Пошук компонент сильної зв’язності

Роботу виконував Амброзяк Захар

Зміст

Постановка завдання

Дано орієнтований граф G без петель та мультиребер, множина вершин V, множина ребер – E. Тоді нехай n – це кількість вершин графа, а m – кількість ребер.

Компонента сильної зв’язності (з англійської **Strongly connected component SCC**) – це така підмножина вершин C ⊆ V, що будь-які вершини цієї підмножини є досяжними одна до одної, тобто ∀u, v ∈ C: u -> v, v -> u, де -> будемо позначати досяжність, тобто існування шляху з першої вершини в другу.

Компоненти сильної зв’язності для графа не перетинаються, тобто це розбиття всіх вершин графу. Звідси можна визначити конденсацію, як граф, що отримується з нашого графу стисненням кожної компоненти сильної зв’язності в одну вершину. Кожній вершині графа конденсації буде відповідати компонента сильної зв’язності графа G, а орієнтоване ребро між двома вершинами Ci та Cj графа конденсації проводиться, якщо буде пара вершин (u, v): u ∈ Ci, v ∈ Cj, між якими існує ребро у графі G, тобто (u, v) ∈ E.

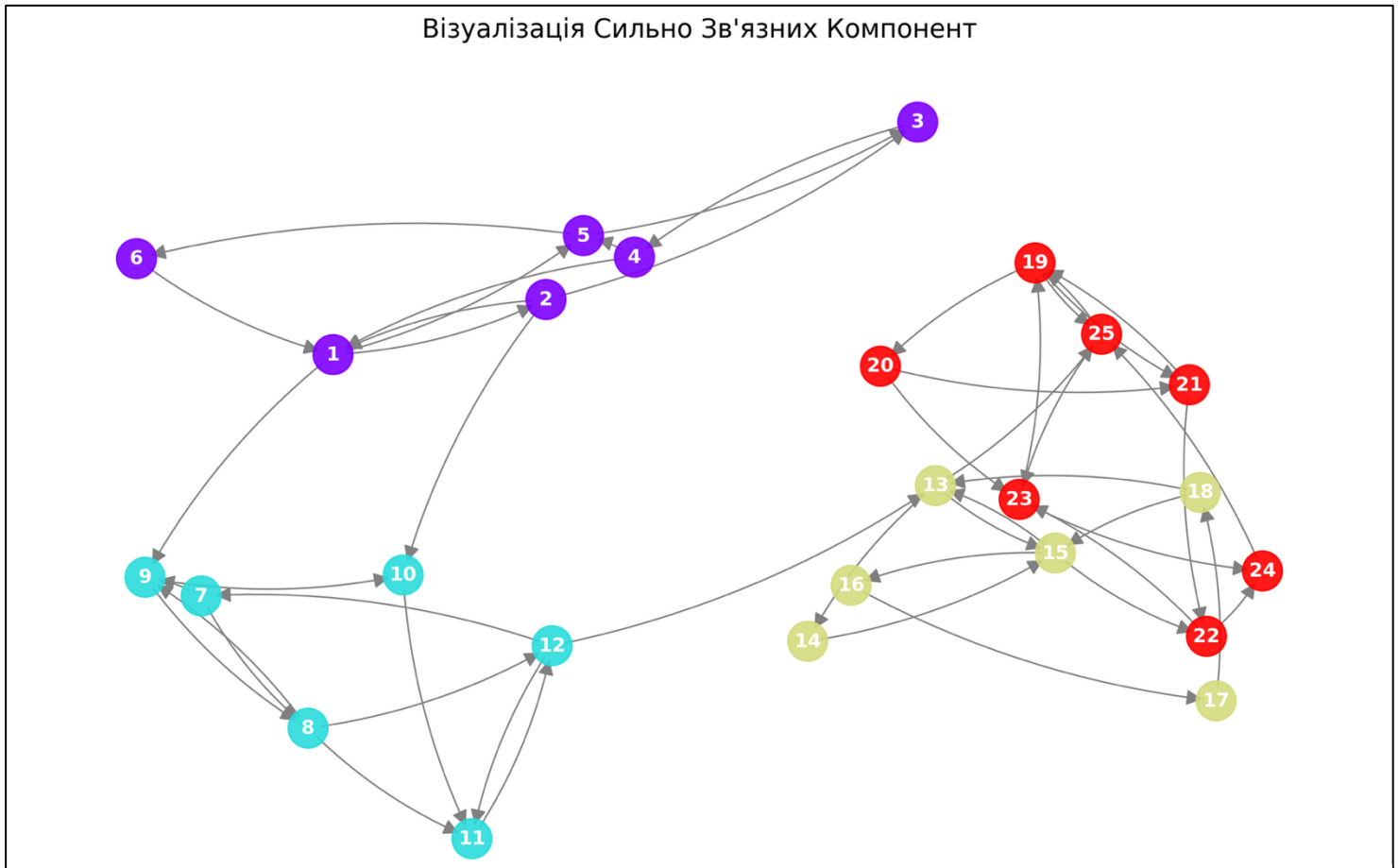


Рисунок 1: приклад графу **Візуалізація зроблена з допомогою Gemini Pro*

Алгоритм

Алгоритм був доведений **Косарайю** в 1978 році. Вся його концепція побудована на двох серіях пошуків у глибину

На першому кроці алгоритму виконуються обходи в глибину, що відвідують весь граф. Щоб це зробити, проходимось по всіх вершинах графа, і якщо вершина ще не відвідана, викликаємо обхід в глибину. При цьому для кожної вершини запам'ятовуємо час виходу t_out[v]. Часи виходу будуть мати важливу роль в алгоритмі, яка виражена в теоремі:

Теорема. Нехай C і C' – це дві різні компоненти сильної зв'язності, і нехай у графі конденсації між ними є ребро (C, C'). Тоді t_out[C] > t_out[C'], де t_out[C] – це час виходу із компоненти C, що визначається як максимум зі значень t_out[v] $\forall$ v $\in$ C

Ця теорема є основою алгоритму пошуку компонент сильної зв'язності. З неї випливає, що будь-яке ребро (C, C') у графі конденсацій йде з компоненти з більшою величиною t_out у компоненту з меншою величиною.

Якщо ми відсортуємо всі вершини v $\in$ V у порядку спадання часу виходу t_out[v], то першою буде деяка вершина u, що належить «кореневій» компоненті сильної зв'язності, тобто в яку не входить жодне ребро в графі конденсацій. Тепер нам хочеться запустити такий обхід з цієї вершини u, який би відвідав тільки цю компоненту сильної зв'язності та не зайшов у жодну іншу; навчившись це робити, ми зможемо виділити всі компоненти сильної зв'язності: видаливши з графа вершини першої виділеної компоненти, ми знову знайдемо вершину з найбільшою величиною tout, запустимо обхід і.т.д.

Щоб навчитися робити такий обхід, розглянемо обернений граф G^T, тобто граф, отриманий з G зміною напрямку кожного ребра на протилежний (a -> b, тоді обернений b -> a). Неважко побачити, що в цьому графі будуть ті самі компоненти сильної зв'язності, що й у початковому графі. Більше того, граф конденсації G^SCC,T для нього буде дорівнювати оберненому графу конденсації початкового графа G^SCC. Це означає, що тепер із розглянутої нами "кореневої" компоненти вже не виходитимуть ребра в інші компоненти.

Таким чином, щоб обійти всю "кореневу" компоненту сильної зв'язності, що має деяку вершину v, достатньо запустити обхід в глибину із вершини v у графі G^T. Цей обхід відвідає всі вершини цієї компоненти сильної зв'язності та тільки їх. Після цього, ми можемо їх умовно видалити з графу, знаходити нову вершину з максимальним t_out і запускати обхід на оберненому графі з неї і.т.д.

Отже ми побудували такий алгоритм виділення компонент сильної зв’язності:

1. Запустити серію обходів у глибину графа G, яка повертає вершини в порядку збільшення часу виходу як деякий список order.
2. Побудувати обернений граф G^T. Запустити серію обходів у глибину цього графу в поряду списку order (у зворотному). Кожна множина вершин, яку ми отримаємо після обходу в глибину і буде компонентою сильної зв’язності.

Складність алгоритму тоді дорівнює O(n + m), оскільки це лише два обходи в глибину. Хоча варто звернути увагу, що високій щільності графу m→n^2 і тоді складність списків наблизиться до O(n^2). Також, якщо граф побудований на матриці суміжності, то ми будемо змушені проходити всю матрицю, тому що потрібно буде перевіряти ребра, яких навіть немає (if matrix[u][v]). Це значно погіршує складність алгоритму – тоді вона буде O(n^2).

Псевдокод

```
Функція DFS1(вершина u):
    Позначити u як відвідану
    Для кожного сусіда u як вершину v в графі G:
        Якщо v ще не відвідана:
            DFS1(v)
    Додати u до черги Order
Функція DFS2(вершина u, список Component):
    Позначити u як відвідану
    Додати u до списку Component
    Для кожного сусіда u як вершину v графі G^T (оберненому):
        Якщо v ще не відвідана:
            DFS2(v, Component)
Алгоритм Косарайю(Граф G):
    N = кількість вершин у G
```

Visited = масив розміру N, заповнений значенням False

Order = порожня черга

1. Для кожної вершини i від 0 до n-1:

    Якщо visited[i] це False:

        DFS1(i)

G^T = обернений граф до G

Заповнити масив Visited значенням False

2. Поки черга Order не порожня:

    u = витягуємо верхній елемент з черги Order

        Якщо visited[u] це False:

            Component = порожній список

            DFS2(u, Component)

            Виводимо Component

Опис програмної реалізації

Весь код був написаний на мові програмування Python, оскільки в ній є широка можливість для візуалізацій та роботи з даними та відносна простота написання коду. Весь код був розділений на чотири файли .py:

1. **graph.py**

    В ньому було реалізовано класи GraphList та GraphMatrix як представники графів, що подані списком суміжностей та матрицею суміжностей відповідно. Самі списки суміжностей та матриці суміжностей реалізовані двовимірними масивами (list of lists). Кожен клас зберігає в собі інформацію про граф: кількість вершин, кількість ребер, щільність. В обох класах є всі необхідні методи, як от генерація графу, зручне виведення на екран всіх ребер, перехід від матриці суміжностей до списку і навпаки, додавання ребер і.т.д. Все це дозволяє

легко працювати з графом та використовувати його для будь-яких алгоритмів на графах.

2. **algorithms.py**

В ньому були реалізовані алгоритми Kosaraju_matrix та Kosaraju_list в залежності від подання графу. Самі алгоритми повністю співпадають з псевдокодом наведеним вище. В них також обраховується час роботи алгоритму, щоб в подальшому робити експерименти

3. **experiments.py**

В ньому реалізовані експерименти для обрахунку часу алгоритмів з подальшою візуалізацією та збереженням даних у таблицю Excel.

4. **graph_visuals.py**

В ньому реалізована візуалізація графу з чотирма сильнозв'язними компонентами. Цей файл був повністю згенерований ШІ.

Посилання на GitHub репозиторій

https://github.com/ZaharAmbrozyak/Kosaraju-Algorithm

Експериментальна частина

Оскільки кількість компонент сильної зв’язності сильно залежать від параметрів графу, то було прийняте рішення обирати найрізноманітніші параметри. Для експериментальної частини було вирішено обрати наступні параметри:

Для кількості вершин взято 10 варіантів: 20, 40, … , 180, 200.

Також було обрано 5 щільностей: 0.1, 0.3, 0.5, 0.7, 0.9.

Кожну пару експериментів (к-сть вершин, щільність) ми проводимо 1000 разів для точності результату та зменшення впливу похибок комп’ютера.

Алгоритм експерименту псевдокодом

Для кожної щільності зі списку щільностей:

Для кожної к-сті вершин зі списку к-сті вершин:

Повторити 1000 разів:

1. Згенерувати список суміжності
2. Пройти алгоритм Косарайю
3. Додати пройдений час алгоритму до суми
4. Зробити те саме для матриці суміжностей

Знайти середній час виконання для 1000 симуляцій алгоритму для обох реалізацій графу і записати це в словник результатів

Таблиці з результатами

Кількість вершин, N	Список суміжностей	Матриця суміжностей
20	0.0086 мс	0.0247 мс
40	0.0227 мс	0.0801 мс
60	0.0317 мс	0.1689 мс
80	0.0389 мс	0.2964 мс
100	0.0618 мс	0.4794 мс
120	0.1187 мс	0.706 мс
140	0.1362 мс	0.9407 мс
160	0.2271 мс	1.2233 мс
180	0.2384 мс	1.4732 мс
200	0.2522 мс	1.768 мс

Таблиця 1. Час виконання алгоритму при щільності 10%

Кількість вершин, N	Список суміжностей	Матриця суміжностей
20	0.0109 мс	0.0273 мс
40	0.0341 мс	0.0912 мс
60	0.0526 мс	0.2001 мс
80	0.0757 мс	0.3496 мс
100	0.1241 мс	0.541 мс
120	0.2065 мс	0.7929 мс
140	0.2898 мс	1.0554 мс
160	0.3398 мс	1.3489 мс
180	0.3651 мс	1.6976 мс
200	0.5038 мс	2.0745 мс

Таблиця 2. Час виконання алгоритму при щільності 30%

Кількість вершин, N	Список суміжностей	Матриця суміжностей
20	0.0155 мс	0.0295 мс
40	0.0446 мс	0.1019 мс
60	0.0748 мс	0.2235 мс
80	0.1108 мс	0.3895 мс
100	0.1881 мс	0.6038 мс
120	0.3041 мс	0.8802 мс
140	0.3945 мс	1.1686 мс
160	0.5154 мс	1.5032 мс
180	0.5746 мс	1.9051 мс
200	0.7687 мс	2.4476 мс

Таблиця 3. Час виконання алгоритму при щільності 50%

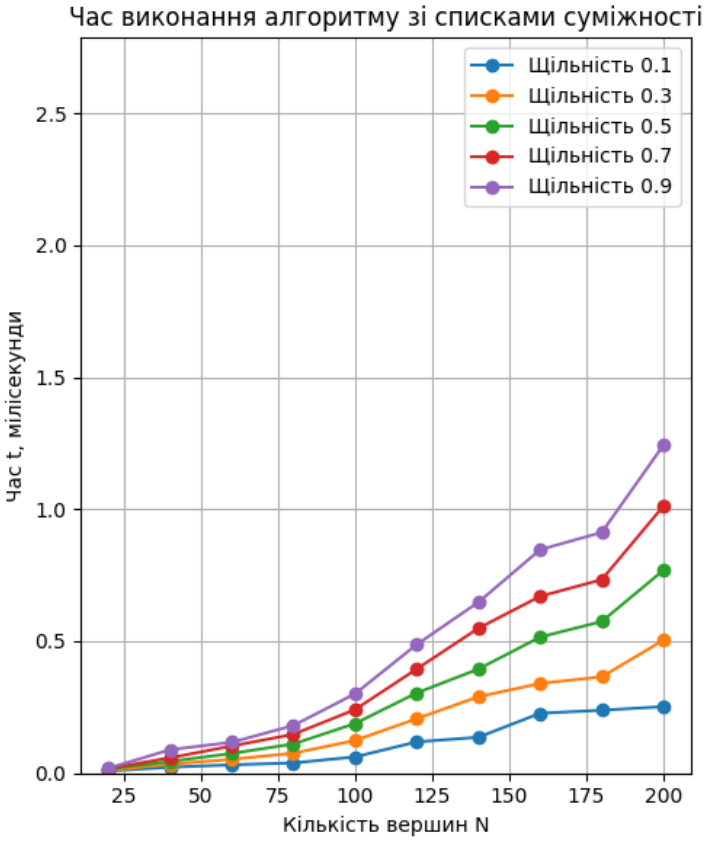
Кількість вершин, N	Список суміжностей	Матриця суміжностей
20	0.0174 мс	0.0324 мс
40	0.0584 мс	0.1111 мс
60	0.103 мс	0.2355 мс
80	0.1473 мс	0.4061 мс
100	0.2411 мс	0.6342 мс
120	0.3948 мс	0.9086 мс
140	0.5491 мс	1.2136 мс
160	0.6699 мс	1.5655 мс
180	0.7331 мс	1.9787 мс
200	1.0117 мс	2.4203 мс

Таблиця 4. Час виконання алгоритму при щільності 70%

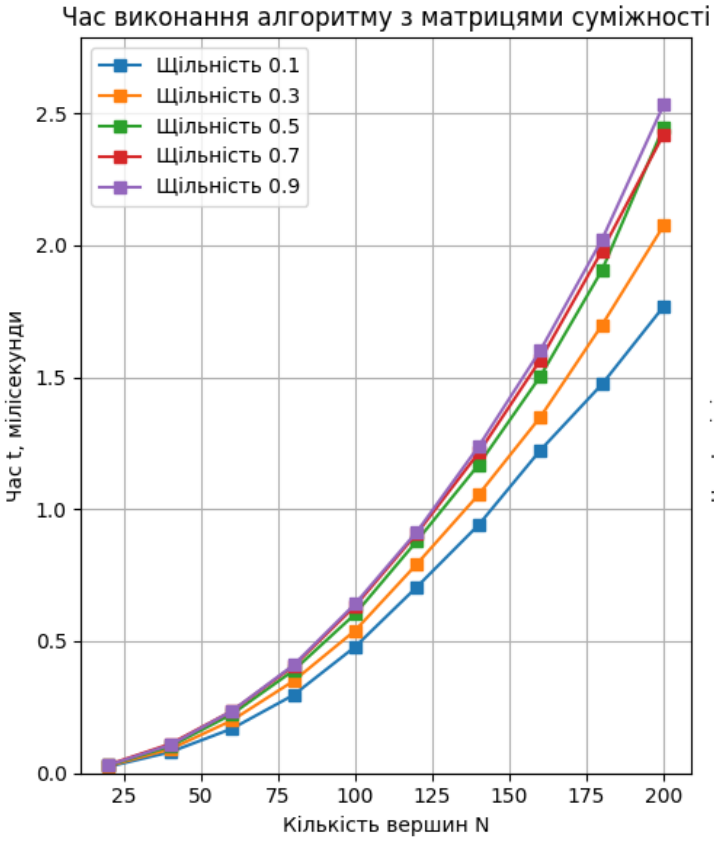
Кількість вершин, N	Список суміжностей	Матриця суміжностей
20	0.0187 мс	0.0308 мс
40	0.0891 мс	0.1074 мс
60	0.1172 мс	0.2358 мс
80	0.1807 мс	0.4101 мс
100	0.3009 мс	0.6423 мс
120	0.4876 мс	0.9152 мс
140	0.6478 мс	1.2381 мс
160	0.8466 мс	1.6015 мс
180	0.9121 мс	2.0211 мс
200	1.2424 мс	2.5319 мс

Таблиця 5. Час виконання алгоритму при щільності 90%

Візуалізація даних з таблиць



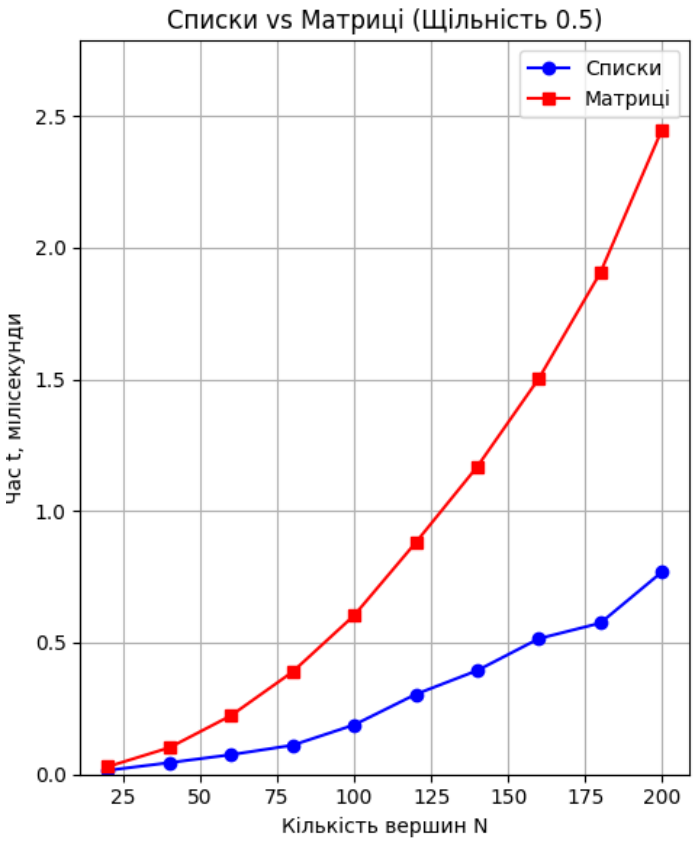
Графік 1: списки суміжності



Графік 2: матриці суміжності

З графіків бачимо, що при збільшенні кількості вершин, або розміру щільності буде зростати час виконання алгоритму. При чому для графу, побудованому на списку суміжності, швидкість зростання часу буде лінійною, а для графу, побудованому на матриці суміжностей – квадратичною.

Порівняльний графік списку суміжності та матриці суміжності



Графік 3: Різниця між матрицями та списками

З графіку можемо явно побачити складність обох алгоритмів:

В матриць суміжності складність алгоритму O(n^2), що можна побачити як вітку параболи.

В списку суміжності складність алгоритму O(n+m), що можна побачити як плавно зростаючу пряму.

Таку тенденцію можна пояснити тим, що сам алгоритм базується на списку суміжності. Якщо його переносити на матрицю суміжності, то в нас почнуть виникати “зайві операції”, що сповільнить алгоритм, оскільки нам доведеться робити перебір всієї матриці n*n.

Спочатку різниця часу виконання обох варіантів не суттєва через малу кількість даних. Проте при збільшені кількості вершин, ситуація кардинально змінюється на користі списків суміжності. Хоча це дивно, оскільки як вже зазначалося раніше, при високій щільності (m→n^2) складності алгоритмів мали б бути однаковими як O(n^2). Можливо це можна пояснити постійними зайвими переборами нулів у матриці, яких немає у списках.

Висновок експерименту:

Алгоритм Косарайю буде знаходити компоненти сильної зв’язності швидше в графі побудованому на списку суміжності, ніж графі побудованому на матриці суміжності.

Висновки

Підводячи підсумки роботи:

1. Було детально проаналізовано вхідні та вихідні дані, теоретичну частину алгоритму та псевдокод.
2. Був написаний код на Python, де був реалізований граф та алгоритм на основі псевдокоду.
3. Результати алгоритму були використані для побудови графіків, таблиць та візуалізацій графу.

З експерименту ми зробили висновок, що алгоритм ліпше працює для графів, що подані списком суміжностей, аніж матрицею суміжностей. З графіків експерименту було доведено теоретичну оцінку складності алгоритмів.

Цей проєкт дозволив попрактикуватися з новим алгоритмом, реалізувати граф та використовувати вихідні дані алгоритму для візуалізацій, графіків та таблиць.

Використані джерела

1. Kosaraju algorithm. https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm - сторінка у вікіпедії для ознайомлення з алгоритмом
2. Erdős-Rényi model. https://en.wikipedia.org/wiki/Erdős–Rényi_model - сторінка у вікіпедії для реалізації генерації графу
3. Gemini. https://gemini.google.com/app - ШІ, який допомогав з візуалізацією даних, проте не використовувався в інших частинах проєкту
4. Strongly connected components. https://cp-algorithms.com/graph/strongly-connected-components.html - сайт з описами алгоритмів, з якого я брав більшу частину теорії