



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

<b>Wykaz ważniejszych oznaczeń i skrótów</b>	<b>6</b>
<b>1 Wstęp i cel pracy</b>	<b>7</b>
1.1 Cel pracy . . . . .	7
1.2 Zakres pracy . . . . .	7
<b>2 Wprowadzenie teoretyczne</b>	<b>8</b>
2.1 Wprowadzenie . . . . .	8
2.2 Geneza historyczna . . . . .	8
2.3 Przegląd literatury . . . . .	9
2.4 Metody badań . . . . .	9
<b>3 Badane algorytmy</b>	<b>11</b>
3.1 Wprowadzenie . . . . .	11
3.2 Algorytm Brute Force . . . . .	11
3.2.1 Działanie . . . . .	11
3.2.2 Złożoność i wydajność . . . . .	11
3.2.3 Pseudokod . . . . .	12
3.3 Algorytm liniowy dla drzew . . . . .	12
3.3.1 Działanie . . . . .	12
3.3.2 Złożoność i wydajność . . . . .	16
3.3.3 Pseudokod . . . . .	16
3.4 Algorytm programowania liniowego I . . . . .	16
3.4.1 Pseudokod . . . . .	16
3.5 Algorytm programowania liniowego II . . . . .	16
3.5.1 Pseudokod . . . . .	16
3.6 Algorytm mrówkowy . . . . .	16
3.6.1 Pseudokod . . . . .	16
3.7 Algorytm aproksymacyjny . . . . .	16
3.7.1 Pseudokod . . . . .	16
<b>4 Podsumowanie i wnioski</b>	<b>22</b>
4.1 Podsumowanie wyników . . . . .	22
4.2 Wnioski i dalsze kierunki badań . . . . .	22
<b>A Załączniki</b>	<b>24</b>
A.1 Dodatkowe materiały . . . . .	24

A.1.1	Schemat obliczeń . . . . .	24
A.1.2	Kod źródłowy . . . . .	24

## 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 znajdowania 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 wykorzystujące 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

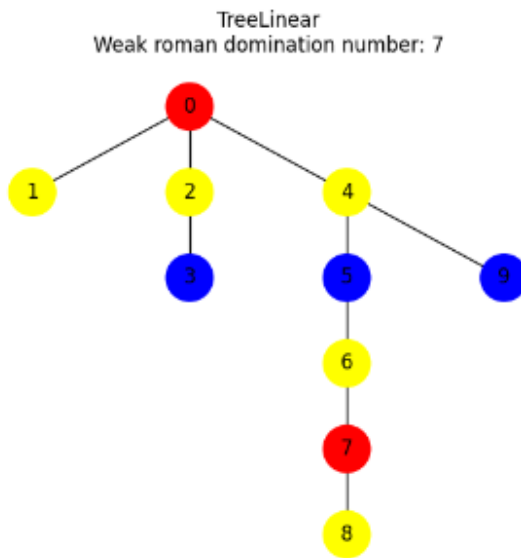
Jako, że problem nie jest powszechnie znany, należy wprowadzić następujące pojęcia [1]:

**Definicja 1** Funkcja dominująca rzymska zdefiniowana jest dla grafu  $G = (V, E)$ , gdzie  $f : V \rightarrow \{0, 1, 2\}$  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 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 utrzymania 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 pamięci 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  $2(1 + \epsilon)(1 + \ln(\Delta - 1))$ -aproxymacyjny. W ramach niniejszej pracy, analizowane będą te dwa modele programowania liniowego oraz aproxymacyjny. 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 kra-

wędzi, 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 słabospójności 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 eksponencjonalną, 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  $O(n + m)$

## Algorytm Brute Force

---

```

1: function FINDROMANDOMINATINGSET(graph)
2:   Initialize  $min\_roman\_number \leftarrow \infty$ 
3:   Initialize  $best\_node\_values \leftarrow None$ 
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 ▷ 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

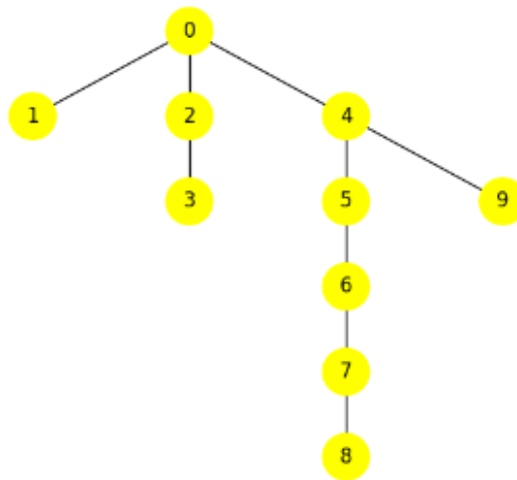
Dla każdego wierzchołka definiujemy następujące parametry:

- $v[R']$  - wartość funkcji dominowania rzymskiego 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 początkowo 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:

$$T[7][n00] = 1$$

$$T[7][child] = 8 \text{ (od wierzchołka 8)}$$

$$T[4][n00] = 1$$

$$T[4][child] = 9 \text{ (od wierzchołka 9)}$$

$$T[2][n00] = 1$$

$$T[2][child] = 3 \text{ (od wierzchołka 3)}$$

$$T[0][n00] = 1$$

$T[0]['child'] = 1$  (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)

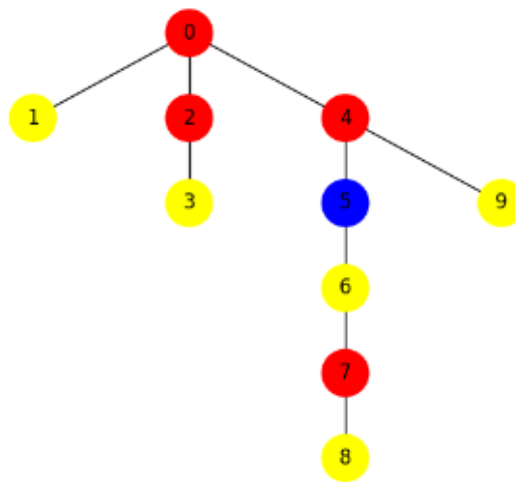
$T[4]['n1'] = 1$  (od wierzchołka 5)

(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 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.

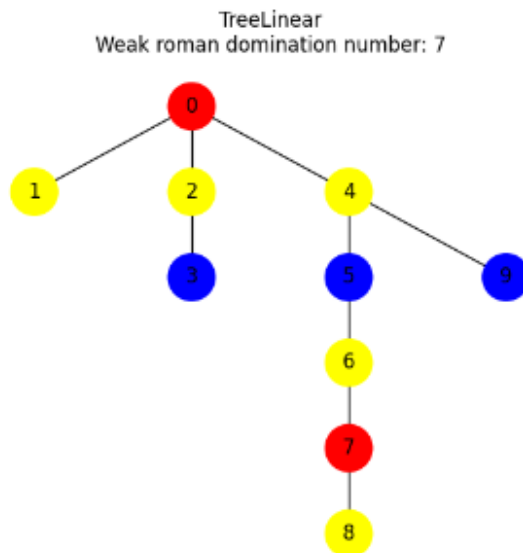
Zdjęcie przedstawia zachowanie algorytmu po fazie 1. Czerwone wierzchołki to wartość  $R = 2$ , niebieskie to  $R = 1$ , a żółte to  $R = 0$ . Widać że przypisanie nie jest jeszcze optymalne.



Rysunek 3.2: Graf  $G$  - po fazie 1

W fazie 2, dla każdego wierzchołka posiadającego ojca, tylko jedno dziecko 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. Poniższy rysunek przedstawia prawidłowe przypisanie wartości  $R$  po fazie 2. 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)



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

## 3.4 Algorytm programowania liniowego I

### 3.4.1 Pseudokod

## 3.5 Algorytm programowania liniowego II

### 3.5.1 Pseudokod

## 3.6 Algorytm mrówkowy

### 3.6.1 Pseudokod

## 3.7 Algorytm aproksymacyjny

### 3.7.1 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'] == (number of root's neighbors - 1) 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( $T$ , root)
2:   for each node  $v$  in 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
```

---

---

### Algorytm programowania liniowego I

---

```
1: function ILP_I( $graph$ )
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  $(i, j) \in E$  ▷ 1 if edge is in  $G'$ 
8:      $y[i, j]$  for  $(i, j) \in E$  ▷ 1 if edge is in spanning tree  $T'$ 
9:      $a[i]$  for  $i \in V$  ▷ 1 if node belongs to  $V1 \cup V2$ 
10:     $b[i]$  for  $i \in V$  ▷ 1 if node belongs to  $V2$ 
11:   Constraints:
12:   for each node  $i$  in  $V$  do ▷ Each node must be defended
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:  $y[i, j] \leq x[i, j]$  ▷ Tree edge must exist in  $G'$ 
17:     Add constraint:  $x[i, j] \leq a[i] + a[j]$  ▷ Tree edges must connect defended nodes
18:   end for
19:   Add constraint:  $\sum y[i, j] = |V| - 1$  ▷ Tree must have  $|V| - 1$  edges
20:   Find cliques of size  $\geq 3$  in  $graph$  and store as  $subsets$ 
21:   for each subset  $S$  in  $subsets$  do ▷ Cycle elimination
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 ▷  $V2$  nodes must be in  $V1 \cup V2$ 
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
```

---

---

## 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:    $n \leftarrow |V|$  ▷ Number of nodes
6:   Define binary variables:
7:    $x[i]$  for  $i \in V$  ▷ 1 if node  $i$  is in set  $X$ 
8:    $y[i]$  for  $i \in V$  ▷ 1 if node  $i$  is in set  $Y$ 
9:    $a[e]$  for  $e \in E$  ▷ 1 if edge  $e$  is in the spanning tree
10:   $t[i]$  for  $i \in V$  ▷ 1 if node  $i$  is the root
11:  Define integer and continuous variables:
12:   $u[i]$  for  $i \in V$  ▷ Integer variable for tree structure
13:   $v[e]$  for  $e \in E$  ▷ Flow variable with bounds  $[-n, n]$ 
14:  Objective:
15:  Minimize  $\sum (x[i] + y[i])$  for all  $i \in V$ 
16:  Constraints:
17:  for each node  $i$  in  $V$  do ▷ Ensure all nodes are covered
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:  $\sum t[i] = 1$  ▷ Only one root exists
28:  for each node  $i$  in  $V$  do ▷ Tree structure constraints
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
```

---

---

#### 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 heuristic based on number of neighbors
11:    Compute probability as  $(pheromone\_level^\alpha) \times (heuristic^\beta)$ 
12:    Append probability to probabilities
13:   end for
14:   Normalize probabilities
15:   return Random weighted choice from {0, 1, 2}
16: end function
17: function BUILDSOLUTION(graph, pheromones)
18:   Initialize node_values as empty dictionary
19:   for each node in graph do
20:     neighbors  $\leftarrow$  list of node's neighbors
21:     Assign node_values[node]  $\leftarrow$  CHOOSENODEVALUE(node, pheromones, neighbors)
22:   end for
23:   return node_values
24: end function
25: function EVALUATESOLUTION(graph, node_values)
26:   if not ISVALIDROMANDOMINATINGSET(graph, node_values) then
27:     return  $\infty$ 
28:   end if
29:   return Sum of all node values
30: end function
```

---

---

**Algorytm mrówkowy - główna petla**

---

```
1: function UPDATEPHEROMONES(graph, pheromones, solutions)
2:   for each edge in pheromones do
3:     Reduce pheromone level using evaporation rate
4:   end for
5:    $best\_solution \leftarrow$  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:    $pheromones \leftarrow$  INITIALIZEPHEROMONES(graph)
14:    $best\_solution \leftarrow None$ 
15:    $best\_roman\_number \leftarrow \infty$ 
16:   for each iteration in num_iterations do
17:     Initialize  $solutions$  as empty list
18:     for each ant in num_ants do
19:        $solution \leftarrow$  BUILDSOLUTION(graph, pheromones)
20:        $roman\_number \leftarrow$  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
```

---

---

**Algorithm 1 Algorytm aproksymacyjny**

---

```
1: function COMPUTEDOMINATINGSET(graph)
2:    $dominating\_set \leftarrow \emptyset$ 
3:    $uncovered\_nodes \leftarrow$  all nodes in graph
4:   while  $uncovered\_nodes$  is not empty do
5:      $max\_degree\_node \leftarrow$  node with highest degree in  $uncovered\_nodes$ 
6:     Add  $max\_degree\_node$  to  $dominating\_set$ 
7:     Remove  $max\_degree\_node$  and its neighbors from  $uncovered\_nodes$ 
8:   end while
9:   return  $dominating\_set$ 
10: end function
11: function EXECUTE(graph)
12:    $dominating\_set \leftarrow$  COMPUTEDOMINATINGSET(graph)
13:    $node\_values \leftarrow \{node : 2 \text{ if } node \in dominating\_set, \text{ else } 0\}$ 
14:    $roman\_number \leftarrow$  sum of values in  $node\_values$ 
15:   return  $(roman\_number, node\_values)$ 
16: 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]
```