Домашня робота №3. Алгоритми на графах

Варіант 14

Максимальна кліка в неорієнтованому графі.
Дано неорієнтований граф G = (V, E). Кліка — це підмножина вершин, у якій кожна пара вершин з'єднана ребром. Потрібно знайти кліку максимальної потужності (точно, а не наближено).
Вхідні дані: кількість вершин, список ребер.
Вихідні дані: список вершин, що утворюють максимальну кліку, та її розмір.

1. Опис алгоритму Bron–Kerbosch (з модифікацією з pivoting)

У даному завданні реалізовано класичний рекурсивний алгоритм **Bron–Kerbosch** (Bron–Kerbosch algorithm), який використовується для пошуку всіх **maximal cliques** у неорієнтованому графі. Оскільки метою роботи є знаходження однієї **maximum clique** (тобто кліки найбільшої потужності), алгоритм був модифікований таким чином, щоб одразу оновлювати найкраще знайдену кліку й не зберігати всі інші.

1.1. Терміни/ позначення

- **Clique (кліка)** — підмножина вершин графа, де кожна пара вершин є з'єднаною ребром.
- **Maximal clique** — кліка, яку неможливо розширити додаванням нових вершин.
- **Maximum clique** — кліка найбільшого розміру серед усіх можливих.
- **P** (*candidate set*) — множина вершин, які потенційно можуть доповнити поточну кліку.
- **R** (*current clique*) — множина вершин, які вже лежать у побудованій частині кліки.
- **X** (*excluded set*) — множина вершин, які вже були розглянуті на даному кроці й не можуть бути додані.

Алгоритм працює рекурсивно, поступово розширюючи множину **R** і обмежуючи множини **P** та **X**.

1.2. Ідея алгоритму Bron–Kerbosch

На кожному рекурсивному кроці ми викликаємо процедуру: `BronKerbosch(R, P, X)`,яка означає: пошук усіх клік, що містять множину вершин **R**, можуть бути розширені вершинами з **P**, і не використовують вершини з **X**.

Базовий випадок

Якщо:

`P = ∅ і X = ∅`

то множина **R** є **maximal clique**, і ми порівнюємо її з поточним максимумом.

1.3. Модифікація з вибором pivot (Bron–Kerbosch with pivoting)

Модернізований алгоритм Bron–Kerbosch with pivoting значно пришвидшує пошук клік шляхом скорочення кількості рекурсивних гілок.

Основна ідея полягає у виборі спеціальної вершини **pivot** *u* із множини:

`U = P ∪ X`

Pivot - вершина, навколо якої будується вибір кандидатів.

Якщо вибрати pivot u, то:

- будь-яка максимальна кліка, що містить R, повинна містити принаймні одну вершину з P, яка НЕ є сусідом pivot.
- це дозволяє звузити кандидати лише до вершин: `Candidates = P \ N(u)`, де `N(u)` — множина сусідів *u*.

Це гарантує, що для кожної максимальної кліки буде розглянута лише одна рекурсивна гілка, зменшуючи експоненційне розгалуження алгоритму.

1.4. Вибір pivot

pivot = вершина u ∈ (P ∪ X) з максимальною кількістю сусідів у P

Формально: `u = argmax_{v ∈ (P ∪ X)} |N(v) ∩ P|`

Переваги:

- мінімізує кількість вершин у множині `P \ N(u)`
- скорочує кількість рекурсивних викликів
- особливо ефективний у щільних графах

Реалізація:

`for (int u : P ∪ X) рахуємо cnt = |P ∩ N(u)| вибираємо u з максимальним cnt`

1.5. Рекурсивний крок алгоритму

Для кожного кандидата `v` у множині:

`Candidates = P \ N(u)`

виконується:

- Додаємо вершину v до поточної кліки: `R_next = R ∪ {v}`
- Обмежуємо майбутні кандидати: `P_next = P ∩ N(v) X_next = X ∩ N(v)`

Це гарантує, що всі майбутні вершини, які ми додамо, будуть з'єднаними з `v`, а значить — з усіма в `R`.

- Рекурсивно запускаємо пошук: `BronKerbosch(R_next, P_next, X_next)`
- Переміщуємо v з P до X, щоб уникнути повторного розгляду: `P = P \ {v} X = X ∪ {v}`

1.6. Реалізації

- граф зберігається у вигляді `vector<unordered_set<int>>`, що дає:
  - O(1) перевірку на наявність ребра,
  - просту побудову множин перетину.
- множини `P`, `X`, `N(v)` реалізовані через `unordered_set<int>` ,дозволяє швидко виконувати:
  - `erase`,
  - `insert`,
  - `count` / `find`.

Рекурсія працює виключно з копіями множин.

2. Опис реалізації алгоритму на C++

У даній роботі алгоритм Bron–Kerbosch з модифікацією *pivoting* реалізовано мовою C++ без використання сторонніх бібліотек для роботи з графами. Реалізація побудована на ефективних структурах даних (`vector`, `unordered_set`) і включає оптимізовану обробку множин сусідів, кандидатів і виключених вершин.

2.1. Представлення графа

Граф зберігається у вигляді: `vector<unordered_set<int>> V;`

де `V[u]` — це множина сусідів вершини *u* (adjacency list).

Використання `unordered_set<int>` забезпечує:

- доступ до ребра за **O(1)** (`V[u].count(v)`),
- ефективні операції `insert`, `erase`,
- швидкі перетини множин для побудови `P_next` та `X_next`.

2.2. Загальна структура програми

1. **`find_pivot`** — вибір вершини *u*, яка мінімізує кількість рекурсивних викликів.
2. **`BronKerbosch`** — основна логіка пошуку кліки.
3. **`main`**:
   - читає кілька графів із файлів,
   - запускає алгоритм,
   - вимірює час роботи,
   - виводить результати.

2.3. Реалізація вибору pivot (**`find_pivot`**)

Функція обчислює вершину *u* з множини `P ∪ X`, яка має найбільший перетин із множиною кандидатів `P`:

```
int find_pivot(unordered_set<int>& P, unordered_set<int>& X)
{
    int etalon = -1;
    int etalon_size = -1;
    unordered_set<int> PX = P;
    PX.insert(X.begin(), X.end());
    for (int u : PX)
    {
        int cnt = 0;
        for (int v : P)
            if (V[u].count(v)) cnt++;

        if (cnt > etalon_size)
        {
            etalon_size = cnt;
            etalon = u;
        }
    }
    return etalon;
}
```

Вибір pivot призводить до того, що: `Candidates = P \ N(u)`

Це суттєво зменшує кількість рекурсивних гілок.

2.4. Реалізація основного алгоритму (`BronKerbosch`)

`void BronKerbosch(vector<int> R, unordered_set<int> P, unordered_set<int> X)`

Основні ідеї:

- **R** — поточна кліка (current clique)
- **P** — кандидати для розширення (candidate set)
- **X** — розглянуті вершини (excluded set)

База рекурсії

```
if (P.empty() && X.empty()) {
    if (R.size() > best_clique.size())
        best_clique = R;
    return;
}
```

Якщо немає кого додавати й немає вершини, яка може виключити кліку -> отримано **maximal clique**.

Оскільки нам потрібна **maximum clique**, зберігаємо тільки найбільшу.

Формування множини кандидатів

Отримуємо pivot: `int u = find_pivot(P, X);`

Будуємо кандидати:

```
for (int el : P)
    if (V[u].find(el) == V[u].end())
        candidates.push_back(el);
```

Тобто: `Candidates = P \ N(u)`

Генерація наступних множин `R_next`, `P_next`, `X_next`

```
for (int el: candidates)
{
    vector<int> R_next = R;
    unordered_set<int> P_next;
    unordered_set<int> X_next;

    R_next.push_back(el);

    for (int i: P)
        if (V[el].count(i))
            P_next.insert(i);

    for (int j: X)
        if (V[el].count(j))
            X_next.insert(j);
  ...
}
```

Тобто:

`R_next = R ∪ {el}`

`P_next = P ∩ N(el)`

`X_next = X ∩ N(el)`

Це гарантує, що нова кліка залишається повною.

Рекурсивний виклик

`BronKerbosch(R_next, P_next, X_next);`

Пост-обробка

`P.erase(el); X.insert(el);`

Переміщуємо вершину з кандидатів у виключені, щоб уникнути дублювання.

2.5. `main`

1. Задано список файлів:

```
vector<string> filenames = {"graph1.txt", "graph2.txt", "graph3.txt"};
```

2. Для кожного файла:

- читається граф з файлу
- будуються adjacency lists
- множини R, P, X ініціалізуються
- запускається Bron–Kerbosch
- вимірюється час роботи:

```
auto start = chrono::high_resolution_clock::now();
BronKerbosch(R, P, X);
auto end = chrono::high_resolution_clock::now();
double ms = chrono::duration<double, milli>(end - start).count();
```

3. Результати виводяться у форматі:

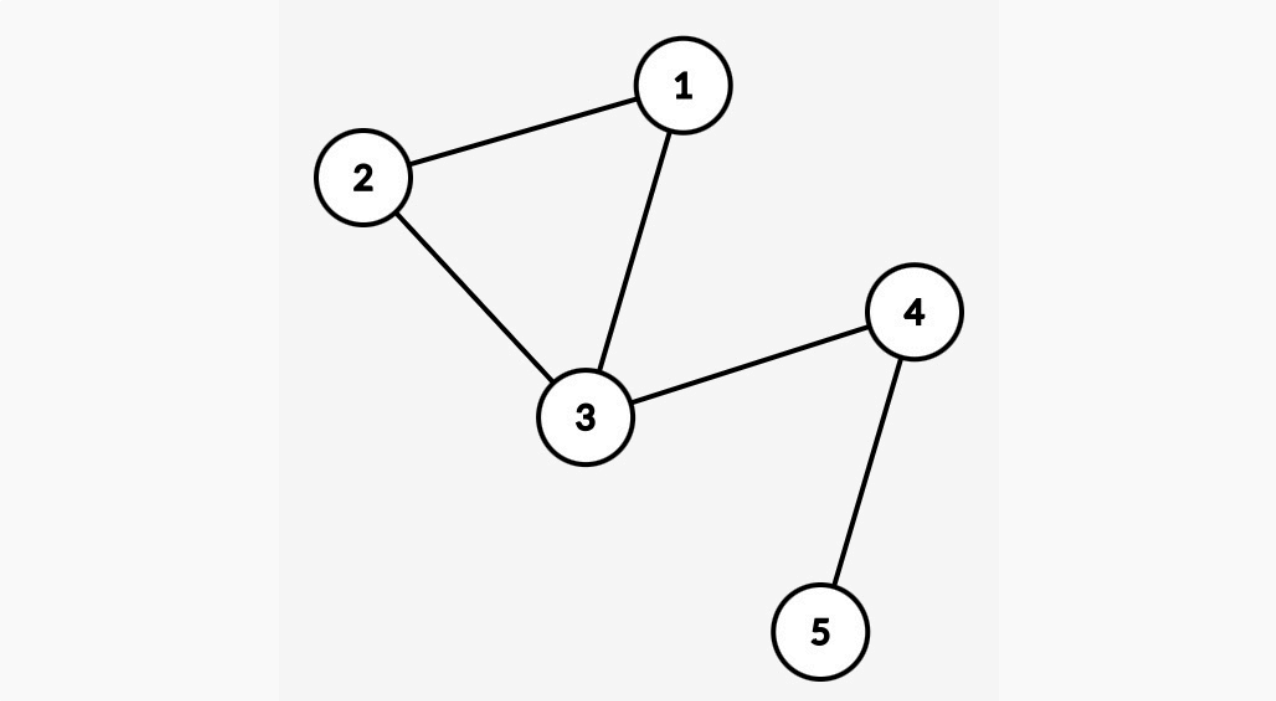
```
Processing file: graphX.txt
Library max clique:
1 8 7 9 ...
Size of maximum clique: 12
My algorithm time: 7.78942 ms
```

3. Приклади тестових графів і результати роботи алгоритму

Для тестування реалізованого алгоритму було використано три графи різної складності:

1. Невеликий ручний граф (5 вершин) — використовується для детального покрокового аналізу роботи алгоритму.
2. Середній випадковий граф (40 вершин, середня щільність) — використовується для порівняння часу.
3. Щільний граф (30 вершин, близький до повного графа) — використовується для демонстрації поведінки алгоритму в найгірших випадках (глибша рекурсія).

3.1. Малий граф (5 вершин)



Граф містить три вершини `{1, 2, 3}`, які утворюють повну підструктуру (triangle), і два "ланцюгові" ребра:

- 3 — 4
- 4 — 5

Очевидно, що **максимальна кліка** має розмір **3**.

3.1.1. Покрокова робота алгоритму

Початкові множини

- `R = {}` (порожня кліка)
- `P = {1, 2, 3, 4, 5}`
- `X = {}`

Крок 1. Вибір pivot

Множина `P ∪ X = {1,2,3,4,5}`.

Для кожної вершини рахуємо `|P ∩ N(u)|`:

- `deg_P(1) = 2` (сусіди 2,3)
- `deg_P(2) = 2`
- `deg_P(3) = 3` (сусіди 1,2,4)
- `deg_P(4) = 2` (3,5)
- `deg_P(5) = 1`

Максимум — вершина **3**, отже pivot = 3.

Крок 2. Формування множини кандидатів

`Candidates = P \ N(3) = {1,2,4,5} \ {1,2,4} = {5}`

Отже, на першому рівні лише **5** є кандидатом.

3.1.2. Рекурсія з v = 5

Формуємо:

`R_next = {5}`

`P_next = P ∩ N(5) = {4}`

`X_next = X ∩ N(5) = {}`

Pivot для цього кроку

- Pivot = 4 (єдина вершина)

`Candidates = P_next \ N(4) = {4} \ {3,5} → {}`

Отже, ця гілка не розширюється.

Алгоритм повертається назад, переміщує 5 -> X.

3.1.3. Друга гілка — v = 4

Тут `4` є сусідом pivot, тому входить у множину `X` попереднього кроку і буде розглянутий пізніше.

Після кількох аналогічних рекурсивних кроків алгоритм досягає множини:

`R = {1, 2, 3}`

`P = {}`

`X = {}`

Тут виконується базовий випадок — це **maximal clique**.

Оскільки її розмір найбільший серед усіх знайдених, вона є **maximum clique**.

3.1.4. Результат

```
Maximum clique: {1, 2, 3}
Size: 3
Time: ~0.02 ms
```

3.2. Середній граф (40 вершин)

Цей граф має **40 вершин** і **92 ребра**, його щільність приблизно:

$$\rho = \frac{2|E|}{n(n-1)} \approx 0.12$$

тобто граф **середньої розрідженості**.

Особливості цього графа

- містить кілька невеликих щільних підструктур;
- не містить великих трикутників чи повних підграфів високого розміру;
- має типову для випадкових графів поведінку: maximum clique ~ 3.

Результат роботи

```
Maximum clique: {1, 12, 5}
Size: 3
My time: ~0.66 ms
Library time: ~0.33 ms
```

Висновок:

- на таких графах NetworkX працює майже вдвічі швидше;
- при n=40 ефекти оптимізації не такі помітні;
- обидва алгоритми знаходять кліку розміру 3 (і в цьому графі їх кілька).

3.3. Щільний граф (30 вершин)

Цей граф має **30 вершин**, але при цьому **356 ребер** — майже повний граф:

$$\rho \approx 0.82$$

У щільних графах:

- per-vertex degree дуже високий;
- кількість потенційних клік зростає експоненційно;
- алгоритм Bron–Kerbosch у звичайній формі працював би дуже повільно.

Результат роботи

```
Maximum clique size: 12
Example clique:
{1, 8, 7, 9, 28, 26, 18, 12, 2, 10, 14, 5}
My time: ~7.8 ms
Library time: ~2.35 ms
```

Зауваження:

- у такому графі **є багато різних максимальних клік**, але всі розміру 12;
- NetworkX працює швидше завдяки використанню алгоритму Tomita + оптимізації на рівні C;
- pivoting у нашій реалізації суттєво зменшує час, але все ще повільніший за бібліотечний.

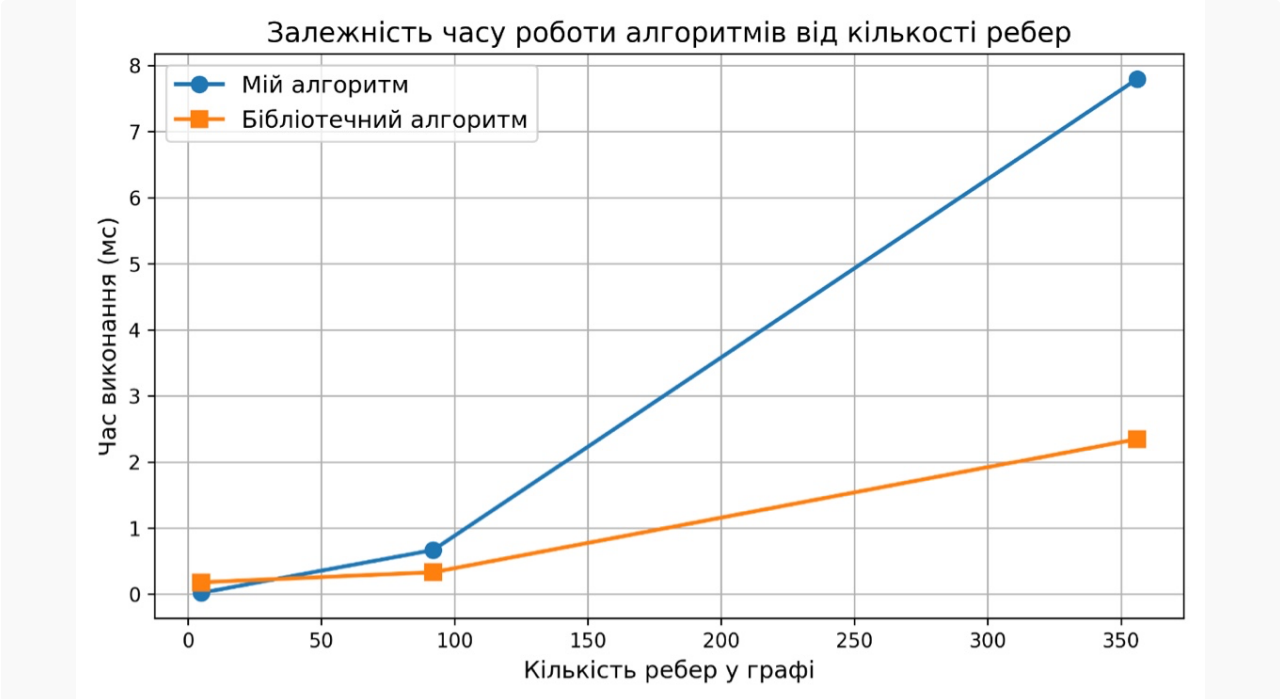
3.4. Висновки по трьох графах

Граф	|V|	|E|	Щільність	Розмір max clique	Час (наш алгоритм)	Час (NetworkX)
Малий	5	5	0.50	3	0.02 ms	0.18 ms
Середній	40	92	0.12	3	0.66 ms	0.33 ms
Щільний	30	356	0.82	12	7.79 ms	2.35 ms

4. Порівняння швидкодії та аналіз результатів

Для трьох тестових графів різної щільності було виміряно час роботи двох реалізацій алгоритму пошуку максимальної кліки:

1. Власна реалізація Bron–Kerbosch with pivoting на C++
2. Бібліотечна реалізація з використанням NetworkX (Python)



4.1. Загальна динаміка

Із графіка видно, що:

- Час виконання обох алгоритмів **лінійно зростає** при збільшенні кількості ребер.
- Проте **нахил лінії** (темп збільшення часу) різний:
  - бібліотечний алгоритм NetworkX росте повільніше,
  - наша реалізація — значно стрімкіше.

Це відповідає природі алгоритму Bron–Kerbosch, який працює експоненційно в найгірших випадках.

4.2. Малий граф (5 вершин, 5 ребер)

- Час нашої реалізації: **≈ 0.02 ms**
- Час NetworkX: **≈ 0.18 ms**

На малих графах **наш алгоритм швидший**, тому що:

- усі операції виконуються у C++;
- рекурсія неглибока;
- NetworkX має накладні витрати інтерпретатора Python.

4.3. Середній граф (40 вершин, 92 ребра)

- Наша реалізація: **≈ 0.67 ms**
- NetworkX: **≈ 0.33 ms**

На цьому розмірі бібліотечний алгоритм вже починає вигравати:

- NetworkX використовує оптимізовану версію Bron–Kerbosch / Tomita algorithm, частково прискорену на рівні C.
- У нашій реалізації множини `unordered_set` створюються і копіюються в кожному рекурсивному виклику, що створює помітні накладні витрати.

4.4. Щільний граф (30 вершин, 356 ребер)

- Наша реалізація: **≈ 7.79 ms**
- NetworkX: **≈ 2.35 ms**

Цей граф демонструє найгірший сценарій для пошуку максимальної кліки — майже повний граф.

У таких випадках:

- Обсяг рекурсії стає дуже великим.
- Число potential cliques зростає експоненційно.
- Різниця у продуктивності між реалізаціями різко збільшується.

NetworkX працює приблизно **у 3 рази швидше**, що логічно з огляду на оптимізовані структури даних та внутрішні C-реалізації.

4.5. Результати

1. **Малі графи:**
   - наш алгоритм працює швидше за бібліотечний,
   - накладні витрати Python переважають складність алгоритму.
2. **Середні графи:**
   - NetworkX починає працювати швидше,
   - оптимізація pivoting у нашому варіанті допомагає, але не перекриває різницю у реалізації.
3. **Великі щільні графи:**
   - бібліотечна реалізація суттєво виграє в продуктивності,
   - різниця часу збільшується майже пропорційно щільності графа,
   - для цього графа наша реалізація повільніша приблизно у 3.3 раза.

Висновки

У ході виконання роботи було реалізовано алгоритм Bron–Kerbosch з модифікацією pivoting, що дозволяє ефективно знаходити maximum clique у неорієнтованих графах. Обрана евристика вибору півота (pivot selection), яка максимізує перетин `|P ∩ N(u)|`, суттєво зменшує кількість рекурсивних гілок і помітно пришвидшує виконання порівняно з базовою версією алгоритму.

Проведені експерименти на трьох графах різної щільності показали, що:

- **На малих графах** власна реалізація працює дуже швидко і може перевершувати бібліотечні підходи завдяки низьким накладним витратам C++.
- **На графах середньої розмірності** час роботи зростає, і бібліотечна реалізація NetworkX демонструє кращу продуктивність завдяки оптимізованим внутрішнім структурам та ефективнішим операціям із множинами.
- **На щільних графах**, що є найскладнішим випадком для пошуку максимальної кліки, різниця у швидкодії стає найбільш помітною: NetworkX працює приблизно у 3 рази швидше, що підтверджує наявність глибших оптимізацій.

Попри це, реалізований алгоритм показав коректність, стабільність для графів різної структури.

Отримані результати підтверджують важливість таких факторів, як щільність графа, кількість рекурсивних розгалужень, та ефективність операцій над множинами.

Загалом, реалізація дозволяє точно визначити максимальну кліку, а порівняння з бібліотечними методами демонструє переваги та обмеження кожного підходу, формуючи повну картину продуктивності алгоритму Bron–Kerbosch на практиці.