Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ”

Кафедра математики факультету інформатики

Кваліфікаційна робота
освітній ступінь — бакалавр

на тему: **"M-Ліпшицеві відображення на графах"**

Виконала: студентка 4-го року навчання,
Спеціальності
113 Прикладна математика
Гуназа Анна Олегівна

Керівник:
к. ф.-м. н. *Козеренко С.О.*

Рецензент ________________________________
(прізвище та ініціали)

Кваліфікаційна робота захищена
з оцінкою ________________________________

Секретар ЕК ________________________________
(підпис)
“____”____________________________ 2024 р.

Київ – 2024

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ"

Кафедра математики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри математики, доц., к.ф-м.н.
___________________ Р.К. Чорней
(підпис)
"____"___________________ 2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на кваліфікаційну роботу
студентці 4-го курсу факультету інформатики
Гуназі Анні Олегівні

Тема: M-Ліпшицеві відображення на графах.
Зміст ТЧ до курсової роботи:

Анотація
Вступ
1. Основні означення
2. Основні результати
3. Програмна реалізація алгоритмів на Python
4. Застосування алгоритмів на прикладах
Висновки
Література

Дата видачі "____"___________ 2024 р. Керівник _________________________
(підпис)
Завдання отримав ________________
(підпис)

Графік підготовки кваліфікаційної роботи до захисту

Номер	Назва етапу курсової	Термін виконання етапу	Примітка
1.	Отримання теми кваліфікаційної роботи.	01.08.2023	
2.	Ознайомлення з темою кваліфікаційної.	01.09.2023	
3.	Розробка плану та структури роботи.	15.09.2023	
4.	Робота з науковою літературою, опис основних означень теорії графів.	15.10.2023	
5.	Аналіз та побудова алгоритмів.	01.01.2024	
6.	Робота над текстовим оформленням результатів.	01.04.2024	
7.	Попередній аналіз кваліфікаційної. Виправлення помилок.	01.05.2024	
8.	Захист кваліфікаційної роботи.	03.06.2024	

Зміст

Анотація **5**

Вступ **6**

1 Основні означення **7**

2 Основні результати **14**

2.1 Мотивація вибору алгоритмів для реалізації 14
2.2 Постановка задач M-ParExt та Strong M-ParExt 15
2.3 Алгоритм для M-ParExt на деревах 16
2.3.1 Псевдокод зі статті [1] 16
2.3.2 Пояснення основних кроків алгоритму 17
2.3.3 Зауваження 18
2.4 Алгоритм для M-ParExt на загальних графах 19
2.4.1 Псевдокод зі статті [1] 21
2.4.2 Опис основних кроків алгоритму 22
2.5 Порівняння M-ParExt на деревах та загальних графах . . . 23
2.6 M-ParExt на графах блоків 24
2.6.1 Мотивація 24
2.6.2 Опис алгоритму 25
2.7 Складність алгоритмів 27
2.8 Строгі M-Ліпшицеві відображення 28

3 Програмна реалізація алгоритмів на Python **30**

3.1 M-ParExt на деревах 30
3.2 Strong M-ParExt на деревах 33
3.3 M-ParExt та Strong M-ParExt на загальних графах 36
3.4 M-ParExt на графах блоків 39

4 Застосування алгоритмів на прикладах **44**

Висновки **52**

Література **53**

Анотація

Кваліфікаційна робота присвячена дослідженню M-Ліпшицевих відображень на графах. У роботі розроблено та проаналізовано алгоритми для задач M-ParExt та Strong M-ParExt на деревах, загальних графах та графах блоків. Вони дозволяють розширювати часткові M-Ліпшицеві відображення на весь граф, зберігаючи при цьому властивість M-досяжності між образами вершин. Представлено теоретичне обґрунтування, програмну реалізацію на Python та приклади застосування.

Ключові слова: Графи, дерева, графи боків, M-Ліпшицеве відображення, розширення часткових відображень, M-ParExt, Strong M-ParExt, алгоритми.

Вступ

З поняттям Ліпшицевого відображення більшість з нас знайома. Воно характеризується наявністю константи L, яка дозволяє контролювати відстань між образами функції. Аналогічне поняття присутнє і в теорії графів та називається M-Ліпшицевим відображенням на графах. Основна ідея полягає у контролюванні відстані між образами суміжних вершин графа.

Така цікава властивість дозволяє вирішувати задачі, де важлива рівномірність розподілення даних. Як приклад можна розглянути моніторинг водних ресурсів: рівень забруднення в сусідніх водних об'єктах повинен бути схожим, щоб запобігти раптовим змінам у якості води. Це допоможе ефективніше управляти водними ресурсами та вчасно виявляти забруднення.

Окрім того, часто ми маємо справу з неповними або пошкодженими даними. Тому визначення, чи певне відображення може бути частиною M-Ліпшицевого відображення, можна розглядати як швидкий спосіб виключення випадків явно суперечливих даних.

Мета цієї роботи складається з наступних пунктів:

1. Дослідити основні властивості М-Ліпшицевих відображень на графах.
2. Реалізувати алгоритми з розширення М-Ліпшицевих відображень на деревах і на загальних графах за наданими псевдокодами.
3. Створити модифікації цих алгоритмів для розширення сильних М-Ліпшицевих відображень.
4. Створити і реалізувати власний алгоритм з розширення М-Ліпшицевих відображень на графах блоків.

1 Основні означення

Означення 1.1. *Неорієнтований граф* – це пара $G = (V, E)$, де

$$V = V(G) \text{ – множина } \textit{вершин},$$
$$E = E(G) \subset V^{(2)} = \{\{a, b\} |\ a, b \in V\} \text{ – множина } \textit{ребер}.$$

Означення 1.2. *Зв'язний граф* – це граф, між будь-якою парою вершин якого існує шлях, що їх сполучає.

Означення 1.3. *Компонента зв'язності* графа – це його максимальний (за включенням) зв'язний підграф.

Означення 1.4. *Простий граф* – це граф без кратних ребер та петель.

Надалі в роботі всі графи вважаються неорієнтованими, зв'язними, і простими, якщо не вказано протилежного. Ребра графа G позначатимемо через $xy \in E(G)$, де $x, y \in V(G)$.

Означення 1.5. *Шлях* між парою вершин $x, y \in V(G)$ – це послідовність вершин $v_0, v_1, \ldots, v_n$, де $x = v_0$ – *початок шляху*, $y = v_n$ – *кінець шляху*, $\forall i = \overline{0, n-1} :\ v_i v_{i+1} \in E(G)$.

Означення 1.6. *Відстань* $d(x, y)$ між парою вершин $x, y \in V(G)$ – це кількість ребер у найкоротшому шляху, що їх сполучає.

Означення 1.7. *Цикл* – це шлях, у якого початок та кінець збігаються.

Означення 1.8. *Дерево* – це зв'язний граф, який не містить циклів.

Означення 1.9. *Двочастковий граф* – граф, множину вершин якого можна розбити на дві підмножини (*частки*) так, щоб жодні дві вершини з

однієї підмножини не були суміжними, тобто

$$V = V_1 \sqcup V_2,\ E \subset \{ab|\ a \in V_1, b \in V_2\}.$$

Означення 1.10. Для $M \in \mathbb{N}$, *M-Ліпшицеве відображення* зв'язного графа $G = (V, E)$ з коренем $v_0 \in V$ - це відображення $f : V \to \mathbb{Z}$ таке, що $f(v_0) = 0$ і для кожного ребра $uv \in E$ виконується $|f(u) - f(v)| \leq M$. Множина всіх M-Ліпшицевих відображень графа G позначається $\mathcal{L}_M(G)$.

Означення 1.11. Відображення $f : V' \to \mathbb{Z}$ графу G, де $V' \subseteq V(G)$, називають *M-досяжним (M-reachable)*, якщо для кожної пари вершин $u, v \in V'$ виконується

$$|f(u) - f(v)| \leq M \cdot d(u, v),$$

де $d(u, v)$ позначає відстань між вершинами u та v у графі G.

Означення 1.12. Відображення $f : V' \to \mathbb{Z}$ графа G, де $V' \subseteq V(G)$, називають таким, що *має корінь (rooted)*, якщо $f^{-1}(0) \neq \emptyset$.

Означення 1.13. Для $M \in \mathbb{N}$ *сильне (strong) M-ліпшицеве відображення* зв'язного графа $G = (V, E)$ з коренем $v_0 \in V$ - це відображення $f : V \to \mathbb{Z}$ таке, що $f(v_0) = 0$ і для кожного ребра $uv \in E$ виконується умова $|f(u) - f(v)| = M$. Множина всіх сильних M-ліпшицевих відображень графа G позначається $L_{\pm M}(G)$.

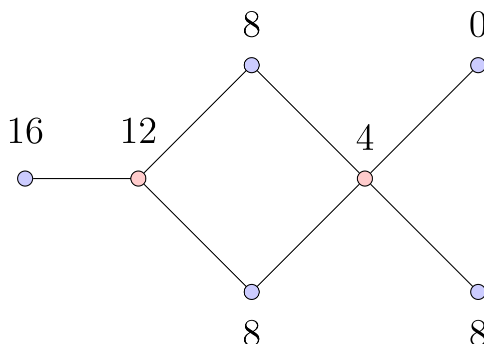


Рис. 1: Граф G_1 з сильним M-Ліпшицевим відображенням

Лема 1.14. [1] *Граф має сильне M-ліпшицеве відображення тоді і тільки тоді, коли він є двочастковим.*

Доведення. Розглянемо граф G, який не є двочастковим, і сильне М-ліпшицеве відображення f графа G. Нагадаємо, що граф є двочастковим тоді й тільки тоді, коли він не містить циклу непарної довжини як підграфа.

Від супротивного, припустимо, що граф G містить непарний цикл C з ребрами $v_1v_2, \dots, v_lv_1$. Позначимо

$$e_i := f(v_{(i+1 \mod l)}) - f(v_{(i \mod l)}), \quad \forall i \in \{1, \dots, l\}.$$

Бачимо, що $e_i \in \{+M, -M\}$. Більше того, $\sum_{i=0}^{n} e_i = 0$ з визначення e_i. Однак ця сума має непарну кількість доданків, тому вона не може дорівнювати нулю. Отримали протиріччя. Отже, граф з сильним M-Ліпшицевим відображенням має цикли лише парної довжини, а тому завжди є двочастковим. □

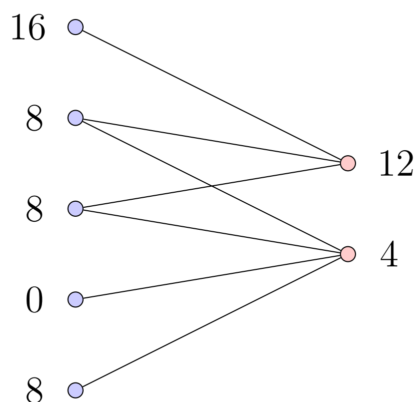


Рис. 2: Двочасткове розбиття графа G_1 з сильним M-Ліпшицевим відображенням

Лема 1.14 має обґрунтування лише в один бік, тому в наступному твердженні ми пропонуємо розглянути й інший.

Твердження 1.15. *Якщо граф є двочастковим, то він має сильне M-ліпшицеве відображення f.*

Доведення. Розглянемо двочастковий граф G з двома частками V_1 і V_2.

Розглянемо відображення $f : V \to \mathbb{Z}$, де V - множина вершин графа. Позначимо через M деяку цілу константу.

Визначимо відображення f наступним чином:

$$f(v) = \begin{cases} 0, & \text{якщо } v \in V_1, \\ M, & \text{якщо } v \in V_2. \end{cases}$$

Тоді для будь-якого ребра $uv \in E(G)$, $(u \in V_1, v \in V_2)$ в графі маємо:

$$|f(u) - f(v)| = |0 - M| = M,$$

що підтверджує сильну M-ліпшицевість відображення f для двочасткового графа G. Таким чином можна визначити тривіальне f для будь-якого

двочасткового графа, проставивши 0 усім вершинам з однієї частки і M - з іншої (як на Рисунку 3), що доводить наше твердження. □

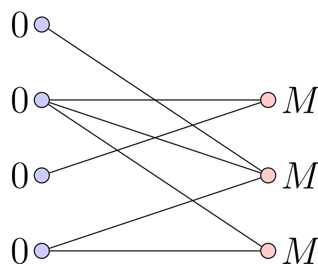


Рис. 3: Приклад сильного M-Ліпшицевого відображення на двочастковому графі

Тепер перейдемо до розгляду графів блоків, перед цим ознайомившись з поняттями точки з'єднання, 2-зв'язності та блоку графа.

Означення 1.16. *Точка з'єднання (cut vertex)* – це вершина, видалення з G якої збільшує кількість зв'язних компонент.

Кажуть, що вершина u *розділяє* дві вершини x та y, якщо x та y знаходяться в одній зв'язній компоненті у G, але у різних зв'язних компонентах у $G - u$. Очевидно, що вершина є точкою з'єднання тоді і тільки тоді, коли вона розділяє деяку пару вершин.

Означення 1.17. Граф називають 2-*зв'язним*, якщо він є зв'язним і не має точок з'єднання.

Означення 1.18. *Блок* у графі - це його максимальний 2-зв'язний підграф.

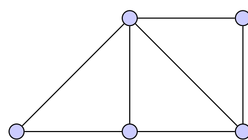


Рис. 4: Приклад двозв'язного графа

Означення 1.19. *Граф блоків* $B(G)$ графу G - це граф перетину на колекції блоків у G. Іншими словами, вершини $B(G)$ відповідають блокам у G, причому два блоки є сусідніми, якщо вони мають спільну вершину (яка повинна бути точкою з'єднання у G). Абстрактний граф H називається *графом блоків*, якщо він ізоморфний $B(G)$ для деякого G.

Наступна характеризація графів блоків часто використовується як визначення графів блоків.

Теорема 1.20. [3] *Граф H є графом блоків тоді і тільки тоді, коли кожен блок у H є повним підграфом.*

Зауважимо, що Теорема 1.20 безпосередньо стверджує, що будь-яке дерево є графом блоків.

Означення 1.21. *Блоко-точкове дерево* $T(G)$ – граф, множина вершин якого є об'єднанням множини блоків і точок з'єднання графа G. При цьому дві вершини є суміжними лише у випадку, якщо одна відповідає блоку B графа G, а інша - точці з'єднання c графа G та $c \in B$.

Теорема 1.22. [4] *Якщо G є зв'язним, то $T(G)$ є деревом.*

Доведення. Якщо $T(G)$ має цикл, цей цикл повинен містити принаймні два блоки B_1 і B_2 як вершини. У $T(G)$ існують два шляхи від B_1 до B_2, а тому і в G. Тоді B_1 і B_2 містяться у тому самому блоку B у G, що суперечить логіці. Отже, $T(G)$ є ациклічним. Оскільки зв'язність G випливає зі зв'язності $T(G)$, тримуємо, що $T(G)$ є деревом. □

Приклад 1.23. Для графу блоків G_1 з Рисунку 5 відповідне блоко-точкове дерево зображено на Рисунку 6.

Блоки виглядають наступним чином:

$B_0 = \{1, 2, 3, 4\}$, $B_1 = \{4, 5, 6\}$, $B_2 = \{5, 11\}$, $B_3 = \{6, 7\}$, $B_4 = \{7, 8, 9\}$, $B_5 = \{7, 10\}$.

Точки з'єднання: $\{4, 5, 6, 7\}$.

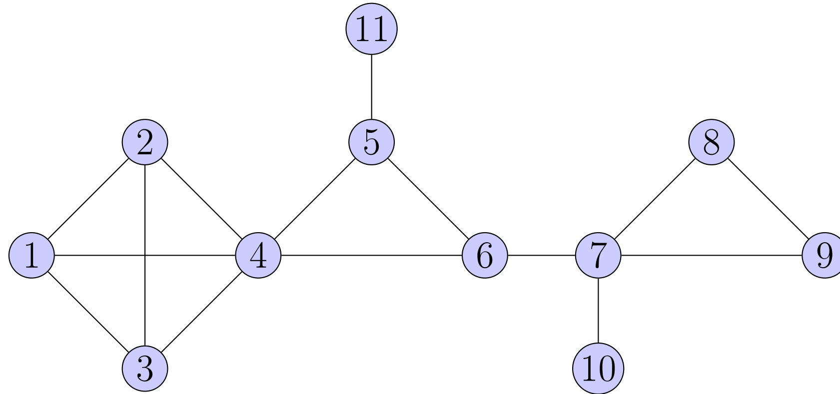


Рис. 5: Граф блоків G_1

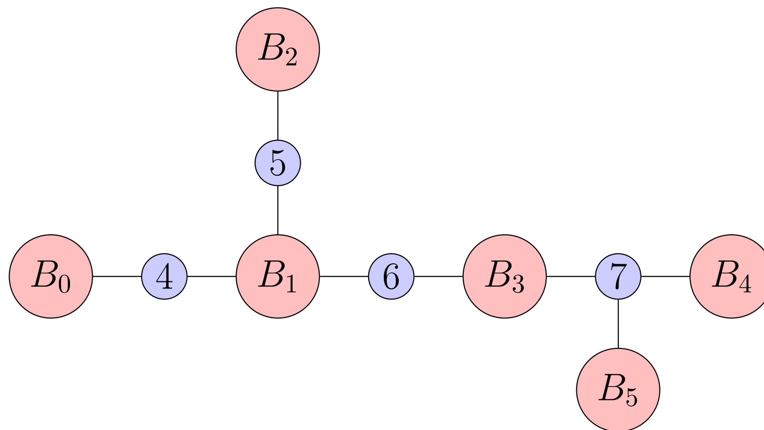


Рис. 6: Блоко-точкове дерево для графа G_1

2 Основні результати

2.1 Мотивація вибору алгоритмів для реалізації

Як було зазначено вище, M-Ліпшицеві відображення кожній вершині графа ставлять у відповідність певне ціле число. І виявляється, що існує бієкція між M-Ліпшицевими відображеннями та гомоморфізмами графів! Розглянемо граф Z_M з множиною вершин $V(Z_M) = Z$ та множиною ребер $E(Z_M) = \{ij : |i - j| \leq M\}$. Кожне M-Ліпшицеве відображення відповідає гомоморфізму графа до Z_M.

Проблема: Проблема гомоморфізму списку - $LHom(H)$

Вхідні дані: Граф G та функція списку $L : V(G) \to 2^{V(H)}$.

Питання: Чи існує гомоморфізм $f : G \to H$ такий, що $f(v) \in L(v)$ для кожного $v \in V(G)$?

Тобто, так само як константа M дозволяє контролювати відстань між образами відображення f для кожної пари суміжних вершин, у проблемі $LHom(H)$ списки обмежують, до яких вершин у цільовому графі може бути відображена кожна вершина у даному графі.

Алгоритм вирішення цієї відомої проблеми наведений у статті [2]. Для випадку $LHom(Z_M)$ його часова складність - $O(|V|^4)$. У наступних пунктах ми розглянемо задачу M-ParExt та її алгоритмічне вирішення на деревах та загальних графах. У [1] було запропоновано алгоритм з меншою часовою складністю, який ми детально проаналізували та запрограмували.

2.2 Постановка задач M-ParExt та Strong M-ParExt

Проблема M-LipExt: Розширення часткового M-Ліпшицевого відображення

Вхідні дані: Зв'язний граф $G = (V, E)$, підмножина $V' \subseteq V$ з функцією $f' : V' \to \mathbb{Z}$.

Питання: Чи існує M-Ліпшицеве відображення f графу G таке, що $f' \subseteq f$?

Назви даного алгоритму M-LipExt та M-ParExt - можуть вживатися взаємозамінно. Аналогічно ставиться задача розширення *сильного* M-Ліпшицевого відображення:

Проблема Strong M-LipExt: Розширення часткового *сильного* M-Ліпшицевого відображення

Вхідні дані: Зв'язний граф $G = (V, E)$, підмножина $V' \subseteq V$ з функцією $f' : V' \to \mathbb{Z}$.

Питання: Чи існує *сильне* M-Ліпшицеве відображення f графу G таке, що $f' \subseteq f$?

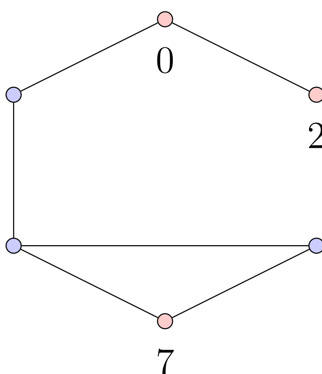


Рис. 7: Приклад часткового відображення з трьома промаркованими вершинами

2.3 Алгоритм для M-ParExt на деревах

2.3.1 Псевдокод зі статті [1]

Algorithm 1 Алгоритм для M-ParExt на деревах

Require: A tree graph G, a vertex set $V' \subseteq V(G)$, and a partial M-Lipschitz mapping $f' : V' \to \mathbb{Z}$.

1: Check if $|f'(v) - f'(u)| \leq M$ for $u, v \in V' : uv \in E(G)$. If not, f' cannot be extended.
2: Set $P(v) := [f'(v), f'(v)]$ for every $v \in V'$.
3: Set $P(v) := [-\infty, \infty]$ for every $v \in V(G) \setminus V'$.
4: **for** every v' in V' **do**
5:     Start the DFS on G from v'.
6:     In DFS, when you process vertex v with $P(v) = [\underline{P}(v), \overline{P}(v)]$, do the following:
7:     **for** every $w \in N_G(v)$ **do**
8:         $P(w) := [\underline{P}(v) - M, \overline{P}(v) + M] \cap P(w)$.
9:     **end for**
10: **end for**
11: Find $r \in V(G)$ such that $0 \in P(v)$ and re-run DFS from Line 5 with $v' = r$.
12: **if** no such r exists **then**
13:     **return** The mapping f' cannot be extended.
14: **end if**
15: Set $f(r) := 0$.
16: **if** $P(v) = \emptyset$ for some $v \in V(G)$ **then**
17:     **return** The mapping f' cannot be extended.
18: **end if**
19: Launch the BFS from r and for every visited vertex $v \neq r$, set $f(v)$ so that for parent vertex p, $f(v) \in [f(p) - M, f(p) + M] \cap P(v)$ holds.
20: **if** the previous BFS could not be completed **then**
21:     **return false**
22: **end if**
23: **return true**

2.3.2 Пояснення основних кроків алгоритму

Згідно з псевдокодом, наведеним вище, алгоритм пошуку M-Ліпшицевого відображення f складається з наступних кроків:

1. Перевірити, чи дане відображення f' є M-Ліпшицевим: відстань між образами суміжних вершин має не перевищувати задане число M.

2. Кожній вершині присвоїти початковий інтервал P: для вершин з визначеною в умові міткою інтервал міститиме лише це значення, для решти $(-\infty, +\infty)$.

3. Починаючи з вершин з відомими мітками, проходимося по всьому графу G (пошуком у глибину), оновлюючи інтервали сусідніх до даної вершин наступним чином: інтервал на даній вершині розширюємо на M одиниць в обидва боки і знаходимо його перетин з інтервалом сусідної вершини.

4. Знаходимо корінь, обравши вершину, у чий інтервал входить 0. Оновлюємо інтервали ще раз, починаючи з кореня.

5. Проходимося по графу G востаннє, щоб проставити мітки для решти вершин. Для цього обране число має належати визначеному на цій вершині інтервалу, а також мати різницю з суміжними вершинами не більш, ніж на M.

Аналогічно реалізовується Strong M-ParExt на деревах. Єдина відмінність від даного алгоритму полягає у використанні множин можливих значень S для вершин замість інтервалів P та відповідному підлаштуванні функцій, які приймали P як параметр, під цю зміну.

2.3.3 Зауваження

У даному псевдокоді ми знайшли дві неточності, які виправили.

1. У першому рядку була умова:

   $|f'(v) - f'(u)| \leq M$ for all $u, v \in V'$.

   Вона є неправильною, адже за означенням [1.8] відстань між образами контролюється лише для суміжних вершин. Тут же було запропоновано перевіряти усі пари вершин. Ми додали це уточнення, і тепер умова виглядає так:

   $|f'(v) - f'(u)| \leq M$ for $u, v \in V' : uv \in E(G)$.

2. У рядку 19, на етапі присвоєння міток, зазначено, що для кожної вершини v та її батька p має виконуватись:

   $f(v) \in [f(p) - M, f(p) + M]$.

   Попри правильність цього твердження, ми вважаємо важливим вказати, що треба обов'язково зважати на інтервал $P(v)$, обчислений для даної вершини раніше. Після виправлення умова виглядає так:

   $f(v) \in [f(p) - M, f(p) + M] \cap P(v)$.

Обидва виправлення виділені червоним у псевдокоді.

2.4 Алгоритм для M-ParExt на загальних графах

Для початку розглянемо теорему та її доведення, що описують алгоритм побудови розширеного відображення на загальних графах.

Теорема 2.1. [1] *Для графа $G = (V, E)$, підмножини $V' \subseteq V$, часткового відображення $f' : V' \to \mathbb{Z}$, наступні твердження еквівалентні:*

1. *Відображення f' можна розширити до M-Ліпшицевого відображення.*
2. *Одне з наступних тверджень виконується:*
   (а) *Відображення f' є M-досяжним та має корінь.*
   (б) *Існує $r \in V \setminus V'$, для якого f'' визначається як $f'' := f' \cup \{(r, 0)\}$, таке що f'' є M-досяжним.*

Доведення. (1) $\Rightarrow$ (2): Розглянемо (a), випадок (b) є аналогічним. Припустимо, що f' можна розширити до M-Ліпшицевого відображення f^*. Оберемо пару вершин $u, v \in V'$ таких, що $|f(u) - f(v)| > M \cdot d(u, v)$. Оберемо найкоротший шлях $u = x_1, \ldots, x_l = v$ між u і v. Кожен доданок у сумі $\sum_{i=2}^{l}(f(x_i) - f(x_{i-1}))$ не перевищує M та є не меншим за $-M$, що суперечить тому, що відображення f^* є M-Ліпшицевим.

(2) $\Rightarrow$ (1): Знову розглянемо випадок (a), тоді (b) доводиться аналогічно. Покажемо, що ми можемо розширити відображення f' на одну вершину і при цьому зберегти його M-досяжність. Застосувавши цю схему індуктивно, доведемо, що з (2) слідує (1).

Оберемо вершину a, що суміжна з вершиною b, такою що $f'(b)$ визначена, а $f'(a)$ - ні. Для кожної вершини $c \in f'(V')$, вершина b є досяжною. Таким чином ми завжди можемо знайти число (мітку $f'(a)$) для a таке, що c буде досяжною з a. Для кожної c ми можемо визначити інтервал I_c, який

міститиме всі можливі значення для a такі, що c є досяжною з a. Це є замкнутий зв'язний інтервал у $\mathbb{Z}$. Крім того, система множин $\{I_c | \forall c \in f'(V')\}$ має властивість Хеллі. І наостанок, для кожних двох $c_1, c_2 \in f'(V')$ існує непорожній перетин. В іншому випадку ми отримали б суперечність з M-досяжністю f'. Отже, обираємо відповідне $k \in \bigcap_{c \in f'(V')} I_c$ і розширяємо f', встановивши $f' := f' \cup \{(a, k)\}$. Також встановлюємо, що $V' := V' \cup \{a\}$. Таким чином, поступово розширюємо відображення f', додаючи щоразу по одній вершині, поки не покриємо усі вершини заданого графа G.

□

Приклад 2.2. Знайти інтервал можливих значень для вершини 2 даного графа G при $M = 2$.

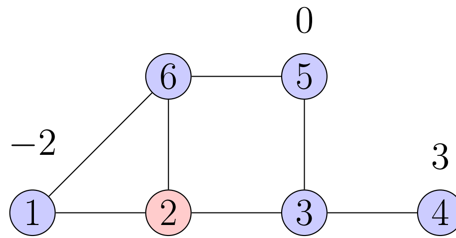


Рис. 8: Граф G

Граф G має три вершини з проставленими мітками. Знайдемо інтервал можливих значень для вершини 2. Як зазначено в доведенні Теореми 2.1, цей інтервал має вигляд: $\bigcap_{c \in f'(V')} I_c$, де c - вершини з мітками. У нашому випадку це вершини $\{1, 4, 5\}$. Маємо:

$$I_1 = [-2 - M, -2 + M] = [-4, 0],$$

$$I_4 = [3 - 2M, 3 + 2M] = [-1, 7],$$

$$I_5 = [0 - 2M, 0 + 2M] = [-4, 4].$$

Тоді $\bigcap_{c \in f'(V')} I_c = [-1, 0]$. Таким чином, вершина 2 може набувати значень з множини $\{-1, 0\}$. Інтервали для решти вершин обчислюються аналогічно.

2.4.1 Псевдокод зі статті [1]

Algorithm 2 Алгоритм для M-ParExt на загальних графах

Require: A graph G, a set of vertices $V' \subseteq V(G)$, and a partial M-Lipschitz mapping $f' : V' \to \mathbb{Z}$.

1: Compute all-pairs distances.
2: **if** f' rooted **then**
3: Set $f'' := f'$ and go to line 8.
4: **end if**
5: **if** f' not rooted **then**
6: **for** every $v' \in V \setminus V'$ **do**
7: Set $f'' := f' \cup \{(v', 0)\}$.
8: **if** f'' is M-reachable **then**
9: **while** some vertex not mapped under f'' **do**
10: Pick a non-mapped vertex a adjacent to some already mapped vertex. Choose some $k \in \bigcap_{c \in f''(V'')} I_c$ with c's and I_c's defined analogously as in the proof of Theorem 7.
11: Set $f'' := f'' \cup (a, k)$.
12: **end while**
13: **return True**
14: **end if**
15: **end for**
16: **end if**
17: **return False**

2.4.2 Опис основних кроків алгоритму

1. Обчислити відстані між усіма парами вершин.

2. Перевірити, чи f' має корінь. Якщо так, перейти до пункту 4, щоб розставити мітки.

3. Якщо f' кореня не має, то проходимося по решті вершин та способом спроб і помилок підбираємо вершину-корінь, щоб відображення $f'' := f' \cup (root, 0)$ було M-досяжним.

4. Розставляємо мітки. Для цього беремо непромарковану вершину та, знайшовши перетин інтервалів як зазначено в Теоремі 1.2, випадковим чином обираємо з нього число. Повторюємо, поки не закінчаться вершини без міток.

Аналогічно реалізовується Strong M-ParExt на загальних графах. Головна відмінність від даного алгоритму полягає у використанні множин можливих значень S для вершин при перевірці на "сильну" M-досяжність. Саме через фіксовану відстань між образами вершин ми не можемо спиратися на умову $|f(u) - f(v)| \leq M \cdot d(u, v)$. Для перевірки "сумісності" сусідніх вершин u та v достатньо перевірити, чи $f(v) \in \{f(u) \pm M\}$. Для розширення цього правила й на несуміжні вершини, ми пропонуємо скористатись множиною S, кожен елемент $s \in S$ якої переходить у $\{f(s) \pm M\}$ на кожному кроці, повторюючи цю ітерацію $d(u, v)$ разів (починаємо з u, $S = \{f(u)\}$). Цю ж логіку використовуємо і при проставленні міток.

2.5 Порівняння M-ParExt на деревах та загальних графах

Переглянувши алгоритми для обох видів графів, може виникнути питання, чому алгоритм для дерев не підходить для загальних графів?

Під час обходу вершин у дереві способом DFS (пошуком у глибину) маємо наступну ситуацію. Уже побувавши у певній вершині, ми впевнені, що інших шляхів до неї немає. Це тому, що дерево – це ациклічний граф, і кожна вершина має тільки одну батьківську вершину, окрім кореня. А отже ми не пропустили жодних зв'язків між вершинами, які б могли вплинути на інтервал можливих значень для даної вершини.

Розглядаючи ж загальні графи, може трапитись, що до вершини ведуть декілька шляхів (Рис. 9). Зайшовши до неї один раз, ми більше не зможемо оновити інтервал і, відповідно, врахувати вплив від інших вершин, які є суміжними з даною.

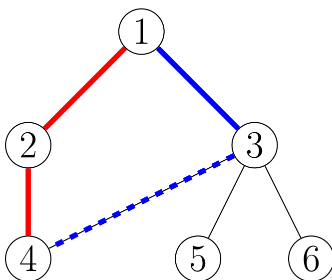


Рис. 9: Вершина 4 вже була відвідана (червоний шлях), тому ребро 34 не може бути враховано

Відповідно, **Алгоритм 1** не є універсальним для загальних графів - він може не врахувати всіх зв'язків між вершинами.

Чи можна застосувати **Алгоритм 2** для дерев? Звісно, але тоді часова складність збільшується з $O(|V|^2)$ до $O(|V|^3)$.

2.6 M-ParExt на графах блоків

2.6.1 Мотивація

Дослідивши та запрограмувавши алгоритми M-ParExt на деревах та загальних графах, у нас виникло питання: чи можливо їх модифікувати для графів блоків так, щоб складність була меншою за $O(|V|^3)$ - як для загальних графів?

Клас загальних графів включає в себе графи блоків, а клас графів блоків, у свою чергу, включає дерева. Крім того, графи блоків мають цікаву структуру, де кожен блок є повним підграфом. Тому власне і виникла ідея спробувати зменшити складність алгоритму на загальних графах за рахунок подібності до дерев або навпаки - доповнити алгоритм на деревах, щоб він працював і на графах блоків.

Отже, у наступному пункті наведено ще один із результатів цієї роботи - власний алгоритм M-ParExt на графах блоків.

Також важливо зазначити, що для графів блоків не існує сильного M-Ліпшицевого відображення, оскільки вони не є двочастковими графами. Єдине виключення – дерева, але вони вже мають окремий алгоритм.

2.6.2 Опис алгоритму

Algorithm 3 Алгоритм M-ParExt на графах блоків

Дано: Граф блоків G, підмножина вершин $V' \subseteq V(G)$, та часткове M-Ліпшицеве відображення $f' : V' \to \mathbb{Z}$.
Знайти: M-Ліпшицеве відображення f графу G таке, що $f' \subseteq f$.

1. Створюємо **block cutpoint tree** T на основі даного графа G.
   Для цього визначаємо список блоків і точок з'єднання. Кожен блок з G переходить у вершину, а кожна точка з'єднання - у вершину, що з'єднує відповідні блоки.

2. Для кожного блоку визначаємо **інтервал** P можливих значень у вигляді: $[max-M, min+M]$, де max та min є найбільшим та найменшим значенням наявних міток в даному блоці, відповідно.
   Окремо зберігаємо список блоків зі скінченним інтервалом. Запишемо їхню кількість як k.
   Якщо існує блок з порожнім інтервалом, то f не існує.

3. Обходимо дерево k разів, щоразу починаючи з блоку з визначеним у (2) інтервалом, оновлюючи інтервали для кожної вершини.
   Використовуємо пошук в глибину з алгоритму для дерев, пропускаючи при цьому точки з'єднання.
   Якщо існує блок з порожнім інтервалом, то f не існує.

4. Визначаємо **інтервали для точок з'єднання** як перетин інтервалів блоків, з якими вони суміжні.
   Якщо існує точка з'єднання з порожнім інтервалом, то f не існує.

5. Знаходимо блок, що може мати в собі **корінь**. Це блок, чий інтервал містить 0. Розставляємо мітки всередині даного блоку.
   Ще раз оновлюємо інтервали, починаючи обхід дерева з цього блоку.
   Якщо кандидата для кореня не знайдено, то f не існує.

6. Наостанок, обходимо решту блоків дерева T, проставлячи мітки вершинам в кожному з них.
   Завершивши цю процедуру, отримаємо шукане відображення f.

Запропонований алгоритм справді базується на алгоритмі M-ParExt для дерев. Як зазначено в пункті (1), ми створюємо блокове дерево з точками зв'язності T і застосовуємо варіацію алгоритму для дерев на ньому, обходячи лише вершини-блоки при оновленні інтервалів.

На етапі присвоєння міток ми заповнюємо один блок за раз, використовуючи ефективний спосіб знаходження інтервалу для кожної вершини. На відміну від загальних графів, блоки є повними графами та, відповідно, не потребують знаходження відстані між вершинами: вони усі рівні 1. Тому для присвоєння мітки певній вершині всередині блоку достатньо буде знайти перетин інтервалу для даного блоку та для цієї вершини: $I = [max_value - M, min_value + M]$, де max_value та min_value є найбільшим та найменшим значенням наявних міток в даному блоці, відповідно. Якщо дана вершина є ще й точкою зв'язності, то враховуємо також її інтервал.

2.7 Складність алгоритмів

Теорема 2.3. [1] *Проблема M-ParExt для дерев має розв'язок у часі* $O(|V|^2)$ *і просторі* $O(|V|)$.

Доведення. Ми виконуємо $O(|V|)$ разів пошук в глибину на графі G, а також здійснюємо константну кількість лінійних обходів структури даних для G. Оскільки G – дерево, загальна часова складність алгоритму становить $O(|V|) \cdot O(|V|-1) = O(|V|^2)$. Щодо просторової складності, вона обмежена $O(|V|)$, оскільки ми зберігаємо лише обмежену кількість інформації для кожної вершини. □

Теорема 2.4. [1] *Проблема M-ParExt є розв'язною за поліноміальний час для загальних графів. Існує алгоритм з часом виконання* $O(|V|^3)$ *та простором* $O(|V|^2)$*. Крім того, якщо екземпляр проблеми M-ParExt має корінь, ми маємо алгоритм, який працює за часом* $O(|V||E|)$ *та простором* $O(|V|^2)$*.*

Доведення. У псевдокоді рядок 1 обчислює відстані між всіма парами вершин за час $\mathrm{O}(|V||E|)$. Рядок 8 вимагає часу $\mathrm{O}(|V|^2)$, і код між рядками 10 і 12 також. Якщо f' має корінь, ми відразу переходимо до циклу for, заощаджуючи фактор $\mathrm{O}(|V|)$, інакше отримуємо додатковий фактор $\mathrm{O}(|V|)$. □

Для Strong M-ParExt маємо ті самі результати по складності. Алгоритм M-ParExt на графах блоків використовує алгоритм на деревах при роботі з блоко-точковим деревом і, водночас, є спрощеною версією алгоритму на загальних графах, адже немає потреби обчислювати відстані між усіма вершинами, а всередині кожного блоку мітки присвоюються дуже ефективно. Тому він оцінюється як складніший алгоритм, ніж на деревах, але простіший, ніж на загальних графах.

2.8 Строгі М-Ліпшицеві відображення

Досліджуючи M-Ліпшицеві відображення, ми натрапили на подібну умову, яка так само контролює "близькість" образів суміжних вершин, але при цьому ще й забезпечує більшу відстань між образами несуміжних вершин! Ця умова є дуже сильною, і далеко не кожен граф допускає такі відображення. Тому ми вирішили назвати їх *строгими M-Ліпшицевими відображеннями*. Далі розглянемо їх детальніше.

Означення 2.5. Граф $G = (V, E)$ називають *індиферентним*, якщо існує додатне число δ (що вимірює "близькість" або "індиферентність") і відображення чисел $f(u)$ для елементів V, таке що для усіх $u, v \in V$, ребро uv є в G тоді й тільки тоді коли $|f(u) - f(v)| \leq \delta$.

Індиферентні графи мають різні характеризації. Розглянемо їх, перед цим ознайомившись з декількома поняттями.

Означення 2.6. Граф $G = (V, E)$ називають *графом інтервалів*, якщо існує сімейство $\{I_i\}$ $(1 < i < n)$ інтервалів на вісі дійсних чисел таке, що різні вершини u, v є суміжними тоді і тільки тоді, коли відповідні інтервали перетинаються. Таке сімейство $\{I_i\}$ інтервалів зазвичай називають інтервальним представленням графа G.

Означення 2.7. Граф G називається *одиничним графом інтервалів*, якщо всі інтервали у представленні мають одиничну довжину.

Означення 2.8. Граф G називається *правильним графом інтервалів*, якщо жоден інтервал у представленні не є власною підмножиною жодного іншого інтервалу.

Означення 2.9. *Клешнею* називають повний двочастковий граф $K_{1,3}$.

Твердження 2.10. [5] *Для графа G наступні твердження є еквівалентними:*

(i) G є одиничним графом інтервалів;

(ii) G є правильним графом інтервалів;

(iii) G є графом інтервалів без породженої клешні;

(iv) G є індиферентним графом.

Теорема 2.11. [5] *Граф $G = (V, E)$ є індиферентним графом тоді й тільки тоді, коли існує лінійний порядок $<$ на V такий, що для кожної трійки вершин u, v, w таких, що*

$$u < v < w, uw \in E, \text{ виконано } uv, vw \in E.$$

Оскільки графи інтервалів допускають таку сильну умову, описану в Означенні 2.5, та мають лінійне впорядкування вершин, на них гарно розв'язується цілий ряд проблем, серед яких є розфарбування вершин, знаходження найкоротшого шляху між вершинами, знаходження центру графа і Гамільтонового шляху. Оптимальні алгоритми для перелічених проблем описуються у статті [5].

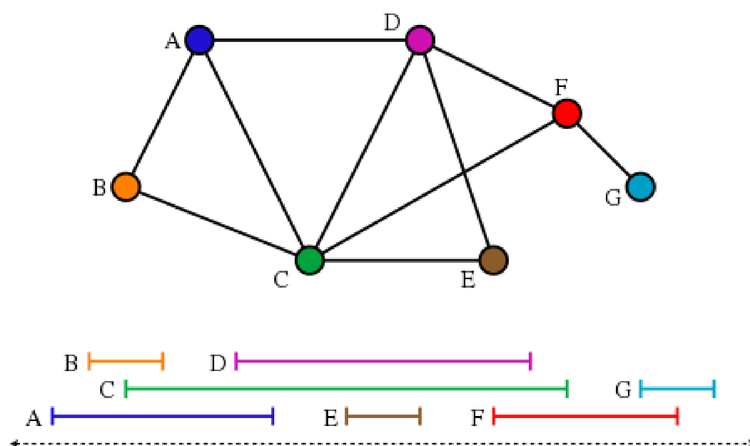


Рис. 10: Приклад графа інтервалів, який не є індиферентним графом (бо існують вкладені інтервали)

3 Програмна реалізація алгоритмів на Python

3.1 M-ParExt на деревах

Усі функції названі згідно з позначеннями у відповідних псевдокодах.

- Імпортуємо необхідні модулі Python.

```
import networkx as nx
import matplotlib.pyplot as plt
import random
from collections import deque
import scipy
import numpy as np
```

- Визначаємо функцію **draw_graph()**, яка зображує даний граф G, використовуючи бібліотеку networkx. Помаранчевим позначені мар- ковані вершини, усередині кожної вершини позначений її індекс, а зверху жирним - мітка, якщо вона була надана.

```
def draw_graph(G, V_prime, f, pos=None):
    node_colors = ['skyblue' if node not in V_prime else 'orange'
for node in G.nodes]
    nx.draw_networkx_nodes(G, pos=pos, node_size=700, node_color=
node_colors)
    nx.draw_networkx_labels(G, pos=pos, labels={node: str(node)
for node in G.nodes}, font_size=8)
    nx.draw_networkx_edges(G, pos=pos)

    for node, (x, y) in pos.items():
        if node in f:
          plt.text(x, y + 0.1, f[node], fontsize=12, fontweight='
bold', ha='center')

    plt.axis('off')
    plt.show()
```

- Визначаємо функцію **check_lipschitz_condition()**, яка перевіряє промаркований підграф на умову Ліпшиця.

```
def check_lipschitz_condition(G, V_prime, f_prime, M):
    for u, v in G.edges():
        if u in V_prime and v in V_prime:
            if abs(f_prime[v] - f_prime[u]) > M:
                print("The mapping f' cannot be extended.")
                return False
    return True

```

- Визначаємо функцію **dfs()**, яка оновлює інтервали для кожної вершини

```
def dfs(G, start_vertex, P):

    dfs_stack = [start_vertex]
    visited = set()

    while dfs_stack:
        current_vertex = dfs_stack.pop()

        if current_vertex in visited:
         continue
        visited.add(current_vertex)

        for neighbor in G.neighbors(current_vertex):
            P[neighbor] = [max(P[current_vertex][0]-M, P[neighbor
][0]),
            min(P[current_vertex][1]+M, P[neighbor][1])]
            dfs_stack.append(neighbor)
```

- Визначаємо функцію **bfs()**, яка для кожної вершини проставляє правильні мітки: варіантів може бути декілька, тоді число обирається рандомним чином з дозволеного інтервалу.

```
def bfs(G, root, f, M, P):
    visited = set()
    queue = deque([(root, None)])
    visited.add(root)

    while queue:
        current_vertex, parent = queue.popleft()
        if not current_vertex in f:
            f[current_vertex] = random.randint(
            max(P[current_vertex][0], f[parent]-M),
            min(P[current_vertex][1], f[parent]+M))
        for neighbor in G[current_vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, current_vertex))
    return f, visited
```

- Визначаємо функцію **find_root()**, що повертає вершину, яку обрано за корінь, тобто інтервал якої містить 0.

```
def find_root(G, P):
    root = None
    for node in G.nodes:
        if P[node][0] <= 0 <= P[node][1]:
            root = node
            break
    return root
```

- Визначаємо основну функцію **m_par_ext_on_trees()**. Вона повертає M-ліпшицеве відображення f, яке розширює f', на дереві G.

```
def m_par_ext_on_trees(G, V_prime, f_prime, M):

    pos = nx.spring_layout(G)
    draw_graph(G, V_prime, f_prime, pos=pos)

    if not check_lipschitz_condition(G, V_prime, f_prime, M):
     return False

    # set initial intervals for the vertices
    P = {v: [f_prime[v], f_prime[v]] if v in V_prime else [-float(
'inf'), float('inf')] for v in G.nodes}

    # update intevals
    for v_prime in V_prime:
     dfs(G, v_prime, P, M)

    # find root and update intervals again
    root = find_root(G, P)
    if root is None:
        print("No such r exists. The mapping f' cannot be extended
.")
        return False

    f = f_prime
    f[root] = 0
    P[root] = [0, 0]
    dfs(G, root, P, M)

    # if the is an empty interval, f doesn't exist
    for node in G.nodes:
        if P[node][0] > P[node][1]:
            print("P[{node}]=empty set. The mapping f' cannot be
extended.")
            return False

    # assign labels to other vertices
    f, visited = bfs(G, root, f, M, P)

    if len(visited) != len(G.nodes):
        print("BFS could not be completed. The mapping f' cannot
be extended.")
        return False

    draw_graph(G, V_prime, f, pos=pos)
    return True

```

3.2 Strong M-ParExt на деревах

Даний алгоритм має за основу M-ParExt на деревах. Тому нижче буде наведено лише функції, які було модифіковано, щоб задовольнити сильну M-Ліпшицевість. Головна відмінність полягає у тому, що замість **інтервалів** P ми використовуємо **множини** S для визначення можливих міток для усіх вершин.

- Визначаємо функцію **strong_dfs()**, яка оновлює множини для кожної вершини.

```python
def strong_dfs(G, start_vertex, S, M):
    print(f"dfs from {start_vertex}")
    dfs_stack = [start_vertex]
    visited = set()

    while dfs_stack:
        current_vertex = dfs_stack.pop()

        if current_vertex in visited:
         continue
        visited.add(current_vertex)

        temp_neighbor = {v + M for v in S[current_vertex]}
        temp_neighbor.add(min(temp_neighbor) - 2 * M)
        for neighbor in G.neighbors(current_vertex):

            if S[neighbor]:
             S[neighbor] = S[neighbor].intersection(temp_neighbor)
            else:
             S[neighbor] = temp_neighbor

            if not S[neighbor]:
                print(" The mapping f' cannot be extended")
                return None

            dfs_stack.append(neighbor)

```

- Визначаємо функцію **find_root()**, яка знаходить вершину, що може стати коренем.

```python
def find_root(S):
    for v, values in S.items():
        if 0 in values:
         return v
    return None

```

- Визначаємо функцію **strong_bfs()**, яка для кожної вершини проставляє мітки. Якщо варіантів декілька, то число обирається випадковим чином.

```
def strong_bfs(G, root, f, M, S):
    visited = set()
    queue = deque([(root, None)])
    visited.add(root)

    while queue:
        current_vertex, parent = queue.popleft()

        if current_vertex != root:
            if not current_vertex in f:
                parent_label = next(iter(S[parent]))
                current = S[current_vertex].intersection({
parent_label- M, parent_label + M})
                f[current_vertex] = random.choice(list(current))

        for neighbor in G[current_vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, current_vertex))
    return f, visited

```

- Визначаємо основну функцію **strong_m_par_ext_on_trees()**, яка повертає сильне M-ліпшицеве відображення f, яке розширює f', на дереві G.

```
def strong_m_par_ext_on_trees(G, V_prime, f_prime, M):

    pos = nx.spring_layout(G)
    draw_graph(G, V_prime, f_prime, pos=pos)

    if any(value % M != 0 for value in f_prime.values()):
        print(f"Not all labels are divisible by M = {M}")
        return False

    if not check_lipschitz_condition(G, V_prime, f_prime, M):
     return False

    # assign sets for vertices
    S = {v: {f_prime[v]} if v in V_prime else {} for v in G.nodes}

    # update sets
    for v_prime in V_prime:
     strong_dfs(G, v_prime, S, M)

    # find root and update sets again
    root = strong_find_root(S)
    if root is None:
```

```
23             print("No such r exists. The mapping f' cannot be extended
    .")
24             return False
25 
26         f = f_prime
27         f[root] = 0
28         S[root] = {0}
29         strong_dfs(G, root, S, M)
30 
31         # if empty set exists, f' cannot be extended
32         for node in G.nodes:
33             if not S[node]:
34                 print("S[{node}]=empty set. The mapping f' cannot be
    extended.")
35                 return False
36 
37         # assign labels for other vertices
38         f = strong_bfs(G, root, f, M, S)
39 
40         draw_graph(G, V_prime, f, pos=pos)
41 
42         return True
43 
```

3.3 M-ParExt та Strong M-ParExt на загальних графах

- Визначаємо функцію **create_sorted_distance_matrix()**, яка повертає матрицю відстаней між вершинами графа G.

```
1 def create_sorted_distance_matrix(G):
2     nodes = sorted(G.nodes())
3     node_to_index = {node: idx for idx, node in enumerate(nodes)}
4 
5     dist_matrix = nx.floyd_warshall_numpy(G, nodelist=nodes)
6     sorted_dist_matrix = dist_matrix[[node_to_index[node] for node
   in nodes]][:, [node_to_index[node] for node in nodes]]
7     sorted_dist_matrix = sorted_dist_matrix.astype(int)
8     return sorted_dist_matrix
9 
```

- Визначаємо функцію **generate_set()**, яка визначає множину можливих значень для вершини, що віддалена від даної вершини v на задану кількість ребер. Використовується лише для Strong M-ParExt.

```
1 def generate_set(v, M, distance):
2     S = {v}
3     for _ in range(distance):
4         new_S = set()
5         for x in S:
6             new_S.add(x - M)
7             new_S.add(x + M)
8         S = new_S
9 
10     return S
11 
```

- Визначаємо функцію **is_M_reachable()**, яка перевіряє заданий набір промаркованих вершин на M-досяжність.

```
1 def is_M_reachable(f, M, all_pairs_distances, strong):
2     pairs = itertools.combinations(f.keys(), 2)
3     if not strong:
4         for u, v in pairs:
5             if abs(f[u] - f[v]) > M * all_pairs_distances[u-1][v
  -1]:
6                  return False
7     else:
8         for u, v in pairs:
9             S = generate_set(f[u], M, all_pairs_distances[u-1][v
  -1])
10                 if not f[v] in S:
11                  return False
12     return True
13 
```

- Визначаємо функції **rooted()** та **pick_non_mapped_vertex()**, які перевіряють, чи відображення f має корінь та повертають ще не промарковану вершину, відповідно.

```
def rooted(f):
 return 0 in f.values()

def pick_non_mapped_vertex(first_list, second_list):
 return list(set(second_list) - set(first_list))[0]

```

- Визначаємо функцію **choose_k()**, яка заданій вершині a повертає мітку k так, щоб для усіх промаркованих вершин зберігалась M-досяжність. У варіанті Strong M-ParExt використовуємо множини.

```
def choose_k(f, a, M, all_pairs_distances, strong):
    if not strong:
        lower_bound, upper_bound = -float('inf'), float('inf')
        for vertex in f:
            v = f[vertex]
            if v-all_pairs_distances[vertex-1][a-1]*M >
lower_bound:
             lower_bound = v - all_pairs_distances[vertex-1][a-1]*
M

            if v+all_pairs_distances[vertex-1][a-1]*M <
upper_bound:
             upper_bound = v + all_pairs_distances[vertex-1][a-1]*
M
        res = random.randint(lower_bound, upper_bound)
    else:
        S = set()
        for v in f:
            distance_to_a = all_pairs_distances[v-1][a-1]
            new_S = generate_set(f[v], M, distance_to_a)

            if not S:
             S = new_S
            S = S.intersection(new_S)
            if not S:
             return None
        res = random.choice(list(S))
    return res

```

- Визначаємо функцію **map_all_vertices()**, яка усім немаркованим вершинам ставить у відповідність цілі числа, зберігаючи при цьому M-досяжність відображення f. Дана функція викликається на етапі,

коли вже знайдений корінь і відомо, що відображення f існує.

```
def map_all_vertices(f, G, M, all_pairs_distances, strong):
    while set(list(f.keys())) != set(G.nodes()):
        a = pick_non_mapped_vertex(list(f.keys()), G.nodes())
        k = choose_k(f, a, M, all_pairs_distances, strong)
        f[a] = k
    return f

```

- Визначаємо основну функцію **M_ParExt()**, яка розширює (сильне) M-Ліпшицеве відображення f' на весь граф G.

```
def M_ParExt(G, V_prime, f_prime, M, strong=False):
    all_pairs_distances = create_sorted_distance_matrix(G)

    f_double_prime = f_prime.copy()
    if rooted(f_prime):
        if is_M_reachable(f_double_prime, M, all_pairs_distances,
strong):
            f_double_prime = map_all_vertices(f_double_prime, G, M
, all_pairs_distances, strong)
        else:
            print("f' is rooted but not M-reachable")
            return False

    else:
        for v_prime in (set(G.nodes()) - set(V_prime)):
            f_double_prime = f_prime.copy()
            f_double_prime[v_prime] = 0

        if is_M_reachable(f_double_prime,M,all_pairs_distances,
strong):
            f_double_prime = map_all_vertices(f_double_prime, G, M
, all_pairs_distances, strong)
            break

    pos = nx.spring_layout(G)
    draw_graph(G, V_prime, f_prime, f_double_prime, pos=pos)

    return True

```

3.4 M-ParExt на графах блоків

- Визначаємо функцію **clique_interval()**, що для заданого блоку знаходить інтервал можливих значень. Також перевіряється, чи для кожного ребра (u, v) в блоці виконується умова $|f(u) - f(v)| \leq M$, якщо $f(u), f(v)$ відомі.

```
def clique_interval(f, block, M):
    labels = [f.get(node) for node in block if node in f]
    if labels:
        if max(labels) - min(labels) > M:
         return None
        return [max(labels) - M, min(labels) + M]
    return [-float('inf'), float('inf')]
```

- Визначаємо функцію **create_block_cutpoint_tree()**, яка створює блоко-точкове дерево T на основі даного графа блоків G. Для цього за допомогою вбудованих функцій знаходиться перелік блоків графа G та його точок з'єднання.

```
def create_block_cutpoint_tree(G):
    blocks = list(nx.biconnected_components(G))
    blocks = {i: lst for i, lst in enumerate(blocks)}
    cutpoints = list(nx.articulation_points(G))
    T = nx.Graph()

    # add cutpoint vertices
    for cutpoint_node in cutpoints:
     T.add_node(f'{cutpoint_node}', type='Cutpoint')

    # add block vertices
    for i, block_nodes in blocks.items():
        T.add_node(f'B{i}', type='Block')
        # add edges
        for cutpoint_node in cutpoints:
            if cutpoint_node in block_nodes:
             T.add_edge(f'B{i}', f'{cutpoint_node}')
    return T, blocks, cutpoints
```

- Визначаємо функцію **set_blocks_intervals()**, яка оновлює інтервали блоків графа T, обходячи вершини пошуком в глибину. Дана функція подібна до **bfs()**, яка була написана вище в алгоритмі для дерев. Головна відмінність полягає у тому, що вершини, які є точками

зв'язності, 'оминаються', адже нас цікавлять лише блоки на даному етапі. Тому нижче в коді можна побачити подвійний цикл **for** для пошуку сусідів сусідів - тобто блоків.

```
def set_blocks_intervals(T, P, start_vertex, M):
    dfs_stack = [start_vertex]
    visited = set()
    dfs_order = []

    while dfs_stack:
        current_vertex = dfs_stack.pop()
        if current_vertex not in visited:
            dfs_order.append(current_vertex)
            visited.add(current_vertex)

            for cutpoint_neighbor in T.neighbors(current_vertex):
                for block_neighbor in T.neighbors(
cutpoint_neighbor):

                    if block_neighbor not in visited:
                        neighbor_index = int(block_neighbor[1])
                        current_index = int(current_vertex[1])
                        P[neighbor_index] = [
                        max(P[current_index][0] - M, P[
neighbor_index][0]),
                        min(P[current_index][1] + M, P[
neighbor_index][1])]

                dfs_stack.append(block_neighbor)
    return dfs_order
```

- Визначаємо функцію **set_cutpoint_intervals()**, яка визначає інтервали для точок з'єднання в блоко-точковому дереві T. Це відбувається після того, як інтервали для блоків вже знайдені. Інтервал для даної точки визначається як перетин інтервалів усіх суміжних з нею блоків.

```
def set_cutpoint_intervals(T, cutpoints, P, f):
    cutpoint_intervals = {}
    for cutpoint_node in cutpoints:
        if cutpoint_node in f:
            cutpoint_intervals[cutpoint_node] = [f[cutpoint_node],
 f[cutpoint_node]]
            continue
        neighbor_blocks = [neighbor for neighbor in T.neighbors(f"
{cutpoint_node}")]

        interval = [-float('inf'), float('inf')]
        for block in neighbor_blocks:
            block_index = int(block[1])
            interval = [max(interval[0], P[block_index][0]),
            min(interval[1], P[block_index][1])]
```

```
            cutpoint_intervals[cutpoint_node] = interval
        return cutpoint_intervals
```

- Визначаємо функцію **find_block_with_root()**, яка шукає блоки-кандидати на вміст кореня і випадковим чином повертає один із них. Для цього береться список блоків, чиї інтервали містять 0. Далі перевіряється, чи існує вершина всередині блоку, яка могла б отримати мітку 0. Може бути, що блок повністю заповнений з самого початку і 0 там немає, або що єдиними кандидатами під корінь є точки з'єднання, в чиї інтевали 0 не входить: у таких випадках цей блок не розглядається.

```
def find_block_with_root(P, blocks, cutpoints_intervals, f):
    blocks_to_check = list(key for key, interval in P.items() if
interval[0] <= 0 <= interval[1])
    rooted_blocks = set()

    if not blocks_to_check:
            return None

    for i in blocks_to_check:
        block = blocks[i]
        # if block already has a vertex with label 0
        if any(f.get(element, None) == 0 for element in block):
         rooted_blocks.add(i)

        for v in block:
            if not v in f:
            # if it's a cutpoint, check if 0 is in its interval
            if v in cutpoints_intervals:
                if cutpoints_intervals[v][0] <= 0 <=
cutpoints_intervals[v][1]:
                 rooted_blocks.add(i)
            else:
             rooted_blocks.add(i)

    if not rooted_blocks:
     return None
    return random.choice(list(rooted_blocks))
```

- Визначаємо функцію **map_block_vertices()**, яка розставляє мітки всередині даного блоку. Враховується, чи в цьому блоці має бути корінь, який у нього інтервал, а також інтервали усіх його точок з'єднання.

```python
def map_block_vertices(block, cutpoints_intervals, M, block_index, P, f, rooted=False):
    max_value = max((f[element] for element in block if element in f), default=-float('inf'))
    min_value = min((f[element] for element in block if element in f), default=+float('inf'))
    if rooted:
        for v in block:
            # if root already exists
            if f.get(v) == 0:
             break
            # if current vertex is not a cutpoint
            if not v in f and not v in cutpoints_intervals:
                f[v] = 0
                P[block_index] = [max(-M, P[block_index][0]), min(M, P[block_index][1])]
                break
            # if it's a cutpoint, check if 0 is in its interval
            if not v in f and v in cutpoints_intervals:
                if max(cutpoints_intervals[v][0], max_value - M, P[block_index][0]) <= 0 \
                <= min(cutpoints_intervals[v][1], min_value + M, P[block_index][1]):
                    f[v] = 0
                    P[block_index] = [max(-M, P[block_index][0]), min(M, P[block_index][1])]
                    break

    for c in cutpoints_intervals:
        if not c in f and c in block:
            f[c] = random.randint(
            max(cutpoints_intervals[c][0], max_value-M, P[block_index][0]),
            min(cutpoints_intervals[c][1], min_value+M, P[block_index][1]))
            cutpoints_intervals[c] = [f[c], f[c]]
            max_value = max(max_value, f[c])
            min_value = min(min_value, f[c])

    for v in block:
        if not v in f:
            f[v] = random.randint(max(max_value - M, P[block_index][0]),
            min(min_value + M, P[block_index][1]))
            max_value = max(max_value, f[v])
            min_value = min(min_value, f[v])
    P[block_index] = [min_value, max_value]
    return f

```

- Визначаємо основну функцію **M_ParExt_block_graphs()**, яка повертає М-ліпшицеве відображення f, розширюючи f', на графі блоків G. Як було зазначено в **Алгоритмі 3**, основні кроки полягають у зна-

ходженні блоко-точкового дерева T, проставленні інтервалів вершинам, знаходженні блока-кандидата для кореня та власне проставлення міток.

```
def M_ParExt_block_graphs(G, f_prime, M):
# Step 1 from Algorithm 3
T, blocks, cutpoints = create_block_cutpoint_tree(G)

# Step 2
P = {i: clique_interval(f_prime, block, M) for i, block in blocks.items()}
if any(interval is None for interval in P.values()):
    print("The mapping f' cannot be extended.")
    return None
labeled_blocks = [index for index, interval in P.items() if not any(val in (float('-inf'), float('inf')) for val in interval)]

# Step 3
for block in labeled_blocks:
 set_blocks_intervals(T, P, 'B' + str(block), M)
if any(interval[0] > interval[1] for interval in P.values()):
    print("There is a block with empty interval. The mapping f' cannot be extended.")
    return None

# Step 4
cutpoints_intervals = set_cutpoint_intervals(T, cutpoints, P, f_prime)
if any(interval[0] > interval[1] for interval in cutpoints_intervals.values()):
    print("There is a cutpoint with empty interval. The mapping f' cannot be extended.")
    return None

# Step 5
rooted_block=find_block_with_root(P,blocks,cutpoints_intervals, f_prime)
if rooted_block is None:
    print("No candidate for root was found. The mapping f' cannot be extended.")
    return None

f = f_prime.copy()
f=map_block_vertices(blocks[rooted_block], cutpoints_intervals, M, rooted_block, P, f, rooted=True)
dfs_order_from_root = set_blocks_intervals(T, P, 'B' + str(rooted_block), M)

# Step 6
for block_label in dfs_order_from_root:
    index = int(block_label[1])
    block = blocks[index]
    f = map_block_vertices(block, cutpoints_intervals, M, index, P, f)
return f
```

4 Застосування алгоритмів на прикладах

Приклад 4.1. Чи існує M-Ліпшицеве відображення f дерева G таке, що $f' \subseteq f$?

$V(G) = \{1, ..., 8\}, E(G) = \{12, 23, 26, 34, 37, 45, 82\}$

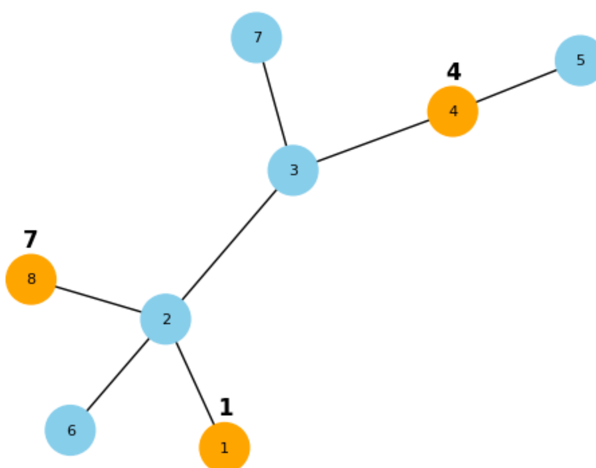
$V'(G) = \{1, 4, 8\}, f' = \{1 : 1, 4 : 4, 8 : 7\}, M = 3$

Розв'язок

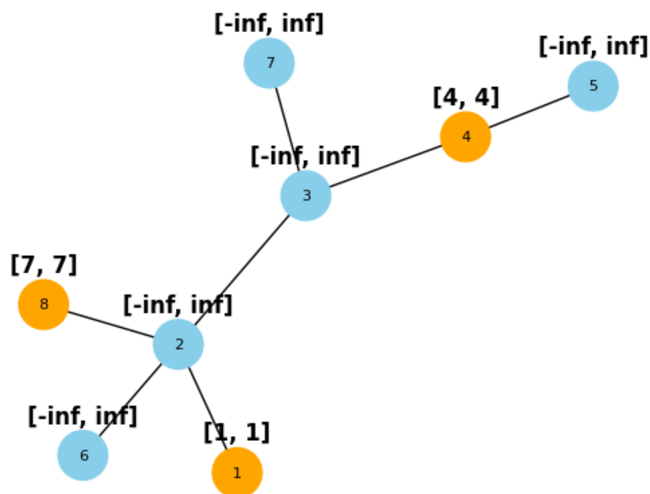
```
G = nx.Graph()
edges = [(1, 2), (2, 3), (2, 6), (3, 4), (3, 7), (4, 5), (8, 2)]
G.add_edges_from(edges)
V_prime = [1, 4, 8]
f_prime = {1: 1, 4: 4, 8: 7}
M = 3
m_par_ext_on_trees(G, V_prime, f_prime, M)

```

Нижче наведений опис головних кроків алгоритму для кращого розуміння, як він працює.

Отже, маємо таке дерево G, промарковані вершини позначені помаранчевим кольором:

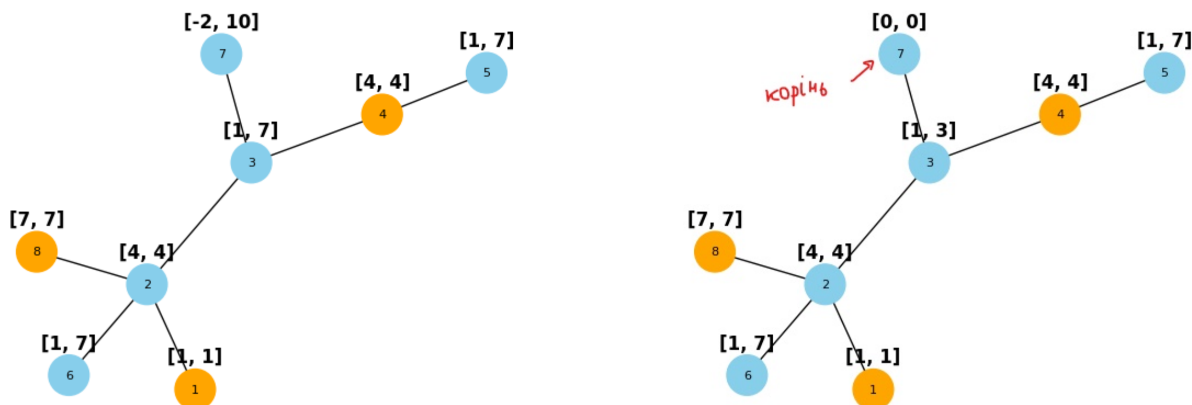


Згідно з алгоритмом, на початку кожній вершині присвоюємо інтервал за правилом: якщо вершина промаркована, то інтервал міститиме лише дане значення; для решти вершин інтервал матиме вигляд $(-\infty, +\infty)$:



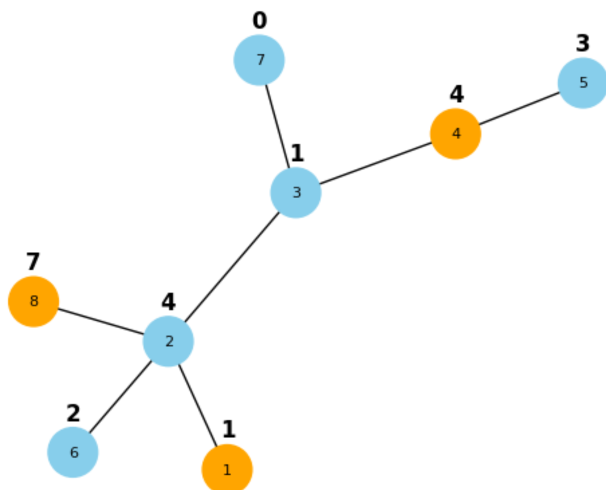
Далі оновлюємо інтервали для всіх вершин так, щоб вони містили всі можливі значення міток, при яких виконується умова M-Ліпщицевості на всьому графі. Для цього тричі обходимо весь граф, щоразу починаючи з нової промаркованої вершини (позначені помаранчевим) та присвоюючи інтервали наступним чином: інтервал на даній вершині розширюємо на M одиниць в обидва боки і знаходимо його перетин з інтервалом сусідньої вершини.

Позначаємо як корінь вершину, чий інтервал містить 0, і відповідно змінюємо її інтервал на $[0, 0]$. Після цієї зміни оновлюємо усі інтервали ще раз.



Маючи корінь і усі інтервали, можемо тепер присвоїти мітки усім вер-

шинам. Для цього випадковим чином вибираємо число з кожного з цих інтервалів і отримуємо результат:



Таким чином, М-Ліпшицеве відображення існує і воно подане на рисунку вище.

Приклад 4.2. Чи існує М-Ліпшицеве відображення f загального графа G таке, що $f' \subseteq f$?

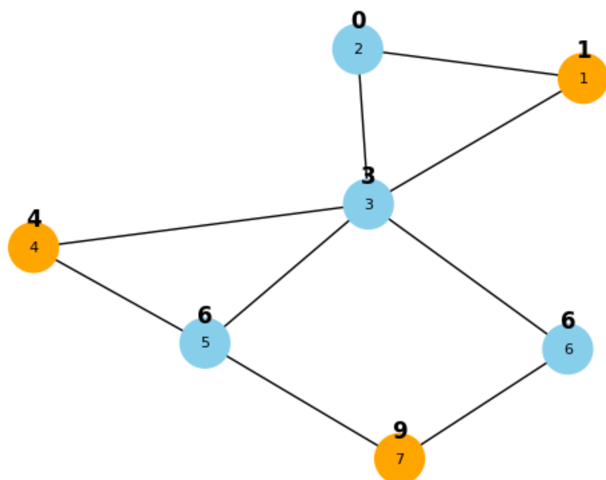
$V(G) = \{1, ..., 7\}$,

$E(G) = \{12, 13, 23, 34, 35, 36, 45, 57, 67\}$

$V'(G) = \{1, 4, 7\}, f' = \{1 : 1, 4 : 4, 7 : 9\}, M = 3$

Розв'язок

```
G = nx.Graph()
edges = [(1, 2), (1, 3), (2, 3), (3, 4), (3, 5), (3, 6), (4, 5), (5, 7)
, (6, 7)]
G.add_edges_from(edges)
V_prime = [1, 4, 7]
f_prime = {1: 1, 4: 4, 7: 9}
M = 3
M_ParExt(G, V_prime, f_prime, M)

```



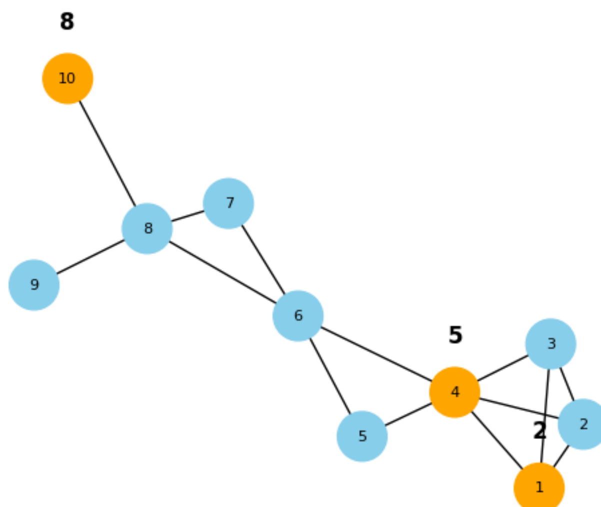
Як бачимо, M-Ліпшицеве відображення графа G існує, адже функція повернула нам повністю промаркований граф. Це значить, що корінь був знайдений і $f'' := f' \cup (root, 0)$ є M-reachable. Далі для кожної непромаркованої вершини знаходився інтервал можливих значень, одне з яких вибиралося випадковим чином і приписувалося поточній вершині.

Приклад 4.3. Чи існує M-Ліпшицеве відображення f графа блоків G таке, що $f' \subseteq f$?
$V(G) = \{1, ..., 10\}$,
$E(G) = \{12, 13, 14, 23, 24, 34, 45, 46, 56, 67, 68, 78, 89, (8, 10)\}$
$V'(G) = \{1, 4, 10\}, f' = \{1 : 2, 4 : 5, 7 : 8\}, M = 4$

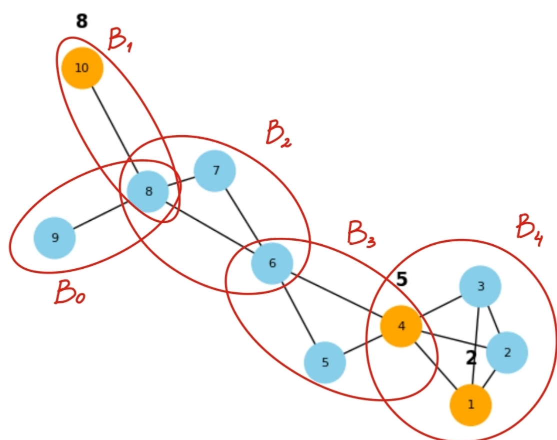
Розв'язок

```
G = nx.Graph()
edges = [(1,2), (1,3), (1,4), (2,3), (2,4), (3,4), (4,5), (4,6), (5,6),
  (6,7), (6,8), (7,8), (8,9), (8,10)]
G.add_edges_from(edges)
V_prime = [1, 4, 10]
f_prime = {1: 2, 4: 5, 10: 8}
M = 4
f = M_ParExt_block_graphs(G, f_prime, M)

```

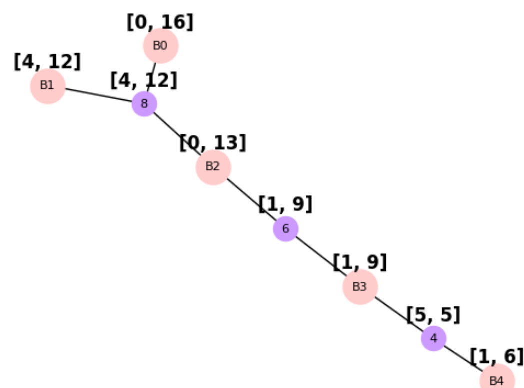
Маємо граф блоків G із проставленими мітками на трьох вершинах:



Згідно із запропонованим алгоритмом, створюється блоко-точкове дерево T, блоки якого зображені рожевими вершинами, а точки з'єднання - фіолетовими. Далі проставляються інтервали для блоків, а потім - точок з'єднання.

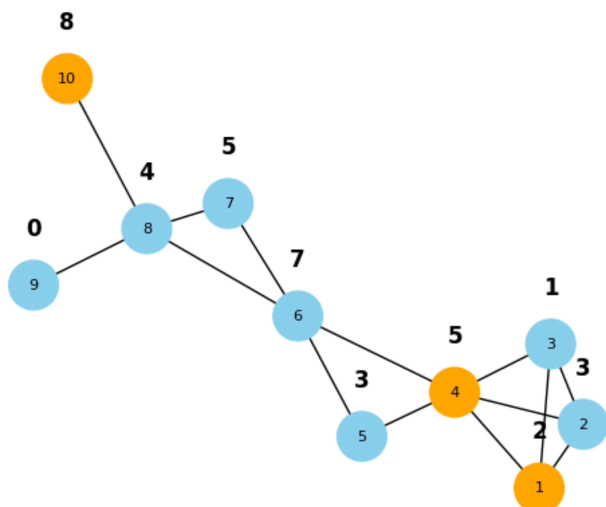


(а) Визначення блоків в G



(б) Блоко-точкове дерево T

Після цього знаходимо блок, який міг би містити в собі корінь. У даному випадку це B_0. Оновлюємо ще раз інтервали, починаючи обхід графа з цього блоку. І наостанок, проходимося по решті блоків, поступово маркуючи усі вершини всередині. У результаті отримуємо M-Ліпшицеве відображення f, яке на графі блоків G виглядає наступним чином:



Приклад 4.4. Чи існує сильне M-Ліпшицеве відображення f дерева G таке, що $f' \subseteq f$?

$V(G) = \{1, ..., 7\}$,

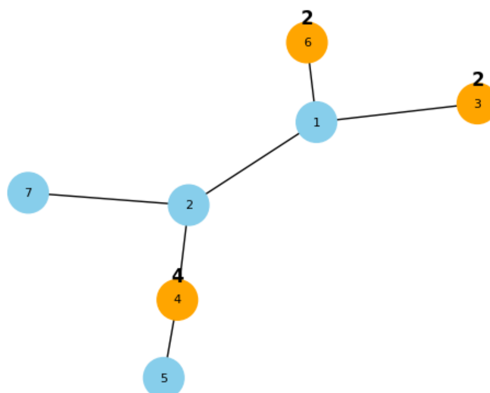
$E(G) = \{12, 13, 45, 16, 24, 27\}$

$V'(G) = \{3, 4, 6\}, f' = \{3 : 2, 4 : 4, 6 : 2\}, M = 2$

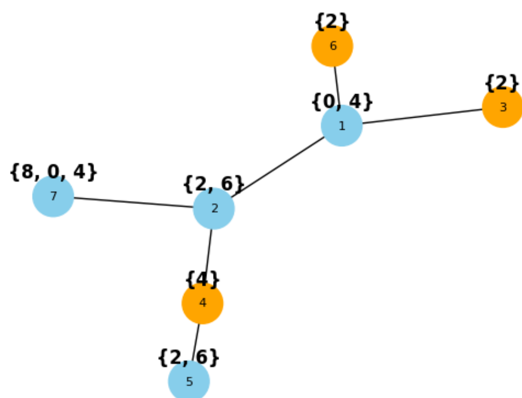
Розв'язок

```
G = nx.Graph()
edges = [(1, 2), (1, 3), (4, 5), (1, 6), (2, 4), (2, 7)]
G.add_edges_from(edges)
V_prime = 3, 4, 6,]
f_prime = {3: 3, 4: 6, 6: 3}
M=3
strong_m_par_ext_on_trees(G, V_prime, f_prime, M

```

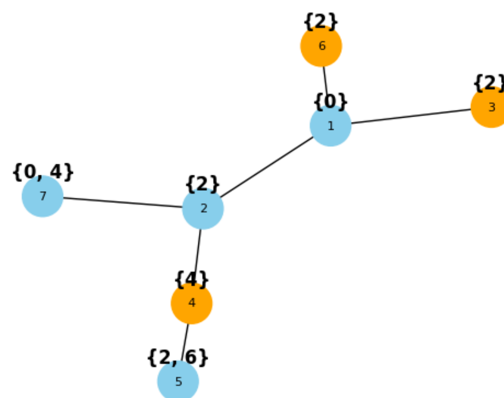
Маємо дерево G із проставленими мітками на трьох вершинах:



Спочатку кожній вершині присвоюється множина можливих значень. Далі обирається вершина-корінь і множини оновлюються ще раз.

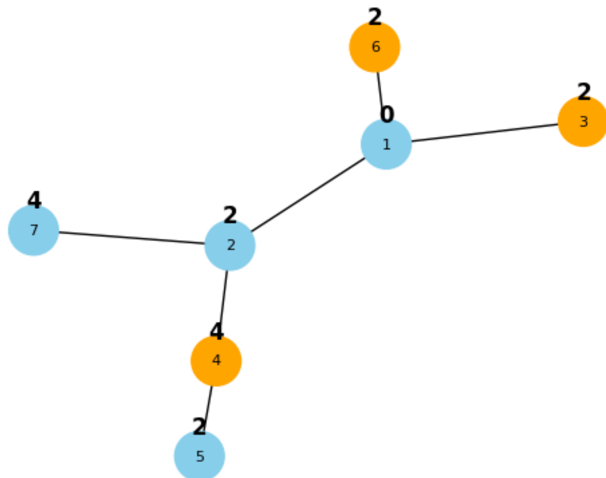


(а) Можливі варіанти міток для вершин до обрання кореня



(б) Можливі варіанти міток для вершин після обрання кореня

Наостанок, проставляємо решту вершин, слідкуючи, щоб між образами зберігалась задана відстань M. Маємо результат:



Приклад 4.5. Чи існує сильне M-Ліпшицеве відображення f загального графа G таке, що $f' \subseteq f$?

$V(G) = \{1, ..., 7\}$,

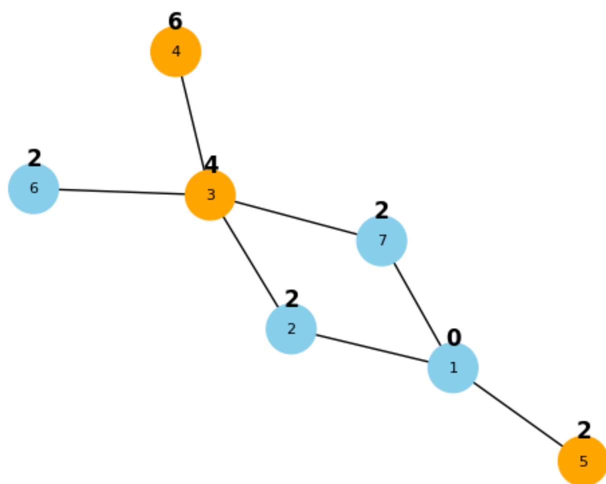
$E(G) = \{12, 15, 17, 23, 34, 36, 37\}$

$V'(G) = \{3, 4, 5\}, f' = \{3 : 4, 4 : 6, 5 : 2\}, M = 2$

Розв'язок

```
G = nx.Graph()
edges = [(1, 2), (4, 3), (1, 5), (2, 3), (3, 6), (1, 7), (7, 3)]
G.add_edges_from(edges)
V_prime = [5, 3, 4]
f_prime = {5: 2, 3: 4, 4: 6}
M = 2
M_ParExt(G, V_prime, f_prime, M, strong=True)

```

Отримане сильне M-Ліпшицеве відображення:



Кандидат під корінь було знайдено, і функція повернула нам повністю промаркований граф. Для знаходження числа для кожної вершини використовувались множини можливих значень.

Висновки

У даній роботі наведено дослідження M-Ліпшицевих відображень на деревах та загальних графах. У псевдокодах до алгоритмів з розширення часткових відображень, поданих у статті [1], було знайдено та виправлено декілька помилок. Було запропоновано Твердження 1.15, що по суті є доведенням Леми 1.14, якій бракувало обґрунтування в іншу сторону. Також були проаналізовані сильні M-Ліпшицеві відображення на деревах та загальних графах. Окремий пункт присвячено опису нашого алгоритму з розширення M-Ліпшицевих відображень на графах блоків. Також згадано про зв'язок цих відображень з сильною умовою існування індиферентних графів.

Практична частина полягає у реалізації алгоритмів для розширення часткових та часткових сильних M-Ліпшицевих відображень на зазначених вище видах графів. Для наочності показані також приклади застосування реалізованих алгоритмів з описом проміжних кроків.

Література

[1] J. Bok, *Algorithmic aspects of M-Lipschitz mappings of graphs*, https://arxiv.org/abs/1801.05496v3, (2018), 1–13.

[2] T. Feder and P. Hell, *List homomorphisms to reflexive graphs.* Journal of Combinatorial Theory, Series B, 1998, 72(2):236—250.

[3] F. Harary, *A characterization of block graphs*, *Canad. Math. Bull.* **6** (1963), 1–6.

[4] F. Harary and G. Prins, *The block-cutpoint-tree of a graph*, *Publ. Math. Debrecen.* **13** (1966), 103–105.

[5] P. J. Looges and S. Olariu, *Optimal greedy algorithms for indifference graphs*, *Computers Math. Applic.* Vol. 25, No. 7, (1993), 15–25.

[6] F.S. Roberts, *Graph Theory and its Applications to Problems of Society*, SIAM, (1978), 15–21, 27–36.

[7] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, (1980), 13–17, 171–188.