



WYDZIAŁ ELEKTRONIKI,
TELEKOMUNIKACJI
I INFORMATYKI

Imię i nazwisko studenta: Paulina Brzecka
Nr albumu: 184701
Poziom kształcenia: studia drugiego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność: Algorytmy i technologie internetowe

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Analiza algorytmów dla dominowania rzymskiego słabo spójnego

Tytuł pracy w języku angielskim: Analysis of algorithms for weakly connected Roman domination

Opiekun pracy: dr inż. Joanna Raczek

STRESZCZENIE

Praca opisuje problem dominowania rzymskiego słabo spójnego. Przedstawia definicję problemu, jego genezę historyczną oraz złożoność obliczeniową. Celem pracy jest analiza istniejących i autorskich algorytmów znajdujących funkcje dominujące rzymskie słabo spójne, ich analiza oraz porównanie w kontekście możliwego praktycznego zastosowania.

Zakres pracy obejmuje systematyczny przegląd literatury, analizę istniejących algorytmów, implementację autorskich rozwiązań oraz ocenę ich skuteczności na wybranych przypadkach testowych. Badane algorytmy wyznaczają zarówno minimalną wartość funkcji dominowania rzymskiego słabo spójnego, jak i też funkcje dominujące rzymskie słabo spójne, niekoniecznie minimalne. Wszystkie jednak wyznaczają prawidłowe, względem definicji funkcje dominujące rzymskie słabo spójne.

Zastosowane metody badawcze obejmowały programowanie liniowe, pomiary czasów oraz wizualizację wyników.

Wyniki wskazują na to, że w zależności od charakteryzacji i cech grafu istnieje możliwość dobrania odpowiedniego algorytmu, rozwiązującego problem w rozsądny czasie i o dobrej jakości rozwiązania.

Ważnym wnioskiem jest przedstawienie teoretycznego potencjału praktycznego zastosowania algorytmów znajdujących funkcje dominujące rzymskie słabo spójne.

Słowa kluczowe: algorytmy, optymalizacja, funkcja dominowania rzymskiego słabo spójnego, liczba dominowania rzymskiego słabo spójnego.

ABSTRACT

The paper describes the problem of weakly connected Roman domination. It presents the definition of the problem, its historical genesis and computational complexity. The aim of the paper is to analyze existing and original algorithms for finding weakly connected Roman domination functions, their analysis and comparison in the context of possible practical application.

The scope of the paper includes a systematic review of the literature, analysis of existing algorithms, implementation of original solutions and evaluation of their effectiveness on selected test cases. The tested algorithms determine both the minimal value of the weakly connected Roman domination function and the weakly connected Roman domination functions, not necessarily minimal. However, all of them determine the correct, in terms of the definition, weakly connected Roman domination function.

The applied research methods included linear programming, time measurements and visualization of results.

The results indicate that depending on the characterization and features of the graph, it is possible to select an appropriate algorithm that solves the problem in a reasonable time and with good quality solutions.

An important conclusion is the presentation of the theoretical potential of practical application of algorithms for finding weakly connected Roman domination functions.

Keywords: algorithms, optimization, weakly connected Roman domination function, weakly connected Roman domination number.

SPIS TREŚCI

Wykaz ważniejszych oznaczeń i skrótów	7
1 Wstęp i cel pracy	8
1.1 Cel pracy	8
1.2 Zakres pracy	8
2 Wprowadzenie teoretyczne	9
2.1 Geneza historyczna	9
2.2 Definicja problemu	9
2.3 Złożoność obliczeniowa	10
2.4 Przegląd literatury	12
2.5 Metody badań	13
3 Badane algorytmy	16
3.1 Wprowadzenie	16
3.2 Algorytm Brute Force	16
3.2.1 Działanie	16
3.2.2 Poprawność	16
3.2.3 Złożoność i wydajność	17
3.2.4 Pseudokod	18
3.3 Algorytm liniowy dla drzew	19
3.3.1 Działanie	19
3.3.2 Poprawność	22
3.3.3 Złożoność i wydajność	23
3.3.4 Pseudokod	23
3.4 Algorytm programowania liniowego I	25
3.4.1 Działanie	25
3.4.2 Poprawność	26
3.4.3 Złożoność i wydajność	27
3.4.4 Pseudokod	27
3.5 Algorytm programowania liniowego II	29
3.5.1 Działanie	29
3.5.2 Poprawność	30
3.5.3 Złożoność i wydajność	31
3.5.4 Pseudokod	32
3.6 Algorytm mrówkowy	34

3.6.1	Działanie	34
3.6.2	Poprawność	35
3.6.3	Złożoność i wydajność	35
3.6.4	Pseudokod	35
3.7	Algorytm aproksymacyjny	38
3.7.1	Działanie	38
3.7.2	Poprawność	39
3.7.3	Złożoność i wydajność	39
3.7.4	Pseudokod	40
3.8	Algorytm zachłanny	41
3.8.1	Działanie	41
3.8.2	Poprawność	41
3.8.3	Złożoność i wydajność	42
3.8.4	Pseudokod	42
4	Wyniki	44
4.1	Wprowadzenie	44
4.2	Hiperparametry algorytmu mrówkowego	44
4.3	Wyniki działania algorytmów	46
4.3.1	Grafy rzadkie	47
4.3.2	Grafy gęste	49
4.3.3	Drzewa	52
4.3.4	Grafy bezskalowe	55
4.4	Wyniki algorytmów niedokładnych	59
4.4.1	Algorytm mrówkowy	59
4.4.2	Algorytm zachłanny	60
4.4.3	Algorytm aproksymacyjny	61
4.4.4	Porównanie algorytmów przybliżonych	63
5	Zastosowania praktyczne	64
5.1	Wprowadzenie	64
5.2	Rozmieszczenie zabezpieczeń sieci energetycznych	64
5.3	Rozmieszczenie agentów monitorujących oszustwa w internetowej sieci społecznościowej	67
6	Podsumowanie i wnioski	72
6.1	Podsumowanie wyników	72
6.2	Wnioski i dalsze kierunki badań	73
Spis rysunków		76

WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

- $WCRDP$ – Weakly Connected Roman Domination Problem - problem dominowania rzymskiego słabo spójnego.
- $WCRDS$ – Weakly Connected Roman Domination Set - zbiór dominujący rzymski słabo spójny.
- $WCRDF$ – Weakly Connected Roman Domination Function - funkcja dominowania rzymskiego słabo spójnego.
- $\gamma_R^{wc}(G)$ – Weakly Connected Roman Domination Number - liczba dominowania rzymskiego słabo spójnego.

ROZDZIAŁ 1. WSTĘP I CEL PRACY

Tematem pracy jest analiza algorytmów znajdujących funkcje dominujące rzymskie słabo spójne w grafach. Problem znajdywania liczby dominowania rzymskiego słabo spójnego jest problemem NP-trudnym. Nie są znane zatem dokładne algorytmy rozwiązuające i weryfikujące problem w czasie wielomianowym. Wersja decyzyjna tego problemu jest NP-zupełna. Dodatkową motywacją jest widoczny potencjał zastosowania algorytmu w rozwiązywaniu praktycznych problemów. Dlatego niniejsza praca dokonuje analizy istniejących w literaturze i proponowanych algorytmów znajdowania minimalnej liczby dominowania rzymskiego słabo spójnego, jak i wyłącznie znajdujących funkcję dominowania rzymskiego słabo spójnego, w celu znalezienia możliwie skutecznych rozwiązań oraz zastosowań. Całe rozważania dotyczą grafów spójnych.

1.1 Cel pracy

Celem badawczym pracy jest opisanie istniejących i opracowanie własnych algorytmów znajdujących funkcje dominujące rzymskie słabo spójne, ich analiza oraz porównanie skuteczności i wyciągnięcie wniosków na temat możliwości ich praktycznego zastosowania.

1.2 Zakres pracy

W ramach pracy dokonano systematycznego przeglądu literatury. W literaturze szeroko zdefiniowano ten problem oraz jego złożoność obliczeniową. Zaproponowano również kilka algorytmów rozwiązujących ten problem - znajdujących minimalną liczbę dominowania rzymskiego słabo spójnego - wykorzystujących m.in. programowanie liniowe - jak i znajdujących funkcję dominowania rzymskiego słabo spójnego (niekoniecznie minimalną). Na podstawie znalezionej literatury zaimplementowane zostały dwa algorytmy programowania liniowego oraz $2(1 + \epsilon)(1 + \ln(\Delta - 1))$ -aproksymacyjny. W ramach własnej pracy, zaimplementowano algorytm brute force, zachłanny, liniowy dla drzew oraz mrówkowy. Niniejsza praca przedstawia problem dominowania rzymskiego słabo spójnego oraz jego złożoność obliczeniową. Opisuje wymienione algorytmy, porównuje je pod kątem wydajności, poprawności oraz czasu działania. Testy przeprowadzono na różnych klasach grafów: gęstych, rzadkich, drzewach oraz bezskalowych. Następnie dokonano analizy potencjalnych praktycznych zastosowań, w których algorytmiczne rozwiązanie tego problemu byłoby przydatne. Na koniec dokonano podsumowania wszystkich wyników dotyczących przeprowadzonych rozważań.

ROZDZIAŁ 2. WPROWADZENIE TEORETYCZNE

2.1 Geneza historyczna

Problem swoją nazwę zawdzięcza imperium rzymskiemu. Po raz pierwszy został opisany w artykule „Defend Roman Empire!”[1]. Obrazuje on problem następująco: Każdy wierzchołek grafu reprezentuje pewną lokalizację (miasto, wzgórze) w Imperium Rzymskim. Lokalizacja (wierzchołek v) jest niechroniona, jeśli nie stacjonują w niej żadne legiony wojska ($f(v) = 0$) oraz chroniona jeśli ($f(v) \in 1, 2$). Wartości te oznaczają liczbę legionów stacjonujących w danej lokalizacji. Niechroniona lokalizacja może być ochroniona poprzez wysłanie legionu stacjonującego w lokalizacji sąsiadującej. W czwartym wieku cesarz Konstantyn Wielki wydał dekret zakazujący przemieszczenia się legionu do lokalizacji sąsiadującej, jeśli sprawi to, że aktualna lokalizacja pozostanie niechroniona. Dlatego, żeby móc wysłać legion do sąsiedniej lokacji, w aktualnej muszą stacjonować dwa legiony. Oczywiście, cesarzowi zależało na jak najmniejszych kosztach utrzymania legionów, a zatem, żeby było ich jak najmniej [2]. Problem możemy zatem zinterpretować grafowo, gdzie lokalizacje to wierzchołki grafu, liczba legionów to wartości na wierzchołkach. W celu zachowania ciągłości efektywnej komunikacji między legionami wymagana jest słabo spójność zbioru dominującego.

2.2 Definicja problemu

Jako, że problem nie jest powszechnie znany, należy go zdefiniować, wprowadzając następujące pojęcia [2]:

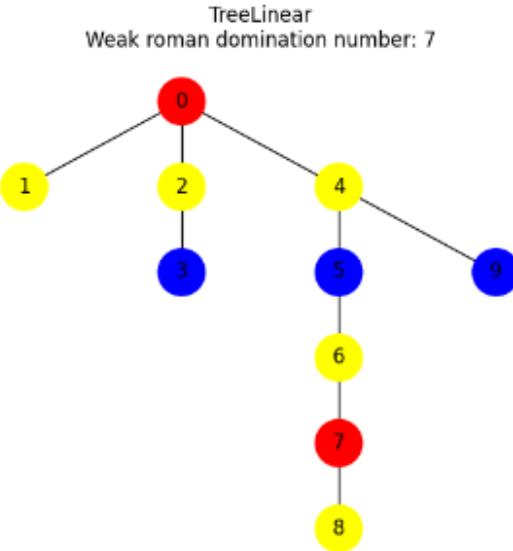
Definicja 1 Funkcja dominująca rzymska zdefiniowana jest dla grafu $G = (V, E)$, gdzie $f : V \rightarrow \{0, 1, 2\}$ spełnia warunek, że dla każdego wierzchołka u , dla którego $f(u) = 0$ jest sąsiadem przynajmniej jednego wierzchołka v , dla którego $f(v) = 2$.

Definicja 2 Dominujący zbiór $D \subseteq V$ jest zbiorem dominującym słabo spójnym grafu G jeśli graf $(V, E \cap (D \times V))$ jest spójny.

Definicja 3 Funkcja dominująca rzymska słabo spójna na grafie G będzie funkcją dominującą rzymską, taką, że zbiór $\{u \in V : f(u) \in \{1, 2\}\}$ jest jednocześnie zbiorem dominującym słabo spójnym.

Definicja 4 Wagę funkcji dominującej rzymskiej słabo spójnej definiujemy jako $f(V) = \sum_{u \in V} f(u)$. Minimalną wartość tej funkcji nazywamy liczbą dominowania rzymskiego słabo spójnego - $\gamma_R^{wc}(G)$.

Definicja 5 Jeśli graf G ma następującą własność: $\gamma_R^{wc}(G) = 2\gamma_{wc}(G)$, to nazywa się go grafem rzymskim słabo spójnym.



Rysunek 2.1: Przykład grafu rzymskiego słabo spójnego.

Od tego momentu zbiór dominujący rzymski słabo spójny będzie definiowany jako *WCRDS*, funkcja dominowania rzymskiego słabo spójnego to *WCRDF*. Problem dominowania rzymskiego słabo spójnego będzie oznaczany jako *WCRDP*.

2.3 Złożoność obliczeniowa

Poniższy szkic dowodu został przedstawiony w artykule: „Weakly connected Roman domination in graphs”[2]. Za pomocą redukcji alfa problem WCRDF zostanie sprowadzony do NP-zupełnego problemu dokładnego pokrycia 3 zbiorami (Exact Cover by 3-Sets, X3C) [3], czym zostanie udowodniona NP-zupełność wersji decyzyjnej WCRDF. Naturalnie, wersja minimalizacyjna - poszukiwanie $\gamma_R^{wc}(G)$ jest wtedy NP-trudne.

Dane: Spójny graf G oraz dodatnia liczba całkowita k .

Pytanie: Czy istnieje słabo spójna funkcja dominacji rzymskiej w grafie G o wagie co najwyżej k ?

Problem WCRDF jest NP-zupełny, nawet dla grafów podziału (graf, w którym można wydzielić klikę oraz zbiór niezależny) oraz grafów dwudzielnych.

Szkic dowodu:

Problem WCRDF należy do klasy NP, ponieważ dla danego przyporządkowania $f : V(G) \rightarrow \{0, 1, 2\}$ można w czasie wielomianowym sprawdzić, czy f ma wagę co najwyżej k oraz czy spełnia warunki WCRDF.

Redukcja jest przeprowadzana z problemu dokładnego pokrycia 3-zbiorami (Exact Cover by 3-Sets, X3C). Dla danej instancji $X = \{x_1, \dots, x_{3q}\}$ i rodziny zbiorów $\mathcal{C} = \{C_1, \dots, C_m\}$, gdzie każdy $C_j \subseteq X$ oraz $|C_j| = 3$, konstruowany jest graf podziału G .

Dla każdego elementu $x_i \in X$ oraz zbioru $C_j \in \mathcal{C}$ tworzone są wierzchołki. Dodaje się krawędź między x_i a C_j wtedy i tylko wtedy, gdy $x_i \in C_j$. Dodatkowo wierzchołki odpowiadające

zbiorom C_j tworzą klikę K_m (poprzez połączenie tych wierzchołków krawędziami). Przyjęto: $k = 2q$.

Można wykazać, że \mathcal{C} zawiera dokładne pokrycie zbioru X wtedy i tylko wtedy, gdy graf G posiada słabo spójną funkcję dominacji rzymskiej o wadze co najwyżej k .

Przykładową konstrukcję można zdefiniować następująco:

$$X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$$

$$\mathcal{C} = \left\{ \begin{array}{l} C_1 = \{x_1, x_2, x_3\}, \\ C_2 = \{x_2, x_4, x_5\}, \\ C_3 = \{x_4, x_5, x_6\} \end{array} \right\}$$

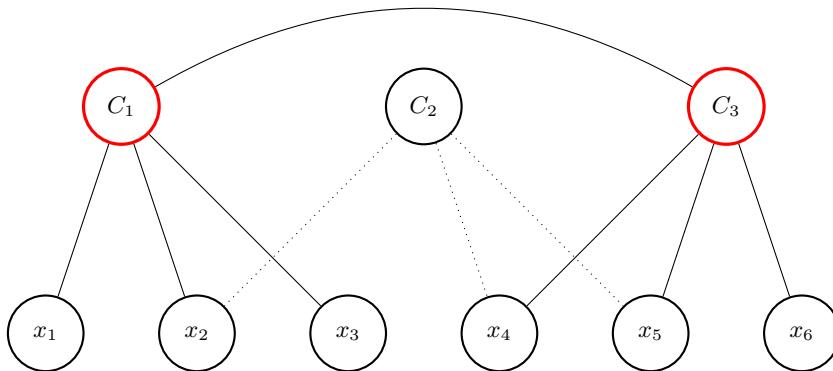
$$C_1 \cup C_3 = \{x_1, x_2, x_3, x_4, x_5, x_6\} = X$$

$$q = 2$$

$$k = 2q = 4$$

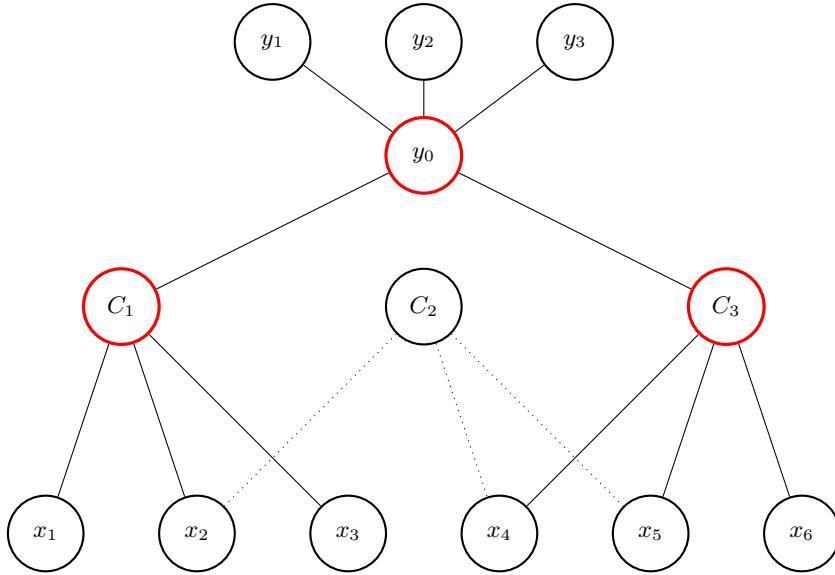
Zgodnie z konstrukcją, wierzchołki x zostaną połączone z wierzchołkami C , zgodnie z $x_i \in C_j$. W tym przypadku liczba zbiorów dokładnego pokrycia 3-zbiorami wynosi 2 i są to zbiory C_1 oraz C_2 . Je również należy połączyć krawędzią. Wierzchołki należące do WCRDS to C_1 oraz C_2 . Skonstruowany graf G składa się z wierzchołków zbioru X i C_1 oraz C_2 . Zgodnie z WCRDF, nadanie wierzchołkom C_1 oraz C_2 wartości 2 spełnia jego definicję.

Poniżej umieszczono rysunek poglądowy (na czerwono wierzchołki należące do WCRDS, a wierzchołek C_2 nie należy do grafu G , istnieje na rysunku tylko dla celów poglądowych):



Aby uzyskać wynik również dla grafów dwudzielnych, należy zmodyfikować konstrukcję: zamiast dodawać krawędzie między wierzchołkami C_j , dodajemy cztery nowe wierzchołki y_0, y_1, y_2, y_3 oraz krawędzie y_0y_1, y_0y_2, y_0y_3 i y_0C_j dla każdego j . Ustawiamy wtedy $k = 2q + 2$.

Konstrukcję dla przykładowego grafu dwudzielnego przedstawia poniższy rysunek. k będzie wynosiło 6.



2.4 Przegląd literatury

W 2004 roku zdefiniowano formalnie funkcję dominowania rzymskiego (RDF) [4] i od tego czasu w wielu publikacjach analizowano właściwości tego parametru oraz inne wersje dominowania rzymskiego, m.in. silną, mieszaną, totalną dominację rzymską.

Henning i Hedetniemi w 2003 roku udowodnili, że problem słabego dominowania rzymskiego jest NP-zupełny w wersji decyzyjnej, nawet dla grafów dwudzielnych [5].

W artykule z 2019 roku [2] wprowadzono pojęcie słabo spójnego dominowania rzymskiego. Zdefiniowano WCRDF oraz $\gamma_R^{wc}(G)$ oraz następujące właściwości:

- Dla każdego spójnego grafu G zachodzi:

$$\gamma_{wc}(G) \leq \gamma_R^{wc}(G) \leq 2\gamma_{wc}(G)$$

co oznacza, że liczba słabo spójnej dominacji rzymskiej jest ograniczona od dołu przez klasyczną liczbę słabo spójnej dominacji oraz od góry przez jej podwojenie.

- Dla spójnego grafu G , $\gamma_{wc}(G) = \gamma_R^{wc}(G)$ zachodzi wtedy i tylko wtedy, gdy $G = K_1$.
- Dla dowolnego spójnego grafu G o n wierzchołkach:

$$\gamma_R^{wc}(G) \leq n$$

z równością wtedy i tylko wtedy, gdy $G \in \{K_1, K_2\}$.

- Jeżeli $\gamma_R^{wc}(G) < n$ oraz $f = (V_0, V_1, V_2)$ jest funkcją minimalną WCRDF, to zachodzi:

$$|V_0| > 0 \quad \text{oraz} \quad |V_2| > 0$$

czyli co najmniej jeden wierzchołek musi mieć wartość 0, a co najmniej jeden wartość 2.

Dodatkowo pokazano, że problem ten jest NP-zupełny, nawet dla grafów dwudzielnych.

W 2021 rozszerzono wiedzę z poprzedniej publikacji o charakteryzację słabo spójnych drzew

rzymskich oraz pokazano, że sprawdzenie, czy graf dwudzielny jest rzymski słabo spójny jest co-NP-trudne [6].

W 2021 roku Chakradhar i współautorzy zaproponowali dwa nowe modele programowania liniowego całkowitoliczbowego oraz algorytm $2(1 + \epsilon)(1 + \ln(\Delta - 1))$ -aproksymacyjny. W ramach niżejzej pracy, analizowane będą te dwa modele programowania liniowego oraz aproksymacyjny [7]. Dodatkowo analizowane będą propozycje własne rozwiązania tego problemu.

W 2023 roku ukazał się artykuł proponujący GRASP (Greedy Randomized Adaptive Search Procedure) - zachłanna losowa procedura adaptacyjnego przeszukiwania dla problemu znajdowania $\gamma_R^{wc}(G)$, poprzez zdefiniowanie funkcji zachłannych: Dscore (dominacja) i Nscore (sąsiedztwo), strategie tabu w celu uniknięcia zapętlenia oraz lokalnych przeszukiwań w celu optymalizacji rozwiązania. Jednakże, autorzy artykułu źle zrozumieli definicję WCRDP, dlatego jego wyniki nie zostały dalej analizowane [8].

Literatura proponuje również praktyczne zastosowanie rozwiązania tego problemu w obecnych czasach. Mianowicie, kosztowne pojazdy służb ratunkowych powinny być rozmieszczane tak, aby były w stanie udzielić pomocy potrzebującym, przy możliwie dużej redukcji kosztów utrzymania takiego pojazdu. Dlatego, na wzór legionów rzymskich, w budynkach służb ratunkowych powinien znajdować się pojazd, bądź być możliwość wypożyczenia go z sąsiedniej (najbliższej) lokalizacji służb ratunkowych [9].

2.5 Metody badań

W celu implementacji i testowania wydajności oraz poprawności algorytmów, stworzono program w języku Python, ze wsparciem następujących bibliotek:

- networkx - pakiet dostarczający funkcje umożliwiające operacje na grafach, wykresach i sieciach
- matplotlib - do wyświetlania wyników działania algorytmów w postaci wykresów grafów
- time - wykorzystywane do pomiarów czasu pracy algorytmów
- pulp - do programowania liniowego
- pandas - do analizy danych z csv
- geopandas - do analizy współrzędnych geograficznych
- itertools - do zagadnień kombinatorycznych

Program umożliwia wprowadzenie dowolnego grafu w postaci listy wierzchołków oraz krawędzi, jak i wygenerowanie losowego grafu. Następnie wybrane algorytmy analizują dany graf poprzez przypisywanie odpowiednich wartości wierzchołkom oraz wyliczania liczby dominowania rzymskiego słabo spójnego. Dla każdego z algorytmów wyliczany i zapisywany jest ich czas działania. Na końcu program wyświetla wykres z nadanymi wartościami na wierzchołkach.

W celu przeprowadzenia testów wygenerowano oraz zapisano w plikach grafy następujących klas:

- gęste - `nx.erdos_renyi_graph(p=0.7)` - z racji wielu krawędzi, co może znacznie wpływać na

- wyniki osiągane przez algorytmy, gdzie p to prawdopodobieństwo wystąpienia krawędzi,
- rzadkie - $nx.erdos_renyi_graph(p=0.3)$ - z racji mniejszej liczby krawędzi, w celu zweryfikowania hipotezy, że dla tych grafów algorytmy będą działały szybciej,
 - drzewa - $nx.random_unlabeled_rooted_tree(n)$ - z racji implementacji specjalnego liniowego algorytmu dla drzew, który znajduje $\gamma_R^{wc}(G)$ oraz specyficznych własności drzew,
 - bezskalowe - $nx.barabasi_albert_graph()$ - grafy nieregularne, z powodu hipotetycznych zastosowań praktycznych, dobrze odzwierciedlają naturalnie powstające na świecie grafy np. sieci społecznościowych, komputerowych, energetycznych.

Grafy te wygenerowano kilka razy dla konkretnej liczby wierzchołków. Liczby wierzchołków, dla których przeprowadzano analizę to [10, 20, 30, 40, 50]. Wszystkie algorytmy zostały użyte dla tego samego zestawu grafów, w celu zachowania miarodajności przy analizie porównawczej. Wszystkie wyniki działania algorytmów na wszystkich wygenerowanych przykładowych grafach zostały zapisane w pliku csv. Zapisano tam również pomiary czasu oraz WCRDF oraz wyznaczoną $\gamma_R^{wc}(G)$. Dodatkowo zapisano też wykresy grafów wraz z przypisanymi wartościami WCRDF.

Czas mierzono za pomocą funkcji `time.perf_counter_ns()`, dając wyniki w nanosekundach o dokładności pomiaru 100 [ns]. Wyniki pomiaru czasu działania są uśrednione, wykonywane pięciokrotnie, aby uniknąć niespodziewanych odchyleń.

Przeprowadzono również dostosowywanie parametrów dla algorytmu mrówkowego na wcześniej wygenerowanych grafach, w celu znalezienia jak najlepszych parametrów dla tego konkretnego problemu. Następnie, na podstawie wielkości błędu dokonano wyboru najlepszego zestawu parametrów, których używano później w docelowej analizie porównawczej.

Algorytmy wyznaczające poprawnie WCRDF, jednakże nie zawsze wyznaczające $\gamma_R^{wc}(G)$, poddano dodatkowej analizie w celu określenia rozbieżności od optymalnego wyniku.

Następnie algorytmy zostały poddane analizie na grafach naturalnie powstały. Dzięki bibliotece geopandas utworzono mapę Polski, a następnie naniesiono na niej punkty odpowiadające węzłom przesyłowym sieci najwyższych napięć w Polsce, na podstawie istniejącego planu. Na grafie starano się odwzorować najważniejsze punkty i krawędzie tej sieci przesyłowej. Z tych danych powstał graf, który następnie zbadano przy użyciu zaimplementowanych algorytmów. Rozmieszczenie wartości WCRDF ma odpowiadać lokalizacjom mechanizmów zabezpieczających sieć przesyłową w razie awarii jednej ze stacji [10].

Następnie analizowano naturalne grafy odzwierciedlające sieci społecznościowe. Są to: mały graf sieci klubu karate Zacharego z 34 wierzchołkami i 78 krawędziami [11], oraz graf dużej sieci społecznościowej znajomych z Facebook'a (4039 wierzchołki i 80234 krawędzi) [12]. W przypadku sieci społecznościowych WCRDF może mieć zastosowanie w optymalnym rozmieszczeniu

agentów wykrywających oszustwa w internetowej sieci społecznościowej.

Dla grafów naturalnie powstały również mierzono czas działania, jak i jakość prezentowanych przez algorytmy wyników.

ROZDZIAŁ 3. BADANE ALGORYTMY

3.1 Wprowadzenie

Niniejszy rozdział opisuje algorytmy dla funkcji dominowania rzymskiego słabo spójnego. Zostaną one przedstawione pod względem schematu działania, złożoności i wydajności, poprawności oraz ich pseudokod. Lista analizowanych algorytmów jest następująca:

- algorytm brute force - Brute Force,
- algorytm liniowy dla drzew - TreeLinear,
- algorytm programowania liniowego I - ILP,
- algorytm programowania liniowego II - ILP2,
- algorytm mrówkowy - AntColony,
- algorytm aproksymacyjny - Approx,
- algorytm zachłanny - Greedy.

3.2 Algorytm Brute Force

3.2.1 Działanie

Jest to w zasadzie trywialna implementacja dokładnego algorytmu wyznaczającego WCRDF poprzez sprawdzenie każdej kombinacji wartości $\{0, 1, 2\}$ na wierzchołkach grafu wejściowego. Każda kombinacja sprawdzana jest pod względem poprawności według definicji słabo spójności w następujący sposób:

- wyznaczany jest zbiór indukowany, który składa się ze zbioru dominującego (wierzchołki z wartościami $\{1, 2\}$) oraz sąsiadów wierzchołków zbioru dominującego,
- sprawdzenie, czy wszystkie wierzchołki z wartością 0 mają sąsiada z wartością 2,
- dla każdego wierzchołka ze zbioru indukowanego dodawane są krawędzie, ale tylko te wychodzące z wierzchołków zbioru dominującego
- następnie sprawdzenie, czy powstały graf jest spójny. Jeśli jest, to zbiór spełnia założenia definicji,
- ze wszystkich prawidłowych kombinacji wybierana jest ta o najmniejszej sumie WCRDF.

3.2.2 Poprawność

Aby wykazać poprawność algorytmu Brute Force, należy udowodnić, że algorytm sprawdza wszystkie możliwe funkcje kandydujące, gwarantując kompletność przeszukania oraz poprawność weryfikacji kombinacji względem definicji WCRDF.

Kompletność przeszukania

Algorytm generuje wszystkie możliwe przypisania wartości $f : V \rightarrow \{0, 1, 2\}$, co daje 3^n kombinacji.

cji, gdzie $n = |V|$, więc żadne potencjalne rozwiązanie nie będzie pominięte.

Poprawność weryfikacji

Dla każdej funkcji f , algorytm sprawdza:

- a) warunek dominacji rzymskiej — sprawdzenie, czy wszystkie wierzchołki z wartością 0 mają sąsiada z wartością 2.
- b) warunek słabej spójności — wyznaczany jest podgraf indukowany, który składa się ze zbioru dominującego (wierzchołki z wartościami 1, 2) oraz sąsiadów wierzchołków zbioru dominującego. Dla każdego wierzchołka ze zbioru indukowanego dodawane są krawędzie, ale tylko te wychodzące z wierzchołków zbioru dominującego. Sprawdzenie spójności tego podgrafa.

Minimalność

Spośród poprawnych kombinacji wartości WCRDF wybierana jest funkcja o najmniejszej sumie wag przypisanych do wierzchołków.

Zatem algorytm poprawnie wyznacza on WCRDF oraz wartość $\gamma_R^{\text{WC}}(G)$.

3.2.3 Złożoność i wydajność

Algorytm ma złożoność wykładniczą, zatem nie będzie wykonywalny w rozsądny czasie dla większych grafów, z uwagi na:

- generowanie wszystkich kombinacji możliwych przypisań: 3^n , gdzie n to liczba wierzchołków grafu,
- sprawdzanie własności zbioru słabo spójnego dla każdego przypisania: n^2

Zatem złożoność czasowa algorytmu wynosi $O(3^n \cdot n^2)$

Złożoność pamięciowa ogranicza się do przechowywania grafu w pamięci i wynosi $O(n + m)$.

3.2.4 Pseudokod

Algorytm Brute Force

```

1: function FindRomanDominatingSet(graph)
2:   Initialize min_roman_number  $\leftarrow \infty$ 
3:   Initialize best_node_values  $\leftarrow \text{None}$ 
4:   nodes  $\leftarrow$  list of nodes in graph
5:   for each assignment of values (0, 1, 2) to all nodes do
6:     node_values  $\leftarrow$  mapping of nodes to values
7:     for each node in graph do                                 $\triangleright$  Sprawdzanie warunku dominacji
8:       if node_values[node] = 0 then
9:         if not any neighbor of node has value 2 then
10:          Continue to next assignment
11:        end if
12:      end if
13:    end for
14:    induced_set  $\leftarrow$  nodes with values {1, 2}
15:    for each node in induced_set do
16:      Add all its neighbors to induced_set
17:    end for
18:    Create empty induced_graph
19:    for each node in induced_set do
20:      if node_values[node] is 1 or 2 then
21:        for each neighbor in graph do
22:          if neighbor in induced_set then
23:            Add edge to induced_graph
24:          end if
25:        end for
26:      end if
27:    end for
28:    if induced_graph is connected then
29:      Compute roman_number  $\leftarrow$  sum of node_values
30:      if roman_number < min_roman_number then
31:        Update min_roman_number and best_node_values
32:      end if
33:    end if
34:  end for
35:  return (min_roman_number, best_node_values)
36: end function

```

3.3 Algorytm liniowy dla drzew

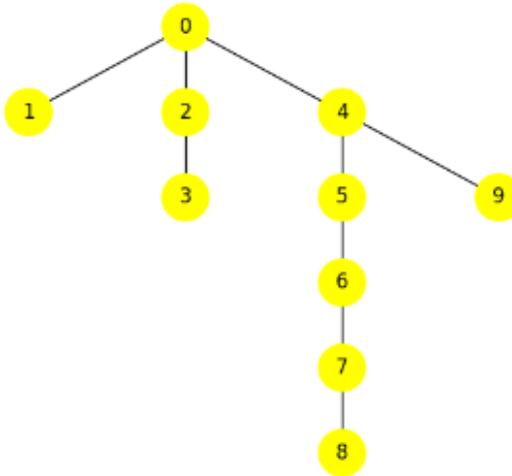
3.3.1 Działanie

Dla każdego wierzchołka definiujemy następujące parametry:

- $v[R']$ - wartość funkcji dominowania rzymskiego słabo spójnego w wierzchołku v , z założeniem, że $v[R'] \in \{0, 1, 2\}$
- $v[n00']$ - oznacza liczbę dzieci $z R = 0$ oraz bez sąsiada $z R = 2$ (dziecko niezdominowane)
- $v[n01']$ - oznacza liczbę dzieci $z R = 0$ i z sąsiadem $z R = 2$ (dziecko zdominowane)
- $v[n1']$ - oznacza liczbę dzieci wierzchołka v z $R = 1$
- $v[n2']$ - oznacza liczbę dzieci wierzchołka v z $R = 2$
- $v[sw']$ - oznacza liczbę dzieci wierzchołka v z $n00 = 1$ i $n01 = 0$. (wierzchołek wspierający)
- $v[ch'] = 1$ jeśli $v[sw'] > 1$ lub jeśli $v[sw'] = 1$ i mający przynajmniej jedno dziecko $z R = 0$; w przeciwnym razie $v[ch'] = 0$. Jeśli $v[ch'] = 1$, wtedy $v[R'] = 2$ i w fazie 2 każde dziecko v z $n00 = 1$ i z $n01 = 0$ dostaje $R = 0$ i jego jedyne dziecko $z R = 0$ zmienia wartość na $R = 1$.
- $v[child']$ - jeśli v jest wierzchołkiem wspierającym, wtedy wartość ta jest numerem liścia sąsiadującego z v .

Algorytm ma 2 fazy. W obu fazach rozpatrywane są wszystkie wierzchołki drzewa według odwrotnego porządku drzewa, czyli od ostatniego wierzchołka do korzenia (reverse tree-order). Wszystkie początkowo zdefiniowane wartości mają wartości 0. W bardzo ogólnym rozumieniu, rozpatrywany jest każdy wierzchołek na podstawie sąsiedztwa, relacji ojciec-dziecko oraz wartości zdefiniowanych parametrów i aktualizację ich w obu fazach. Wartości R przy każdym wierzchołku, to wartości funkcji dominowania rzymskiego słabo spójnego.

Z racji sporego stopnia skomplikowania algorytmu, jego działanie zostanie przedstawione na przykładzie. Dany jest graf G będący drzewem ukorzenionym o 10 wierzchołkach, ponumerowanych wartościami od 0 do 9. Zakładamy, że wyznaczenie ojca każdego z wierzchołków jest trywialne, dlatego podczas rozważań wyznaczanie ojca wierzchołka będzie pomijane.



Rysunek 3.1: Graf G - przykładowe drzewo

1. Jeśli wierzchołek jest liściem i nie jest korzeniem to zwiększamy wartość 'n00' ojca o 1.

Zatem dla liści 1,3,8,9, wartości ich ojców wyglądają następująco:

$$T[7]['n00'] = 1$$

$$T[7]['child'] = 8 \text{ (od wierzchołka 8)}$$

$$T[4]['n00'] = 1$$

$$T[4]['child'] = 9 \text{ (od wierzchołka 9)}$$

$$T[2]['n00'] = 1$$

$$T[2]['child'] = 3 \text{ (od wierzchołka 3)}$$

$$T[0]['n00'] = 1$$

$$T[0]['child'] = 1 \text{ (od wierzchołka 1)}$$

2. Jeśli wierzchołek nie jest liściem:

- (a) Sprawdzamy czy wierzchołek posiada tylko jedno niezdominowane dziecko i posiada ojca. W tym przypadku ojciec będzie wierzchołkiem wspierającym.

$$T[6]['sw'] = 1 \text{ (od wierzchołka 7)}$$

$$T[0]['sw'] = 2 \text{ (od wierzchołka 4 i 2)}$$

- (b) Sprawdzamy sumę wartości dzieci zdominowanych, niezdominowanych oraz liczbę dzieci dla których wierzchołek jest wspierający. Jeśli ta suma jest większa od 1, to wartość tego wierzchołka ustawiamy na 2, a parametr 'ch' na 1.

$$T[0]['R'] = 2 \text{ (od wierzchołka 0)}$$

$$T[0]['ch'] = 1 \text{ (od wierzchołka 0)}$$

- i. Jeśli wierzchołek ma ojca to parametr 'n2' ojca zwiększamy o 1, a jeśli wierzchołek posiada tylko jedno niezdominowane dziecko zmniejszamy parametr wspierający u ojca.

- (c) Jeśli wierzchołek nie jest wspierający:

- i. Jeśli wierzchołek posiada niezdominowane dzieci lub jedno dziecko i żadnych dzieci z wartością 2 lub dzieci zdominowane, to wtedy wierzchołek będzie miał wartość

2. Dla istniejącego ojca wierzchołka zwiększamy 'n2'.

$T[7][R'] = 2$ (od wierzchołka 7)

$T[6][n2'] = 1$ (od wierzchołka 7)

$T[4][R'] = 2$ (od wierzchołka 4)

$T[0][n2'] = 2$ (od wierzchołka 4 i 2)

$T[2][R'] = 2$ (od wierzchołka 2)

ii. Jeśli wierzchołek posiada niezdominowane dziecko, to danemu wierzchołkowi przypisujemy wartość 0, a temu dziecku wartość 1, a dla ojca zmniejszamy wartość wspierania.

iii. Jeśli wierzchołek posiada tylko dzieci zdominowane, to danemu wierzchołkowi przypisujemy wartość 1, a ojcu zwiększamy wartość 'n1'.

$T[5][R'] = 1$ (od wierzchołka 5)

$T[4][n1'] = 1$ (od wierzchołka 5)

(d) Jeśli wartość wierzchołka wynosi 0, posiada on dzieci z wartością 2 oraz ojca, to zwiększamy wartość ojca 'n01' o 1,

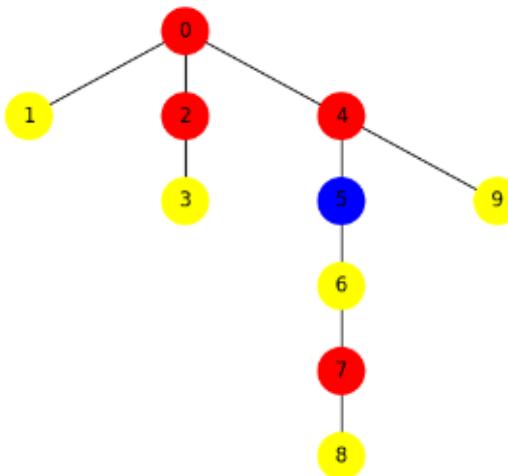
$T[5][n01'] = 1$ (od wierzchołka 6)

(e) Jeśli wartość wierzchołka wynosi 0, nie posiada on dzieci z wartością 2 oraz ojca, to zwiększamy wartość ojca 'n00' o 1,

3. Korzeń należy rozpatrzyć dodatkowo. Jeśli nie posiada on dzieci z wartościami 2 i sam ma wartość 0, to przypisujemy mu $R = 2$,

4. Jeśli liczba dzieci korzenia z $R = 1$ jest równa liczbie dzieci pomniejszonej o 1, to korzeń również ma wartość 1.

Zdjęcie przedstawia zachowanie algorytmu po fazie 1. Czerwone wierzchołki to wartość $R = 2$, niebieskie to $R = 1$, a żółte to $R = 0$. Widać, że przypisanie nie jest jeszcze optymalne.



Rysunek 3.2: Graf G - po fazie 1

W fazie 2, dla każdego wierzchołka posiadającego ojca, tylko jedno dziecko niezdomino-

wane i parametr ojca 'ch' wynoszący 1, wtedy musimy „zmienić” układ, poprzez ustawienie 0 na obecnym wierzchołku, ustawienie dziecka na 1 oraz zwiększenie liczby dzieci niedominowanych ojca wierzchołka. Ten warunek spełniony jest dla wierzchołków 4 i 2. Zatem:

$$T[4]['R'] = 0 \text{ (od wierzchołka 4)}$$

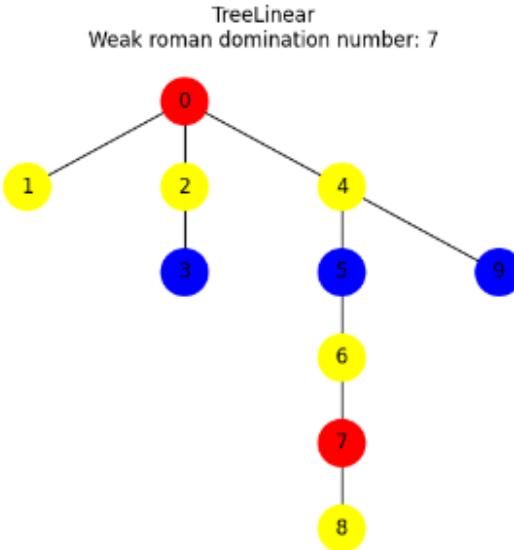
$$T[9]['R'] = 1 \text{ (dziecko wierzchołka 4)}$$

$$T[2]['R'] = 0 \text{ (od wierzchołka 2)}$$

$$T[3]['R'] = 1 \text{ (dziecko wierzchołka 2)}$$

$$T[0]['n00'] = 3 \text{ (ojciec wierzchołków 4 i 2)}$$

Poniższy rysunek przedstawia prawidłowe, optymalne przypisanie wartości R po fazie 2.



Rysunek 3.3: Graf G - po fazie 2 - finalna wersja

3.3.2 Poprawność

Poprawność algorytmu dla drzew zostanie wyznaczona poprzez szkic dowodu indukcyjnego. Dla każdego drzewa $T = (V, E)$ zakłada się, że algorytm liniowy poprawnie wyznacza funkcję $f : V \rightarrow \{0, 1, 2\}$ będącą WCRDF o minimalnej wadze $\gamma_R^{\text{wc}}(T)$.

Krok początkowy:

- Dla $|V| = 1$, $f(v) = 1$, dominuje sam siebie. Algorytm wykrywa brak dzieci i przypisuje $R = 1$ zgodnie z warunkiem korzenia bez $n1$).
- Dla $|V| = 2$: liśc otrzymuje $R = 0$, a korzeń $R = 2$, ponieważ nie ma innych dzieci dominujących. Sytuacja zostaje poprawnie wykryta na podstawie $n00 = 1$, $n01 = 0$, co daje przypisanie $R = 2$ ojcu.

Krok indukcyjny:

Załóżmy, że algorytm wyznacza poprawnie $\gamma_R^{\text{wc}}(T)$ dla wszystkich drzew o liczbie wierzchołków $< n$. Wykażemy, że działa również dla dowolnego drzewa T o $|V| = n$.

Niech r będzie korzeniem drzewa T , a T_1, \dots, T_k jego poddrzewami o korzeniami v_1, \dots, v_k .

Z założenia indukcyjnego dla każdego T_i funkcja f_i wyznaczona przez algorytm spełnia warunki WCRDF i jest minimalna na T_i .

Spójność i dominacja:

Podczas przetwarzania r , algorytm zbiera informacje o:

- liczbie dzieci z $f = 0$ i bez dominacji ($n00$),
- liczbie dzieci wspierających (sw),
- liczbie dzieci z $f = 1$ i $f = 2$.

W zależności od tych wartości:

- jeśli $n00 > 1$ lub dzieci są zbyt słabo wspierane – r przyjmuje $f(r) = 2$,
- jeśli $n00 = 1$ i wspierające dziecko posiada jedno dziecko z $f = 0$, to jego dziecko dostaje $f = 1$ w Fazie 2,
- jeśli wszystkie dzieci są już zdominowane, r może przyjąć $f = 1$,
- jeśli r jest zdominowany przez któreś dziecko z $f = 2$ i jego dołączenie do zbioru nie jest konieczne do zapewnienia spójności, r może przyjąć $f(r) = 0$.

W każdej z tych sytuacji:

- każde v z $f(v) = 0$ ma sąsiada z $f = 2$,
- zbiór $D = \{v : f(v) \in \{1, 2\}\}$ jest słabo spójny, ponieważ każde D_i jest spójne, a r łączy je jako nadrzędny węzeł.

Minimalność:

Decyzje w algorytmie są podejmowane lokalnie optymalnie:

- przypisanie $f = 2$ tylko wtedy, gdy inne opcje (np. $f = 1$ lub propagacja przez dzieci) nie wystarczają,
- preferowanie struktury $[1, 1]$ nad $[2, 0]$ tam, gdzie jest to możliwe,
- przypisania są zgodne z lokalną strukturą drzewa.

W związku z tym suma wag WCRDF jest najmniejsza możliwa przy spełnieniu warunków definicji, co kończy szkic dowodu.

3.3.3 Złożoność i wydajność

Algorytm ma złożoność liniową $O(n)$, gdzie n to liczba wierzchołków drzewa. W obu fazach rozpatrywane są wszystkie wierzchołki, od liści do korzenia. Uprzednio wyznaczana zostaje mapa ojcostwa, również liniowo. Algorytm zatem jest skalowalny i szybki dla większych grafów, natomiast ograniczony do jednej ich klasy - drzew.

Złożoność pamięciowa jest liniowa względem liczby wierzchołków, ze względu na przechowywanie drzewa w pamięci.

3.3.4 Pseudokod

Algorytm liniowy dla drzew - Faza 1

```
1: function Phase1(T, root)
2:   father_map  $\leftarrow$  Compute parent-child relationships using BFS
3:   nodes_ids  $\leftarrow$  List of all nodes in T
4:   for each node  $v$  in reversed(nodes_ids) do
5:     father  $\leftarrow$  father_map[ $v$ ]
6:     if  $v$  is a leaf and  $v \neq root$  then
7:       Increase  $T[father]['n00']$ 
8:       Set  $T[father]['child'] \leftarrow v$ 
9:     else
10:      if  $T[v]['n00'] == 1$  and  $T[v]['n01'] == 0$  and father exists then
11:        Increase  $T[father]['sw']$ 
12:      end if
13:      if  $T[v]['sw'] + T[v]['n00'] + T[v]['n01'] > 1$  then
14:        Set  $T[v]['R'] = 2$ 
15:        if father exists then
16:          Increase  $T[father]['n2']$ 
17:          if  $T[v]['n00'] == 1$  and  $T[v]['n01'] == 0$  then
18:            Decrease  $T[father]['sw']$ 
19:          end if
20:        end if
21:         $T[v]['ch'] = 1$ 
22:      end if
23:      if  $T[v]['sw'] == 0$  then
24:        if  $T[v]['n00'] > 1$  or ( $T[v]['n00'] == 1$  and ( $T[v]['n2'] == 0$  or  $T[v]['n01'] > 0$ ))
25:          then
26:            Set  $T[v]['R'] = 2$ 
27:            if father exists then
28:              Increase  $T[father]['n2']$ 
29:            end if
30:            else if  $T[v]['n00'] == 1$  then
31:              Set  $T[v]['R'] = 0$ 
32:              Set  $T[T[v]['child']]['R'] = 1$ 
33:              if father exists then
34:                Decrease  $T[father]['sw']$ 
35:              end if
36:            end if
37:            if  $T[v]['n00'] == 0$  and  $T[v]['n01'] > 0$  then
38:              Set  $T[v]['R'] = 1$ 
39:              if father exists then
40:                Increase  $T[father]['n1']$ 
41:              end if
42:            end if
43:            if  $T[v]['R'] = 0$  and  $T[v]['n2'] > 0$  and father exists then
44:               $T[father]['n01'] \leftarrow T[father]['n01'] + 1$ 
45:            end if
46:            if  $T[v]['R'] = 0$  and  $T[v]['n2'] = 0$  and father exists then
47:               $T[father]['n00'] \leftarrow T[father]['n00'] + 1$ 
48:            end if
49:          end if
50:        end for
51:        if  $T[root]['n2'] == 0$  and  $T[root]['R'] == 0$  then
52:          Set  $T[root]['R'] = 2$ 
53:        end if
54:        if  $T[root]['n1'] == (\text{number of root's neighbors})$  then
55:          Set  $T[root]['R'] = 1$ 
56:        end if
57:      return T
58:    end function
```

Algorytm liniowy dla drzew - Faza 2

```

1: function Phase2( $T$ ,  $\text{root}$ )
2:   for each node  $v$  in  $\text{reversed}(nodes\_ids)$  do
3:      $father \leftarrow father\_map[v]$ 
4:     if  $father$  exists then
5:       if  $T[v]['n00'] == 1$  and  $T[father]['ch'] == 1$  and  $T[v]['n01'] == 0$  then
6:         Set  $T[v]['R'] = 0$ 
7:         Set  $T[T[v]['child']]['R'] = 1$ 
8:         Increase  $T[father]['n00']$ 
9:       end if
10:      end if
11:    end for
12:    return  $T$ 
13: end function

```

3.4 Algorytm programowania liniowego I

Jest to algorytm programowania liniowego zaimplementowany na podstawie artykułu „Algorithmic complexity of weakly connected Roman domination in graphs”[7].

3.4.1 Działanie

Dany jest graf G o zbiorze wierzchołków V i zbiorze krawędzi E . G' to podgraf indukowany powstały z wierzchołków zbioru dominującego. Ten model programowania liniowego wymaga zdefiniowania następujących zmiennych:

$$x_e = \begin{cases} 1, & e \in E' \\ 0, & e \notin E' \end{cases} \quad y_e = \begin{cases} 1, & e \in T' \\ 0, & e \notin T' \end{cases}$$

$$a_v = \begin{cases} 1, & v \in V_1 \cup V_2 \\ 0, & v \in V_0 \end{cases} \quad b_v = \begin{cases} 1, & v \in V_2 \\ 0, & v \in V_0 \cup V_1 \end{cases}$$

gdzie $v \in V$, $e \in E$ i T' to drzewo rozpinające podgrafa G' .

Definiujemy funkcję celu, czyli minimalizację wagi zbioru dominującego rzymskiego słabo spójnego:

$$Z: \min \left(\sum_{v \in V} a_v + \sum_{v \in V} b_v \right)$$

oraz ograniczenia:

Wierzchołek o wartości 0, bedzie miał co najmniej jednego sasiada z wartością 2:

$$a_v + \sum_{k \in N_G(v)} b_k \geq 1, \quad v \in V, \tag{1}$$

Krawędź istnieje w drzewie rozpinającym T' , jeśli ta krawędź należy do G' . Ograniczenie to zapewnia spójność w G' :

$$y_e \leq x_e, \quad e \in E, \tag{2}$$

Wybór krawędzi, które mają wartość należącą do zbioru dominującego na przynajmniej jednym swoim końcu:

$$x_e \leq a_{i_e} + a_{j_e}, \quad e \in E_{G'}, \quad (3)$$

Drzewo rozpinające T' ma liczbę krawędzi równą liczbie wierzchołków grafu pomniejszoną o 1:

$$\sum_{e \in E} y_e = n - 1, \quad (4)$$

Drzewo rozpinające T' nie posiada cykli:

$$\sum_{i_e, j_e \in S} y_e \leq |S| - 1, \quad S \subseteq V, \quad |S| \geq 3, \quad (5)$$

Podzbiór wierzchołków z $b_v = 1$, czyli (V_2) , jest podzbiorem wierzchołków z $a_v = 1$, czyli $(V_1 \cup V_2)$

$$b_v \leq a_v, \quad v \in V. \quad (6)$$

Warunki 1, 2, 5 gwarantują, że T' jest drzewem rozpinającym grafu G' .

3.4.2 Poprawność

Poniżej przedstawiona zostaje analiza, dlaczego model programowania liniowego i opisany powyżej poprawnie odwzorowuje problem wyznaczania WCRDF, a jego rozwiązanie minimalizuje wartość $\gamma_R^{\text{wc}}(G)$.

Każde dopuszczone rozwiązanie modelu ILP jest poprawną funkcją WCRDF.

Rozważane jest dowolne przypisanie zmiennych a_v, b_v, x_e, y_e spełniające ograniczenia (1)-(6).

- Ograniczenie (1): dla każdego v z $a_v = 0$ (czyli $f(v) = 0$), istnieje co najmniej jeden sąsiad k z $b_k = 1$ (czyli $f(k) = 2$). To spełnia warunek dominacji rzymskiej.
- Ograniczenia (2)-(5): definiują graf G' (na podstawie x_e), oraz jego drzewo rozpinające T' (na podstawie y_e). W szczególności:
 - (2): T' zawiera się w G' ,
 - (3): każda krawędź x_e łączy dwa wierzchołki z przynajmniej jednym $a = 1$,
 - (4)-(5): T' ma $n - 1$ krawędzi i nie zawiera cykli, czyli jest drzewem,
- Zatem T' rozciąga się na zbiorze $D = \{v : a_v = 1\}$ i łączy go w sposób spójny — spełnia to warunek słabej spójności.
- Ograniczenie (6): każdy wierzchołek z $b_v = 1$ (czyli $f(v) = 2$) musi mieć również $a_v = 1$ — co odwzorowuje relację $V_2 \subseteq V_1 \cup V_2$.

Każda poprawna funkcja WCRDF może być zakodowana jako przypisanie zmiennych.

Dla dowolnej funkcji $f : V \rightarrow \{0, 1, 2\}$ spełniającej definicję WCRDF, możemy przypisać:

$$a_v = \begin{cases} 1 & \text{jeśli } f(v) > 0 \\ 0 & \text{jeśli } f(v) = 0 \end{cases} \quad b_v = \begin{cases} 1 & \text{jeśli } f(v) = 2 \\ 0 & \text{jeśli } f(v) \neq 2 \end{cases}$$

Podgraf G' indukowany przez wierzchołki z $a_v = 1$ jest spójny (bo f spełnia definicję WCRDF), więc istnieje drzewo rozpinające T' — można przypisać x_e i y_e tak, by spełnić wszystkie ograniczenia (2)-(5).

Funkcja celu odwzorowuje wagę funkcji f

Wartość funkcji celu to:

$$\sum_{v \in V} a_v + \sum_{v \in V} b_v = |\{v : f(v) \in \{1, 2\}\}| + |\{v : f(v) = 2\}| = \sum_v f(v)$$

Zatem minimalizacja funkcji celu odpowiada minimalizacji wagi funkcji WCRDF.

3.4.3 Złożoność i wydajność

Liczba zmiennych dla grafu o n wierzchołkach i m krawędziach wynosi $O(n + m)$.

1. Dominacja rzymska: jedno na każdy wierzchołek, zatem n ograniczeń.
2. Zależność między y_e a x_e : jedno na każdą krawędź, zatem m ograniczeń.
3. Ograniczenie dostępu tylko wśród bronionych wierzchołków: jedno na każdą krawędź, zatem m ograniczeń.
4. Liczba krawędzi w drzewie rozpinającym: jedno globalne ograniczenie, zatem 1.
5. Brak cykli (ograniczenia cykliczne): jedno dla każdego podzbioru $S \subseteq V$, gdzie $|S| \geq 3$. W najgorszym przypadku liczba tych podzbiorów rośnie wykładniczo: $O(2^n)$.
6. Relacja $b_v \leq a_v$: jedno na każdy wierzchołek, zatem n ograniczeń.

Zatem w najgorszym przypadku liczba ograniczeń wynosi $O(n + m + 2^n)$. Złożoność obliczeniowa wynosi $O(2^n)$. Ze względu na ograniczenia cykliczne trudność rozwiązywania jest不稳定na.

Złożoność pamięciowa wynosi $O(n + m + 2^n)$ ze względu na przechowywanie zmiennych i ograniczeń.

3.4.4 Pseudokod

Algorytm programowania liniowego I

```
1: function ILP(graph)
2:    $V \leftarrow$  list of nodes in graph
3:    $E \leftarrow$  list of edges in graph
4:   Initialize ILP model model
5:   Set objective: Minimize  $\sum(a[i] + b[i])$  for all nodes  $i \in V$ 
6:   Define binary variables:
7:      $x[i, j]$  for  $(i, j) \in E$                                  $\triangleright$  1 jeśli krawędź jest w  $G'$ 
8:      $y[i, j]$  for  $(i, j) \in E$                                  $\triangleright$  1 jeśli krawędź jest w drzewie rozpinającym  $T'$ 
9:      $a[i]$  for  $i \in V$                                      $\triangleright$  1 jeśli wierzchołek należy do  $V1 \cup V2$ 
10:     $b[i]$  for  $i \in V$                                      $\triangleright$  1 jeśli wierzchołek należy do  $V2$ 
11:   Constraints:
12:     for each node  $i$  in  $V$  do                       $\triangleright$  Każdy wierzchołek musi być broniony
13:       Add constraint:  $a[i] + \sum b[k] \geq 1$ , where  $k$  are neighbors of  $i$ 
14:     end for
15:     for each edge  $(i, j)$  in  $E$  do
16:       Add constraint:  $y[i, j] \leq x[i, j]$                  $\triangleright$  Krawędź drzewa musi istnieć w  $G'$ 
17:       Add constraint:  $x[i, j] \leq a[i] + a[j]$              $\triangleright$  Krawędzie drzewa muszą łączyć bronione
           wierzchołki
18:     end for
19:     Add constraint:  $\sum y[i, j] = |V| - 1$            $\triangleright$  Drzewo musi mieć  $|V| - 1$  krawędzi
20:     Find cliques of size  $\geq 3$  in graph and store as subsets
21:     for each subset  $S$  in subsets do                   $\triangleright$  Eliminacja cykli
22:       Add constraint:  $\sum y[i, j] \leq |S| - 1$  for edges  $(i, j) \in S$ 
23:     end for
24:     for each node  $i$  in  $V$  do                       $\triangleright$  Wierzchołki z  $V2$  muszą należeć do  $V1 \cup V2$ 
25:       Add constraint:  $b[i] \leq a[i]$ 
26:     end for
27:     Solve ILP model
28:     Extract solution:
29:     for each node  $i$  in  $V$  do
30:        $solution[i] \leftarrow round(a[i].X) + 2 * round(b[i].X)$ 
31:     end for
32:     return (model.objVal, solution)
33: end function
```

3.5 Algorytm programowania liniowego II

Jest to algorytm programowania liniowego zaimplementowany na podstawie artykułu „Algorithmic complexity of weakly connected Roman domination in graphs”[7], podobnie jak poprzedni.

3.5.1 Działanie

Model ten opiera się na przepływach. W tym modelu rozpatrywany jest graf jako wierzchołek, który zużywa jednostkę przepływu, przepływając przez krawędzie grafu. Na wejściu definiowany jest zewnętrzny przepływ, którego liczba jednostek jest równa liczbie wierzchołków grafu. Zachowana jest zasada przepływu sieci, dlatego trzeba wprowadzić przepływ zewnętrzny w jak największej ilości przez pojedynczy wierzchołek.

Należy zdefiniować następujące zmienne:

$$x_i = \begin{cases} 1, & i \in V_1 \cup V_2 \\ 0, & i \in V_0 \end{cases} \quad y_i = \begin{cases} 1, & i \in V_2 \\ 0, & i \in V_0 \cup V_1 \end{cases}$$

$$a_e = \begin{cases} 1, & e \in E' \\ 0, & e \notin E' \end{cases} \quad t_i = \begin{cases} 1, & \text{wierzchołek korzenia} = i \\ 0, & \text{pozostałe wierzchołki} \end{cases}$$

$$u_i \in N \cup \{0\}, \quad v_e \in [-n, n].$$

gdzie:

t_i zdefiniowany dla każdego wierzchołka grafu. Identyfikuje, gdzie zewnętrzny wierzchołek jest traktowany jako wejście.

u_i reprezentuje liczbę jednostek przepływu zewnętrznego dla danego wierzchołka grafu.

v_e oznacza jednostki przepływające przez dane krawędzie.

a_e oznacza przynależność krawędzi do podgrafa indukowanego.

x_i, y_i oznaczają przynależność do zbioru dominującego rzymskiego słabo spójnego.

Minimalizującą liczbę dominowania rzymskiego słabo spójnego jako funkcję celu należy zdefiniować następująco:

$$Z: \min \left(\sum_{i \in V} x_i + \sum_{i \in V} y_i \right),$$

mając dane ograniczenia:

Wierzchołek z wartością 0 sąsiaduje z przynajmniej jednym wierzchołkiem z wartością 2:

$$x_i + \sum_{j \in N_G(i)} y_j \geq 1, \quad i \in V, \tag{1}$$

Ograniczenie zmiennych x i y , aby były zgodne ze zdefiniowaną przynależnością:

$$y_i \leq x_i, \quad i \in V, \quad (2)$$

Krawędź $e \in E'$ ma przynajmniej jeden koniec z wierzchołkiem o wartości przynajmniej 1 (należącym do zbioru dominującego):

$$a_e \leq x_{i_e} + x_{j_e}, \quad e \in E, \quad (3)$$

Jest tylko jeden wierzchołek w grafie G' , gdzie jest dostarczony przepływ zewnętrzny:

$$\sum_{i \in V} t_i = 1, \quad (4)$$

Wielkość przepływu co najwyżej n :

$$u_i \leq n \cdot t_i, \quad i \in V, \quad (5)$$

Przepływ ma miejsce tylko w krawędziach należących do E' :

$$v_e \leq n \cdot a_e, \quad e \in E, \quad (6)$$

$$v_e \geq -n \cdot a_e, \quad e \in E, \quad (7)$$

Reprezentacja zasady zachowania sieci:

$$u_i + \sum_{e:j_e=i} v_e - \sum_{e:i_e=i} v_e = 1, \quad i \in V, \quad (8)$$

Każdy wierzchołek $v \in V$ musi mieć przynajmniej jeden ustalony wierzchołek w E'

$$a_e \geq 1, \quad e \in E, \quad (9)$$

3.5.2 Poprawność

Poniżej przedstawiona zostaje analiza, dlaczego model programowania liniowego II opisany powyżej poprawnie odwzorowuje problem wyznaczania WCRDF, a jego rozwiązanie minimalizuje wartość $\gamma_R^{wc}(G)$.

Poprawność odwzorowania funkcji WCRDF.

Zmiennym w modelu odpowiadają następujące interpretacje:

- $x_i = 1 \iff f(i) \in \{1, 2\}$,
- $y_i = 1 \iff f(i) = 2$,
- $a_e = 1 \iff$ krawędź e należy do grafu G' indukowanego przez zbiór dominujący,

- $t_i = 1 \iff$ wierzchołek i to „źródło” przepływu,
- u_i - liczba jednostek przepływu wchodząca do wierzchołka i ,
- v_e - przepływ przez krawędź e .

Warunek dominacji rzymskiej zapewnia ograniczenie (1), bo jeżeli $x_i = 0$ ($f(i) = 0$), to musi istnieć sąsiad j z $y_j = 1$, czyli $f(j) = 2$:

$$x_i + \sum_{j \in N(i)} y_j \geq 1$$

Warunek $y_i \leq x_i$ (ograniczenie (2)) zapewnia, że zgodnie z definicją, że wierzchołek z $f(i) = 2$ należy także do zbioru z $f(i) \geq 1$.

Warunek łączności: zbiór $D = \{i : x_i = 1\}$ musi tworzyć spójny podgraf G' , którego spójność jest zapewniona przez:

- Ograniczenia (3), (6), (7): krawędzie przepływu są ograniczone do tych z końcami w D ,
- Ograniczenia (4), (5), (8): modeluje się jednostkowy przepływ z jednego wierzchołka źródłowego ($t_i = 1$) przez cały zbiór D ,
- Ograniczenie (8): zasada zachowania przepływu — każdy wierzchołek „zużywa” jedną jednostkę, więc przepływ musi dotrzeć do każdego z nich,
- Ograniczenie (9): każdy wierzchołek musi być incydentny do co najmniej jednej krawędzi a_e , czyli nie być odizolowany.

Zatem cała struktura odpowiada spójnemu podgrafowi G' oraz jego słabo spójnej strukturze dominacyjnej.

Odwzorowanie każdej funkcji WCRDF.

Dla każdej poprawnej funkcji WCRDF $f : V \rightarrow \{0, 1, 2\}$ można ustawić zmienne:

$$x_i = \begin{cases} 1, & \text{jeśli } f(i) > 0 \\ 0, & \text{jeśli } f(i) \leq 0 \end{cases} \quad y_i = \begin{cases} 1, & \text{jeśli } f(i) = 2 \\ 0, & \text{jeśli } f(i) \neq 2 \end{cases}$$

Zbiór $D = \{i : x_i = 1\}$ indukuje spójny graf G' , więc można wybrać a_e dla jego krawędzi i zdefiniować przepływ v_e , u_i zgodnie ze standardową metodą budowania drzew przepływu. Wybierając dowolny wierzchołek z $f = 2$ jako korzeń ($t_i = 1$), spełnimy wszystkie ograniczenia.

Minimalizacja funkcji celu.

Funkcja celu:

$$\min \left(\sum x_i + \sum y_i \right) = \sum_{i: f(i)=1} 1 + \sum_{i: f(i)=2} 2 = \sum f(i)$$

odpowiada dokładnie wadze funkcji f . Minimalizacja celu odpowiada wyznaczeniu $\gamma_R^{\text{wc}}(G)$.

3.5.3 Złożoność i wydajność

Suma zmiennych w sformułowanym modelu wynosi $3n+m$, będących wartościami logicznymi, n integralnych i m ciągłych.

- Pokrycie i relacje x, y — ograniczenia zapewniające, że każdy wierzchołek jest zdominowany oraz że jeśli $y_i = 1$, to $x_i = 1$. Dają łącznie $n + n = 2n$ ograniczeń.
- Krawędziowe ograniczenia — dotyczące relacji między zmiennymi a_e i x_i , oraz ograniczające wartości przepływu v_e przez krawędzie. Są to ograniczenia (3), (6) i (7), po jednym na każdą krawędź, co daje łącznie $3m$ ograniczeń.
- Struktura drzewa przepływu — obejmuje:
 - jedno ograniczenie wymuszające dokładnie jeden korzeń przepływu (warunek $\sum t_i = 1$),
 - n ograniczeń ograniczających wartość u_i (jednostki przepływu) do $n \cdot t_i$,
 - n równań przepływu dla każdego wierzchołka (bilans: wejście + wyjście = 1), co łącznie daje $2n + 1$ ograniczeń.
- Wymuszenie incydencji — ograniczenia (9) wymuszają, by każdy wierzchołek miał co najmniej jedną krawędź w E' (grafie dominującym). Daje to dodatkowe n ograniczeń.

Liczba ograniczeń wynosi $7n + 3m + 2$. Wynika to z tego, że nie wszystkie ograniczenia to są nierówności. Złożoność pamięciowa wynosi $O(n + m)$.

W najgorszym przypadku złożoność czasowa będzie eksponencjalna w n . Trudność rozwiązywania jest złożona przez przepływy, ale bardziej stabilna, niż w algorytmie ILP, który ze względu na dużą liczbę cykli może znacznie wydłużyć swój czas działania.

3.5.4 Pseudokod

Algorytm programowania liniowego II

```
1: function ILP_II(graph)
2:   Initialize ILP model model with minimization objective
3:    $V \leftarrow$  list of nodes in graph
4:    $E \leftarrow$  list of edges in graph
5:    $n \leftarrow |V|$                                       $\triangleright$  Liczba wierzchołków
6:   Define binary variables:
7:    $x[i]$  for  $i \in V$                           $\triangleright$  1 jeśli wierzchołek i należy do zbioru X
8:    $y[i]$  for  $i \in V$                           $\triangleright$  1 jeśli wierzchołek i należy do zbioru Y
9:    $a[e]$  for  $e \in E$                           $\triangleright$  1 jeśli krawędź e należy do drzewa rozpinającego
10:   $t[i]$  for  $i \in V$                            $\triangleright$  1 jeśli wierzchołek i jest korzeniem
11: Define integer and continuous variables:
12:   $u[i]$  for  $i \in V$                           $\triangleright$  Zmienna całkowita do struktury drzewa
13:   $v[e]$  for  $e \in E$                           $\triangleright$  Zmienna przepływu z ograniczeniami  $[-n, n]$ 
14: Objective:
15: Minimize  $\sum(x[i] + y[i])$  for all  $i \in V$ 
16: Constraints:
17: for each node i in V do                 $\triangleright$  Zapewnij pokrycie wszystkich wierzchołków
18:   Add constraint:  $x[i] + \sum y[j] \geq 1$ , where  $(i, j) \in E$ 
19:   Add constraint:  $y[i] \leq x[i]$ 
20:   Add constraint:  $\sum a[e] \geq 1$ , where e contains i
21: end for
22: for each edge  $e = (i_e, j_e)$  in E do
23:   Add constraint:  $a[e] \leq x[i_e] + x[j_e]$ 
24:   Add constraint:  $v[e] \leq n \cdot a[e]$ 
25:   Add constraint:  $v[e] \geq -n \cdot a[e]$ 
26: end for
27: Add constraint:  $\sum t[i] = 1$                    $\triangleright$  Tylko jeden korzeń istnieje
28: for each node i in V do                 $\triangleright$  Ograniczenia struktury drzewa
29:   Add constraint:  $u[i] \leq n \cdot t[i]$ 
30:   Add constraint:  $u[i] + \sum v[e] - \sum v[e] = 1$ , for edges e entering/exiting i
31: end for
32: Solve ILP model
33: Extract solution:
34: for each node i in V do
35:    $solution[i] \leftarrow round(x[i].varValue) + 2 \times round(y[i].varValue)$ 
36: end for
37: return (model.objVal, solution)
38: end function
```

3.6 Algorytm mrówkowy

3.6.1 Działanie

Należy zdefiniować następujące zmienne i heurystyki:

- num_ants - liczba mrówek w każdej iteracji.
- num_iterations - liczba iteracji algorytmu.
- ρ - współczynnik parowania feromonów, określający, jak szybko feromony zanikają.
- τ_{init} - początkowa wartość feromonów na wszystkich krawędziach.
- α - wpływ poziomu feromonów na decyzję wyboru ścieżki.
- β - wpływ heurystyki lokalnej (tutaj liczby sąsiadów) na decyzję wyboru ścieżki.

Na początku należy zainicjować feromony na każdej krawędzi wejściowego grafu.

Dla zdefiniowanej liczby iteracji algorytmu, a następnie dla każdej mrówki:

1. na podstawie heurystyk, parametrów, sąsiadów wierzchołków i feromonów budowane jest rozwiązanie dla pojedynczej mrówki w postaci prawdopodobieństw wartości na wierzchołkach grafu.

Każdy wierzchołek i przyjmuje wartość $v \in \{0, 1, 2\}$ z prawdopodobieństwem:

$$P_i(v) = \frac{\left(\sum_{j \in N(i)} \tau_{ij} \right)^\alpha \cdot (\eta_i)^\beta}{\sum_{v' \in \{0, 1, 2\}} \left(\sum_{j \in N(i)} \tau_{ij} \right)^\alpha \cdot (\eta_i)^\beta}$$

gdzie:

- τ_{ij} to poziom feromonów na krawędzi (i, j) ,
- $\eta_i = |N(i)|$ to liczba sąsiadów wierzchołka i będąca heurystyką

2. rozwiązanie sprawdzane jest pod katem poprawności względem definicji. Sprawdzane jest, czy wierzchołki z wartościami 0, posiadają sąsiada z wartościami 2 oraz warunek słabo spójności. Jeśli warunek nie jest spełniony, rozwiązanie otrzymuje nieskończoną wartość wagi, co sprawia, że nie jest brane pod uwagę w rozwiązaniu. W przeciwnym razie wartości na wierzchołkach grafu są sumowane.

Po przejściu wszystkich mrówek, aktualniamy wartości feromonów na grafie, zmniejszając ich wartość o współczynnik ewaporacji oraz aktualizację wartości feromonów na podstawie najlepszego dotychczasowego rozwiązania.

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij} + \Delta \tau_{ij}$$

gdzie:

- ρ to współczynnik parowania feromonów,
- $\Delta \tau_{ij}$ to ilość feromonów dodana w oparciu o najlepsze rozwiązanie:

$$\Delta \tau_{ij} = \frac{1}{f(best_solution^*)}$$

gdzie $best_solution^*$ to najlepsze znalezione rozwiązanie, a $f(best_solution^*)$ to suma wartości wierzchołków grafu najlepszego rozwiązania.

3.6.2 Poprawność

Algorytm mrówkowy, choć dąży do najmniejszej wartości sumy wag WCRDF, to nie zakłada, że znalezione zostanie $\gamma_R^{wc}(G)$, dlatego sprawdzana jest wyłącznie poprawność słabo spójności. Dzieje się to w sposób analogiczny jak w algorytmie Brute Force, co zostało pokazane w punkcie 3.2.2.

3.6.3 Złożoność i wydajność

Algorytm działa przez $num_iterations$ iteracji i dla każdej mrówki num_ants . Budowanie rozwiązania wykonuje się w czasie $O(n)$, gdzie n to liczba wierzchołków grafu G , a m to liczba jego krawędzi, sprawdzenie rozwiązania w czasie $O(n^2)$. Zatem złożoność wynosi $O(num_iterations \cdot num_ants \cdot n^2)$, co sprawia, że algorytm dla większych grafów może być kosztowny obliczeniowo. W pamięci przechowywane są feromony $O(m)$ oraz rozwiązania $O(num_ants \cdot n)$, więc złożoność pamięciowa wynosi $O(m + num_ants \cdot n)$.

3.6.4 Pseudokod

Algorytm mrówkowy -inicjalizacja

```
1: function InitializePheromones(graph)
2:   pheromones  $\leftarrow$  Assign initial pheromone value to all edges
3:   return pheromones
4: end function
5: function ChooseNodeValue(node, pheromones, neighbors)
6:   values  $\leftarrow \{0, 1, 2\}$ 
7:   Initialize probabilities as empty list
8:   for each value in  $\{0, 1, 2\}$  do
9:     Compute pheromone_level as sum of pheromones of neighboring edges
10:    Compute probability as  $(\text{pheromone\_level}^\alpha) \times (\text{heuristic}^\beta)$ 
11:    Append probability to probabilities
12:   end for
13:   Normalize probabilities
14:   return Random weighted choice from  $\{0, 1, 2\}$ 
15: end function
16: function BuildSolution(graph, pheromones)
17:   Initialize node_values as empty dictionary
18:   for each node in graph do
19:     neighbors  $\leftarrow$  list of node's neighbors
20:     Assign node_values[node]  $\leftarrow$  ChooseNodeValue(node, pheromones, neighbors)
21:   end for
22:   return node_values
23: end function
24: function IsValidRomanDominatingSet(graph, node_values)
25:   for each node in graph do
26:     if node_values[node] = 0 and no neighbor has value 2 then
27:       return False
28:     end if
29:   end for
30:   induced_set  $\leftarrow$  nodes with values  $\{1, 2\}$ 
31:   for each node in induced_set do
32:     Add all neighbors to induced_set
33:   end for
34:   Create empty graph induced_graph
35:   for each node in induced_set do
36:     if node_values[node]  $\in \{1, 2\}$  then
37:       for each neighbor in graph do
38:         if neighbor in induced_set then
39:           Add edge between node and neighbor in induced_graph
40:         end if
41:       end for
42:     end if
43:   end for
44:   if induced_graph is not connected then
45:     return False
46:   end if
47:   return True
48: end function
```

Algorytm mrówkowy - główna pętla

```
1: function UpdatePheromones(graph, pheromones, solutions)
2:   for each edge in pheromones do
3:     Reduce pheromone level using evaporation rate
4:   end for
5:   best_solution ← Solution with minimum Roman number
6:   for each node in best_solution do
7:     for each neighbor of node do
8:       Increase pheromone level on edge (node, neighbor)
9:     end for
10:   end for
11: end function
12: function Execute(graph)
13:   pheromones ← InitializePheromones(graph)
14:   best_solution ← None
15:   best_roman_number ← ∞
16:   for each iteration in num_iterations do
17:     Initialize solutions as empty list
18:     for each ant in num_ants do
19:       solution ← BuildSolution(graph, pheromones)
20:       roman_number ← EvaluateSolution(graph, solution)
21:       Append (solution, roman_number) to solutions
22:       if roman_number < best_roman_number then
23:         Update best_roman_number and best_solution
24:       end if
25:     end for
26:     UpdatePheromones(graph, pheromones, solutions)
27:   end for
28:   return (best_roman_number, best_solution)
29: end function
```

3.7 Algorytm aproksymacyjny

3.7.1 Działanie

Algorytm ten został również zaimplementowany na podstawie artykułu [7]. Zdefiniowano tam następujące kroki algorytmu:

1. Wyznaczenie zbioru dominującego spełniającego warunki zbioru dominującego spójnego, dalej *Connected dominating set* - CDS.
2. Dla wierzchołków ze zbioru dominującego przypisywana jest wartość 2, dla pozostałych wierzchołków przypisywana jest wartość 0.

Udowodniono, że takie rozwiązań jest aproksymacyjne, o współczynniku aproksymacji równym $2(1 + \epsilon)(1 + \ln(\Delta - 1))$. Aby móc przetestować w pełni działanie algorytmu, zaimplementowano go w całości, na podstawie sugestii obecnych w artykule:

1. Wyznaczenie minimalnego zbioru dominującego spójnego. Problem ten również jest NP-trudny, dlatego zbiór ten wyznaczono przy użyciu programowania liniowego, w celu zachowania poprawności rozwiązania oraz - dla wielu przypadków - w rozsądny czasie. Ten fragment algorytmu zaimplementowano również na podstawie literatury[13].

Niech $G = (V, E)$ będzie grafem nieskierowanym, gdzie:

- V - zbiór wierzchołków,
- $E \subseteq V \times V$ - zbiór krawędzi.

Należy zdefiniować funkcję celu jako minimalizację liczby wierzchołków w zbiorze dominującym:

$$Z : \min \sum_{v \in V} x_v$$

Należy zdefiniować następujące zmienne decyzyjne:

$$x_v = \begin{cases} 1, & \text{jeśli } v \in \text{CDS} \\ 0, & \text{jeśli } v \notin \text{CDS} \end{cases}$$

oraz $f_{uv} \in Z_{\geq 0}$ — ilość jednostek przepływu z u do v , dla każdej krawędzi $(u, v) \in E$

Ograniczenia:

Dominacja — każdy wierzchołek musi być zdominowany sam lub przez sąsiada:

$$\forall v \in V : x_v + \sum_{u \in N(v)} x_u \geq 1, \quad (1)$$

Bilans przepływu dla każdego wierzchołka $v \neq r$:

$$\forall v \in V \setminus \{r\} : \sum_{u \in N(v)} f_{uv} - \sum_{u \in N(v)} f_{vu} = x_v, \quad (2)$$

Źródło przepływu — wysyła dokładnie $\sum x_v - 1$ jednostek przepływu:

$$\sum_{u \in N(r)} f_{ru} = \sum_{v \in V} x_v - 1, \quad (3)$$

Ograniczenie przepływu — przepływ możliwy tylko między wierzchołkami należącymi do CDS:

$$\forall (u, v) \in E : f_{uv} \leq (|V| - 1) \cdot x_u \quad \text{oraz} \quad f_{uv} \leq (|V| - 1) \cdot x_v, \quad (4)$$

2. Dla wierzchołków CDS przypisywana jest wartość 2, dla pozostałych wierzchołków przypisywana jest wartość 0.

3.7.2 Poprawność

Algorytm opiera się na wyznaczeniu spójnego zbioru dominującego (CDS) przy pomocy programowania całkowitoliczbowego, a następnie przypisaniu wszystkim wierzchołkom należącym do tego zbioru wartości $f(v) = 2$, a pozostałym $f(v) = 0$. Wystarczy zatem wykazać, że model ILP rzeczywiście wyznacza spójny zbiór dominujący, a takie przypisanie będzie spełniało definicję WCRDF, ale niekoniecznie optymalnie.

Poprawność modelu ILP wyznaczającego CDS

Model zawiera zmienne binarne x_v określające, czy wierzchołek $v \in V$ należy do zbioru dominującego. Warunki modelu:

- Dominacja: dla każdego wierzchołka $v \in V$, zachodzi:

$$x_v + \sum_{u \in N(v)} x_u \geq 1$$

co gwarantuje, że każdy wierzchołek jest albo w CDS, albo sąsiaduje z wierzchołkiem z CDS.

- Spójność: warunki przepływu zapewniają, że zbiór $\{v : x_v = 1\}$ tworzy spójny podgraf:
 - Jedno źródło r wysyła $\sum x_v - 1$ jednostek przepływu.
 - Każdy inny wierzchołek w CDS musi otrzymać dokładnie jedną jednostkę.
 - Przepływ f_{uv} jest dozwolony tylko wtedy, gdy $x_u = 1$ i $x_v = 1$.

Taka konstrukcja odpowiada przesyłaniu przepływu po drzewie rozpinającym na CDS, co zapewnia jego spójność.

3.7.3 Złożoność i wydajność

W pesymistycznym przypadku rozwiązanie modelu programowania liniowego wynosi $O(2^n)$, aby przeszukać całą przestrzeń binarną. Model zawiera n zmiennych x_v oraz $2m$ zmiennych przepływu f_{uv} . Dodatkowo zawiera $n + 2m$ ograniczeń. Zatem liczba zmiennych oraz ograniczeń daje złożoność $O(n + m)$.

3.7.4 Pseudokod

Algorytm aproksymacyjny

```

1: function ComputeDominatingSet( $G$ )
2:    $N \leftarrow$  list of nodes in  $G$ 
3:    $E \leftarrow$  list of edges in  $G$ 
4:    $x_v \in \{0, 1\}$  for all  $v \in N$ 
5:   Choose arbitrary root  $r \in N$ 
6:    $f_{uv} \in Z_{\geq 0}$  for all  $(u, v) \in E$  and reversed
7:   Minimize  $\sum_{v \in N} x_v$ 
8:   for all  $v \in N$  do
9:     Ensure:  $x_v + \sum_{u \in \text{Neighbors}(v)} x_u \geq 1$  ▷ Dominacja
10:    end for
11:    for all  $v \in N, v \neq r$  do
12:      Inflow  $\leftarrow \sum_{u \in \text{Neighbors}(v)} f_{uv}$ 
13:      Outflow  $\leftarrow \sum_{u \in \text{Neighbors}(v)} f_{vu}$ 
14:      Ensure: Inflow – Outflow =  $x_v$  ▷ Bilans przepływu
15:    end for
16:    Ensure:  $\sum_{u \in \text{Neighbors}(r)} f_{ru} = \sum_{v \in N} x_v - 1$  ▷ Źródło przepływu
17:    for all  $(u, v) \in E$  and  $(v, u)$  do
18:      Ensure:  $f_{uv} \leq (|N| - 1) \cdot x_u$ 
19:      Ensure:  $f_{uv} \leq (|N| - 1) \cdot x_v$  ▷ Warunki spójności
20:    end for
21:    Solve
22:    return  $\{v \in N : x_v = 1\}$ 
23: end function
24: function Execute( $G$ )
25:    $N \leftarrow$  list of nodes in  $G$ 
26:    $D \leftarrow$  ComputeDominatingSet( $G$ )
27:   for all  $v \in N$  do
28:     if  $v \in D$  then
29:        $value[v] \leftarrow 2$ 
30:     else
31:        $value[v] \leftarrow 0$ 
32:     end if
33:   end for
34:    $R \leftarrow \sum_{v \in N} value[v]$ 
35:   return  $(R, value)$ 
36: end function

```

3.8 Algorytm zachłanny

3.8.1 Działanie

1. Początkowo należy zdefiniować zbiory wierzchołków chronionych (początkowo pusty) i niechronionych (początkowo wszystkie wierzchołki grafu).
2. Wybierany jest wierzchołek o największym stopniu, jako wierzchołek startowy. Przypisywana mu jest wartość 2, a następnie jest on dodawany do zbioru wierzchołków chronionych, usuwając go z niechronionych. Analogicznie postępowanie jest z sąsiadami tego wierzchołka, tj. usunięcie ich ze zbioru wierzchołków niechronionych, dodając do chronionych.
3. Następnie, dopóki istnieją wierzchołki w zbiorze wierzchołków niechronionych:
 - Iteracja po zbiorze wierzchołków chronionych, wyznaczając ich sąsiadów. Jeśli znajduje się wierzchołek niechroniony, jest on dodawany do zbioru wierzchołków „kandydujących”.
 - Jeśli zbiór „kandydatów” jest pusty, pętla jest przerywana,
 - W przypadku znalezienia „kandydatów”, wybierany jest wierzchołek, który zabezpieczy jak najwięcej wierzchołków jeszcze niechronionych.
 - Następnie wykonywane jest sprawdzenie, czy będą istnieć nowo-ochronieni sąsiedzi. Jeśli nie, to nie ma potrzeby ustawiać temu wierzchołkowi wartości 2, przypisywana jest mu wartość 1. Jeśli istnieją tacy sąsiedzi, to przypisywana jest temu wierzchołkowi wartość 2.

3.8.2 Poprawność

Algorytm zachłanny działa dla grafów spójnych i nie gwarantuje rozwiązań optymalnego, natomiast zapewnia, że zwracane jest WCRDF zgodne z definicją. Dowód poprawności opiera się na dwóch kluczowych własnościach:

Warunek dominacji rzymskiej Algorytm rozpoczyna od przypisania wartości 2 wierzchołkowi o największym stopniu oraz zabezpieczenia jego sąsiadów. Następnie, dopóki istnieją wierzchołki niechronione (czyli z przypisaną wartością 0), wybierany jest wierzchołek v , który dominuje jak największą liczbę sąsiadów.

W każdej iteracji:

- Jeśli v zabezpiecza nowych sąsiadów, otrzymuje wartość $f(v) = 2$,
- Jeśli nie zabezpiecza nowych wierzchołków, ale należy do zbioru dominującego, otrzymuje $f(v) = 1$,
- Po każdej takiej decyzji zabezpieczani są również jego sąsiedzi.

W ten sposób każdy wierzchołek z wartością $f(v) = 0$ musi być sąsiadem jakiegoś wierzchołka z $f = 2$, ponieważ algorytm nie zatrzymuje się, dopóki zbiór niechronionych wierzchołków nie jest pusty.

Warunek słabo spójności Zbiór $D = \{v \in V : f(v) \in \{1, 2\}\}$ jest budowany iteracyjnie. W każdej iteracji dodawany jest do niego wierzchołek v , który jest sąsiadem jakiegoś wcześniej dodanego wierzchołka, wybranego ze zbioru „kandydatów”.

Zatem:

- pierwszy wierzchołek startowy tworzy początek spójnego zbioru D ,
- każdy kolejny dodany wierzchołek jest połączony z wcześniejszym,
- czyli zbiór D rozszerza się w sposób, który zapewnia spójność grafu słabo indukowanego przez D .

Co kończy dowód.

3.8.3 Złożoność i wydajność

Załóżmy, że $n = |V|$ to liczba wierzchołków grafu G oraz $m = |E|$ to liczba krawędzi grafu G . Inicjacja wszystkich zmiennych wykonuje się w czasie liniowym. Iteracyjna ochrona wszystkich wierzchołków w najgorszym przypadku może wynieść $O(n)$, bo każda iteracja zabezpiecza przy najmniej jeden wierzchołek. Dodatkowo iterujemy przez niezabezpieczone jeszcze wierzchołki w maksymalnie $O(n)$ oraz ich sąsiadów, czyli w najgorszym wypadku po wszystkich krawędziach grafu - $O(m)$. Dodatkowe aktualizacje w głównej pętli wykonują się w czasie maksymalnie liniowym. Dlatego ostatecznie złożoność obliczeniowa wynosi $O(n(m+n))$. Zakładając, że dla grafów rzadkich $m = O(n)$, a dla grafów gęstych $m = O(n^2)$ to złożoność czasowa wynosi odpowiednio $O(n^2)$ oraz $O(n^3)$.

Złożoność pamięciowa obejmuje struktury danych przechowujące graf wejściowy, dlatego wynosi ona $O(n+m)$.

Algorytm nie jest dokładny, natomiast w stosunku do innych proponowanych algorytmów, charakteryzuje się wielomianową złożonością obliczeniową.

3.8.4 Pseudokod

Algorytm zachłanny

```
1: function Execute( $G$ )
2:    $f[v] \leftarrow 0$  for all  $v \in \text{nodes of } G$ 
3:    $\text{secured\_nodes} \leftarrow \emptyset$ 
4:    $\text{uncovered\_nodes} \leftarrow \text{set of all nodes in } G$ 
5:    $\text{start\_node} \leftarrow \text{node with maximum degree}$ 
6:    $f[\text{start\_node}] \leftarrow 2$ 
7:   Add  $\text{start\_node}$  to  $\text{secured\_nodes}$ 
8:   Remove  $\text{start\_node}$  from  $\text{uncovered\_nodes}$ 
9:   for all neighbors  $\text{neighbor}$  of  $\text{start\_node}$  do
10:    Add  $\text{neighbor}$  to  $\text{secured\_nodes}$ 
11:    Remove  $\text{neighbor}$  from  $\text{uncovered\_nodes}$ 
12:   end for
13:   while  $\text{uncovered\_nodes}$  is not empty do
14:      $\text{candidate\_nodes} \leftarrow \emptyset$ 
15:     for all  $v \in \text{secured\_nodes}$  do
16:       for all neighbors  $\text{neighbor}$  of  $v$  do
17:         if  $\text{neighbor} \in \text{uncovered\_nodes}$  then
18:           Add  $\text{neighbor}$  to  $\text{candidate\_nodes}$ 
19:         end if
20:       end for
21:     end for
22:     if  $\text{candidate\_nodes}$  is empty then
23:       break
24:     end if
25:      $\text{node} \leftarrow \text{node in } \text{candidate\_nodes} \text{ covering the most uncovered neighbors}$ 
26:      $\text{new\_covered\_neighbors} \leftarrow \text{number of uncovered neighbors of } \text{node}$ 
27:     if  $\text{new\_covered\_neighbors} > 0$  then
28:        $f[\text{node}] \leftarrow 2$ 
29:     else
30:        $f[\text{node}] \leftarrow 1$ 
31:     end if
32:     Add  $\text{node}$  to  $\text{secured\_nodes}$ 
33:     Remove  $\text{node}$  from  $\text{uncovered\_nodes}$ 
34:     for all neighbors  $\text{neighbor}$  of  $\text{node}$  do
35:       Add  $\text{neighbor}$  to  $\text{secured\_nodes}$ 
36:       Remove  $\text{neighbor}$  from  $\text{uncovered\_nodes}$ 
37:     end for
38:   end while
39:    $\text{min\_roman\_domiantion\_number} \leftarrow \sum_v f[v]$ 
40:   return ( $\text{min\_roman\_domiantion\_number}, f$ )
41: end function
```

ROZDZIAŁ 4. WYNIKI

4.1 Wprowadzenie

Dla wszystkich, zaimplementowanych w poprzednim rozdziale algorytmów przeprowadzono testy pokazujące skuteczność ich działania na wybranych klasach grafów. Na początku przeprowadzono testy dla różnych kombinacji hiperparametrów algorytmu mrówkowego, w celu uzyskania najlepszego zestawu parametrów dla tego problemu, który będzie następnie wykorzystywany w dalszych rozważaniach. Następnie przedstawiono wyniki dla wybranych klas grafów:

- grafy rzadkie
- grafy gęste
- drzewa
- grafy bezskalowe

Grafy każdej klasy wygenerowano z następującą liczbą wierzchołków: [10, 20, 30, 40, 50], dla każdej liczby wierzchołków po 3 razy. Dla spójności, wszystkie algorytmy, tam gdzie to ma sens, testowano na tych samych wygenerowanych grafach. Dalej w tabelach zostaną przedstawione wyniki poglądowe jednego z trzech takich zestawów. Wszystkie średnie pomiary czasów w tabelach przedstawione w sekundach. Pomiary dla każdego przypadku wykonywano pięciokrotnie w celu otrzymania wiarygodnego wyniku. Pomiary mierzone w nanosekundach z niepewnością pomiaru wynoszącą 100 nanosekund, wykorzystując funkcję *time.perf_counter_ns()*. Dodatkowo dla każdej klasy przedstawiono wykresy grafów z WCRDF po użyciu algorytmów oraz wykres porównujący czas działania każdego z algorytmów w zależności od liczby wierzchołków, w skali logarytmicznej.

W późniejszej części rozdziału rozważano również skuteczność algorytmów wyznaczających prawidłowe WCRDF, ale niekoniecznie optymalne $\gamma_R^{wc}(G)$.

Na podstawie wyników zostaną przedstawione obserwacje oraz wnioski dla każdej z klas grafów oraz dla algorytmów niedokładnie wyznaczających $\gamma_R^{wc}(G)$.

4.2 Hiperparametry algorytmu mrówkowego

W algorytmie mrówkowym zostały zdefiniowane następujące parametry:

- α - wpływ poziomu feromonów na decyzję wyboru ścieżki. Przyjęte do testów wartości $\alpha = [3, 1]$
- β - wpływ heurystyki lokalnej (liczby sąsiadów) na decyzję wyboru ścieżki. Przyjęte do testów wartości $\beta = [5, 20]$
- ρ - współczynnik parowania feromonów, określający, jak szybko feromony zanikają. Przyjęte do testów wartości $\rho = [0.5, 0.7]$
- liczba mrówek w każdej iteracji. Przyjęte do testów wartości [100, 150]

W poniższej tabeli zostały przedstawione, wybrane zestawy tych hiperparametrów osiągające najmniejszy średni błąd. Średni błąd był wyznaczany na podstawie dokładnego wyniku osiąganego przez algorytmy dokładnie wyznaczające $\gamma_R^{wc}(G)$ dla danego przykładu grafu.

α	β	ρ	liczba mrówek	średni błąd
3	20	0.5	150	7.933333
3	20	0.7	150	8.000000
1	5	0.7	150	8.000000
1	20	0.7	150	8.066667
1	20	0.5	100	8.066667

Tabela 4.1: Wpływ kombinacji hiperparametrów na błąd rozwiązania

Średni błąd wygląda na znaczny, ale można zauważyc, że ogólnie algorytm jest stabilny oraz kombinacja hiperparametrów nieznacznie wpływa na jakość osiąganych wyników. Dominacja wartości $\beta = 20$ wskazuje na, to że algorytm bardziej polega na heurystyce lokalnej, którą jest liczba sąsiadów wierzchołka, w kontekście rozwiązania tego problemu, co jest spodziewanym wynikiem.

Dodatkowo, wolniejsze parowanie feromonów (ρ) zdaje się sprzyjać lepszym wynikom. Oznacza to, że dłuższe zapamiętywanie informacji o dobrych rozwiązaniach wpływa korzystnie na wynik.

Kolejna tabela przedstawia zestaw parametrów z najmniejszym średnim błędem dla danej klasy grafu.

Klasa grafów	α	β	ρ	liczba mrówek	średni błąd
ErdosRenyi_dense	1	20	0.5	100	6.5
ErdosRenyi_sparse	3	20	0.5	150	8.666667
RandomTree	1	5	0.7	150	7.5
ScaleFree	3	20	0.7	150	7.25

Tabela 4.2: Wpływ kombinacji hiperparametrów na błąd rozwiązania na konkretnych klasach grafów

Dla grafów gęstych udało się dobrać korzystny zestaw hiperparametrów, dający stosunkowo niski średni błąd. W tym przypadku algorytm woli bardziej się skupić na heurystyce lokalnej niż na wpływie feromonów, co jest spodziewanym zachowaniem ze względu na to, że w grafach gęstych bardzo często dla WCRDF to wierzchołki o największym stopniu są najbardziej promowane.

Dla grafów rzadkich widoczny jest dodatkowo większy wpływ feromonów. Ze względu na rzadzącą strukturę grafu, korzystnym zachowaniem jest zapamiętanie dobrego rozwiązania, a następnie eksploatawanie go. Podobnie dla grafów bezskalowych, jednakże kładziony jest tutaj nacisk na

większą eksplorację, ze względu na to, że struktura grafu jest podobna do grafów rzadkich. Drzewa w stosunku do pozostałych klas grafu mają najbardziej specyficzną strukturę i w tych wynikach się to również odzwierciedla. W tym przypadku nie jest kładziony nacisk na heurystykę lokalną, ponieważ w drzewach bardzo często wierzchołki o wysokich stopniach niekoniecznie należą do zbioru dominującego. Algorytm w tym przypadku jest bardziej nastawiony na eksplorację, bo można zaobserwować zrównoważony wpływ heurystyki lokalnej i feromonu, przy dużym współczynniku parowania.

Na podstawie powyższych parametrów można stwierdzić, że są one adekwatne do właściwości konkretnej struktury klasy grafu. Mimo to, średni błąd we wszystkich przypadkach jest znaczny, ale dosyć stabilny, nie ma wielkich różnic między klasami grafów.

Po tej analizie dokonano wyboru hiperparametrów, które będą użyte w dalszych rozważaniach. Został wybrany jeden zestaw ze względu na stabilność wartości średniego błędu między różnymi zestawami hiperparametrów, jak i między klasami grafów. Hiperparametry te zostały przedstawione w poniżej tabeli.

α	β	ρ	liczba mrówek
3	20	0.5	150

Tabela 4.3: Wybrany zestaw parametrów

Wybór ten oznacza, że heurystyka lokalna w postaci stopnia wierzchołka grafu oraz feromony będą miały istotny wpływ na wyniki algorytmu. Algorytm będzie zapamiętywał dobre wyniki i je eksploatował.

4.3 Wyniki działania algorytmów

Wybrane wyniki działania algorytmów zostały przedstawione w tabelach. Wyniki podzielono na sekcje względem liczby wierzchołków oraz posortowano według rosnącego czasu działania w obrębie każdej z sekcji. Algorytmy ILP, ILP2, Brute Force oraz TreeLinear dla drzew są w tych przypadkach algorytmami optymalnie wyznaczającymi wartość $\gamma_R^{wc}(G)$ i to one stanowią wartość odniesienia się do dokładności pozostałych algorytmów.

Dodatkowo zostaną zaprezentowane wykresy przedstawiające średni czas działania każdego z algorytmów w stosunku do liczby wierzchołków w skali logarytmicznej. Mierzone punkty zostały połączone liniami w celu lepszej wizualizacji i odzwierciedlenia tendencji wzrostu.

Następnie przedstawiono grafiki ukazujące rozkłady WCRDF dokonane przez algorytmy dla przykładowego, 20-wierzchołkowego grafu analizowanej klasy, gdzie wartości 0 to żółte wierzchołki, 1 - niebieskie, a 2 to czerwone.

4.3.1 Grafy rzadkie

Grafy rzadkie charakteryzują się znacznie mniejszą liczbą krawędzi niż maksymalna możliwa, czyli $|E| \ll \frac{n(n-1)}{2}$ oraz posiada przede wszystkim wierzchołki o niskim stopniu. Z racji mniejszej liczby krawędzi algorytmy programowania liniowego mogą sobie dobrze radzić, ze względu na mniejszą liczbę ograniczeń i zmiennych. Ze względu na nieregularną strukturę i komponenty o różnym stopniu połączenia, algorytmy deterministyczne mogą znacząco odbiegać od optymalnego wyniku.

Algorytm	Liczba wierzchołków	Liczba krawędzi	$\gamma_R^{wc}(G)$	Średni czas (s)
Greedy	10	15	5	0,0000253
	10	15	5	0,03225824
	10	15	5	0,06530336
	10	15	6	0,0716367
	10	15	5	1,22720146
	10	15	5	1,9432132
ILP	20	61	7	0,00009044
	20	61	7	0,17597988
	20	61	8	0,81655686
	20	61	7	1,28852564
	20	61	11	4,302339
ILP2	30	122	10	0,00018332
	30	122	9	0,4925489
	30	122	10	1,27100858
	30	122	9	3,89847968
	30	122	19	10,9692404
Approx	40	215	11	0,00015998
	40	215	8	0,4383133
	40	215	10	3,70053236
	40	215	8	6,50026534
	40	215	25	12,11287052
AntColony	50	353	10	0,00027048
	50	353	8	3,9218204
	50	353	10	4,86207684
	50	353	30	28,00923878
	50	353	8	88,8271173

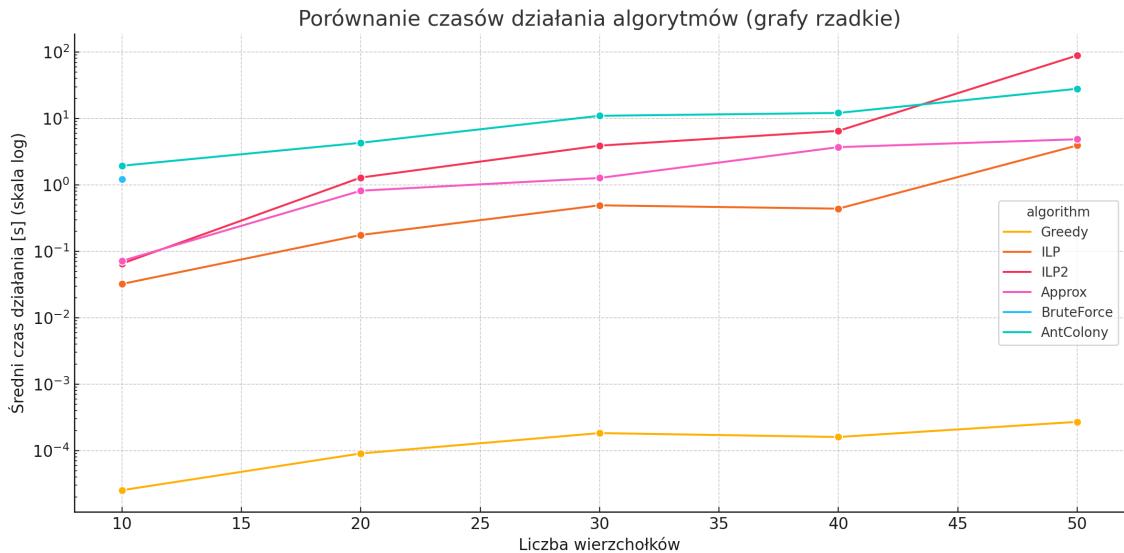
Tabela 4.4: Wyniki dla grafów rzadkich

Analizując powyższe wyniki można zauważyc, że algorytm zachłanny Greedy działa bardzo dobrze dla rzadkich grafów. Charakteryzuje się bardzo szybkim czasem działania, a jakość

rozwiązania jest bliska optymalnej.

Algorytm mrówkowy wraz ze wzrostem liczby wierzchołków zaczyna dawać coraz gorsze wyniki, mimo wyboru dobrego dla tej klasy zestawu parametrów. Warto zaznaczyć również, że jest jednym z wolniej działających algorytmów.

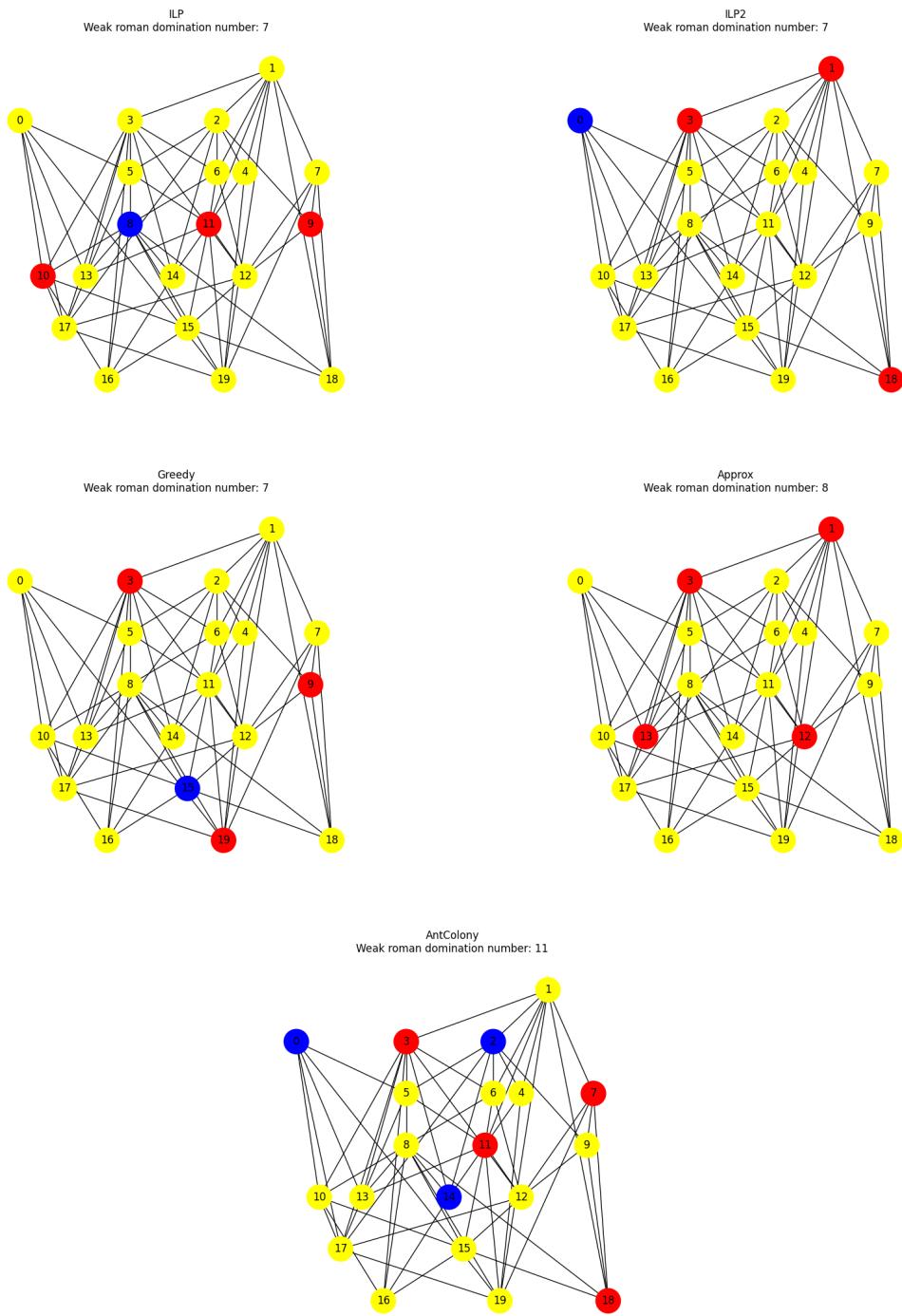
Wynik dla algorytmu Brute Force można zaobserwować wyłącznie dla 10 wierzchołków. Dzieje się tak, ponieważ zgodnie z pierwotnymi założeniami, już dla kilkunastu wierzchołków w dowolnym grafie, algorytm ten przestał zwracać wyniki w rozsądny czasie.



Rysunek 4.1: Porównanie średniego czasu działania algorytmów na grafach rzadkich w stosunku do liczby wierzchołków w grafie.

W tym przypadku znacznie odznacza się algorytm Greedy, który działa ponad 1000 razy szybciej niż pozostałe algorytmy, przy czym nie osiąga wyników znacznie odbiegających od optymalnego. Algorytmy ILP i ILP2 dają optymalny wynik, w rozsądny czasie działania, przy czym ILP jest szybszy.

Naturalnie, wraz ze wzrostem liczby wierzchołków, a co za tym idzie i krawędzi, czas pracy wszystkich algorytmów wydłuża się. Dla algorytmu zachłannego są to jednak na tyle krótkie czasy, że można stwierdzić, że algorytm będzie skalowalny i również wykonywalny w rozsądny czasie dla większych grafów. Pozostałe algorytmy notują znaczny skok w czasie działania, zwłaszcza dla większej liczby wierzchołków.



Rysunek 4.2: Wyniki dla przykładowego grafu rzadkiego.

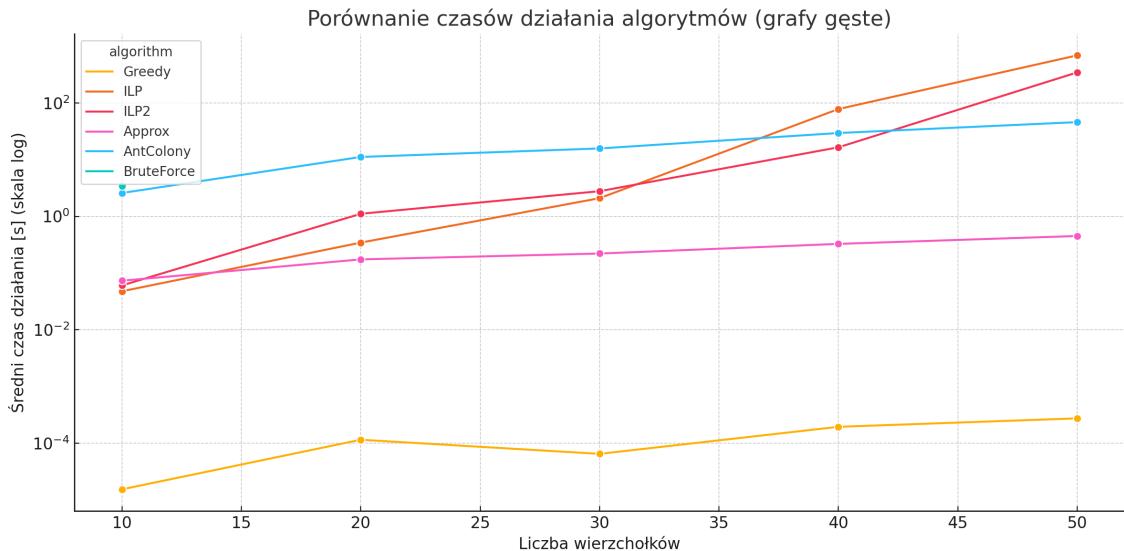
4.3.2 Grafy gęste

Grafy gęste charakteryzują się dużą liczbą krawędzi, bliską $|E| \ll \frac{n(n-1)}{2}$. Wierzchołki mają duże stopnie i dominacja jednego wierzchołka pokrywa znaczną część grafu. Dlatego algorytmy przybliżone powinny łatwo i szybko znajdować dobre rozwiązania. Z racji dużej liczby zmiennych i ograniczeń algorytmy ILP/ILP2 powinny charakteryzować się dłuższym czasem działania.

Algorytm	Liczba wierzchołków	Liczba krawędzi	$\gamma_R^{wc}(G)$	Średni czas (s)
Greedy	10	31	2	0,00001534
ILP	10	31	2	0,04773922
ILP2	10	31	2	0,06054642
Approx	10	31	2	0,07325536
AntColony	10	31	2	2,58175592
Brute Force	10	31	2	3,39913678
Greedy	20	132	4	0,00011488
Approx	20	132	4	0,17418222
ILP	20	132	4	0,34445046
ILP2	20	132	4	1,10987306
AntColony	20	132	9	11,15454742
Greedy	30	288	4	0,00006476
Approx	30	288	4	0,22087496
ILP	30	288	4	2,09136932
ILP2	30	288	4	2,7780365
AntColony	30	288	14	15,7567997
Greedy	40	522	5	0,0001943
Approx	40	522	4	0,32690006
ILP2	40	522	4	16,49655814
AntColony	40	522	21	29,47564454
ILP	40	522	4	77,93719362
Greedy	50	839	5	0,00027366
Approx	50	839	4	0,45039484
AntColony	50	839	28	45,90004724
ILP2	50	839	4	345,33312052
ILP	50	839	4	690,41462788

Tabela 4.5: Wyniki dla grafów gęstych

Algorytmy Brute Force, ILP, ILP2, Approx oraz Greedy wyznaczają optymalne, bądź bliższe optimum wartości $\gamma_R^{wc}(G)$. Algorytm mrówkowy wraz ze wzrostem liczby wierzchołków w grafie, jakość rozwiązania wypada coraz gorzej. Dla 50 wierzchołków jest to nawet 7-krotnie gorszy wynik od optymalnego. Algorytm Approx w większości przypadków wyznaczał dla grafów gęstych optymalne wyniki. Może mieć to związek z korzystną dla tego algorytmu struktura grafu, gdzie wierzchołki zbioru dominującego są zwykle swoimi sąsiadami, a wybór jednego wierzchołka daje korzystną ochronę wielu sąsiadów.



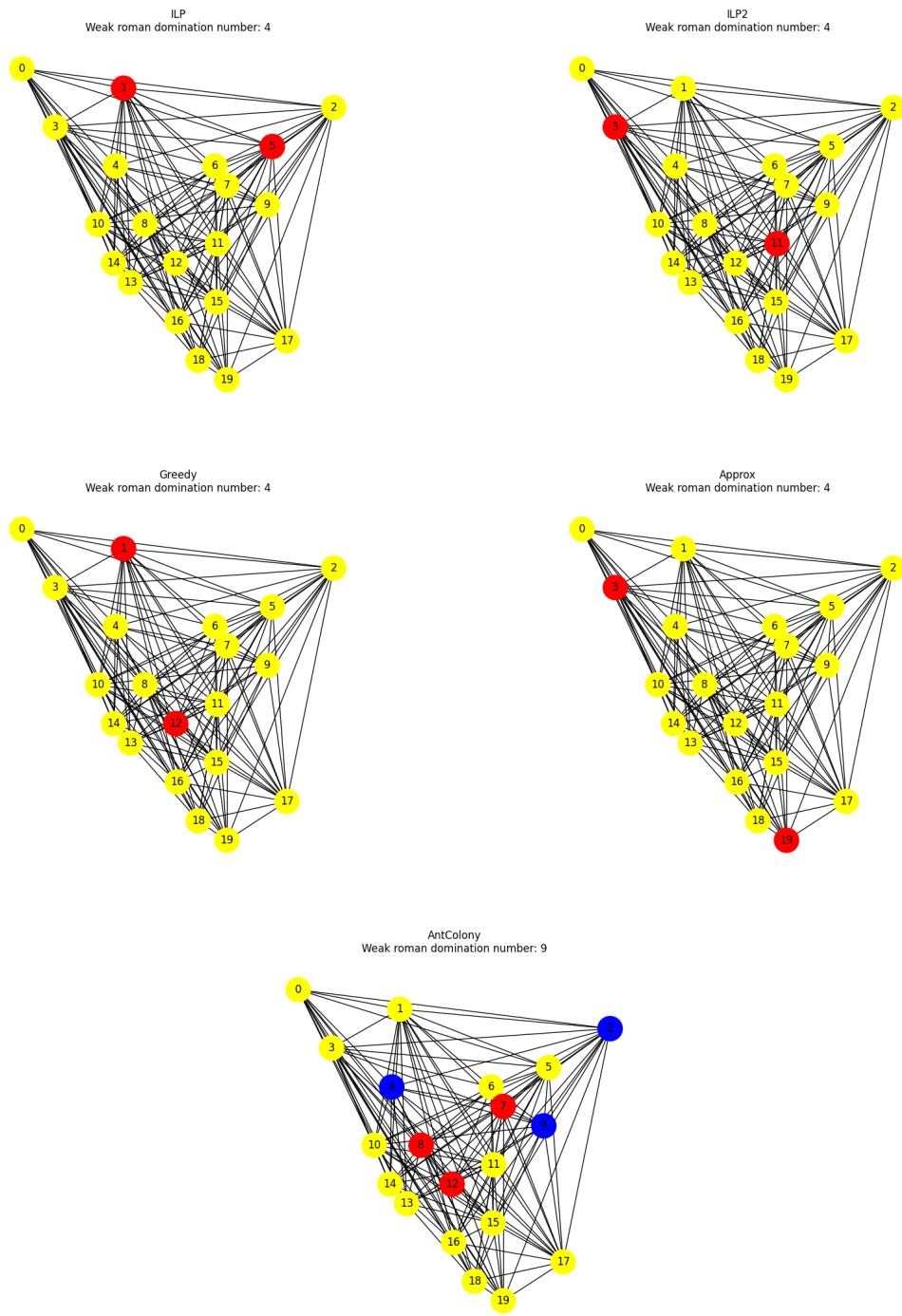
Rysunek 4.3: Porównanie średniego czasu działania algorytmów na grafach gęstych w stosunku do liczby wierzchołków w grafie.

W tym przypadku znacznie odznacza się algorytm Greedy, który działa ponad 1000 razy szybciej niż pozostałe algorytmy, przy czym nie osiąga wyników znacznie odbiegających od optymalnego. Algorytm Approx jest również efektywny czasowo, mimo, że do jego konstrukcji użyto programowania liniowego. Najprawdopodobniej powodem takiego wyniku jest korzystna struktura grafu dla tego algorytmu.

Naturalnie, wraz ze wzrostem liczby wierzchołków, a co za tym idzie i krawędzi, czas pracy wszystkich algorytmów wydłuża się. Dla algorytmu zachłannego są to jednak na tyle krótkie czasy, że można stwierdzić, że algorytm będzie skalowalny i również wykonywalny w rozsądny czasie dla większych grafów.

Algorytmy ILP i ILP2 notują mocny skok czasowy dla $n \geq 40$. Jest on wyraźniejszy i większy niż dla grafów rzadkich, co jest spodziewanym zachowaniem, z racji większej liczby krawędzi do przeanalizowania. W tym przypadku udało się potwierdzić hipotezę, że algorytm ILP2 będzie szybszy od ILP, co ma miejsce dla większej liczby wierzchołków. Wynika to z dużej liczby cykli grafu gęstego do przeanalizowania przez algorytm ILP. ILP2, oparty na przepływach, działa znacznie szybciej. Algorytm mrówkowy w stosunku do pozostałych jest dosyć stabilny czasowo, jednakże jego jakość jest mierna.

Gęste grafy posiadają dużo krawędzi, więc posiadają one wiele możliwości pokrycia dominującego. Algorytmy Approx i Greedy wykorzystują ochronę wielu wierzchołków z jednego wybranego, dlatego w tym przypadku wypadają one bardzo korzystnie czasowo i jakościowo. Ze względu na wzrost krawędzi, cierpią na tym algorytmy dokładne, ILP i ILP2. Heurystyka algorytmu mrówkowego jest nieefektywna dla tej klasy grafu.



Rysunek 4.4: Wyniki dla przykładowego grafu gęstego.

4.3.3 Drzewa

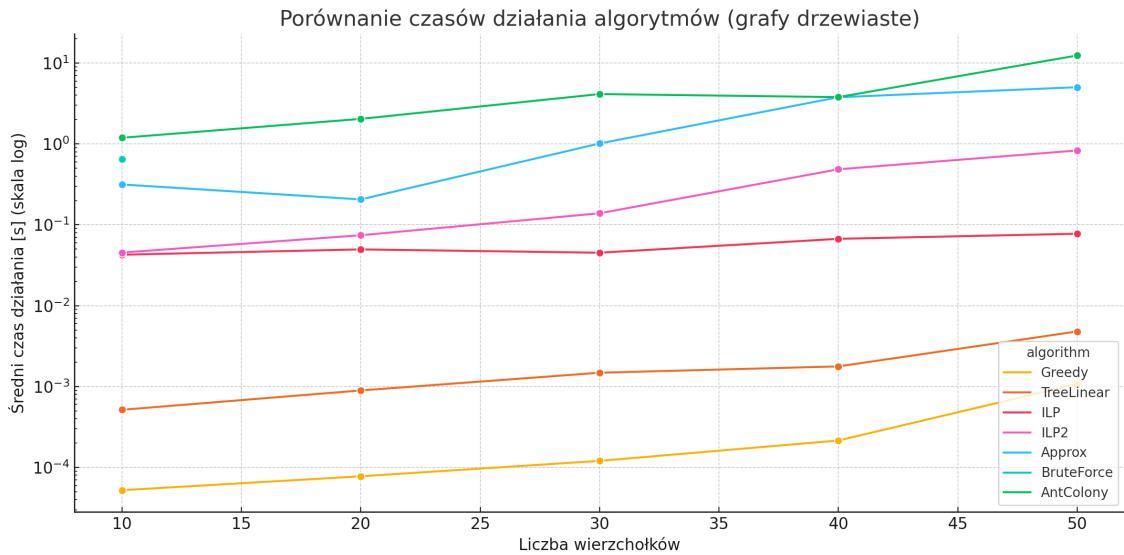
Drzewa to grafy acykliczne, posiadające $n-1$ krawędzi dla n wierzchołków. Są to grafy bardzo rzadkie, przez co posiadają rozproszoną strukturę dominacji. Dlatego algorytmy deterministyczne i przybliżone, które nie są specjalnie dostosowane pod drzewa, mogą osiągać niezadowalające wyniki. Dla takiej uporządkowanej struktury za to można znaleźć efektywny algorytm specjalny tylko dla tej klasy grafu.

Algorytm	Liczba wierzchołków	Liczba krawędzi	$\gamma_R^{wc}(G)$	Średni czas (s)
Greedy	10	9	8	0,00005232
TreeLinear	10	9	7	0,00051616
ILP	10	9	7	0,04267752
ILP2	10	9	7	0,04535534
Approx	10	9	14	0,3153601
Brute Force	10	9	7	0,65010432
AntColony	10	9	8	1,19096304
Greedy	20	19	12	0,00007772
TreeLinear	20	19	12	0,00089554
ILP	20	19	12	0,04966132
ILP2	20	19	12	0,07429058
Approx	20	19	16	0,20606074
AntColony	20	19	18	2,04032124
Greedy	30	29	20	0,0001205
TreeLinear	30	29	17	0,00148292
ILP	30	29	17	0,04517938
ILP2	30	29	17	0,13887008
Approx	30	29	36	1,0127093
AntColony	30	29	30	4,13727904
Greedy	40	39	33	0,00021536
TreeLinear	40	39	29	0,0017755
ILP	40	39	29	0,06708872
ILP2	40	39	29	0,48569172
Approx	40	39	48	3,7790714
AntColony	40	39	43	3,79114808
Greedy	50	49	45	0,00107456
TreeLinear	50	49	35	0,00480282
ILP	50	49	35	0,07759686
ILP2	50	49	35	0,82981668
Approx	50	49	66	5,0276923
AntColony	50	49	59	12,4118761

Tabela 4.6: Wyniki dla drzew

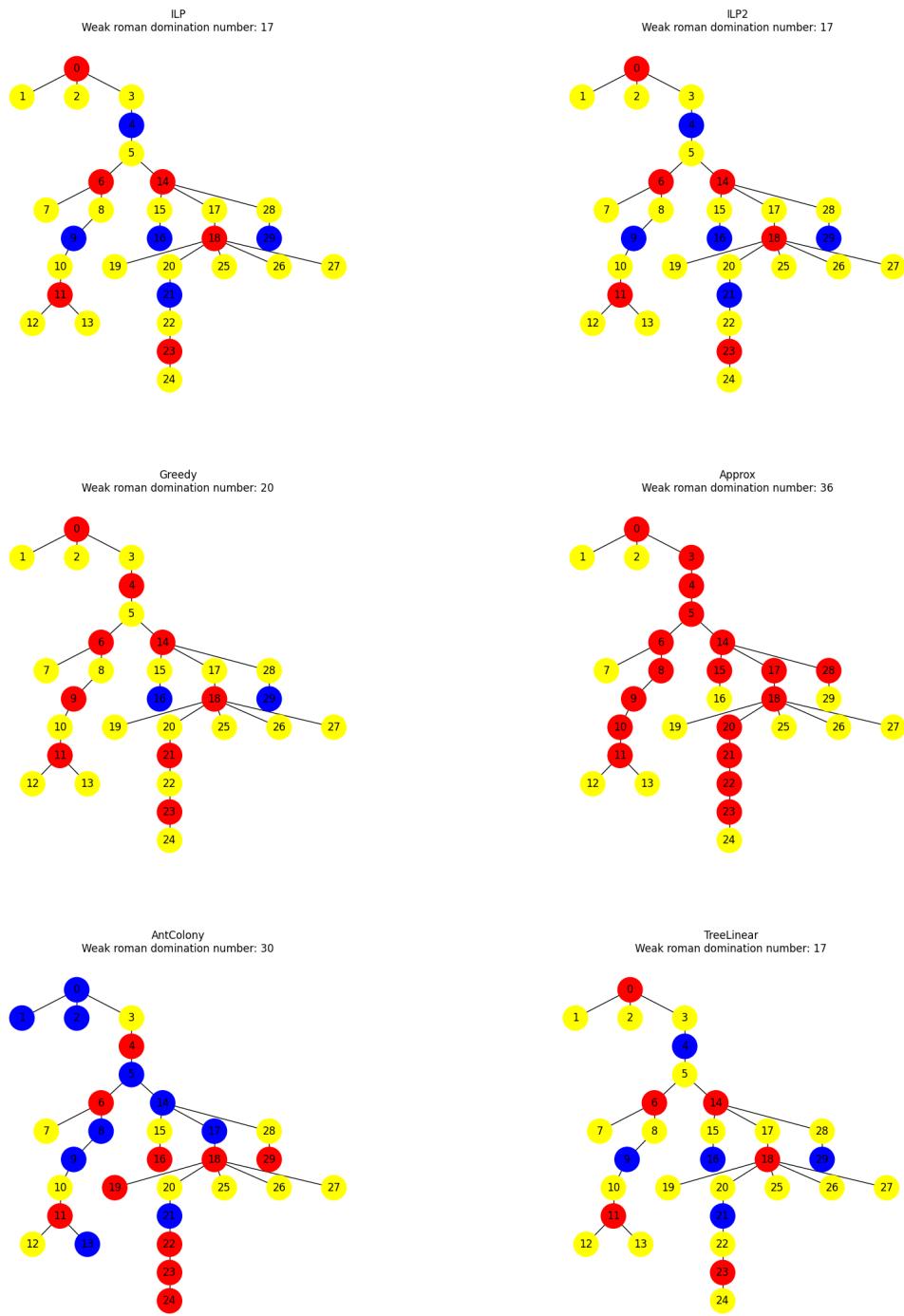
W tym przypadku algorytmy dokładnie wyznaczające $\gamma_R^{wc}(G)$ poradziły sobie również w rozsądny czasie. Wynika to z bardzo rzadkiej struktury grafu. W pozostałych przypadkach, wraz ze wzrostem liczby wierzchołków wyniki osiągane przez algorytmy przybliżone były coraz gorsze jakościowo, co wynika ze specyficznej struktury grafu, gdzie wybór wierzchołków zbioru dominującego słabo spójnego jest nietypowy w stosunku do innych klas grafów. Godny odnotowania

jest tutaj fakt, że akurat w tym przypadku algorytm mrówkowy poradził sobie lepiej niż Approx, ze względu na to, że Approx tworzy zbiór dominujący spójny z samymi wartościami 2, co dla drzew będzie wysoce nieoptymalne.



Rysunek 4.5: Porównanie średniego czasu działania algorytmów na drzewach w stosunku do liczby wierzchołków w grafie.

Algorytm liniowy dla drzew daje w tym zestawieniu najlepszy stosunek czasu działania do jakości. Nie dość, że daje optymalny wynik, to dodatkowo w szybkim czasie. Algorytm jest skalo-walny. Algorytmy ILP/ILP2 z powodu rzadkiej struktury grafu również wykonują się w rozsądny czasie. ILP2 działa wolniej w tym przypadku od ILP. Zachłanny wykonuje się bardzo szybko, ale daje niekorzystne wyniki. TreeLinear jest tylko trochę wolniejszy, ale za to daje optymalny wynik. Algorytmy Approx oraz AntColony działają na tyle wolno i nieoptymalnie, że można stwierdzić, że nie nadają się dla drzew. Najlepszym wyborem dla tej klasy grafów jest algorytm liniowy dla drzew.



Rysunek 4.6: Wyniki dla przykładowego drzewa.

4.3.4 Grafy bezskalowe

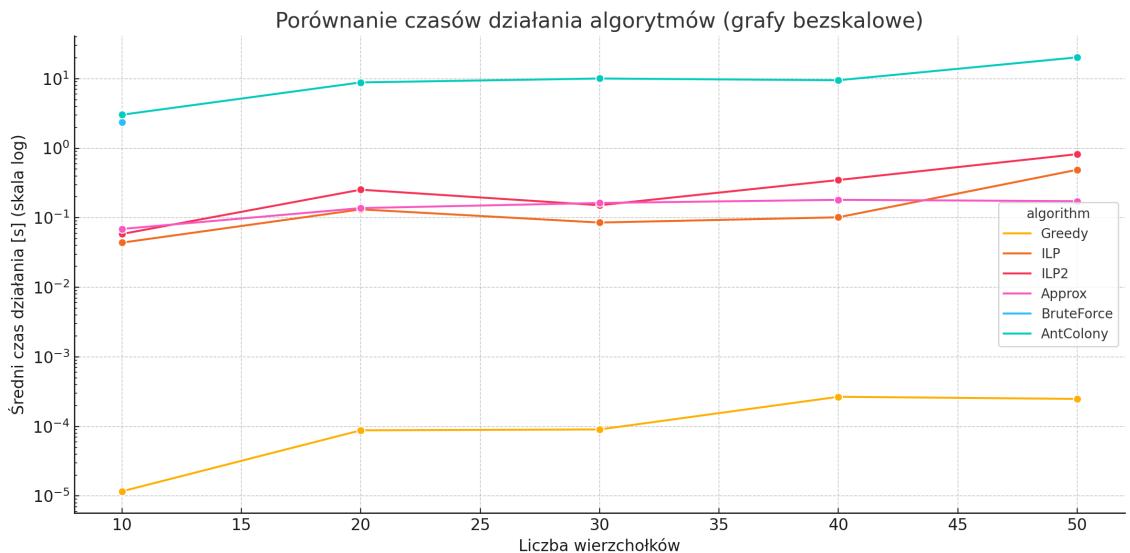
Grafy bezskalowe to grafy naturalnie występujące w rzeczywistości, na przykład w biologii, sieciach WWW i społecznościowych i są analizowane ze względu na potencjalne zastosowania praktyczne problemu WRCDF. Grafy te charakteryzują się wieloma wierzchołkami o małym stopniu i kilkoma wierzchołkami o dużym stopniu. Są to huby. Rozkład stopni wierzchołków w grafach tego typu jest potęgowy. Graf ten nie jest typowo gęsty, bo przeważają wierzchołki o niskim stop-

niu. Dlatego korzystne wyniki powinny osiągać algorytmy oparte na programowaniu liniowym.

Algorytm	Liczba wierzchołków	Liczba krawędzi	$\gamma_R^{wc}(G)$	Średni czas (s)
Greedy	10	25	2	0,00001166
ILP	10	25	2	0,04384846
ILP2	10	25	2	0,05840188
Approx	10	25	2	0,06902438
Brute Force	10	25	2	2,3746041
AntColony	10	25	3	3,02645522
Greedy	20	75	5	0,00008788
ILP	20	75	4	0,1319412
Approx	20	75	4	0,13767914
ILP2	20	75	4	0,25321824
AntColony	20	75	9	8,82321618
Greedy	30	125	9	0,00009044
ILP	30	125	6	0,08527778
ILP2	30	125	6	0,15141182
Approx	30	125	6	0,16305526
AntColony	30	125	18	10,0423744
Greedy	40	175	10	0,00026662
ILP	40	175	7	0,1015079
Approx	40	175	8	0,18101464
ILP2	40	175	7	0,3487031
AntColony	40	175	24	9,48435894
Greedy	50	225	14	0,00024856
Approx	50	225	10	0,17308662
ILP	50	225	10	0,48920074
ILP2	50	225	10	0,82045656
AntColony	50	225	35	20,21980648

Tabela 4.7: Wyniki dla grafów bezskalowych

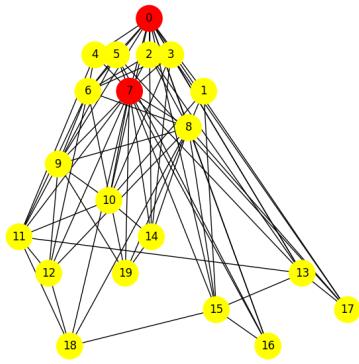
Algorytmy ILP/ILP2/Approx zadowalająco pokrywają huby i w rozsądny czasie, co w strukturze tego grafu jest bardzo ważne. Algorytm zachłanny jest niedokładny, ale wielkość błędu nie rośnie znacznie wraz ze wzrostem liczby wierzchołków. Algorytm mrówkowy nie dość, że za każdym razem działa naj wolniej ze wszystkich propozycji, to daje wraz ze wzrostem liczby wierzchołków coraz gorsze wyniki. Po raz kolejny wskazuje to na brak doboru odpowiedniej heurystyki do tego problemu. Oczywiście algorytm Brute Force daje poprawny wynik, aczkolwiek jest on zupełnie nieskalowalny na większą liczbę wierzchołków niż kilkanaście.



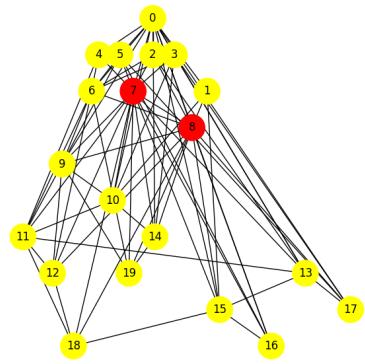
Rysunek 4.7: Porównanie średniego czasu działania algorytmów na grafach bezskalowych w stosunku do liczby wierzchołków w grafie.

Algorytmy Approx i Greedy są czasowo najefektywniejsze. Należy pamiętać, że Approx został zaimplementowany na podstawie programowania liniowego, więc wraz ze wzrostem liczby wierzchołków może zacząć działać bardzo długo. Algorytmy dokładne ILP/ILP2 działają w podobnym czasie co Approx. Algorytm mrówkowy jest przynajmniej 10 razy wolniejszy od innych algorytmów, co czyni go bardzo nieefektywnym.

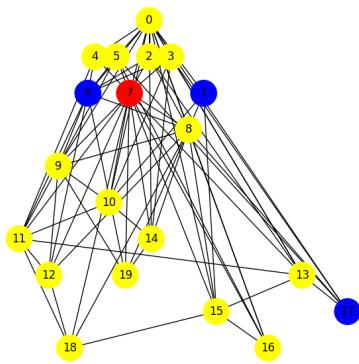
ILP
Weak roman domination number: 4



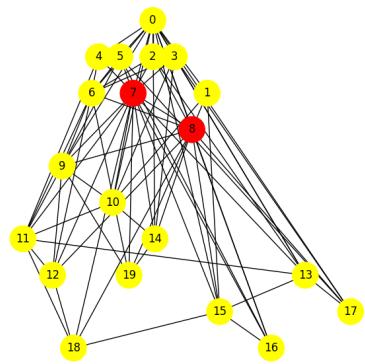
ILP2
Weak roman domination number: 4



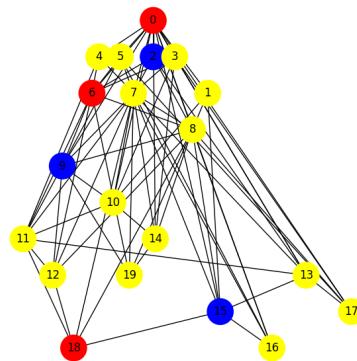
Greedy
Weak roman domination number: 5



Approx
Weak roman domination number: 4



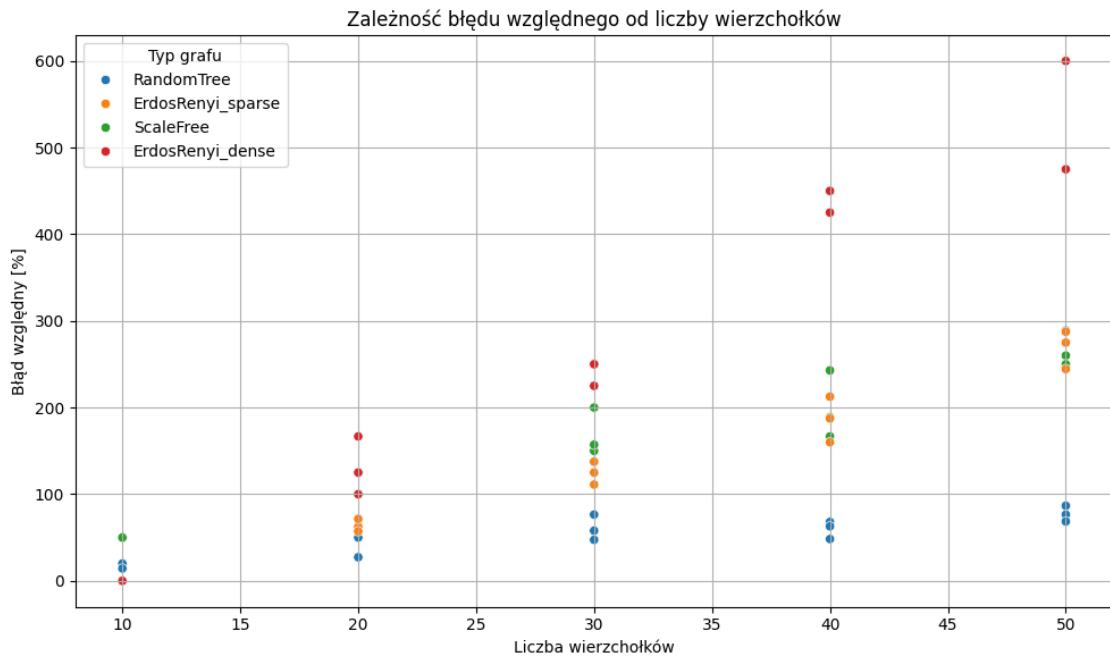
AntColony
Weak roman domination number: 9



Rysunek 4.8: Wyniki dla przykładowego grafu bezskalowego.

4.4 Wyniki algorytmów niedokładnych

4.4.1 Algorytm mrówkowy



Rysunek 4.9: Zależność błędu względnego algorytmu mrówkowego od liczby wierzchołków.

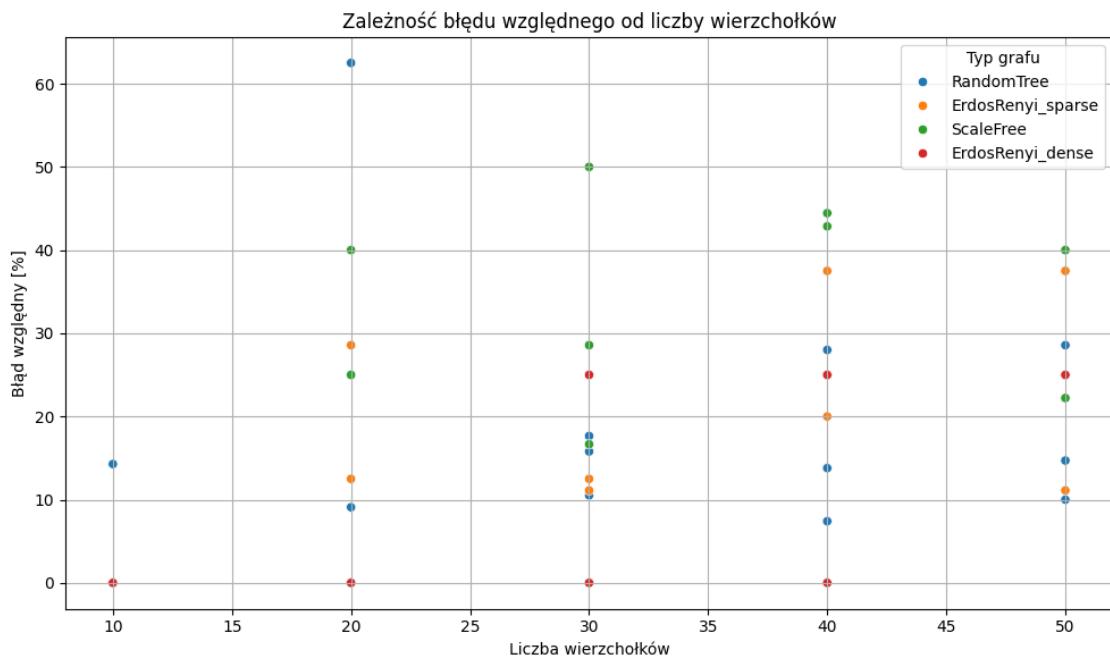
Wraz ze wzrostem liczby wierzchołków można zaobserwować coraz większą rozbieżność od wyniku optymalnego. Dla mniejszych grafów prościej trafić na rozwiązanie optymalne, niż eksplorowanie całej przestrzeni rozwiązań w większych grafach.

Największą rozbieżność można zaobserwować dla grafów gęstych, gdzie błąd względny osiąga nawet 600%. Wynika to z tego, że w grafach gęstych jest dużo wierzchołków o wysokich stopniach, więc heurystyka oparta na stopniach wierzchołków nie jest w stanie wyróżnić wierzchołków najbardziej nadających się do zbioru dominującego.

W odróżnieniu do grafów gęstych, algorytm osiągał zadowalające wyniki dla drzew. Wynika to z tego, że drzewa są grafami rzadkimi i nie ma tam wielu połączeń wierzchołkowych do analizy. W odróżnieniu do innych klas grafów, struktura drzew sprzyja lokalnym decyzjom dotyczącym dominacji, dlatego też algorytm mrówkowy radzi sobie nienajgorzej, chociaż wciąż mocno odbiega od optymalnego wyniku.

Pośrednie wyniki osiągają grafy rzadkie i bezskalowe.

4.4.2 Algorytm zachłanny



Rysunek 4.10: Zależność błędu względnego algorytmu zachłanego od liczby wierzchołków.

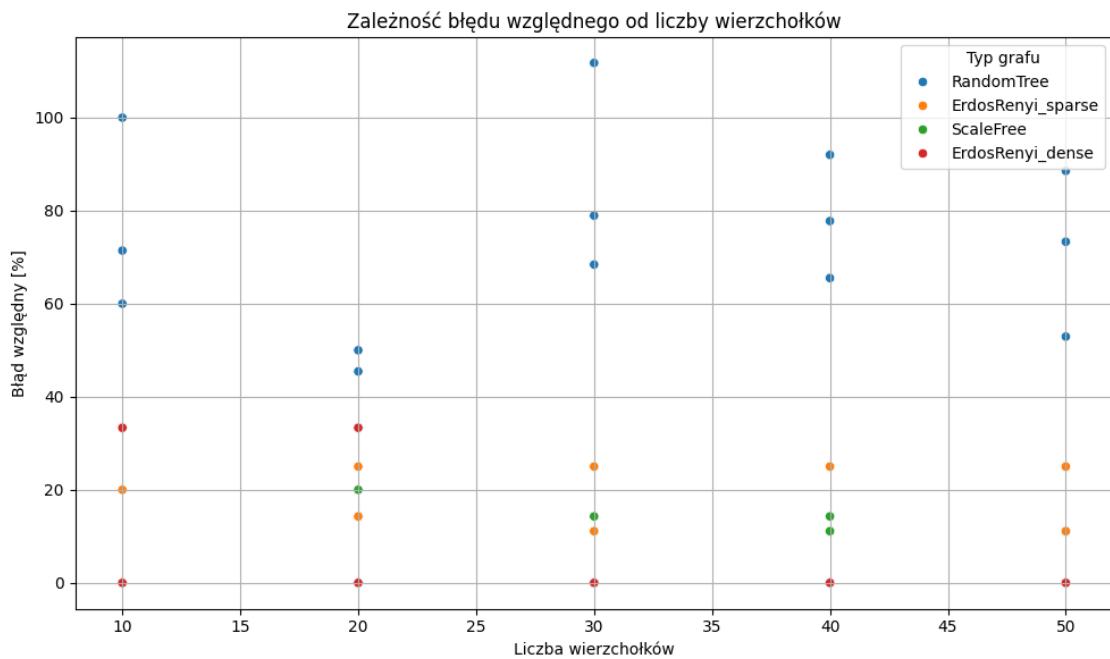
Algorytm zachłanny charakteryzuje się ogólnie niskim błędem względnym na tle innych prezentowanych algorytmów. Godny odnotowania jest fakt, że wraz ze wzrostem liczby wierzchołków w grafach, algorytm nie traci znacznie na jakości, dlatego na podstawie tych przykładów można wysnuć hipotezę, że algorytm może być jakościowo skalowalny.

Największy średni błąd względny algorytm osiąga dla grafów rzadkich i bezskalowych. Prawdopodobnym powodem tego są nieregularne struktury tych grafów i konieczność nieoczywistych decyzji odnośnie wyboru wierzchołka należącego do zbioru dominującego słabo spójnego.

Algorytm zachłanny wyjątkowo dobrze radzi sobie z grafami gęstymi. Wynika to z tego, że promowane są wierzchołki, które pokrywają jak najwięcej sąsiadów. Dla grafów gęstych strategia ta jest bardzo korzystna i sprawia, że algorytm jest w stanie osiągnąć wynik optymalny, bądź bardzo bliski optymalnemu.

Z racji tego, że algorytm zachłanny tworzy dominację bardziej lokalnie, od wierzchołka o największym stopniu, bez zwracenia uwagi na całą strukturę grafu, to w zależności od konkretnego przypadku testowego drzewa, wyniki mogą być bardzo dobre, bądź wysoce nieoptimalne. Na prezentowanym wykresie taki przypadek, największego błędu względnego: ponad 60%, jest właśnie dla drzewa.

4.4.3 Algorytm aproksymacyjny



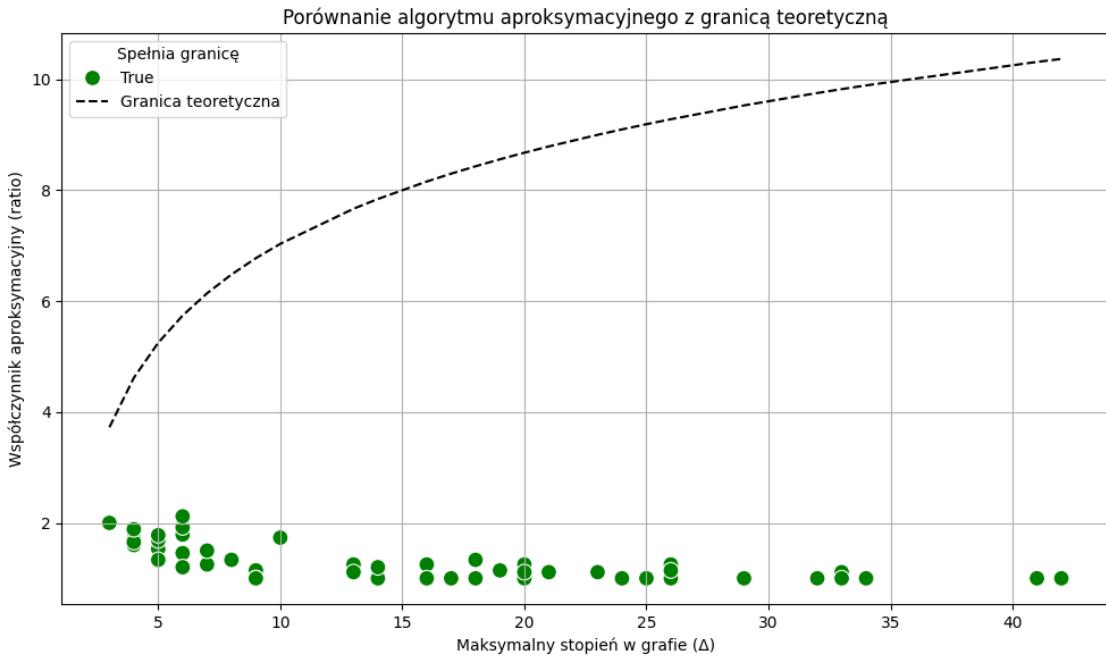
Rysunek 4.11: Zależność błędu względnego algorytmu aproksymacyjnego od liczby wierzchołków.

Dla grafów gęstych algorytm Approx osiąga bardzo dobre wyniki, w wielu przypadkach optymalne, ze względu na dużą możliwość dominacji. Algorytm w sposób dokładny wyznacza spójny zbiór dominujący, dla grafów gęstych takie sposób wyznaczania WCRDF okazuje się być w wielu przypadkach wystarczający.

Z kolei dla drzew jest to bardzo zła strategia. Algorytm przez to, że wyznacza CDS i przypisuje wszystkim wierzchołkom CDS wartości 2, sprawia, że istnieje bardzo dużo nadmiernych przypisań, przez co dla drzew algorytm potrafi osiągnąć nawet ponad 100% błędu względnego.

Algorytm aproksymacyjny radzi sobie dobrze z grafami rzadkimi i bezskalowymi, z błędem względnym na poziomie 20-40%. Wyznacza on prawidłowo CDS i dla tych klas grafów najwyraźniej wystarcza to dla osiągnięcia niskiego poziomu błędu względnego.

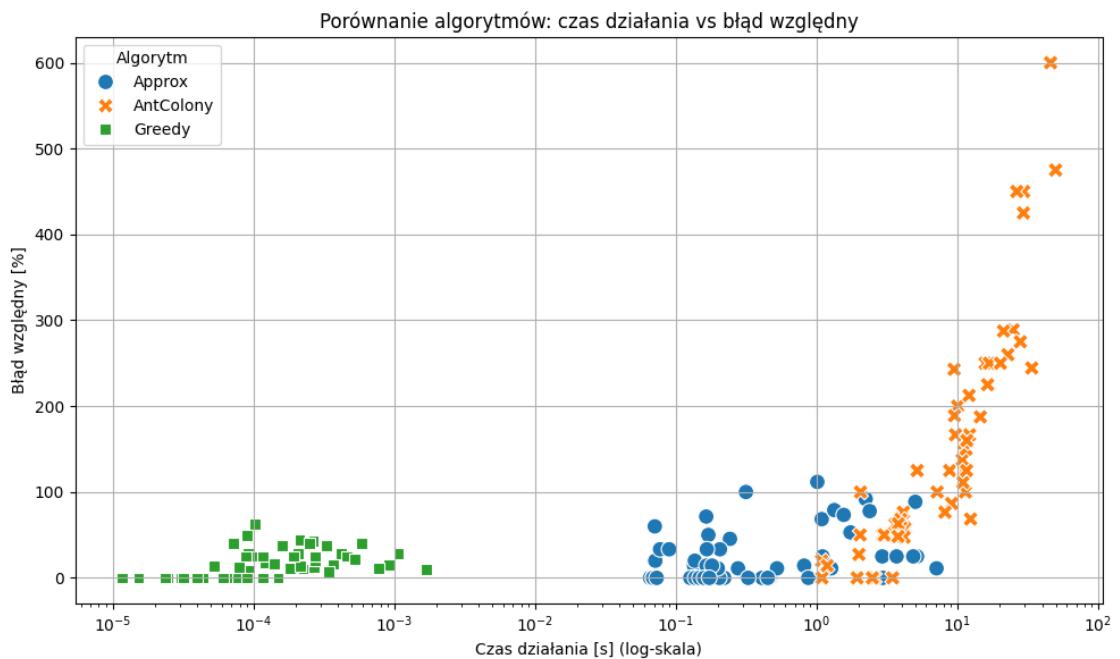
Wykres przedstawia porównanie rzeczywistego współczynnika aproksymacji algorytmu Approx z jego teoretyczną granicą ustaloną w artykule [7] wynoszącą $2(1 + \epsilon)(1 + \ln(\Delta - 1))$ w zależności od maksymalnego stopnia w grafie.



Rysunek 4.12: Porównanie algorytmu aproksymacyjnego z granicą teoretyczną $2(1+\epsilon)(1+\ln(\Delta-1))$.

Można zauważyc, że wszystkie wygenerowane przypadki testowe mieszczą się w granicy teoretycznej. Granica teoretyczna znajduje się daleko od wygenerowanych przykładowych grafów. Zatem w praktyce algorytm radzi sobie znacznie lepiej niż granica teoretyczna zakłada. Prawdopodobnie istnieją przypadki znajdujące się blisko tej granicy, ale taki przypadek nie został odnaleziony w procesie testowania. Dlatego można potwierdzić fakt, że algorytm jest zgodny z współczynnikiem aproksymacji i nie przekracza przewidzianego maksimum, a przynajmniej nie znaleziono przypadku, żeby tak nie było. Można też zauważyc, że wraz ze wzrostem maksymalnego stopnia wierzchołka w testowym grafie, współczynnik aproksymacyjny algorytmu jest coraz niższy. Można z tego wywnioskować fakt, że algorytm jest w stanie dobrze wykorzystać obecność wierzchołków o wysokim stopniu, gdyż bardzo często są one częścią zbioru dominującego.

4.4.4 Porównanie algorytmów przybliżonych



Rysunek 4.13: Algorytmy przybliżone: czas działania w stosunku do błędu względnego.

W ujęciu porównawczym wszystkich algorytmów wyznaczających $\gamma_R^{wc}(G)$ w sposób przybliżony, algorytm zachłanny prezentuje najlepszy stosunek czasu działania do jakości rozwiązania. Jego rozwiązanie, dla wygenerowanych przepadków testowych można otrzymać w przedziale 10^{-5} do 10^{-3} , a błąd względny utrzymuje się na poziomie 0%-50%.

Algorytm Approx prezentuje czas działania w przedziale kilku sekund, potrafi znaleźć więcej lepszych rozwiązań niż Greedy, o czym świadczy większe zagęszczenie niebieskich punktów na poziomie „0” niż w przypadku Greedy. Jednak czasem, zwłaszcza dla drzew, algorytm ten daje nieoptimalne wyniki.

Algorytm mrówkowy, w porównaniu z resztą radzi sobie najgorzej. Czas działania jest najdłuższy, bo rzędu od kilku do kilkudziesięciu sekund, dając przy tym znacznie gorsze wyniki, z największymi błędami względnymi. Świadczy to o tym, że w obecnej konfiguracji i dobranej heurystyce, algorytm mrówkowy nie radzi sobie z tym problemem.

ROZDZIAŁ 5. ZASTOSOWANIA PRAKTYCZNE

5.1 Wprowadzenie

Rozdział ten skupia się na potencjalnych zastosowaniach praktycznych dla problemu WCRDF oraz poszukiwania $\gamma_R^{wc}(G)$. Patrząc na genezę historyczną problemu, warto poszukać zastosowań w życiu codziennym, w celu maksymalizacji ochrony przy niskim jej koszcie [2]. Literatura proponuje parę zastosowań. Jednym z nich jest próba optymalnego rozmieszczenia pojazdów służb ratunkowych. Utrzymanie i kupno takiego pojazdu jest kosztowne, ale jednocześnie cała sieć budynków służb ratunkowych powinna mieć szybki dostęp do takiego pojazdu, aby móc dotrzeć do potrzebujących. Pojazdy można rozmieścić na wzór legionów rzymskich, w budynkach służb ratunkowych powinien znajdować się pojazd, bądź być możliwość wypożyczenia go z sąsiedniej (najbliższej) lokalizacji służb ratunkowych [9].

W rozdziale przeprowadzono analizę potencjalnych zastosowań:

- rozmieszczenie zabezpieczeń sieci energetycznych,
- rozmieszczenie agentów wykrywających oszustwa w internetowej sieci społecznościowej.

Warto zaznaczyć, że oba zastosowania są potencjalne i teoretyczne, nie została przeprowadzona analiza dotycząca faktycznego działania wykorzystywanych systemów.

Do analizy zostały wykorzystane prawdziwe lub uproszczone grafy występujące w rzeczywistości. Następnie grafy te poddano działaniom wybranych algorytmów, tam gdzie było to możliwe. Dodatkowo dokonano analizy jakościowej i czasowej tych algorytmów, w kontekście prezentowanego zastosowania.

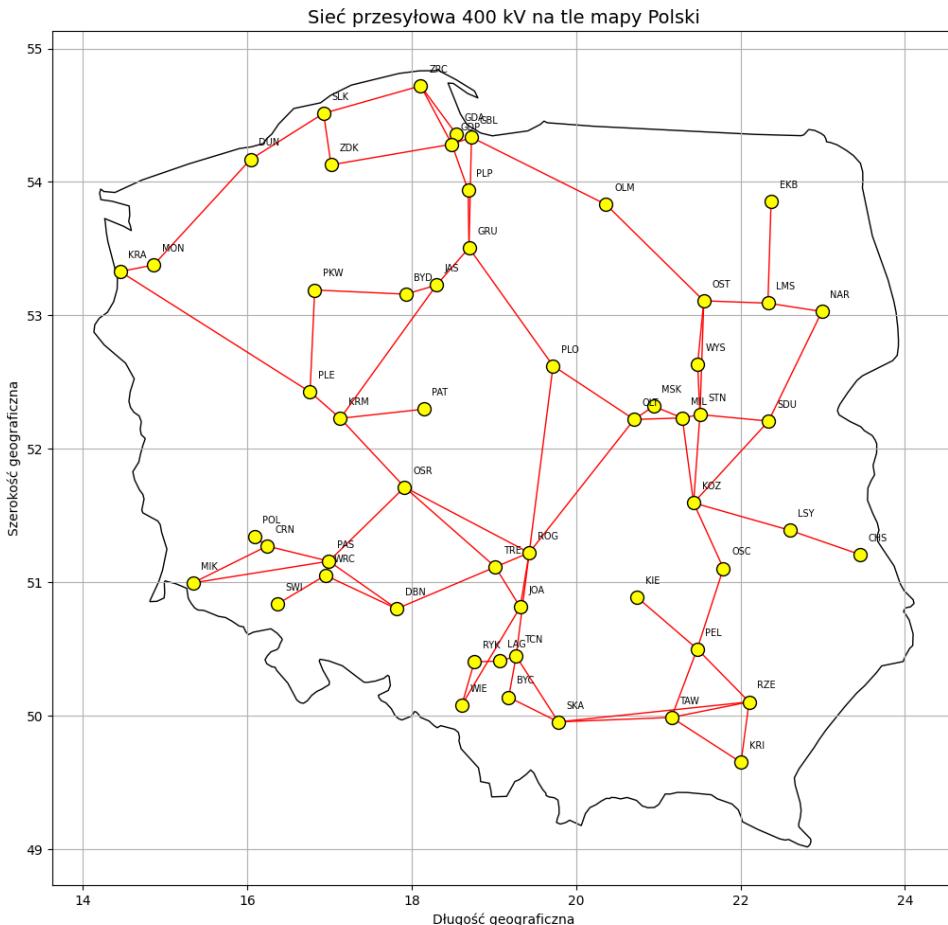
5.2 Rozmieszczenie zabezpieczeń sieci energetycznych

Inspiracją tego zastosowania był blackout na półwyspie Iberyjskim pod koniec kwietnia 2025 roku [14], w którym w wyniku awarii początkowo jednej podstacji, doszło do wyłączenia prądu w kilku krajach.

Dlatego propozycją zapobiegania takim problemom może być próba rozmieszczenia systemów alternatywnych lub zabezpieczających takich, które mogą ochronić albo przejąć obciążenie wadliwej stacji energetycznej, aby zapobiec większej awarii. Dominowanie rzymskie słabo spójne będzie miało tutaj zastosowanie ze względu na chęć utrzymania komunikacji między stacjami energetycznymi oraz braku chęci pozostawienia stacji bez działającej alternatywy. Naturalnie, takie systemy wiążą się z niemałymi kosztami, a więc warto, aby było ich jak najmniej. Zatem zastosowanie algorytmów rozwiązujących problem minimalnej WCRDF wydaje się odpowiednie, przynajmniej w ujęciu teoretycznym.

W tym przypadku jako graf można przyjąć stacje energetyczne i transformatory, natomiast jako krawędzie można przyjąć sieci energetyczne, czyli połączenia między stacjami i transformatorami.

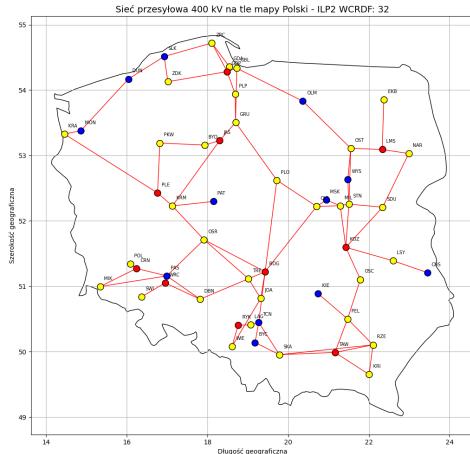
W celu próby realizacji tego pomysłu odtworzono graf połączeń sieci i stacji energetycznych najwyższych napięć w Polsce, na podstawie istniejącego już schematu [10]. W praktyce sieć jest bardziej rozbudowana, jednakże tutaj skupiono się na najwyższym napięciu 400kV. Jej graf przedstawiono na tle mapy Polski. Z powierzchownej analizy tego grafu można zauważyć, że jest to graf bardzo rzadki, posiada wiele wierzchołków o niskim stopniu. Cechuje się brakiem hubów, nie ma wierzchołków o dużym stopniu. Posiada on 55 wierzchołków i 77 krawędzi. Wykres tego grafu jest przedstawiony na grafice poniżej.



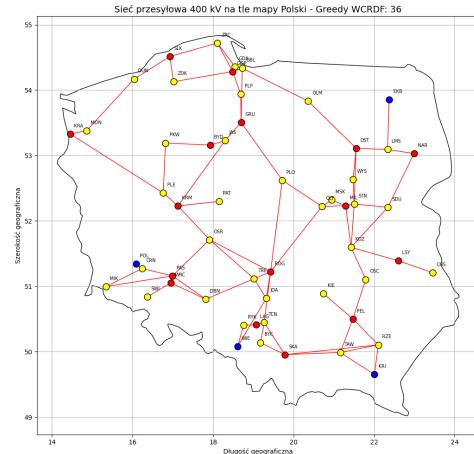
Rysunek 5.1: Sieć przesyłowa największych napięć na tle mapy Polski

Dla grafów bardzo rzadkich, na podstawie analizy z poprzedniego rozdziału można wysnuć hipotezę, że algorytmy programowania liniowego z racji niewielkiej liczby krawędzi będą dawały optymalne wyniki w rozsądny czasie dla tej skali problemu. Algorytm zachłanny powinien prezentować dobry stosunek jakości do czasu działania. Jeśli chodzi o algorytm Approx, z racji rzadkiego grafu, może on prezentować gorszą jakość rozwiązania.

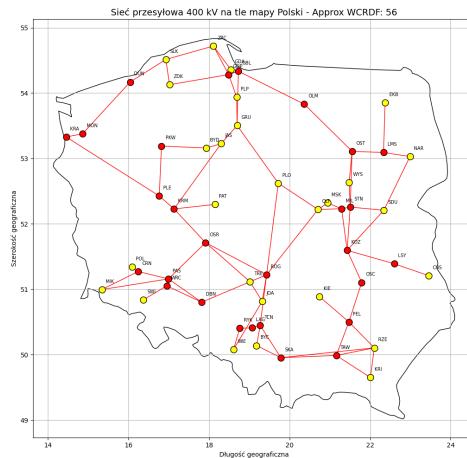
Poniższe grafiki przedstawiają wyniki działania algorytmów ILP2, Greedy oraz Approx. Standardowo, żółte wierzchołki to 0, niebieskie to 1, a czerwone to 2.



(a) ILP2



(b) Greedy



(c) Approx

Rysunek 5.2: Efekt działania algorytmów ILP, Greedy i Approx dla problemu rozmieszczenia za-bezpieczeń sieci energetycznych.

Algorytm	$\gamma_R^{wc}(G)$	Czas działania [s]
ILP2	32	2,7014378
Greedy	36	0,0003222
Approx	56	647,6564676

Tabela 5.1: Wynik działania wybranych algorytmów dla problemu rozmieszczenia zabezpieczeń sieci energetycznych.

Algorytmowi ILP2 udało odnaleźć optymalne rozwiązanie w ciągu dwóch sekund, co jest bardzo dobrym wynikiem. Nieznacznie gorszy wynik prezentuje algorytm zachłanny. Natomiast zgodnie z przewidywaniami, algorytm Approx osiągnął znacznie gorszy wynik od pozostałych, ze względu na to, że szuka on zbioru dominującego spójnego, co w przypadku grafu bardzo rzadkiego mocno zawyża wyniki. Charakteryzuje się też znacznie dłuższym czasem działania od pozostałych.

To rozważanie pokazuje, że nawet dla nieco większych grafów rzadkich, obserwowanych w rzeczywistym życiu, można dokładnie rozwiązać problem $\gamma_R^{wc}(G)$, dzięki algorytmom programowania liniowego i nie ma konieczności stosowania algorytmów przybliżonych. Jednakże, warto podkreślić tutaj potencjał algorytmu zachłannego, który osiągnął nieznacznie gorszy wynik, a jest bez wątpienia najbardziej skalowalnym algorytmem ze wszystkich prezentowanych.

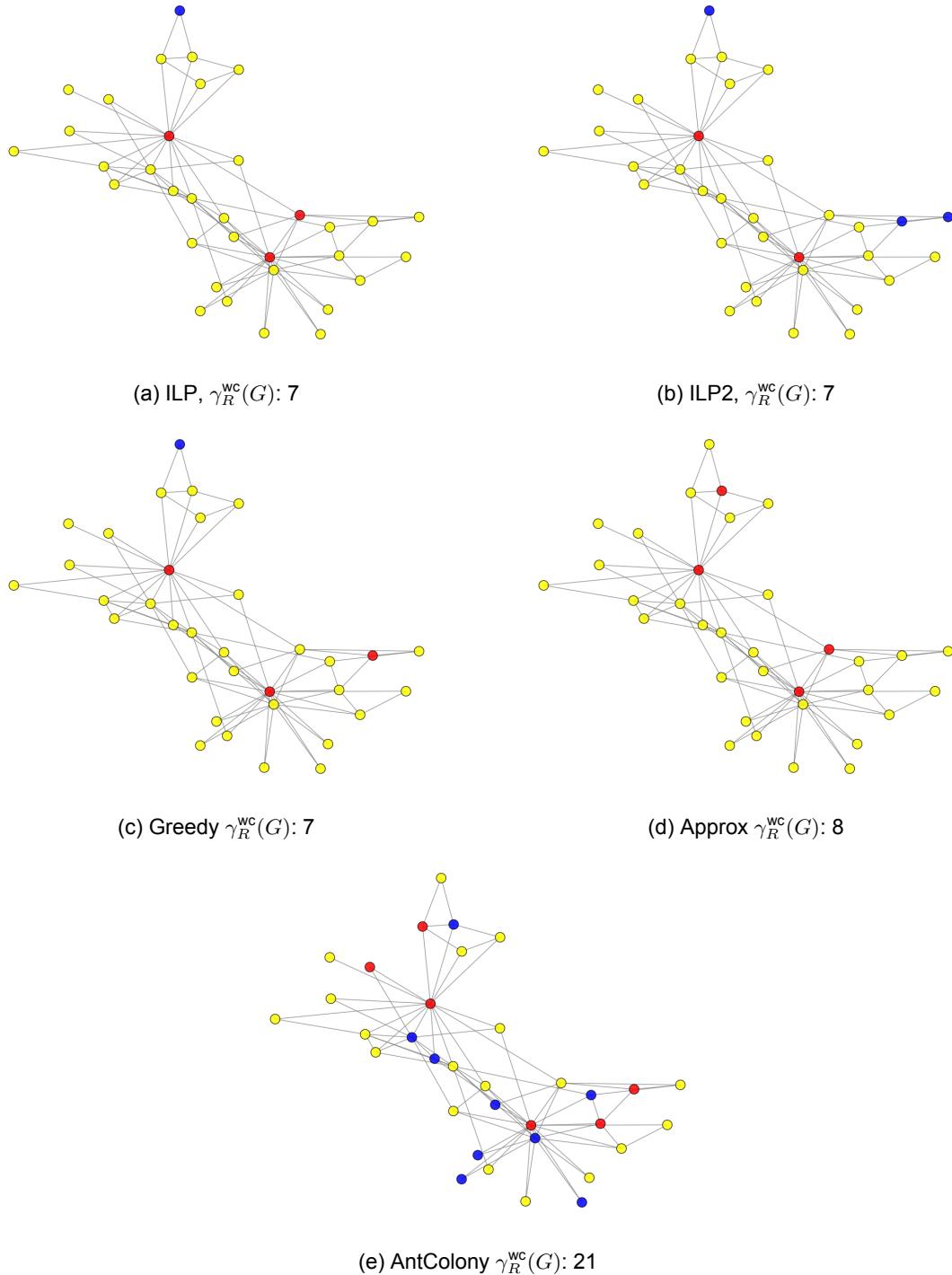
5.3 Rozmieszczenie agentów monitorujących oszustwa w internetowej sieci społecznościowej

Kolejnym z potencjalnych zastosowań może być próba optymalnego rozmieszczenia agentów monitorujących oszustwa w internetowej sieci społecznościowej. Grafem wejściowym będzie w tym przypadku sieć znajomych w wybranej sieci społecznościowej, gdzie wierzchołki to użytkownicy, a krawędzie to interakcje lub znajomości między nimi. W dobie wielu oszustw internetowych, wyłudzeń, kradzieży kont, dobrym pomysłem może być rozmieszczenie agentów monitorujących oraz reagujących na zajście jakiegoś przestępstwa internetowego. Wierzchołki z 2 mogą w tym przypadku monitorować siebie oraz sąsiedztwo, a wierzchołki z 1 wyłącznie siebie. Zachowanie komunikacji i reagowanie między sąsiadami jest tutaj bardzo ważne, aby ofiar oszustwa nie było więcej, dlatego widać tutaj zastosowanie słabo spójności. Dodatkowo, w razie „wypożyczenia” agenta do innego wierzchołka, wierzchołek „wypożyczający” wciąż pozostaje chroniony, a ochrona sąsiadów jest w tym przypadku kluczowa, bo znajomi znajdują się w obszarze podwyższonego ryzyka do szerszego oszustwa. Naturalnie, przechowywanie i zarządzanie jak najmniejszą liczbą instancji agentów jest w tym przypadku pożądane. Dlatego też sensownym wydaje się zastosowanie w przypadku takich sieci algorytmów znajdujących $\gamma_R^{wc}(G)$. Poniższe rozważania analizują dwa przypadki rzeczywistych sieci społecznościowych.

Pierwszą z nich jest popularny w literaturze do analizy sieci społecznościowych graf klubu

karate Zacharego [11]. Graf ten jest odzwierciedleniem interakcji społecznych członków pewnego klubu karate, badanego przez Wayne W. Zachary'ego [15]. Posiada on 34 wierzchołki i 78 krawędzi. Z racji swojego rozmiaru, algorytmy dla tego grafu wykonały się w rozsądny czasie. Graf ma cechy bezskalowego, gdyż posiada kilka wierzchołków o wyższym stopniu niż pozostałe oraz wiele o niewielkim. Dlatego wyniki powinny być analogiczne, jak prezentowane wcześniej dla grafów bezskalowych, a algorytmy programowania liniowego powinny prezentować dobry stosunek czasu działania do jakości rozwiązania, algorytm mrówkowy niekoniecznie.

Poniższe wizualizacje przedstawiają efekty:



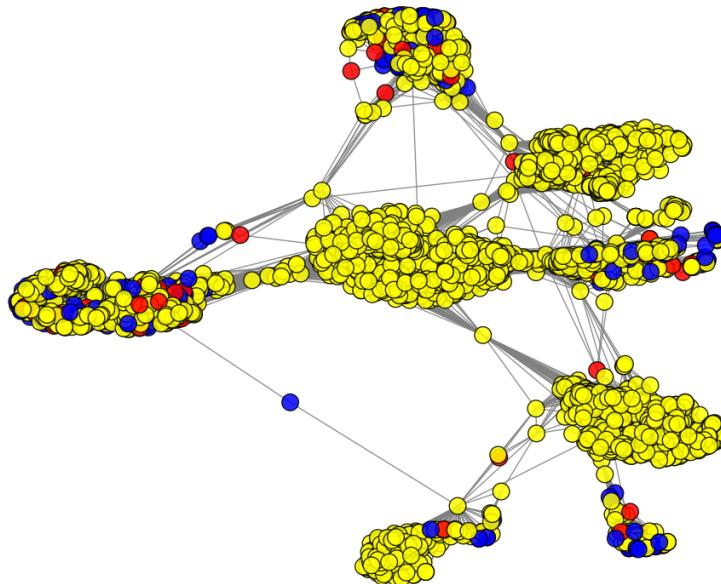
Rysunek 5.3: Efekt działania algorytmów dla problemu rozmieszczenia agentów w małej, rzeczywistej sieci społecznościowej.

Algorytm	$\gamma_R^{wc}(G)$	Czas działania [s]
ILP2	7	0,0865662
ILP	7	0,0624058
Greedy	7	0,0001245
Approx	8	0,1533187
AntColony	21	5,1319121

Tabela 5.2: Wynik działania wybranych algorytmów dla problemu rozmieszczenia agentów w małej sieci społecznościowej.

Zgodnie z przewidywaniami, algorytmy programowania liniowego prezentują zadowalające wyniki. Warto zwrócić uwagę, że algorytm zachłanny wyznaczył optymalną wartość liczby dominowania rzymskiego słabo spójnego. Należy jednak pamiętać, że dla grafów bezskalowych wyniki działania tego algorytmu w zależności od grafu mogą się różnić dokładnością. Algorytm mrówkowy po raz kolejny prezentuje niezadowalające wyniki jakościowe i czasowe.

Kolejną analizowaną siecią społecznościową jest sieć znajomych z serwisu Facebook [12]:



Rysunek 5.4: Efekt działania algorytmu zachłannego dla problemu rozmieszczenia agentów w dużej, rzeczywistej sieci społecznościowej. $\gamma_R^{wc}(G)$: 411, czas działania 2,9348995 s.

Składa się on z 4039 wierzchołków i 80234 krawędzi. Charakteryzacja i wizualizacja grafu wskazuje na to, że można go przypisać do klasy grafów bezskalowych. Z racji wielkości grafu udało się na nim tylko wykonać algorytm zachłanny, pozostałe algorytmy nie wykonały się w rozsądny czasie. Algorytm wyznaczył $\gamma_R^{wc}(G)$ jako 411 w czasie 2,9348995 s, co jest stosunkowo krótkim

czasem wykonania jak na wielkość grafu. Po raz kolejny wyróżnia to skalowalność algorytmu zuchłanego. Nie ma niestety możliwości weryfikacji czy jest to wynik optymalny, ale patrząc na wcześniejsze rozważania, nie powinien być to wynik drastycznie daleki od optymalnego. Pokazuje to, że nawet dla bardzo dużych danych wejściowych, algorytm zachłanny potrafi dać wynik praktycznie w czasie rzeczywistym, w odróżnieniu do innych, prezentowanych algorytmów.

ROZDZIAŁ 6. PODSUMOWANIE I WNIOSKI

6.1 Podsumowanie wyników

Celem badawczym pracy było opisanie istniejących i opracowanie własnych algorytmów znajdujących funkcje dominujące rzymskie słabo spójne, ich analiza oraz porównanie skuteczności i wyciągnięcie wniosków na temat możliwości ich praktycznego zastosowania.

Udało się zrealizować wszystkie postawione zadania, mianowicie:

- przedstawienie problemu WCRDF oraz jego złożoności obliczeniowej,
- przedstawienie algorytmów znajdujących WCRDF oraz algorytmy znajdujące dokładną $\gamma_R^{wc}(G)$ oraz przybliżoną, korzystając z różnych technik programowania,
- zaimplementowanie i przetestowanie wybranych algorytmów,
- wyciągnięcie wniosków na podstawie wyników testów,
- dokonanie analizy pod kątem praktycznego zastosowania rozwiązania tego problemu.

W pracy przetestowano siedem algorytmów:

- algorytmy optymalnie wyznaczające wartość $\gamma_R^{wc}(G)$: Brute Force, ILP i ILP2 oraz TreeLinear - liniowy dla drzew,
- algorytmy przybliżone: zachłanny - Greedy, aproksymacyjny - Approx i mrówkowy - AntColonies.

Eksperymenty przeprowadzono na grafach gęstych, rzadkich, drzewach oraz bezskalowych. Dodatkowo sprawdzono działanie niektórych algorytmów na grafach rzeczywistych.

Wyniki wskazują, że:

- Algorytm BruteForce, chociaż prezentuje dokładne, minimalne rozwiązanie problemu, jest najgorzej skalowalny. Już dla kilkunastu wierzchołków grafu, czas działania robi się nierośądnie długi.
- Algorytmy ILP i ILP2 zapewniają najlepsze jakościowo rozwiązania, lecz ich czas działania rośnie znacząco wraz z rozmiarem badanego grafu, co wynika ze złożoności obliczeniowej. Dla grafów gęstych, o dużej liczbie cykli, algorytm ILP2 oparty na przepływach faktycznie zaczyna działać szybciej niż ILP, opartym na ograniczeniach cyklicznych.
- Algorytm liniowy dla drzew znajduje optymalne rozwiązanie w szybkim czasie i jest skalowalny dla większych instancji drzew. Niestety jego prawidłowe działanie ograniczone jest do jednej klasy grafów.
- Algorytm Greedy wyróżnia się ekstremalnie niskim czasem działania oraz zaskakującą dobrą jakością rozwiązań w wielu klasach grafów. Jako jedyny spośród prezentowanych algorytmów jest skalowalny do większych grafów.
- Approx zapewnia stabilne i poprawne rozwiązania, z niskim błędem względnym, jednak znacząco pogarsza się dla drzew i struktur bardzo rzadkich. Ponadto udało się zweryfikować,

że dla testowanych przykładów grafów udało się pokazać, że wyniki są zgodne z prezentowanym w literaturze współczynnikiem aproksymacji.

- AntColony prezentuje najgorszy spośród wszystkich algorytmów czas działania oraz jakość rozwiązania.

Ponadto wykazano, że wyniki algorytmów różnią się w zależności od klasy grafu, co sugeruje konieczność doboru metody w zależności od specyfiki problemu.

6.2 Wnioski i dalsze kierunki badań

Na podstawie przeprowadzonych badań i analiz sformułowano następujące wnioski:

- WCRDF może stanowić potencjalnie użyteczny model do minimalizacyjnego rozmieszczenia zabezpieczeń w sieciach społecznościowych, czy innych naturalnie utworzonych grafach.
- Algorytmy dokładne (ILP, ILP2) są dobrym punktem, ale niestety są mało skalowalne.
- Algorytm Greedy może być z powodzeniem stosowany w praktyce, szczególnie w dużych instancjach wymagających szybkich przybliżeń.
- AntColony w obecnej implementacji nie nadaje się do skutecznego rozwiązywania tego problemu. Dla każdej klasy grafu i prezentował najbardziej niekorzystne wyniki wśród wszystkich testowanych algorytmów.
- Zastosowanie WCRDF do problemów praktycznych, czyli np. rozmieszczenia zabezpieczeń w sieci energetycznej czy agentów wykrywających oszustwa w sieci społecznościowej zostało uzasadnione i warte dalszych badań.

Dalsze kierunki badań mogą obejmować:

- Rozszerzenie testów na inne klasy, jak i na inne wielkości grafów.
- Podniesienie jakości wyników algorytmu mrówkowego, między innymi poprzez inną implementację heurystyki lokalnej oraz strategii feromonowej oraz badania w celu znalezienia jak najlepszej heurystyki.
- Próba opracowania algorytmów dokładnych, rozwiązywalnych w czasie wielomianowym dla innych klas grafów.
- Weryfikacja i przełożenie teoretycznych rozważań na temat praktycznych zastosowań tego problemu na praktyczną analizę i realizację.

WYKAZ LITERATURY

1. STEWART, I. Defend the Roman Empire!, w: USA: Scientific, 1999, s. 136–139. Dostęp: 03.03.2025.
2. DR INŻ. JOANNA RACZEK, DR JOANNA CYMAN. *Weakly connected Roman domination in graphs*. Dostępne także z: <https://mostwiedzy.pl/en/publication/weakly-connected-roman-domination-in-graphs,150016-1>. Dostęp: 03.03.2025.
3. *Exact cover by 3-sets*. Dostępne także z: https://en.wikipedia.org/wiki/Exact_cover. Dostęp: 11.05.2025.
4. ERNIE J COCKAYNE, PAUL A DREYER JR., SANDRA M HEDETNIEMI, STEPHEN T HEDETNIEMI. *Roman domination in graphs*. 2004. Dostępne także z: <https://www.sciencedirect.com/science/article/pii/S0012365X03004473>. Dostęp: 11.05.2025.
5. MICHAEL A. HENNING, STEPHEN T. HEDETNIEMI. *Defending the Roman Empire—A new strategy*. 2003. Dostępne także z: <https://www.sciencedirect.com/science/article/pii/S0012365X02008117>. Dostęp: 11.05.2025.
6. JOANNA RACZEK, RITA ZUAZUA. *Progress on Roman and Weakly Connected Roman Graphs*. 2021. Dostępne także z: <https://www.mdpi.com/2227-7390/9/16/1846>. Dostęp: 11.05.2025.
7. PADAMUTHAM CHAKRADHAR, PALAGIRI VENKATA SUBBA REDDY, I KHANDELWAL HIMANSHU. *Algorithmic complexity of weakly connected Roman domination in graphs*. 2021. Dostępne także z: <https://www.worldscientific.com/doi/epdf/10.1142/S1793830921501251>. Dostęp: 15.03.2025.
8. DANGDANG NIU, XIAOLIN NIE, LILIN ZHANG, HONGMING ZHANG, MINGHAO YIN. *A greedy randomized adaptive search procedure (GRASP) for minimum weakly connected dominating set problem*. 2023. Dostępne także z: <https://www.sciencedirect.com/science/article/abs/pii/S0957417422023569>. Dostęp: 11.05.2025.
9. MARIJA IVANOVIĆ. *Improved integer linear programming formulation for weak Roman domination problem*. 2017. Dostępne także z: <https://link.springer.com/article/10.1007/s00500-017-2706-4>. Dostęp: 15.03.2025.
10. *Plan istniejącej sieci przesyłowej najwyższych napięć*. 2025. Dostępne także z: <https://www.pse.pl/obszary-dzialalnosci/krajowy-system-elektroenergetyczny/plan-sieci-elektroenergetycznej-najwyzszych-napiec/istniejaca>. Dostęp: 11.05.2025.
11. *Karate club graph*. Dostępne także z: https://networkx.org/documentation/stable/reference/generated/networkx.generators.social.karate_club_graph.html. Dostęp: 11.05.2025.
12. *Social circles: Facebook*. Dostępne także z: <https://snap.stanford.edu/data/ego-Facebook.html>. Dostęp: 11.05.2025.

13. *Solving the Connected Dominating Set Problem and Power Dominating Set Problem by Integer Programming*. 2012. Dostępne także z: https://link.springer.com/chapter/10.1007/978-3-642-31770-5_33. Dostęp: 22.05.2025.
14. *Awaria zasilania na Półwyspie Iberyjskim*. 2025. Dostępne także z: https://pl.wikipedia.org/wiki/Awaria_zasilania_na_P%C3%B3%C5%82wyspie_Iberyjskim. Dostęp: 21.05.2025.
15. *Zachary's karate club*. Dostępne także z: https://en.wikipedia.org/wiki/Zachary%27s_karate_club. Dostęp: 22.05.2025.

SPIS RYSUNKÓW

2.1 Przykład grafu rzymskiego słabo spójnego.	10
3.1 Graf G - przykładowe drzewo	20
3.2 Graf G - po fazie 1	21
3.3 Graf G - po fazie 2 - finalna wersja	22
4.1 Porównanie średniego czasu działania algorytmów na grafach rzadkich w stosunku do liczby wierzchołków w grafie.	48
4.2 Wyniki dla przykładowego grafu rzadkiego.	49
4.3 Porównanie średniego czasu działania algorytmów na grafach gęstych w stosunku do liczby wierzchołków w grafie.	51
4.4 Wyniki dla przykładowego grafu gęstego.	52
4.5 Porównanie średniego czasu działania algorytmów na drzewach w stosunku do liczby wierzchołków w grafie.	54
4.6 Wyniki dla przykładowego drzewa.	55
4.7 Porównanie średniego czasu działania algorytmów na grafach bezskalowych w stosunku do liczby wierzchołków w grafie.	57
4.8 Wyniki dla przykładowego grafu bezskalowego.	58
4.9 Zależność błędu względnego algorytmu mrówkowego od liczby wierzchołków. . . .	59
4.10 Zależność błędu względnego algorytmu zachłannego od liczby wierzchołków. . . .	60
4.11 Zależność błędu względnego algorytmu aproksymacyjnego od liczby wierzchołków. .	61
4.12 Porównanie algorytmu aproksymacyjnego z granicą teoretyczną $2(1+\epsilon)(1+\ln(\Delta-1))$. .	62
4.13 Algorytmy przybliżone: czas działania w stosunku do błędu względnego.	63
5.1 Sieć przesyłowa największych napięć na tle mapy Polski	65
5.2 Efekt działania algorytmów ILP, Greedy i Approx dla problemu rozmieszczenia za-bezpieczeń sieci energetycznych.	66
5.3 Efekt działania algorytmów dla problemu rozmieszczenia agentów w małej, rzeczy-wistej sieci społecznościowej.	69
5.4 Efekt działania algorytmu zachłannego dla problemu rozmieszczenia agentów w dużej, rzeczywistej sieci społecznościowej. $\gamma_R^{wc}(G)$: 411, czas działania 2,9348995 s. .	70

SPIS TABEL

4.1	Wpływ kombinacji hiperparametrów na błąd rozwiązania	45
4.2	Wpływ kombinacji hiperparametrów na błąd rozwiązania na konkretnych klasach grafów	45
4.3	Wybrany zestaw parametrów	46
4.4	Wyniki dla grafów rzadkich	47
4.5	Wyniki dla grafów gęstych	50
4.6	Wyniki dla drzew	53
4.7	Wyniki dla grafów bezskalowych	56
5.1	Wynik działania wybranych algorytmów dla problemu rozmieszczenia zabezpieczeń sieci energetycznych.	67
5.2	Wynik działania wybranych algorytmów dla problemu rozmieszczenia agentów w małej sieci społecznościowej.	70