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 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 aproksymacyjny o współczynniku aproksymacji $2(1 + ln(\Delta + 1))$. 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.*

2.2 Geneza historyczna

Problem swoją nazwę zawdzięcza imperium rzymskiemu. Po raz pierwszy został opisany w artykule „Defend Roman Empire!”. 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

Omówienie istniejących metod i podejść z literatury, które odnoszą się do problemu badawczego. Jednym z istotnych źródeł w tym zakresie jest strona National Center of Biotechnology Information, która dostarcza bogatych danych do analiz [2].

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 słabospójności w następujący sposób:

- wyznaczamy zbiór indukowany, który składa się ze zbioru dominującego (wierzchołki z wartościami {1, 2}) oraz sąsiadów wierzchołków zbioru dominującego,
- 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 wiekszych 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 $min\_roman\_number \leftarrow \infty$
  Initialize $best\_node\_values \leftarrow None$
  $nodes \leftarrow$ list of nodes in $graph$
  for each assignment of values $(0, 1, 2)$ to all nodes **do**
    $node\_values \leftarrow$ mapping of nodes to values
    $induced\_set \leftarrow$ 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 $roman\_number \leftarrow$ 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 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

```
function PHASE1(T, root)
    father_map ← Compute parent-child relationships using BFS
    nodes_ids ← List of all nodes in T
    for each node v in reversed(nodes_ids) do
        father ← father_map[v]
        if v is a leaf and v ≠ root then
            if father exists then
                Increase T[father]['n00']
                Set T[father]['child'] ← v
            end if
        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) 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
            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)
    father_map ← Compute parent-child relationships using BFS
    nodes_ids ← List of all nodes in T
    for each node v in reversed(nodes_ids) do
        father ← father_map[v]
        if father exists then
            T[father]['child'] ← v
            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
```

Algorytm programowania liniowego I

```
function ILP_I(graph)
    V ← list of nodes in graph
    E ← list of edges in graph
    Initialize ILP model model
    Set objective: Minimize ∑(a[i] + b[i]) for all nodes i ∈ V
    Define binary variables:
    x[i, j] for (i, j) ∈ E                          ▷ 1 if edge is in G′
    y[i, j] for (i, j) ∈ E                          ▷ 1 if edge is in spanning tree T′
    a[i] for i ∈ V                                  ▷ 1 if node belongs to V1 ∪ V2
    b[i] for i ∈ V                                  ▷ 1 if node belongs to V2
    Constraints:
    for each node i in V do                         ▷ Each node must be defended
        Add constraint: a[i] + ∑ b[k] ≥ 1, where k are neighbors of i
    end for
    for each edge (i, j) in E do
        Add constraint: y[i, j] ≤ x[i, j]           ▷ Tree edge must exist in G′
        Add constraint: x[i, j] ≤ a[i] + a[j]       ▷ Tree edges must connect defended nodes
    end for
    Add constraint: ∑ 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: ∑ y[i, j] ≤ |S| − 1 for edges (i, j) ∈ S
    end for
    for each node i in V do                         ▷ V2 nodes must be in V1 ∪ V2
        Add constraint: b[i] ≤ a[i]
    end for
    Solve ILP model
    Extract solution:
    for each node i in V do
        solution[i] ← round(a[i].X) + 2 ∗ round(b[i].X)
    end for
    return (model.objVal, solution)
end function
```

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

Algorytm mrówkowy - inicjalizacja

function InitializePheromones(graph)
    $pheromones \leftarrow$ Assign initial pheromone value to all edges
    return $pheromones$
end function
function ChooseNodeValue(node, pheromones, neighbors)
    $values \leftarrow \{0, 1, 2\}$
    Initialize $probabilities$ as empty list
    for each $value$ in $\{0, 1, 2\}$ **do**
        Compute $pheromone\_level$ as sum of pheromones of neighboring edges
        Compute $heuristic$ based on number of neighbors
        Compute probability as $(pheromone\_level^{\alpha}) \times (heuristic^{\beta})$
        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**
        $neighbors \leftarrow$ list of node's neighbors
        Assign $node\_values[node] \leftarrow$ ChooseNodeValue(node, pheromones, neighbors)
    end for
    return $node\_values$
end function
function EvaluateSolution(graph, node_values)
    if not IsValidRomanDominatingSet(graph, node_values) **then**
        return ∞
    end if
    return Sum of all node values
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
```

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. OPENAI. *Treść wygenerowana przy użyciu narzędzi ChatGPT (wersja 2024)*. 2024. Dostępne także z: `https://openai.com`. Dostęp: 15.01.2025.
3. KOWALSKI, J.; KABACKI, J. Simulation of Network Systems in Education. W: *Proceedings of the XXIV Autumn International Colloquium Advanced Simulation of Systems*. Ostrava, Czechy: ASIS, 2002, s. 213–218. 9–11 września 2002.

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]
```