Звіт по домашньому завданню № 3

Магарита Ія
Варіант: 19 Мінімальна спрямована арборесценція (Chu–Liu / Edmonds)

Дано орієнтований зважений граф (G = (V, E)) і корінь (r).
Завдання - знайти мінімальну спрямовану арборесценцію: для всіх вершин, досяжних з (r), вибрати рівно по одному вхідному ребру (крім самого кореня), так щоб сумарна вага була мінімальною.

На вихід:
- список ребер у форматі "батько, вага",
- сумарна вага дерева.

ідейно

Реалізований алгоритм Чу–Ліу/Едмондса — класичний підхід для знаходження мінімальної арборесценції.
Алгоритм умовно складається з трьох блоків:

1. Виділення підграфу, що справді має значення - тобто всіх вершин, які досяжні з кореня.
2. **Вибір для кожної вершини її мінімального вхідного ребра**.
3. Перевірка на цикл серед цих ребер.
    - Якщо циклу немає - задача вирішена.
    - Якщо цикл є - він стискається в **одну** вершину, ваги коригуються, і алгоритм викликається рекурсивно.
    - Після повернення з рекурсії цикл **розтискається**, і ребра в ньому відновлюються з оригінальних мінімальних вхідних ребер.

реалізація

Вся логіка знаходиться у функції:

find_abrorescential(n, r, edges)

Вона повертає масив parent_final, де parent_final[v] = (parent, weight).

1. обчислення досяжних вершин

Спершу робиться bfs_reachable(...).

Це потрібно, бо класичний алгоритм вимагає, щоб у кожної вершини був хоча б один вхід - але лише якщо вона взагалі **може** бути досягнута з кореня.
Тому недосяжні вершини нам не треба брати до уваги, і надалі вони не впливають на результат.

2. вибір мінімального вхідного ребра

Для всіх (v ≠ r):

- шукається ребро (u -> v) з мінімальною вагою,
- записується як потенційний кандидат у дерево.

Це стандартний перший крок Chu–Liu/Edmonds.

3. пошук циклу

Оскільки ми вибрали рівно по одному вхідному ребру для всіх, хто не є коренем, цикл може виникнути лише всередині цього "скелету".

Функція find_cycle(...) ітерується від кожної вершини та слідкує, чи не потрапляємо ми в уже відвідану.

Якщо цикл знайдено - повертається список вершин циклу.

4. стиснення циклу

Цикл замінюється на одну "штучну" вершину C = n + 1.
Усі ребра, що:

- виходили з циклу стають C -> v
- входили в цикл стають u -> C, але з корекцією ваги

Корекція ваги виглядає так:
w'(u -> C) = w(u -> v) - w(мінімальне вхідне для v)
Це дозволяє "компенсувати" вже враховані ваги та не втратити оптимальність.

5. рекурсія та розтискання

Рекурсивно викликаємо алгоритм для нового графу.
Після того, як мінімальна арборесценція знайдена в стиснутому графі:

- перевіряємо, чи має C батька,

- якщо так - тоді ми відновлюємо ребро, яке входило в цикл ззовні й було мінімальним (для цього ми обрали якусь вершину циклу, тож всі інші входи в неї прибираємо - і циклу більш нема),

- внутрішні ребра циклу відновлюються з minimal_edge[v].

Труднощі з якими я зіштовхнулась

У первинній версії алгоритм падав на деяких контрприкладах через такі штуки:

1. Використання ребер з недосяжних вершин

Це створювало арборесценцію, де деякі вершини переставали бути досяжними з кореня - що по факту суперечить означенню задачі.
Як рішення - фільтрація edges після BFS (що очевидно).

2. Неповне "розтискання" вершини циклу

Після рекурсії вершини поза циклом могли мати батьком c_id, але потрібно було знати, яка саме оригінальна вершина циклу була їхнім батьком.
Як рішення: я ввела expand_outgoing_parent, яка займається тим, щоб заміняти c_id на реальний вузол.

Після цих мінімальних виправлень алгоритм проходить всі рандомні тести і збігається з брутфорс-реалізацією і подібним.

І потім я перевірила корекність реалізації.

Перевірка коректності реалізації через networkx

Щоб не просто вірити своїй імплементації алгоритму Чу–Ліу/Едмондса на слово, я написала окремий модуль для перевірки коректності з використанням бібліотеки **networkx**.

Що робить цей модуль

Є три основні частини:

1. Обчислення результату через мою реалізацію
2. Обчислення арборесценції через networkx
3. Порівняння результатів на випадкових графах

2. правильний варіант через networkx

Функція nx_arborescence_weight(n, r, edges) робить наступне:

- Спершу знаходить множину вершин, досяжних з кореня r (знову, через BFS), щоб працювати тільки з тим підграфом, який реально має значення.
- Будує орієнтований граф nx.DiGraph() і додає всі ребра u -> v з їхніми вагами.
- Далі вводиться штучний супер-корінь S = 0:
    - додається ребро S -> r з вагою 0,
    - для всіх інших досяжних вершин v ≠ c додаються ребра S -> v з дуже великою вагою M.
- Викликається nx.minimum_spanning_arborescence(...) для цього розширеного графа.
- Після цього:
    - усі ребра, що виходять із S або входять у S, просто викидаються;
    - залишається тільки арборесценція між "нормальними" вершинами (1 … n);
    - рахується її сумарна вага.

Та й все.

3. Порівняння та стресс-тести

Функція:

stress_test(num_tests=200, n_min=2, n_max=10, seed=0)

робить таке:

- Генерує num_tests випадкових орієнтованих графів з:
    - кількістю вершин n у діапазоні [n_min, n_max],
    - випадковим коренем r,
    - випадковими ребрами, що додаються з певною ймовірністю;
- гарантує, що з кореня є хоч одне вихідне ребро;
- для кожного графа:
    - порівнює сумарну вагу моєї арборесценції та арборесценції від networkx,
    - якщо є розбіжність — виводить детальний лог (через compare_single_graph(...)) і одразу зупиняє тестування;

- якщо всі тести пройдено — друкує повідомлення, що усі випадкові приклади збіглися.

В підсумку, якщо networkx мовчить - значить все справді добре.

Висновок

Загалом, завдання дозволило глибоко зрозуміти, як працює Chu–Liu/Edmonds "під капотом", особливо етапи стиснення та відновлення циклів. А ще випробувати фізичні можливості мого організму, в силу наявності інших дедлайнів.

Дякую за прочитання.