POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

Imię i nazwisko studenta: Paulina Brzęcka
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

Streszczenie pracy opisuje problem naukowy polegający na opracowaniu i wdrożeniu metody analizy danych w środowisku wielowymiarowym. Celem pracy jest zaproponowanie algorytmu optymalizacyjnego, który pozwala na efektywne przetwarzanie dużych zbiorów danych. Zakres pracy obejmuje analizę istniejących metod, implementację nowego rozwiązania oraz ocenę jego skuteczności na wybranych przypadkach testowych. Zastosowana metoda badawcza obejmowała modelowanie matematyczne, programowanie w języku Python oraz wizualizację wyników. Wyniki wskazują na istotne przyspieszenie obliczeń przy zachowaniu wysokiej dokładności. Najważniejszym wnioskiem jest możliwość zastosowania zaproponowanego algorytmu w rzeczywistych aplikacjach analitycznych.

Słowa kluczowe: algorytmy, przetwarzanie danych, optymalizacja.

ABSTRACT

The abstract describes a scientific problem focusing on the development and implementation of a data analysis method in a multidimensional environment. The objective of this thesis is to propose an optimization algorithm that enables efficient processing of large datasets. The scope of the work includes an analysis of existing methods, the implementation of a new solution, and the evaluation of its effectiveness on selected test cases. The research methodology involved mathematical modeling, Python programming, and visualization of results. The results indicate significant acceleration in computations while maintaining high accuracy. The key conclusion is the feasibility of applying the proposed algorithm in real-world analytical applications.

Keywords: algorithms, data processing, optimization.

SPIS TREŚCI

WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

e – Niepewność pomiaru.

f – Częstotliwość [Hz].

k – Stała Boltzmanna $1.38 \cdot 10^{-23}$ Ws/K.

1. WSTĘP I CEL PRACY

Tematem pracy jest analiza algorytmów znajdujących funkcję dominującą rzymską słabospójną w grafach. Problem znajdywania liczby dominowania rzymskiego słabospójnego jest problemem NP-trudnym. Wersja decyzyjna tego problemu jest NP-zupełna. Nie istnieją zatem dokładne algorytmy rozwiązujące problem w czasie wielomianowym. Dlatego niniejsza praca dokonuje analizy dostępnych i proponowanych algorytmów rozwiązujących ten problem w sposób zarówno dokładny, jak i przybliżony, w celu znalezienia możliwie skutecznych rozwiązań oraz zastosowań.

1.1 Cel pracy

Celem pracy jest analiza algorytmów dla dominowania rzymskiego słabo spójnego, w tym opisanie już istniejących rozwiązań oraz opracowanie własnych, porównanie ich skuteczności oraz możliwych praktycznych zastosowań.

1.2 Zakres pracy

W ramach pracy dokonano systematycznego przeglądu literatury. W literaturze proponowano wiele algorytmów dokładnych o czasie wykładniczym, między innymi algorytmy wykorzystujace programowanie liniowe. Dodatkowo, w wielu publikacjach skupiono się na algorytmach dla konkretnych klas grafów. W literaturze zostały również zdefiniowane algorytmy niedokładne, aproksymacyjne, o różnej jakości rozwiązania. 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 dokładny brute force, liniowy dokładny dla drzew oraz mrówkowy. Niniejsza praca opisuje wymienione algorytmy, porównuje je pod kątem wydajności, poprawności oraz czasu działania.

2. WPROWADZENIE TEORETYCZNE

2.1 Wprowadzenie

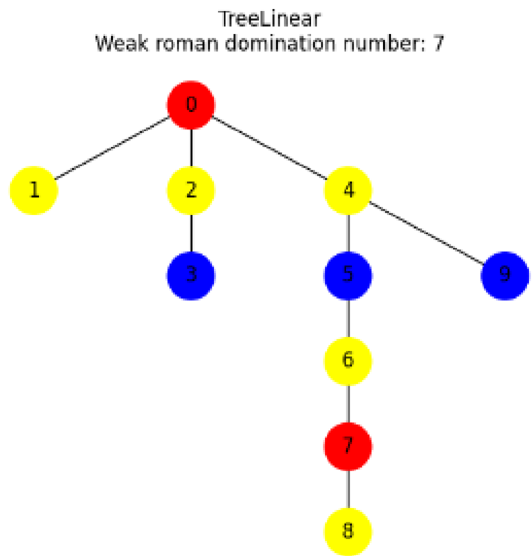
Jako, że problem nie jest powszechnie znany, należy wprowadzić następujące pojęcia [1]:

Definicja 1 *Funkcja dominująca rzymska zdefiniowana jest dla grafu $G = (V, E)$, gdzie $f : V- > \{0, 1, 2\}$ 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 słabospójnym grafu G jeśli graf $(V, E \cap (D \times V))$ jest spójny.*

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

Definicja 4 *Wagę funkcji dominującej rzymskiej słabospójnej definiujemy jako $f(V) = \sum_{u \in V} f(u)$. Minimalną wartość tej funkcji nazywamy liczbą dominowania rzymskiego słabospójnego.*



Rysunek 2.1: Przykład grafu rzymskiego słabo spójnego.

2.2 Geneza historyczna

Problem swoją nazwę zawdzięcza imperium rzymskiemu. Po raz pierwszy został opisany w artykule „Defend Roman Empire!”.[2] 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 utrzymywania legionów, a zatem, żeby było ich jak najmniej. [1]

2.3 Przegląd literatury

Cockayne ze współautorami w 2005 roku zdefiniowali po raz pierwszy problem dominacji rzymskiej słabo spójnej. Od tego czasu powstało wiele prac naukowych badających ten parametr i proponujące algorytmy dla wszystkich, jak i wybranych klas grafów. Henning i Hedetniemi w 2003 roku udowodnili, że problem dominowania rzymskiego słabospójnego jest NP-zupełny w wersji decyzyjnej, nawet dla grafów dwudzielnych. Natomiast dla grafów blokowych, wyznaczanie liczby dominowania rzymskiego słabospójnego może być osiągnięte w czasie liniowym, co pokazali w swojej pracy Liu wraz ze współautorami (2010).
Trywialna złożoność tego problemu wynosi $O(3^n)$. Natomiast w 2013, Chapelle i współautorzy zaproponowali algorytm dokładny dla wszystkich klas grafów o złożoności czasowej $O(2^n)$, wymagający jednak pamięci wykładniczej oraz algorytm $O(2,2279^n)$ z wykorzystaniem pamieci wielomianowej. Burger i współautorzy w 2013 zaproponowali również model programowania liniowego całkowitoliczbowego. [4]
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 niniejszej pracy, analizowane będą te dwa modele programowania liniowego oraz aproksymacyjny. Dodatkowo analizowane będą propozycje własne rozwiązania tego problemu.[3].
Literatura proponuje również 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. [4]

2.4 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
- gurobipy - do programowania liniowego

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 słabospójnego. Dla każdego z algorytmów wyliczany i zapisywany jest ich czas działania. Na końcu program wyświetla wykres z nadanymi wartościami na wierzchołkach.

3. BADANE ALGORYTMY

3.1 Wprowadzenie

Niniejszy rozdział opisuje algorytmy dla funkcji dominowania rzymskiego słabospójnego. Analizowane one będą pod kątem złożoności, wydajności, poprawności oraz potencjalnego zastosowania. Wszystkie algorytmy wyznaczają dodatkowo zbiór dominowania rzymskiego słabospójnego. Lista analizowanych algorytmów jest następująca:

- algorytm brute force
- algorytm liniowy dla drzew
- algorytm programowania liniowego I
- algorytm programowania liniowego II
- algorytm mrówkowy
- algorytm aproksymacyjny

3.2 Algorytm Brute Force

3.2.1 Działanie

Jest to w zasadzie trywialna implementacja dokładnego algorytmu wyznaczającego funkcję oraz zbiór dominujący rzymski słabospójny 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:

- wyznaczamy zbiór indukowany, który składa się ze zbioru dominującego (wierzchołki z wartościami {1, 2}) oraz sąsiadów wierzchołków zbioru dominującego,
- sprawdzamy czy wszystkie wierzchołki z wartością 0 mają sąsiada z wartością 2,
- dla każdego wierzchołka ze zbioru indukowanego dodajemy krawędzie, ale tylko te wychodzące z wierzchołków zbioru dominującego
- następnie sprawdzamy, czy powstały graf jest spójny. Jeśli jest, to zbiór spełnia założenia definicji.

3.2.2 Złożoność i wydajność

Algorytm ma złożoność wykładniczą, zatem nie będzie wykonywalny w rozsądnym czasie dla większych grafów.

- generowanie wszystkich kombinacji możliwych przypisań: 3^n, gdzie n to liczba wierzchołków grafu,
- sprawdzanie własności zbioru słabospójnego 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 $O(n + m)$

3.2.3 Pseudokod

Algorytm Brute Force

```
function FindRomanDominatingSet(graph)
    Initialize min_roman_number ← ∞
    Initialize best_node_values ← None
    nodes ← list of nodes in graph
    for each assignment of values (0, 1, 2) to all nodes do
        node_values ← mapping of nodes to values
        for each node in graph do                    ▷ Sprawdzanie warunku dominacji
            if node_values[node] = 0 then
                if not any neighbor of node has value 2 then
                    Continue to next assignment
                end if
            end if
        end for
        induced_set ← nodes with values {1, 2}
        for each node in induced_set do
            Add all its neighbors to induced_set
        end for
        Create empty induced_graph
        for each node in induced_set do
            if node_values[node] is 1 or 2 then
                for each neighbor in graph do
                    if neighbor in induced_set then
                        Add edge to induced_graph
                    end if
                end for
            end if
        end for
        if induced_graph is connected then
            Compute roman_number ← sum of node_values
            if roman_number < min_roman_number then
                Update min_roman_number and best_node_values
            end if
        end if
    end for
    return (min_roman_number, best_node_values)
end function
```

3.3 Algorytm liniowy dla drzew

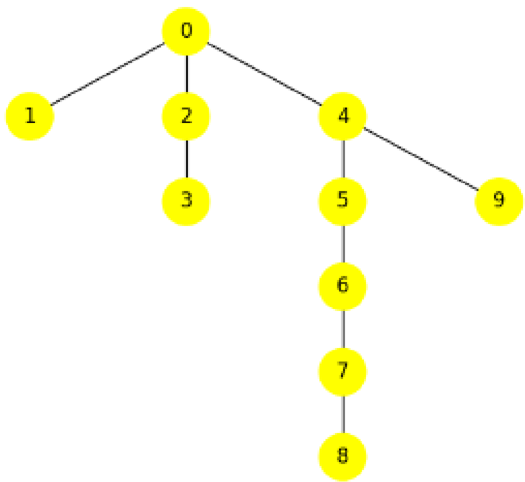
3.3.1 Działanie

Dla każdego wierzchołka definiujemy następujące parametry:

- $v['R']$ - wartość funkcji dominowania rzymskiego słabospójnego 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 rozpatrujemy wszystkie wierzchołki drzewa według odwrotnego porządku drzewa, czyli od ostatniego wierzchołka do korzenia (reverse tree-order). Wszystkie poczatkowo zdefiniowane wartości mają wartości 0. W bardzo ogólnym rozumieniu, rozpatrujemy 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 dominowania rzymskiego słabospójnego, a suma tych wartości to funkcja dominowania rzymskiego słabospójnego.
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 i niezłożone czasowo, 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 (od wierzchołka 8)
   T[4][’n00’] = 1
   T[4][’child’] = 9 (od wierzchołka 9)
   T[2][’n00’] = 1
   T[2][’child’] = 3 (od wierzchołka 3)
   T[0][’n00’] = 1

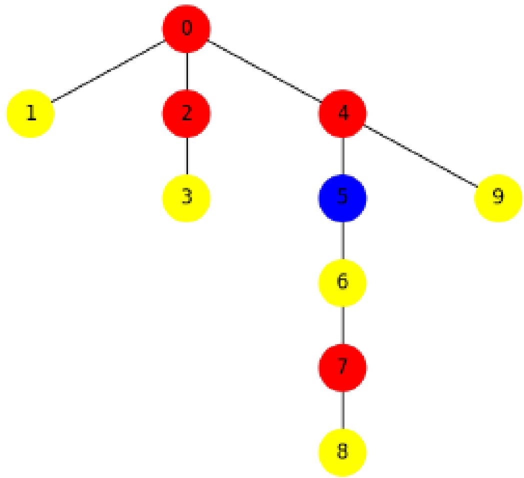
T[0]['child'] = 1 (od wierzchołka 1)

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 (od wierzchołka 7)
   T[0]['sw'] = 2 (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 (od wierzchołka 0)
   T[0]['ch'] = 1 (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 rozpatrzeć 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 niezdominowane 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 (od wierzchołka 4)

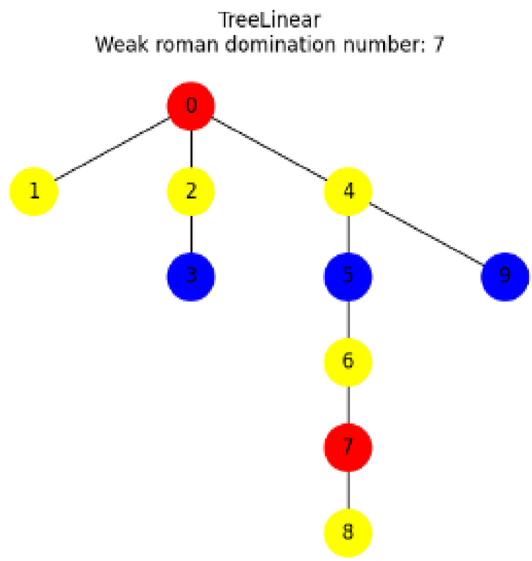
T[9]['R'] = 1 (dziecko wierzchołka 4)

T[2]['R'] = 0 (od wierzchołka 2)

T[3]['R'] = 1 (dziecko wierzchołka 2)

T[0]['n00'] = 3 (ojciec wierzchołków 4 i 2)

Poniższy rysunek przedstawia prawidłowe przypisanie wartości R po fazie 2.



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

3.3.2 Złożoność i wydajność

Algorytm ma złożoność liniową $O(n)$, gdzie n to liczba wierzchołków drzewa. Algorytm zatem jest skalowalny i szybki dla większych grafów, natomiast ograniczony do jednej ich klasy - drzew.

3.3.3 Pseudokod

Algorytm liniowy dla drzew - Faza 1

function PHASE1(T, root)
$father\_map \leftarrow$ Compute parent-child relationships using BFS
$nodes\_ids \leftarrow$ List of all nodes in T
for each node v in reversed($nodes\_ids$) **do**
$father \leftarrow father\_map[v]$
if v is a leaf and $v \neq root$ **then**
Increase $T[father]['n00']$
Set $T[father]['child'] \leftarrow v$
else
if $T[v]['n00'] == 1$ and $T[v]['n01'] == 0$ and $father$ exists **then**
Increase $T[father]['sw']$
end if
if $T[v]['sw'] + T[v]['n00'] + T[v]['n01'] > 1$ **then**
Set $T[v]['R'] = 2$
if $father$ exists **then**
Increase $T[father]['n2']$
if $T[v]['n00'] == 1$ and $T[v]['n01'] == 0$ **then**
Decrease $T[father]['sw']$
end if
end if
$T[v]['ch'] = 1$
end if
if $T[v]['sw'] == 0$ **then**
if $T[v]['n00'] > 1$ or ($T[v]['n00'] == 1$ and ($T[v]['n2'] == 0$ or $T[v]['n01'] > 0$)) **then**
Set $T[v]['R'] = 2$
if $father$ exists **then**
Increase $T[father]['n2']$
end if
else if $T[v]['n00'] == 1$ **then**
Set $T[v]['R'] = 0$
Set $T[T[v]['child']]['R'] = 1$
if $father$ exists **then**
Decrease $T[father]['sw']$
end if
end if
if $T[v]['n00'] == 0$ and $T[v]['n01'] > 0$ **then**
Set $T[v]['R'] = 1$
if $father$ exists **then**
Increase $T[father]['n1']$
end if
end if
end if
if $T[v]['R'] = 0$ **and** $T[v]['n2'] > 0$ **and** $father$ exists **then**
$T[father]['n01'] \leftarrow T[father]['n01] + 1$
end if
if $T[v]['R'] = 0$ **and** $T[v]['n2'] = 0$ **and** $father$ exists **then**
$T[father]['n00'] \leftarrow T[father]['n00] + 1$
end if
end if
end for
if $T[root]['n2'] == 0$ and $T[root]['R'] == 0$ **then**
Set $T[root]['R'] = 2$
end if
if $T[root]['n1'] ==$ (number of root's neighbors - 1) **then**
Set $T[root]['R'] = 1$
end if
return T
end function

Algorytm liniowy dla drzew - Faza 2

```
function PHASE2(T, root)
    for each node v in reversed(nodes_ids) do
        father ← father_map[v]
        if father exists then
            if T[v]['n00'] == 1 and T[father]['ch'] == 1 and T[v]['n01'] == 0 then
                Set T[v]['R'] = 0
                Set T[T[v]['child']]['R'] = 1
                Increase T[father]['n00']
            end if
        end if
    end for
    return T
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”[3].

3.4.1 Działanie

Ten model programowania liniowego wymagania zdefiniowania następujących zmiennych:

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

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

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

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

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

oraz ograniczenia:

Wierzchołek o wartości 0, bedzie miał przynajmniej 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', jesli 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:

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

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

$$\sum_{e \in E} y_e = n - 1, \tag{4}$$

Drzewo rozpinające T' nie posiada cykli:

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

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

$$b_v \leq a_v, \quad v \in V. \tag{6}$$

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

3.4.2 Złożoność i wydajność

Liczba zmiennych dla grafu o n wierzchołkach i m krawędziach wynosi $O(n+m)$. Z powodu wykładniczej natury nierówności liczba ograniczeń wynosi $O(2^n)$.

3.4.3 Pseudokod

Algorytm programowania liniowego I

function ILP_I(graph)
    $V \leftarrow$ list of nodes in $graph$
    $E \leftarrow$ list of edges in $graph$
    Initialize ILP model $model$
    Set objective: Minimize $\sum(a[i] + b[i])$ for all nodes $i \in V$
    Define binary variables:
    $x[i, j]$ for $(i, j) \in E$ ▷ 1 if edge is in G'
    $y[i, j]$ for $(i, j) \in E$ ▷ 1 if edge is in spanning tree T'
    $a[i]$ for $i \in V$ ▷ 1 if node belongs to $V1 \cup V2$
    $b[i]$ for $i \in V$ ▷ 1 if node belongs to $V2$
    Constraints:
    for each node i in V **do** ▷ Each node must be defended
        Add constraint: $a[i] + \sum b[k] \geq 1$, where k are neighbors of i
    end for
    for each edge (i, j) in E **do**
        Add constraint: $y[i, j] \leq x[i, j]$ ▷ Tree edge must exist in G'
        Add constraint: $x[i, j] \leq a[i] + a[j]$ ▷ Tree edges must connect defended nodes
    end for
    Add constraint: $\sum y[i, j] = |V| - 1$ ▷ Tree must have $|V| - 1$ edges
    Find cliques of size ≥ 3 in $graph$ and store as $subsets$
    for each subset S in $subsets$ **do** ▷ Cycle elimination
        Add constraint: $\sum y[i, j] \leq |S| - 1$ for edges $(i, j) \in S$
    end for
    for each node i in V **do** ▷ V2 nodes must be in V1 ∪ V2
        Add constraint: $b[i] \leq a[i]$
    end for
    Solve ILP model
    Extract solution:
    for each node i in V **do**
        $solution[i] \leftarrow round(a[i].X) + 2 * round(b[i].X)$
    end for
    return $(model.objVal, solution)$
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”[3], podobnie jak poprzedni.

3.5.1 Działanie

Model ten opiera się na przepływach. W tym modelu rozpatrujemy graf jako wierzchołek który zużywa jednostkę przepływu, przepływając przez krawędzie grafu. Na wejściu definiujemy zewnętrzny przepływ, którego liczba jednostek jest równa liczbie wierzchołków grafu. Zachowujemy zasadę przepływu sieci, dlatego próbujemy wprowadzić przepływ zewnętrzny w jak najwiekszej ilości przez pojedynczy wierzchołek.
Definiujemy następujące zmienne:

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

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

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

gdzie:
t_i zdefiniowany dla każdego wierzchołka grafu. Identyfikuje, gdzie zewnetrzny wierzchołek jest traktowany jako wejście.
u_i reprezentuje liczbę jednostek przepływu zewnętrznego dla danego wierzchołka grafu.
v_e oznacza jednostki przepływające przez dane krawędzie.
x_i, y_i oznaczają przynależność do zbioru dominującego rzymskiego słabo spójnego.

Minimalizującą liczbę dominowania rzymskiego słabospójnego jako funkcję celu definiujemy następująco:

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

mając dane ograniczenia:
Wierzchołek z wartością 0 sąsiaduje z przynajmniej jednym wierzchołkiem z wartością 2:

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

Prawidłowe zdefiniowanie zmiennych x i y:

$$y_i \leq x_i, \quad i \in V, \tag{2}$$

Krawędź $e \in E'$ ma przynajmniej jeden koniec z wierzchołkiem o wartości przynajmniej 1:

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

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

$$\sum_{i \in V} t_i = 1, \tag{4}$$

Wielkość przepływu wielkości co najwyżej n:

$$u_i \leq n \cdot t_i, \quad i \in V, \tag{5}$$

Przepływ ma miejsce tylko w krawędziach należących do E':

$$v_e \leq n \cdot a_e, \quad e \in E, \tag{6}$$

$$v_e \geq -n \cdot a_e, \quad e \in E, \tag{7}$$

Reprezentacja zasady zachowania sieci:

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

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

$$a_e \geq 1, \quad e \in E, \tag{9}$$

3.5.2 Złożoność i wydajność

Suma zmiennych w sformułowanym modelu wynosi $3n+m$, będących wartościami logicznymi, n integralnych i m ciągłych.
Liczba ograniczeń wynosi $7n + 3m + 2$. Wynika to z tego, że nie wszystkie ograniczenia to są nierówności.

3.5.3 Pseudokod

Algorytm programowania liniowego II

function ILP_II(graph)
  Initialize ILP model $model$ with minimization objective
  $V \leftarrow$ list of nodes in $graph$
  $E \leftarrow$ list of edges in $graph$
  $n \leftarrow |V|$ ▷ Number of nodes
  Define binary variables:
  $x[i]$ for $i \in V$ ▷ 1 if node i is in set X
  $y[i]$ for $i \in V$ ▷ 1 if node i is in set Y
  $a[e]$ for $e \in E$ ▷ 1 if edge e is in the spanning tree
  $t[i]$ for $i \in V$ ▷ 1 if node i is the root
  Define integer and continuous variables:
  $u[i]$ for $i \in V$ ▷ Integer variable for tree structure
  $v[e]$ for $e \in E$ ▷ Flow variable with bounds $[-n, n]$
  Objective:
  Minimize $\sum(x[i] + y[i])$ for all $i \in V$
  Constraints:
  for each node i in V **do** ▷ Ensure all nodes are covered
    Add constraint: $x[i] + \sum y[j] \geq 1$, where $(i, j) \in E$
    Add constraint: $y[i] \leq x[i]$
    Add constraint: $\sum a[e] \geq 1$, where e contains i
  end for
  for each edge $e = (i_e, j_e)$ in E **do**
    Add constraint: $a[e] \leq x[i_e] + x[j_e]$
    Add constraint: $v[e] \leq n \cdot a[e]$
    Add constraint: $v[e] \geq -n \cdot a[e]$
  end for
  Add constraint: $\sum t[i] = 1$ ▷ Only one root exists
  for each node i in V **do** ▷ Tree structure constraints
    Add constraint: $u[i] \leq n \cdot t[i]$
    Add constraint: $u[i] + \sum v[e] - \sum v[e] = 1$, for edges e entering/exiting i
  end for
  Solve ILP model
  Extract solution:
  for each node i in V **do**
    $solution[i] \leftarrow round(x[i].varValue) + 2 \times round(y[i].varValue)$
  end for
  return $(model.objVal, solution)$
end function

3.6 Algorytm mrówkowy

3.6.1 Działanie

Definiujemy 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 (np. 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 prawdopodobieńsw 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 sasiada 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 sumujemy wartości na wierzchołkach grafu.

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

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

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 Złożoność i wydajność

Złożoność wynosi $O(num\_iterations \cdot num\_ants \cdot V^2)$, co sprawia, że algorytm dla większych grafów może być kosztowny obliczeniowo.
Złożoność pamięciowa wynosi $O(V^2 + num\_ants \cdot V)$.

3.6.3 Pseudokod

Algorytm mrówkowy - inicjalizacja

```
function InitializePheromones(graph)
    pheromones ← Assign initial pheromone value to all edges
    return pheromones
end function
function ChooseNodeValue(node, pheromones, neighbors)
    values ← {0, 1, 2}
    Initialize probabilities as empty list
    for each value in {0, 1, 2} do
        Compute pheromone_level as sum of pheromones of neighboring edges
        Compute probability as (pheromone_level^α) × (heuristic^β)
        Append probability to probabilities
    end for
    Normalize probabilities
    return Random weighted choice from {0, 1, 2}
end function
function BuildSolution(graph, pheromones)
    Initialize node_values as empty dictionary
    for each node in graph do
        neighbors ← list of node's neighbors
        Assign node_values[node] ← ChooseNodeValue(node, pheromones, neighbors)
    end for
    return node_values
end function
function IsValidRomanDominatingSet(graph, node_values)
    for each node in graph do
        if node_values[node] = 0 and no neighbor has value 2 then
            return False
        end if
    end for
    induced_set ← nodes with values {1, 2}
    for each node in induced_set do
        Add all neighbors to induced_set
    end for
    Create empty graph induced_graph
    for each node in induced_set do
        if node_values[node] ∈ {1, 2} then
            for each neighbor in graph do
                if neighbor in induced_set then
                    Add edge between node and neighbor in induced_graph
                end if
            end for
        end if
    end for
    if induced_graph is not connected then
        return False
    end if
    return True
end function
```

Algorytm mrówkowy - główna petla

```
function UpdatePheromones(graph, pheromones, solutions)
    for each edge in pheromones do
        Reduce pheromone level using evaporation rate
    end for
    best_solution ← Solution with minimum Roman number
    for each node in best_solution do
        for each neighbor of node do
            Increase pheromone level on edge (node, neighbor)
        end for
    end for
end function
function Execute(graph)
    pheromones ← InitializePheromones(graph)
    best_solution ← None
    best_roman_number ← ∞
    for each iteration in num_iterations do
        Initialize solutions as empty list
        for each ant in num_ants do
            solution ← BuildSolution(graph, pheromones)
            roman_number ← EvaluateSolution(graph, solution)
            Append (solution, roman_number) to solutions
            if roman_number < best_roman_number then
                Update best_roman_number and best_solution
            end if
        end for
        UpdatePheromones(graph, pheromones, solutions)
    end for
    return (best_roman_number, best_solution)
end function
```

3.7 Algorytm aproksymacyjny

3.7.1 Pseudokod

Algorithm 1 Algorytm aproksymacyjny

```
function ComputeDominatingSet(graph)
    dominating_set ← ∅
    uncovered_nodes ← all nodes in graph
    while uncovered_nodes is not empty do
        max_degree_node ← node with highest degree in uncovered_nodes
        Add max_degree_node to dominating_set
        Remove max_degree_node and its neighbors from uncovered_nodes
    end while
    return dominating_set
end function
function Execute(graph)
    dominating_set ← ComputeDominatingSet(graph)
    node_values ← {node : 2 if node ∈ dominating_set, else 0}
    roman_number ← sum of values in node_values
    return (roman_number, node_values)
end function
```

4. PODSUMOWANIE I WNIOSKI

4.1 Podsumowanie wyników

Przedstawienie kluczowych wyników pracy oraz ich znaczenia w kontekście postawionego celu badawczego.

4.2 Wnioski i dalsze kierunki badań

Na podstawie wyników pracy sformułowano następujące wnioski:

- Wniosek 1: [Opis pierwszego wniosku].
- Wniosek 2: [Opis drugiego wniosku].

Dalsze badania mogłyby obejmować:

- Rozszerzenie algorytmu na inne typy danych.
- Testy w środowisku rzeczywistym.

WYKAZ LITERATURY

1. 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.
2. STEWART, I. Defend the Roman Empire!, w: USA: Scientific, 1999, s. 136–139. Dostęp: 03.03.2025.
3. 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.
4. 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.

A. ZAŁĄCZNIKI

A.1 Dodatkowe materiały

Przykładowe materiały pomocnicze:

- Schematy obliczeniowe,
- Dodatkowe wykresy wyników,
- Fragmenty kodu źródłowego (jeśli dotyczy).

A.1.1 Schemat obliczeń

Prezentacja dodatkowych szczegółów dotyczących analizy obliczeniowej.

A.1.2 Kod źródłowy

Wybrane fragmenty implementacji algorytmów w języku Python:

```
def example_function(data):
    return [x**2 for x in data if x > 0]
```