_to_index[node] for node in perm]

    # adjacency[v] is closed neighborhood of v.
    adjacency = dict((i, set([i])) for i in range(1, n+1))    # O(n) time
    used = set()
```

```
    for i in perm2:    # O(n) time
        if i in used:
            used.remove(i)
        else:
            for j in used:
                adjacency[i].add(j)
                adjacency[j].add(i)
            used.add(i)

    # find low(i) - smallest element (index) in neighborhood N[i].
    low = [0] * (n+1)    # low[0] is not used
    for i in range(1, n+1):    # total O(n+m) time
        low[i] = min(adjacency[i])

    maxlow = [0] * (n+1)    # maxlow[0] is not used
    for i in range(1, n+1):    # total O(n+m) time
        maxlow[i] = max(low[j] for j in range(low[i], i+1))

    # M_i = {j | j > i and j is a neighbor of i}
    M = dict((i, []) for i in range(1, n+1))    # O(n) time
    for i in range(1, n+1):    # total O(n+m) time
        for j in adjacency[i]:
            if j > i:
                M[i].append(j)

    Wt = [0.0] * (n+1)  # Wt[0] = 0, Wt[i] is weight of MD[i]
    choice = [0] * (n+1)  # choice[i] = indeks wybranego wierzcholka dla MD[i]

    for i in range(1, n+1):
        min_weight = float('inf')
        min_j = 0    # sztuczna wartosc

        # check all candidates: L_i = {maxlow[i], ..., i} and M_i = {j > i}
        candidates = list(range(maxlow[i], i+1)) + M[i]
        for j in candidates:
            total_weight = weight_dict[peo[j-1]] + Wt[low[j]-1]
            if total_weight < min_weight:
                min_weight = total_weight
                min_j = j

        choice[i] = min_j
        Wt[i] = min_weight

    # odtwarzamy rozwiazanie uzywajac iteracyjnego backtracking
    result_set = set()
    i = n
    while i > 0:
        j = choice[i]
        result_set.add(peo[j-1])
        i = low[j] - 1

    return Wt[n], result_set
```

3.7. Wyznaczanie najmniejszego zbioru dominującego dla grafów permutacji

Problem wyznaczania zbioru dominującego dla grafów permutacji został rozwiązany w roku 1985 przez Farbera i Keila [35], którzy podali pseudokod algorytmu o złożoności $O(n^2)$.

Dane wejściowe: Liczba wierzchołków n oraz permutacja P o długości n (lub $n+1$ z zerem na pozycji 0). W implementacji dla wygody gdy $|P| = n$, dopełniamy P przez wstawienie zera na pozycję 0.

Problem: Wyznaczenie najmniejszego zbioru dominującego w grafie permutacji.

Dane wyjściowe: Wyznaczony najmniejszy zbiór dominujący.

Opis algorytmu: Algorytm rozpoczynamy od inicjalizacji tablicy stanów $D[i][j]$ z wartościami ∞ dla wszystkich pozycji, gdzie $D[0][0] = 0$ reprezentuje stan początkowy. Algorytm wykorzystuje programowanie dynamiczne działające na samej permutacji P oraz jej odwrotności P^{-1}, bez jawnego budowania grafu. Utrzymujemy tablicę zbiorów $D[i][j]$, gdzie każdy wpis reprezentuje minimalny zbiór dominujący dla prefiksu $\{1, \ldots, i\}$ przy założeniu, że największy z „prawych końców" przykrytych na dolnym rzędzie ma wartość j. Dla każdego kolejnego wierzchołka i rozważane są trzy przypadki:
— **brak zmiany:** $D[i][j] = D[i-1][j]$ - wierzchołek i nie jest dodawany do zbioru dominującego,
— **domknięcie największego prawego końca:** $D[i][j] = D[i-1][j'] + 1$ gdzie j' jest odpowiednio zaktualizowane,
— **dodanie bieżącego wierzchołka:** $D[i][j] = D[i-1][j] + 1$ - wierzchołek i jest dodawany do zbioru dominującego,
a wybór dokonany jest według najmniejszej liczności, czyli $D[i][j]$ to będzie minimum z powyższych trzech wartości.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n^2)$, gdzie n to liczba wierzchołków. Mamy podwójną pętlę po indeksach i, j, więc wymagane jest budowanie zbiorów $D[i][j]$ w stałym czasie. Pewne operacje na zbiorach zajmują czas $O(n)$ (obliczanie maksimum), więc tych operacji nie może być wykonywanych więcej niż $O(n)$. Dla grafów permutacji drabiny złożoność ma pewien narzut ponad $O(n^2)$ spowodowany sklejaniem zbiorów.

Uwagi: Implementacja w module pdset1.py zwraca jawny najmniejszy zbiór dominujący. Zastosowanie permutacyjnego modelu P pozwala uniknąć budowy grafu.

Listing 3.8. Moduł pdset1.

```
def permutation_minimum_dset(perm):
    """Find a minimum dominating set in a permutation
    graph given by a permutation."""
    assert perm[0] == 0 # fake value
```

```
    n = len(perm)
    permnv = [None] * n
    for i in range(n):
        permnv[perm[i]] = i

    # D[i][j] = dominating set Yij with minimal cardinality
    D = [[[] for _ in range(n)] for _ in range(n)]

    for j in range(n):
        if j >= perm[1]:
            D[1][j] = [1]
        else:
            D[1][j] = []

    def max_perm(S):
        """Returns the maximum perm[v] for v in S"""
        return max((perm[v] for v in S), default=-1)

    for i in range(2, n):
        p = max(perm[j] for j in range(1, i+1))
        k = permnv[p]

        for j in range(n):
            if j < perm[i]:
                D[i][j] = D[i - 1][j]
            elif j == perm[i]:
                if max_perm(D[i - 1][j]) > perm[i]:
                    D[i][j] = D[i - 1][j]
                else:
                    D[i][j] = D[i - 1][j] + [k] # copying
            else:  # j > perm[i]
                # A: we do not add i
                if max_perm(D[i - 1][j]) > perm[i]:
                    A = D[i - 1][j]
                else:
                    A = D[i - 1][j] + [k] # copying

                # B: we add i and look at D[i-1][perm[i] - 1]
                B = D[i - 1][perm[i] - 1] + [i] # copying

                if len(A) < len(B):
                    D[i][j] = A
                elif len(B) < len(A):
                    D[i][j] = B
                else:
                    D[i][j] = A if max_perm(A) > max_perm(B) else B

    return set(perm[i] for i in D[n-1][n-1])
```

3.8. Wyznaczanie najmniejszej wagi zbioru dominującego dla grafów permutacji

Problem wyznaczania najmniejszej wagi zbioru dominującego dla grafów permutacji został również rozwiązany w roku 1985 przez Farbera i Keila [35], dla tego problemu autorzy podali pseudokod algorytmu o złożoności $O(n^3)$.

Dane wejściowe: Permutacja P definiująca graf permutacji oraz lista wag w przypisanych wierzchołkom tego grafu.

Problem: Wyznaczenie najmniejszej wagi zbioru dominującego w grafie permutacji.

Dane wyjściowe: Wyznaczona minimalna waga zbioru dominującego oraz jawnie podana lista wierzchołków tworzących najmniejszy zbiór dominujący.

Opis algorytmu: Algorytm rozpoczynamy od inicjalizacji tablicy trójwymiarowej $S[i][j][k]$, gdzie $S[i][j][k]$ reprezentuje minimalną wagę zbioru dominującego dla podgrafu indukowanego przez wierzchołki $\{1, 2, \ldots, i\}$, z dodatkowym warunkiem, że wierzchołek o wartości k musi być zdominowany przez wierzchołki o wartościach nie większych niż j.

W początkowym etapie algorytmu inicjalizujemy rozwiązania dla pierwszego wierzchołka, ustawiając $S[1][j][k] = (w[P(1)], [P(1)])$ dla $k = P(1)$ oraz $S[1][j][k] = \text{None}$ dla $k \neq P(1)$. W kolejnych etapach algorytmu dla każdego wierzchołka i rozważamy trzy przypadki w zależności od relacji między j a $P(i)$:

— gdy $j < P(i)$: kopiujemy rozwiązanie z poprzedniego etapu lub dodajemy bieżący wierzchołek do najlepszego rozwiązania z poprzednich wartości k,
— gdy $j = P(i)$: kopiujemy rozwiązanie z poprzedniej pozycji $j - 1$ lub ustawiamy None jeśli warunki nie są spełnione,
— gdy $j > P(i)$: wybieramy lepsze z dwóch rozwiązań - bez dodawania bieżącego wierzchołka lub z dodaniem bieżącego wierzchołka do rozwiązania z pozycji $P(i) - 1$.

W każdym przypadku aktualizujemy tablicę S na podstawie wcześniej obliczonych rozwiązań, zachowując informację o wagach i wartościach wierzchołków. Po przetworzeniu wszystkich etapów wybieramy najlepsze rozwiązanie spośród wszystkich kandydatów $S[n][n][k]$, porównując ich wagi i zwracając rozwiązanie o najmniejszej wadze.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n^3)$, gdzie n to liczba wierzchołków.

Uwagi: Implementacja w module wpdset1.py zwraca jawny najmniejszy zbiór dominujący. Zastosowanie permutacyjnego modelu P pozwala uniknąć budowy grafu.

Listing 3.9. Moduł wpdset1.

```
def permutation_minimum_weight_dset(permutation, weights):
```

```
    """Computes a minimum weight dominating set in a permutation
    graph given by a permutation and weights."""
    if len(weights) > 0 and weights[0] != 0:
        weights = [0] + weights
    if len(permutation) > 0 and permutation[0] != 0:
        permutation = [0] + permutation

    n = len(permutation) - 1
    assert (len(weights) == n + 1 and
            min(weights[1:]) >= 0), "Invalid weights or size."

    # inverse permutation: inv_perm[value] = position
    inv_perm = [0] * (n + 1)
    for pos in range(1, n + 1):
        inv_perm[permutation[pos]] = pos

    # S[i][j][k] stores (weight, vertex_values) for stages i, j, k
    S = [[[None] * (n + 1) for i in range(n + 1)]
         for j in range(n + 1)]

    for j in range(0, n + 1):
        for k in range(1, n + 1):
            if k == permutation[1]:
                S[1][j][k] = (weights[permutation[1]], [1])
            else:
                S[1][j][k] = None

    for stage in range(2, n + 1):
        for k in range(1, n + 1):
            if inv_perm[k] <= stage:
                S[stage][0][k] = (weights[k], [inv_perm[k]])
            else:
                S[stage][0][k] = None

        for j in range(1, n + 1):
            for k in range(1, n + 1):
                if j < permutation[stage]:
                    if k != permutation[stage]:
                        S[stage][j][k] = S[stage - 1][j][k]
                    else:
                        best_solution = None
                        for l in range(1, k):
                            candidate = S[stage][j][l]
                            if candidate is not None:
                                new_weight = (weights[permutation[stage]] +
                                              candidate[0])
                                if (best_solution is None or
                                    new_weight < best_solution[0]):
                                    new_vertices = ([stage] +
                                                    candidate[1])
                                    best_solution = (new_weight, new_vertices)

                        if best_solution is None:
                            vertex_weight = weights[permutation[stage]]
                            best_solution = (vertex_weight,
                                             [stage])
                        S[stage][j][k] = best_solution
```

```
                elif j == permutation[stage]:
                    if (k >= j) and (inv_perm[k] <= stage):
                        S[stage][j][k] = S[stage][j - 1][k]
                    else:
                        S[stage][j][k] = None

                else:  # j > permutation[stage]
                    if k == permutation[stage]:
                        S[stage][j][k] = S[stage][k][k]
                    elif (k > permutation[stage]) and (inv_perm[k] <= stage):
                        # option A: don't add current vertex
                        option_A = S[stage - 1][j][k]
                        option_A_weight = (float('inf')
                            if option_A is None else option_A[0])

                        # option B: add current vertex
                        base_solution = (S[stage][permutation[stage] - 1][k]
                            if permutation[stage] - 1 >= 0 else None)
                        if base_solution is None:
                            option_B = None
                        else:
                            new_weight = (base_solution[0] +
                                weights[permutation[stage]])
                            new_vertices = ([stage] +
                                base_solution[1])
                            option_B = (new_weight, new_vertices)

                        # choose the better option
                        if option_B is None or option_A_weight <= option_B[0]:
                            S[stage][j][k] = option_A
                        else:
                            S[stage][j][k] = option_B
                    else:
                        S[stage][j][k] = None

    # finding best solution among all final candidates
    best_solution = None
    for k in range(1, n + 1):
        candidate = S[n][n][k]
        if (candidate is not None and
            (best_solution is None or candidate[0] < best_solution[0])):
            best_solution = candidate

    if best_solution is None:
        return float('inf'), []

    return best_solution[0], set(permutation[i] for i in best_solution[1])
```

3.9. Wyznaczanie najmniejszej wagi niezależnego zbioru dominującego dla grafów permutacji

Problem wyznaczania najmniejszej wagi niezależnego zbioru dominującego dla grafów permutacji został rozwiązany w roku 1987 przez Brandstadta

i Kratscha [41], dla tego problemu podali pseudokod algorytmu o złożoności $O(n^2)$.

Dane wejściowe: Permutacja P definiująca graf permutacji oraz lista wag w przypisanych wierzchołkom tego grafu.

Problem: Wyznaczenie najmniejszej wagi niezależnego zbioru dominującego w grafie permutacji.

Dane wyjściowe: Wyznaczona najmniejsza waga niezależnego zbioru dominującego oraz jawnie podana lista wierzchołków tworzących najmniejszy zbiór dominujący.

Opis algorytmu: Algorytm rozpoczynamy od inicjalizacji tablic kosztów $c[i]$ i poprzedników $parent[i]$. Następnie w pierwszym etapie obliczamy dynamicznie minimalny koszt $c[i]$ rozwiązania, które kończy się wyborem wierzchołka na pozycji i-tej w permutacji. Dla każdej pozycji i inicjalizujemy $c[i] = w(P(i))$ i przechodzimy w lewo po pozycjach j, takich że $j < i$ w poszukiwaniu takiego najlepszego wierzchołka do dołączenia do rozwiązania, które nie złamie niezależności. Warunek niezależności wymuszamy zmienną t inicjalizowaną na -1, taką że dopuszczamy tylko te j, dla których wartości $P(j)$ rosną względem wcześniej sprawdzonych (odpowiada to brakowi krawędzi między wybranymi wierzchołkami). Dla każdego dopuszczonego j sprawdzamy czy połączenie $i+j$ ma mniejszą wagę niż aktualne $c[i]$. Jeśli tak, to aktualizujemy $c[i]$ i zapamiętujemy rodzica pozycji i-tej. Następnym krokiem jest wybór ogona rozwiązania, w tym celu przechodzimy po permutacji od końca, utrzymując próg $right$ i traktując jako kandydatów tylko pozycje, takie że $P(i) > right$. Dla każdego takiego i sprawdzamy czy $c[i] < c[best\_index]$ i jeśli tak, to aktualizujemy $best\_index$. Ostatnim krokiem jest odtworzenie naszego rozwiązania poprzez cofanie się po poprzednikach od naszego wybranego ogona i budowanie zbioru wierzchołków.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n^2)$, gdzie n to liczba wierzchołków.

Uwagi: Dostępne są dwie implementacje tego algorytmu:
— wipdset1.py – przyjmuje permutacje liczb od 1 do n oraz listę wag indeksowaną pozycjami, zwraca listę wierzchołków,
— wipdset2.py – przyjmuje dowolne permutacje liczb od 0 do $n-1$ oraz listę lub słownik z wagami indeksowanymi wartościami permutacji, zwraca zbiór wierzchołków.
Zastosowanie permutacyjnego modelu P pozwala uniknąć budowy grafu. Tak jak w przypadku dwóch poprzednich algorytmów, problem wyznaczania najmniejszej wagi niezależnego zbioru dominującego dla grafów permutacji został również rozwiązany w roku 1985 przez Farbera i Keila [35]. Podali oni tam pseudokod algorytmu o złożoności $O(n^3)$, jednakże rozwiązanie podane przez Brandstadta i Kratscha jest prostsze i szybsze.

Listing 3.10. Moduł wipdset2.

```
def permutation_minimum_weight_independent_dset2(perm, weights):
    """Computes a minimum weight independent dominating set
    in a perm graph defined by a perm and weights.
    """
    n = len(perm)
    assert len(weights) == n

    # stage 1: compute c(i) values
    c = [float("inf")] * n
    parent = [-1] * n   # from right to left indices

    for i in range(n):
        t = -1   # below the smallest allowed index 0
        c[i] = weights[perm[i]]   # perm[i] must be in MIS
        for j in range(i-1, -1, -1):
            if perm[j] < perm[i] and t == -1:   # first extension of MIS
                c[i] = weights[perm[i]] + c[j]
                t = perm[j]   # MIS decreased from P(i) to P(j) going left
                parent[i] = j
            elif -1 < t < perm[j] < perm[i]:
                new_ci = weights[perm[i]] + c[j]
                if new_ci <= c[i]:   # if equal weight, we prefer better perm[j]
                    c[i] = new_ci
                    parent[i] = j
                # t remembers the largest left neighbor of P(i) that was checked
                t = perm[j]

    # stage 2: select minimum c(i) that is tail of mis
    best_index = n-1
    right = perm[best_index]   # try the right end for MIS

    for i in range(n-2, -1, -1):   # go left from position n-1
        if perm[i] > right:   # otherwise perm[i] would be in the old MIS
            if c[i] <= c[best_index]: # if equal weight, we prefer bigger MIS
                best_index = i
            right = perm[i]   # new right end for MIS, which was checked

    # reconstruct solution
    mis = set()
    i = best_index
    while i != -1:   # O(n)
        mis.add(perm[i])
        i = parent[i]
    return c[best_index], mis
```

4. Podsumowanie

W niniejszej pracy przedstawiono szereg problemów związanych ze zbiorami dominującymi, ich odmiany i algorytmy znajdujące rozwiązania. Przy okazji omówiono pewne metody wykraczające poza tematykę zbiorów dominujących.

Problem zbiorów dominujących polega na znalezieniu najmniejszego podzbioru wierzchołków grafu, takiego że każdy wierzchołek grafu należy do tego podzbioru lub sąsiaduje z co najmniej jednym wierzchołkiem należącym do tego zbioru. Jest to problem o szerokim zastosowaniu w praktyce, między innymi w telekomunikacji i lokalizacyjnych problemach. Problem jest NP-zupełny dla grafów ogólnych, jednak dla wielu klas grafów istnieją efektywne algorytmy wielomianowe, które zostały przedstawione w ramach tej pracy.

Ważnym aspektem teoretycznym jest zrozumienie, gdzie przebiega granica między rodzinami grafów, dla których problem znajdowania najmniejszego zbioru dominującego jest możliwy do rozwiązania w czasie wielomianowym, a tymi, dla których staje się już trudny (NP-zupełny). Tabela 4.1 przedstawia klasyfikację wybranych rodzin grafów pod tym względem.

Tabela 4.1. Granica wielomianowej rozwiązywalności problemu zbiorów dominujących dla wybranych rodzin grafów.

wielomianowa	**wykładnicza (NP-zupełny)**
drzewa, sp-grafy	grafy planarne
grafy przedziałowe	grafy cięciwowe
grafy permutacji	grafy kołowe
grafy permutacji	grafy porównywalności
drzewa	grafy dwudzielne

Każdy wiersz w tabeli porównuje dwie zbliżone rodziny grafów, gdzie druga (wykładnicza) jest uogólnieniem pierwszej (wielomianowej). Przykładowo, drzewa to szczególny przypadek grafów planarnych, ale problem jest łatwy dla drzew, a trudny dla ogólnych grafów planarnych. Podobnie grafy przedziałowe to podzbiór grafów cięciwowych, ale problem ten staje się trudny przy przejściu do ogólnych grafów cięciwowych.

Metoda *primal-dual* pozwala na wyznaczenie jednocześnie najmniejszego zbioru dominującego i największego zbioru 2-niezależnego. Pokazano zastosowanie metody dla drzew i grafów przedziałowych.

Drzewa mają strukturę rekurencyjną, co jest wykorzystywane w metodzie składania drzew (metoda Borie). W pracy pokazane zostało zastosowanie tej metody do znajdowania najmniejszych zbiorów dominujących, zwykłych i niezależnych, z wagami lub bez wag. Warto zauważyć, że wiele rodzin grafów

ma strukturę rekurencyjną, co oczywiście było wykorzytywane w literaturze do tworzenia wydajnych algorytmów.

Metoda programowania dynamicznego pozwala zachować zalety podejścia rekurencyjnego przy jednoczesnym uniknięciu jego wad. Programowanie dynamiczne jest wykorzystane w przypadku algorytmów działających na grafach permutacji, które mają strukturę liniową (nie mają trójek asteroidalnych). Idąc wzdłuż permutacji tworzy się pewne zbiory pomocnicze, aby na końcu uzyskać wymagany zbiór dominujący.

Wszystkie powyższe metody znalazły zastosowanie w implementacjach algorytmów do znajdowania najmniejszych zbiorów dominujących, najmniejszych niezależnych zbiorów dominujących, zbiorów dominujących o najmniejszej wadze oraz niezależnych zbiorów dominujących o najmniejszej wadze, powstałych w ramach tej pracy opisanych dokładnie w rozdziale 3. Każda z implementacji została sprawdzona pod kątem poprawności z wykorzystaniem testów jednostkowych powstałych z pomocą modułu unittest oraz pod kątem wydajności i złożoności czasowej z wykorzystaniem modułu timeit. Otrzymane wyniki potwierdziły teoretyczne złożoności algorytmów. Bardziej szczegółowe informacje na temat testów znajdują się w rozdziale A.

A. Testy algorytmów

W tej części pracy opiszemy wyniki testów algorytmów z rozdziału 3. Algorytmy zostały przetestowane pod kątem poprawności oraz wydajności z użyciem narzędzi dostępnych w języku Python. Każdy algorytm został sprawdzony pod względem poprawności przy użyciu testów jednostkowych przygotowanych z modułem unittest [42]. Oceny wydajności oraz wyznaczenie złożoności obliczeniowej zostały z kolei przeprowadzone z użyciem modułu timeit [43]. Dla każdego algorytmu badania były wykonywane trzykrotnie, a następnie brano medianę, ponieważ występuje tutaj przypadkowość wyników wynikająca z wpływu innych procesów w systemie operacyjnym.

A.1. Test wyznaczania najmniejszego zbioru dominującego dla drzew

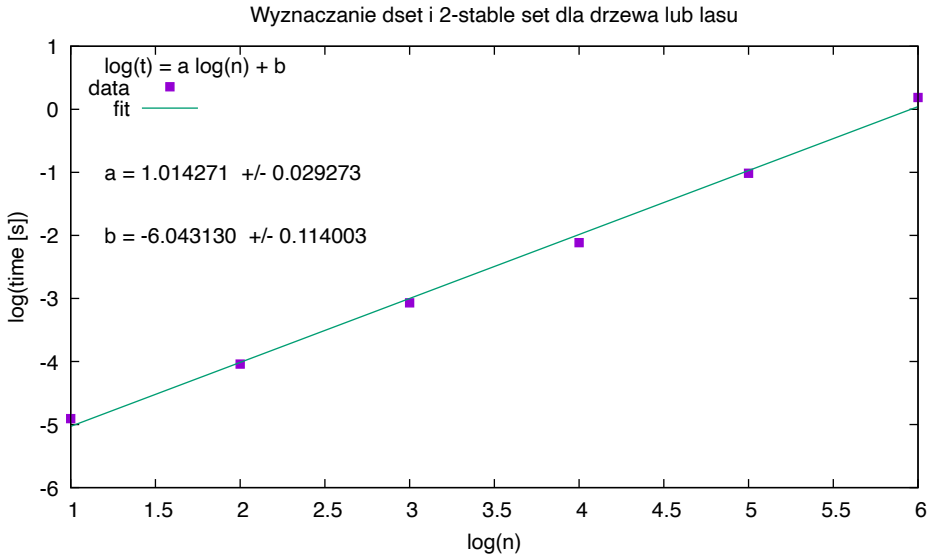
Algorytm z klasy TreeDominatingSet3 został przetestowany na losowo wygenerowanych drzewach o różnych liczbach wierzchołków. Drzewa zostały wygenerowane za pomocą funkcji make_tree z klasy GraphFactory, należącej do pakietu graphtheory [11]. Wynikiem testów jest wartość współczynnika dopasowania $a = 1.014(30)$ ukazana na wykresie A.1. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n)$.

A.2. Test wyznaczania najmniejszego zbioru dominującego dla drzew z użyciem klasy TreePEO

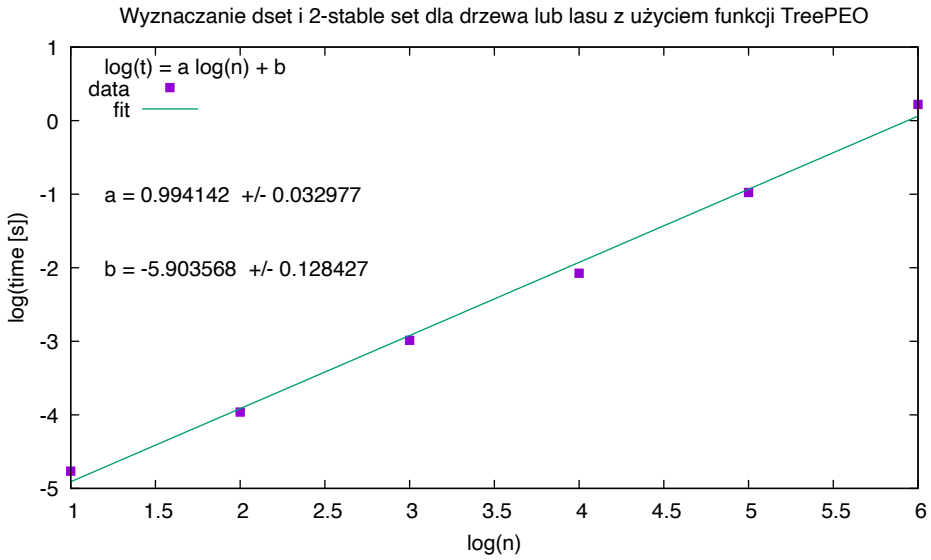
Algorytm z klasy TreeDominatingSet4 również został przetestowany w identyczny sposób co algorytm z klasy TreeDominatingSet3. Wynikiem testów jest wartość współczynnika dopasowania $a = 0.994(33)$ ukazana na wykresie A.2. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n)$. Jak można zauważyć, różnice między podejściami są minimalne, z lekką przewagą algorytmu używającego klasy TreePEO.

A.3. Test wyznaczania najmniejszego niezależnego zbioru dominującego dla drzew

Algorytm z klasy TreeIndependentDominatingSet1 został przetestowany na losowo wygenerowanych drzewach o różnych liczbach wierzchołków. Drzewa zostały wygenerowane za pomocą funkcji make_tree z klasy GraphFactory, należącej do pakietu graphtheory [11]. Wynikiem testów jest wartość współ-



Rysunek A.1. Wykres pomiarów algorytmu znajdującego najmniejszy zbiór dominujący dla drzew i lasów. Wartość współczynnika dopasowania $a = 1.014(30)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n)$.



Rysunek A.2. Wykres pomiarów algorytmu znajdującego najmniejszy zbiór dominujący dla drzew i lasów z użyciem klasy TreePEO. Wartość współczynnika dopasowania $a = 0.994(33)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n)$.

czynnika dopasowania $a = 1.058(19)$ ukazana na wykresie A.3. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n)$.

A.4. Test wyznaczania zbioru dominującego z najmniejszą wagą dla drzew

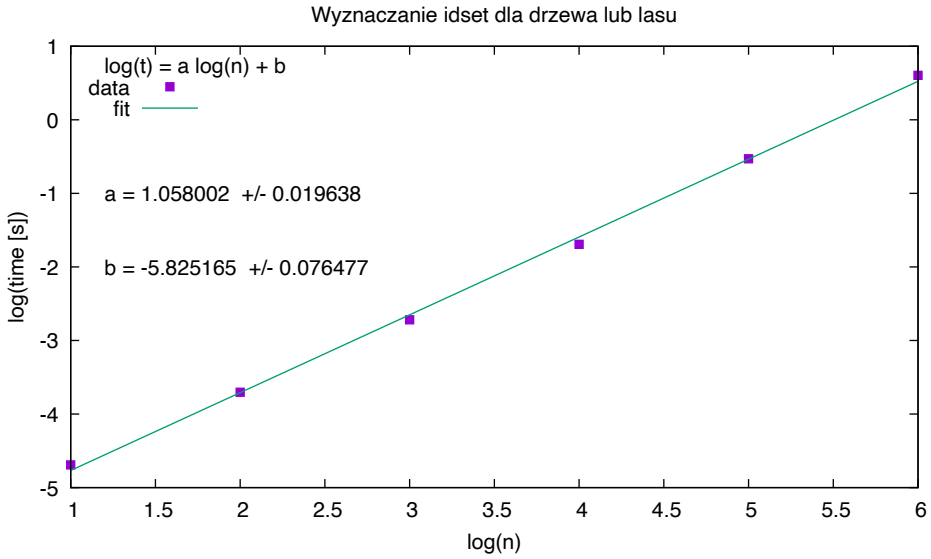
Algorytm z klasy WeightedDominatingSet1 został przetestowany na losowo wygenerowanych drzewach o różnych liczbach wierzchołków. Drzewa zostały wygenerowane za pomocą funkcji make_tree z klasy GraphFactory, należącej do pakietu graphtheory [11]. Do przydzielania wag wierzchołkom wykorzystano algorytm BipartiteGraphBFS z tego samego pakietu, który koloruje drzewo na dwa kolory, a następnie przypisuje wagi 1 lub 3 w zależności od koloru wierzchołka. Wynikiem testów jest wartość współczynnika dopasowania $a = 1.122(14)$ ukazana na wykresie A.4. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n)$. Pewna nadwyżka ponad wartość jeden wynika z działania metody _calc_weight, która wielokrotnie oblicza wagę zbiorów dominujących oraz jawnego trzymania zbiorów dominujących. Można tego uniknąć kosztem pewnego skomplikowania kodu: w wywołaniach rekurencyjnych łącznie ze zbiorami należy też przekazywać wagi zbiorów.

A.5. Test wyznaczania niezależnego zbioru dominującego z najmniejszą wagą dla drzew

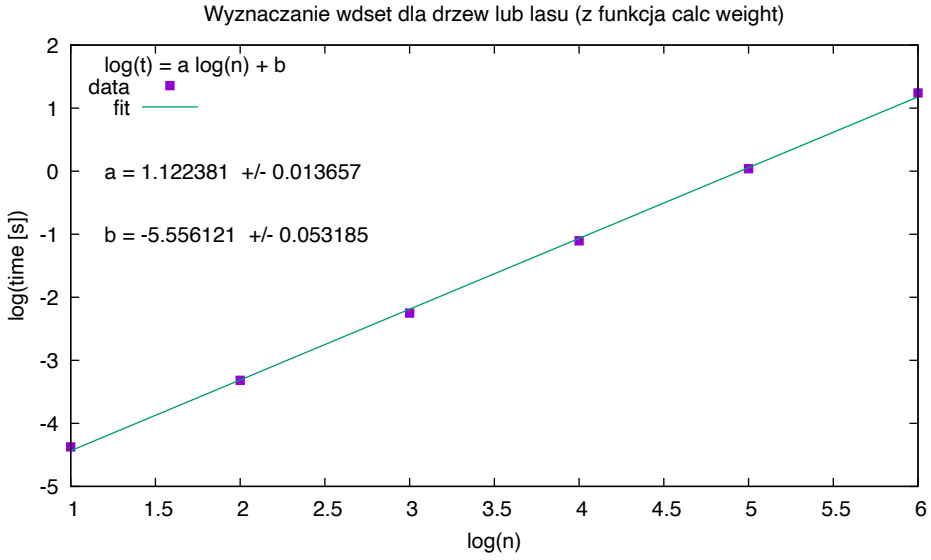
Algorytm z klasy TreeWeightedIndependentDominatingSet1 został przetestowany w identyczny sposób jak algorytm z klasy WeightedDominatingSet1. Wynikiem testów jest wartość współczynnika dopasowania $a = 1.082(14)$ ukazana na wykresie A.5. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n)$. Tak samo jak w przypadku algorytmu dla ważonych zbiorów dominujących dla drzew, pewna nadwyżka ponad wartość jeden wynika z działania metody _calc_weight, która wielokrotnie oblicza wagę zbiorów dominujących oraz jawnego trzymania zbiorów dominujących. Można tego uniknąć kosztem pewnego skomplikowania kodu: w wywołaniach rekurencyjnych łącznie ze zbiorami należy też przekazywać wagi zbiorów. Na wykresach A.6 i A.7 ukazano wyniki działania dwóch dodatkowo zrobionych wersji algorytmu, które różnią się obecnością funkcji _calc_weight aby zobaczyć wpływ tej funkcji na złożoność czasową.

A.6. Test wyznaczania najmniejszego zbioru dominującego dla grafów przedziałowych 2-drzewowych

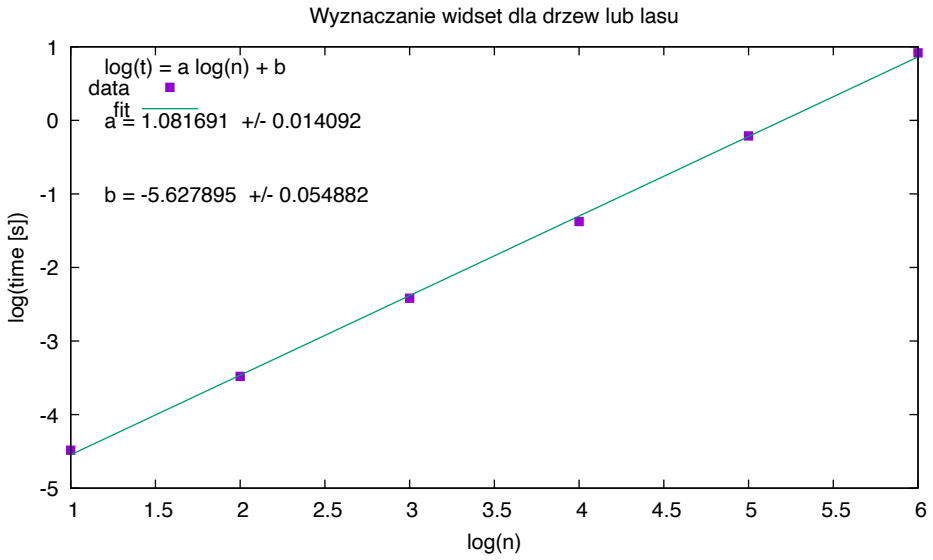
Algorytm z funkcji interval_minimum_dset2 został przetestowany na losowo wygenerowanych grafach 2-drzewowych w postaci podwójnej permutacji. Grafy przedziałowe zostały wygenerowane za pomocą funkcji make_2tree_interval



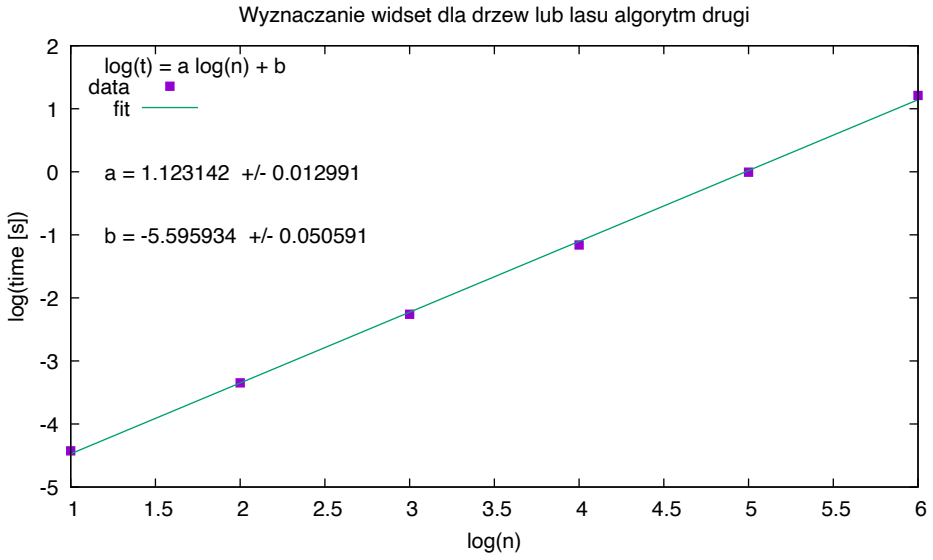
Rysunek A.3. Wykres pomiarów algorytmu znajdującego najmniejszy niezależny zbiór dominujący dla drzew i lasów. Wartość współczynnika dopasowania $a = 1.058(19)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n)$.



Rysunek A.4. Wykres pomiarów algorytmu znajdującego najmniejszy ważony zbiór dominujący dla losowo wygenerowanych drzew z użyciem klasy WeightedDominatingSet1 Wartość współczynnika dopasowania $a = 1.122(14)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n)$.



Rysunek A.5. Wykres pomiarów algorytmu znajdującego najmniejszy ważony niezależny zbiór dominujący dla losowo wygenerowanych drzew z użyciem klasy TreeWeightedIndependentDominatingSet1. Wartość współczynnika dopasowania $a = 1.082(14)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n)$. Pewna nadwyżka ponad wartość jeden wynika z działania metody _calc_weight, która wielokrotnie oblicza wagę zbiorów dominujących oraz jawnego trzymania zbiorów dominujących.



Rysunek A.6. Wykres pomiarów algorytmu znajdującego najmniejszy ważony niezależny zbiór dominujący dla losowo wygenerowanych drzew z użyciem klasy TreeWeightedIndependentDominatingSet2. Wartość współczynnika dopasowania $a = 1.123(13)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n)$. Pewna nadwyżka ponad wartość jeden wynika z działania metody _calc_weight, która wielokrotnie oblicza wagę zbiorów dominujących oraz jawnego trzymania zbiorów dominujących.

znajdującej się w module intervaltools z pakietu graphtheory. Funkcja ta tworzy grafy o małej liczbie krawędzi rzędu $O(n)$. Wynikiem testów jest wartość współczynnika dopasowania $a = 0.9899(53)$ ukazana na wykresie A.8. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n)$.

A.7. Test wyznaczania najmniejszego zbioru dominującego dla grafów przedziałowych k-drzewowych

Algorytm z funkcji interval_minimum_dset2 został przetestowany na losowo wygenerowanych grafach k-drzewowych. Grafy przedziałowe tym razem zostały wygenerowane za pomocą funkcji make_ktree_interval znajdującej się w module intervaltools z pakietu graphtheory. Funkcja ta tworzy grafy z klikami maksymalnymi zawierającymi $k+1$ wierzchołków. Jeżeli k będzie proporcjonalne do n, to otrzymamy liczbę krawędzi rzędu $O(n^2)$. Wynikiem testów jest wartość współczynnika dopasowania $a = 1.910(41)$ ukazana na wykresie A.9. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n^2)$.

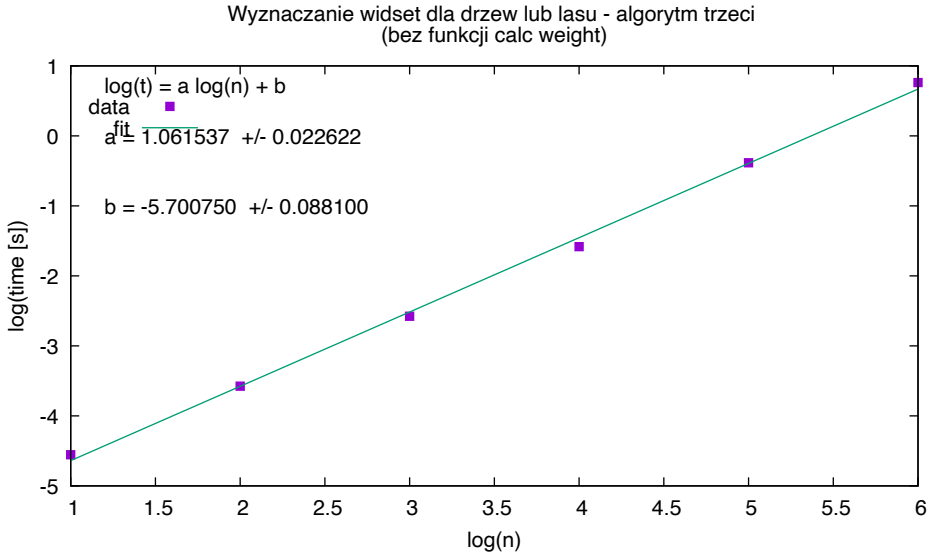
Algorytm z funkcji interval_minimum_dset2 został raz jeszcze przetestowany na losowo wygenerowanych grafach k-drzewowych. Wynikiem testów jest wartość współczynnika dopasowania $a = 0.965(18)$ ukazana na wykresie A.10. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n + m)$, czyli liniową względem rozmiaru danych wejściowych (rozmiaru grafu).

A.8. Test wyznaczania zbioru dominującego o najmniejszej wadze dla grafów przedziałowych 2-drzewowych

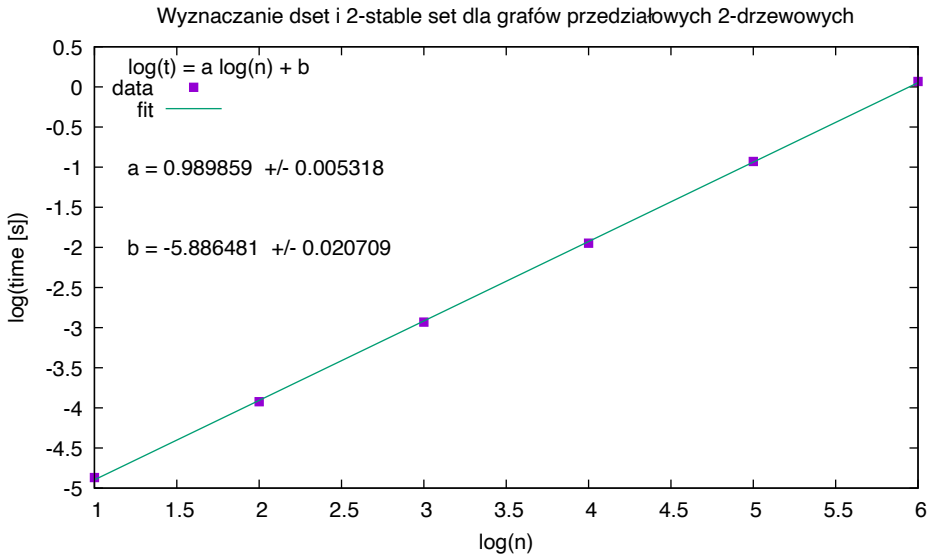
Algorytm z funkcji interval_minimum_weight_dset został przetestowany na losowo wygenerowanych grafach 2-drzewowych w postaci podwójnej permutacji. Grafy przedziałowe zostały wygenerowane za pomocą funkcji make_2tree_interval znajdującej się w module intervaltools z pakietu graphtheory. Funkcja ta tworzy grafy o małej liczbie krawędzi rzędu $O(n)$. Każdemu wierzchołkowi przypisane zostaly wagi: parzystym wierzchołkom przypisano wagę 1, a nieparzystym wierzchołkom przypisano wagę 2. Wynikiem testów jest wartość współczynnika dopasowania $a = 0.9856(54)$ ukazana na wykresie A.11. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n + m)$.

A.9. Test wyznaczania najmniejszego zbioru dominującego dla grafów permutacji

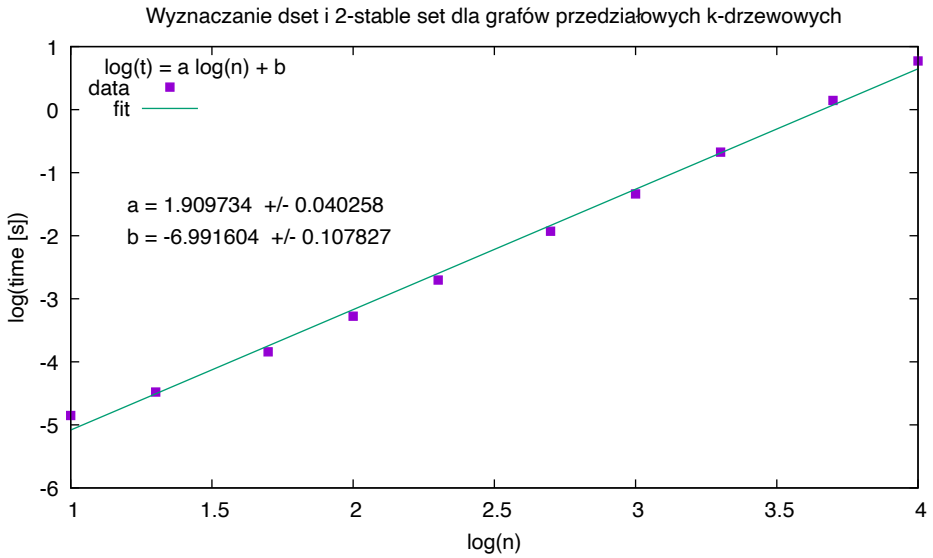
Algorytm z funkcji permutation_minimum_dset został przetestowany na losowo wygenerowanych permutacjach o różnych długościach. Wynikiem testów jest wartość współczynnika dopasowania $a = 1.958(15)$ ukazana na wykresie A.12. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n^2)$. Algorytm został również przetestowany na drabinie. Wynikiem testów jest



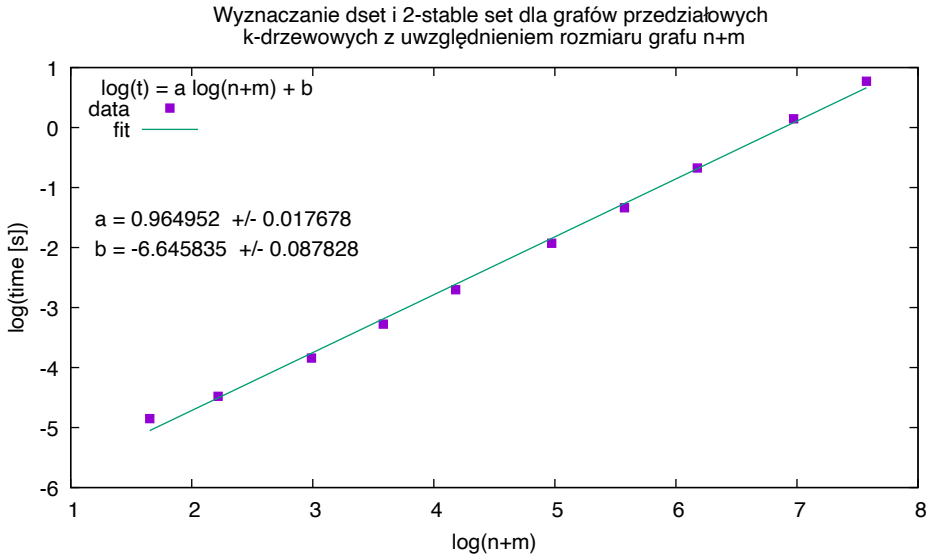
Rysunek A.7. Wykres pomiarów algorytmu znajdującego najmniejszy ważony niezależny zbiór dominujący dla losowo wygenerowanych drzew z użyciem klasy TreeWeightedIndependentDominatingSet3. Wartość współczynnika dopasowania $a = 1.062(23)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n)$. Pewna nadwyżka ponad wartość jeden wynika z jawnego trzymania zbiorów dominujących.



Rysunek A.8. Wykres pomiarów algorytmu znajdującego najmniejszy zbiór dominujący dla grafów przedziałowych 2-drzewowych z użyciem funkcji interval_minimum_dset2. Wartość współczynnika dopasowania $a = 0.9899(53)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n)$.



Rysunek A.9. Wykres pomiarów algorytmu znajdującego najmniejszy zbiór dominujący dla grafów przedziałowych k-drzewowych z użyciem funkcji interval_minimum_dset2. Wartość współczynnika dopasowania $a = 1.910(41)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n^2)$ dla grafów gęstych.



Rysunek A.10. Wykres pomiarów algorytmu znajdującego najmniejszy zbiór dominujący dla tych samych grafów przedziałowych k-drzewowych co na rysunku A.9, ale z uwzględnieniem rozmiaru grafu $n + m$. Wartość współczynnika dopasowania $a = 0.965(18)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n+m)$, czyli liniowej względem rozmiaru danych wejściowych.

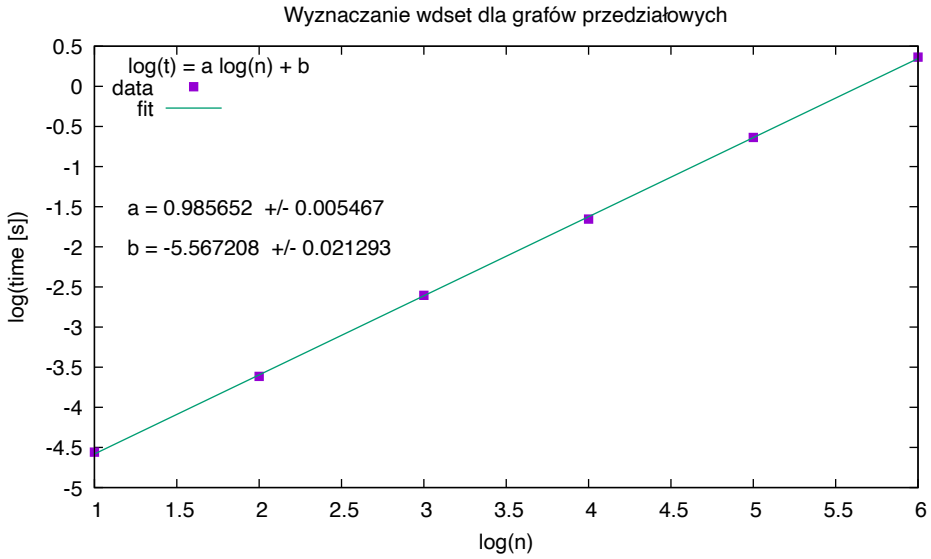
wartość współczynnika dopasowania $a = 2.390(64)$ ukazana na wykresie A.13. Pewna nadwyżka ponad $O(n^2)$ wynika z faktu, że sklejanie zbiorów jest kosztowne i daje pewien narzut.

A.10. Test wyznaczania najmniejszego ważonego zbioru dominującego dla grafów permutacji

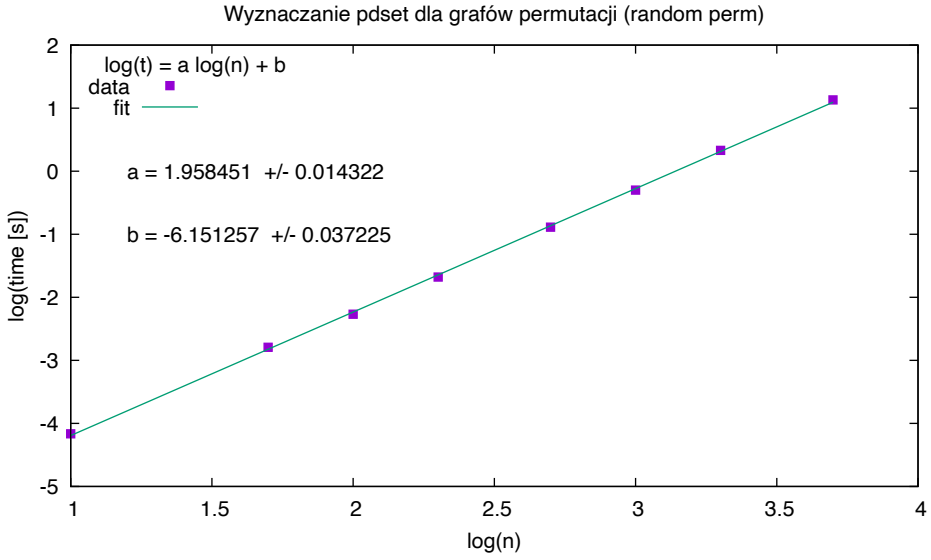
Algorytm z funkcji permutation_weighted_minimum_dset został przetestowany na losowo wygenerowanych permutacjach o różnych długościach. Do przypisywania wag wierzchołkom została przygotowana funkcja, która parzystym wierzchołkom przypisuje wagę 1, a nieparzystym 3. Wynikiem testów jest wartość współczynnika dopasowania $a = 2.965(16)$ ukazana na wykresie A.14. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n^3)$.

A.11. Test wyznaczania najmniejszego ważonego niezależnego zbioru dominującego dla grafów permutacji

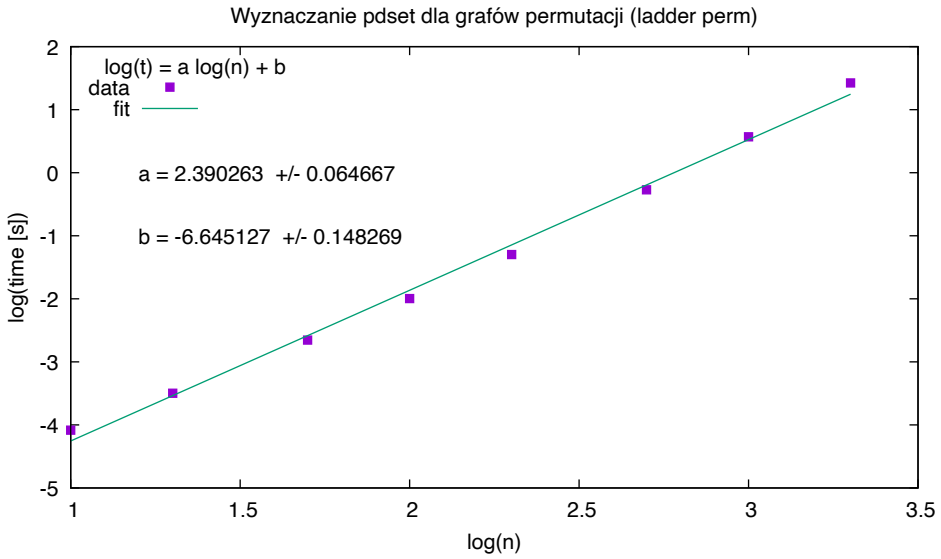
Algorytm z funkcji permutation_weighted_minimum_independent_dset2 został przetestowany na losowo wygenerowanych permutacjach o różnych długościach. Do przypisywania wag wierzchołkom została przygotowana funkcja, która parzystym wierzchołkom przypisuje wagę 1, a nieparzystym 3. Wynikiem testów jest wartość współczynnika dopasowania $a = 1.909(32)$ ukazana na wykresie A.15. Potwierdza ona teoretyczną złożoność obliczeniową wynoszącą $O(n^2)$.



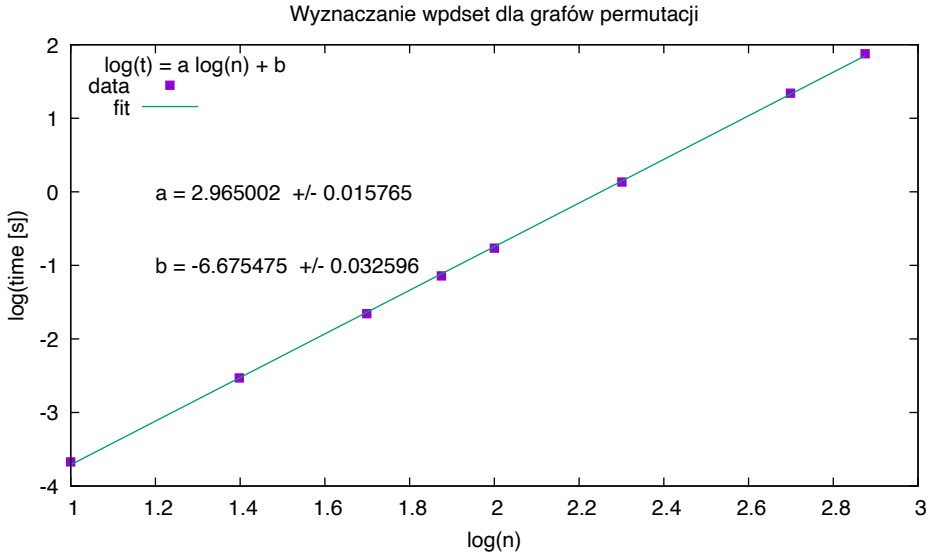
Rysunek A.11. Wykres pomiarów algorytmu znajdującego zbiór dominujący o najmniejszej wadze dla grafów przedziałowych 2-drzewowych z użyciem funkcji interval_minimum_weight_dset. Wartość współczynnika dopasowania $a = 0.9856(54)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n+m)$.



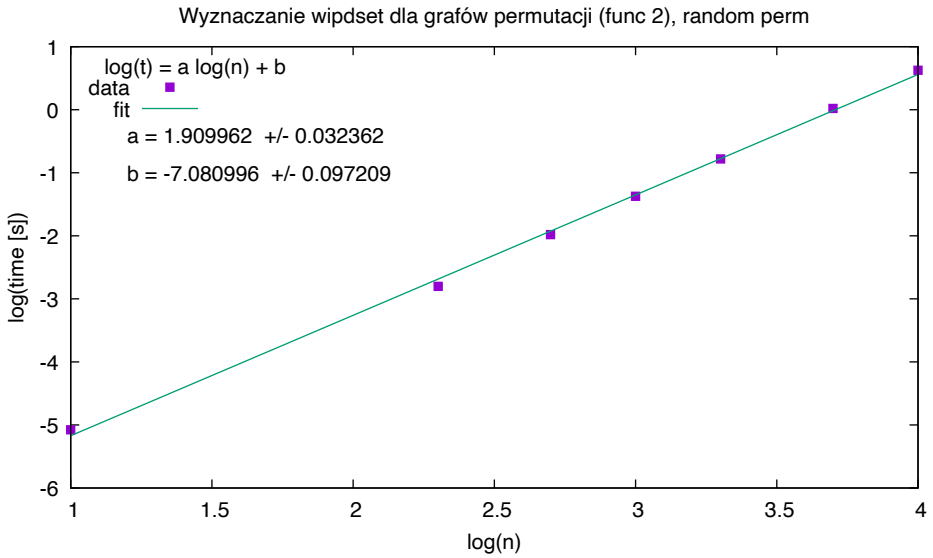
Rysunek A.12. Wykres pomiarów algorytmu znajdującego najmniejszy zbiór dominujący dla losowo wygenerowanych permutacji o różnych długościach. Wartość współczynnika dopasowania $a = 1.958(15)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n^2)$.



Rysunek A.13. Wykres pomiarów algorytmu znajdującego najmniejszy zbiór dominujący dla grafówpermutacji drabiny o różnych długościach. Wartość współczynnika dopasowania $a = 2.390(64)$ co wskazuje na pewien narzut ponad $O(n^2)$ spowodowany sklejaniem zbiorów.



Rysunek A.14. Wykres pomiarów algorytmu znajdującego najmniejszy ważony zbiór dominujący dla losowo wygenerowanych permutacji o różnych długościach. Wartość współczynnika dopasowania $a = 2.965(16)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n^3)$.



Rysunek A.15. Wykres pomiarów algorytmu znajdującego najmniejszy ważony niezależny zbiór dominujący dla losowo wygenerowanych permutacji o różnych długościach. Wartość współczynnika dopasowania $a = 1.909(32)$ jest potwierdzeniem teoretycznej złożoności obliczeniowej $O(n^2)$.

Bibliografia

[1] Wikipedia, Dominating set, 2025, `https://en.wikipedia.org/wiki/Dominating_set`.

[2] Teresa W. Haynes, Stephen T. Hedetniemi, Peter J. Slater, *Fundamentals of Domination in Graphs*, Marcel Dekker, Inc., New York 1998. str. 16-25.

[3] My T. Thai, Feng Weng, Dhruva Chakrabarty, Zhuo Feng, *Connected Dominating Sets in Wireless Networks with Different Transmission Ranges*, IEEE Transactions on Mobile Computing, Vol. 6, No. 7, 2007.

[4] Gerard Jennhwa Chang, *Algorithmic Aspects of Domination in Graphs*, National Taiwan University, Taiwan 2011, str. 1-4.

[5] Sambor Guze, *An application of the selected graph theory domination concepts to transportation networks modelling*, Gdynia Maritime University, Gdynia 2017.

[6] Thang N. Dinh, D. T. Nguyen and My T. Thai, *A Unified Approach for Domination Problems on Different Network Topologies*, W: P. M. Pardalos, D.-Z. Du, R. L. Graham (eds), Handbook of Combinatorial Optimization, Second Edition. Springer, New York, 2013.

[7] Jose C. Nacher, Tatsuya Akutsu, *Dominating scale-free networks with variable scaling exponent: heterogeneous networks are not difficult to control*, New Journal of Physics, Volume 14, 2012.

[8] Stephen Eubank, Hasan Guclu, V. S. Anil Kumar, Madhav V. Marathe, Aravind Srinivasan, Zoltan Toroczkai, Nan Wang, *Modelling disease outbreaks in realistic urban social networks*, Nature 429, 180-184 (2004).

[9] Robert B. Allan and Renu Laska, *On domination and independent domination numbers of a graph*, Discrete Mathematics, Volume 23, Issue 2, 1978.

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

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

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

[13] Jonathan L. Gross, Jay Yellen, Ping Zhang, *Handbook of Graph Theory, Second Edition*, CRC Press, Boca Raton 2014.

[14] Reinhard Diestel, *Graph Theory*, Springer, New York 2000.

[15] Edward A. Bender, S. Gill Williamson, *Lists, Decisions and Graphs*, 2010.

[16] S. T. Hedetniemi, R. C. Laskar, *Bibliography on domination in graphs and some basic definitions of domination parameters* W: *In Annals of discrete mathematics (Vol. 48, pp. 257-277).* 1990, Elsevier.

[17] E. Sampathkumar, H. B. Walikar, *The connected domination of a graph* W: *Math. Phys. Sci. 13 No.6.* 1979, India.

[18] Mahalingam, Gayathri, *Connected Domination in Graphs.* USF Tampa Graduate Theses and Dissertations (2005). `https://digitalcommons.usf.edu/etd/2961`

[19] Chung-Shou Liao, Gerard J. Chang, *Algorithmic Aspects of Domination in Graphs*, Taiwanese Journal Of Mathematics 6(3), 415-420 (2002).
[20] P. J. Slater, *R-domination in graphs*, Association for Computing Machinery, New York, 1976.
[21] Gayla S. Domke, Johannes H. Hattingh, Stephen T. Hedetniemi, Renu C. Laskar, Lisa R. Markus, *Restrained domination in graphs*, Discrete Mathematics 203, 61-69 (1999).
[22] Aleksander Krawczyk, *Badanie grafów Halina z językiem Python*, Uniwersytet Jagielloński, Kraków 2016.
[23] Małgorzata Olak, *Badanie grafów cięciwowych z językiem Python*, Uniwersytet Jagielloński, Kraków 2017.
[24] G. J. Chang, *Algorithmic Aspects of Domination in Graphs*. W: P. M. Pardalos, D.-Z. Du, R. L. Graham (eds), Handbook of Combinatorial Optimization, Second Edition. Springer, New York, 2013.
[25] Min-Jen Jou and Jenq-Jong Lin, *Algorithms for Weighted Domination Number and Weighted Independent Domination Number of a Tree*, International Journal of Contemporary Mathematical Sciences 13, 133-140 (2018).
[26] Konrad Gałuszka, *Badanie grafów szeregowo-równoległych z językiem Python*, Uniwersytet Jagielloński, Kraków 2018.
[27] Maciej Mularski, *Badanie grafów przedziałowych z językiem Python*, Uniwersytet Jagielloński, Kraków 2023.
[28] G. Ramalingam, C. Pandu Rangan, *A unified approach to domination problems in interval graphs*, Information Processing Letters 27, 271-274 (1988).
[29] Kellogg S. Booth and J. Howard Johnson, *Dominating Sets in Chordal Graphs*, SIAM Journal on Computing Vol. 11, No. 1, 1982.
[30] Maciej Niezabitowski, *Dekompozycja drzewowa w teorii grafów*, Uniwersytet Jagielloński, Kraków 2019.
[31] Guillermo Durán, Luciano N. Grippo, Martín D. Safe, *Structural results on circular-arc graphs and circle graphs: A survey and the main open problems*, 2013.
[32] Wen-Lian Hsu and Kuo-Hui Tsai, *Linear time algorithms on circular-arc graphs*, Information Processing Letters 40, 123–129 (1991).
[33] Oliwia Gil, *Grafy łuków na okręgu*, Uniwersytet Jagielloński, Kraków 2025.
[34] C. Rhee, Y. D. Liang, S. K. Dhall, S. Lakshmivarahan, *An $O(N+M)$-time algorithm for finding a minimum-weight dominating set in a permutation graph*, SIAM Journal on Computing 25(2), 404-419 (1996).
[35] Martin Farber, J. Mark Keil, *Domination in permutation graphs*, Journal of Algorithms 6, 309-321 (1985).
[36] Albert Surmacz, *Badanie grafów kołowych z językiem Python*, Uniwersytet Jagielloński, Kraków 2023.
[37] Keil J. M., *The complexity of domination problems in circle graphs*, Discrete Applied Mathematics 42, 51-63 (1993).
[38] Angelika Siwek, *Badanie grafów bez trójek asteroidalnych z językiem Python*, Uniwersytet Jagielloński, Kraków 2024.
[39] D. Kratsch, *Domination and total domination in asteroidal triple-free graphs*, Discrete Appl. Math. 99, 111-123 (2000).
[40] Wikipedia, Planar graph, 2025, `https://en.wikipedia.org/wiki/Planar_graph`.
[41] Andreas Brandstadt and Dieter Kratsch, *On domination problems for permutation graphs and other graphs*, Theoretical Computer Science 54, 181-197 (1987).

[42] Python Programming Language - Official Documentation Website, https://docs.python.org/3/library/unittest.html.
[43] Python Programming Language - Official Documentation Website, https://docs.python.org/3/library/timeit.html.