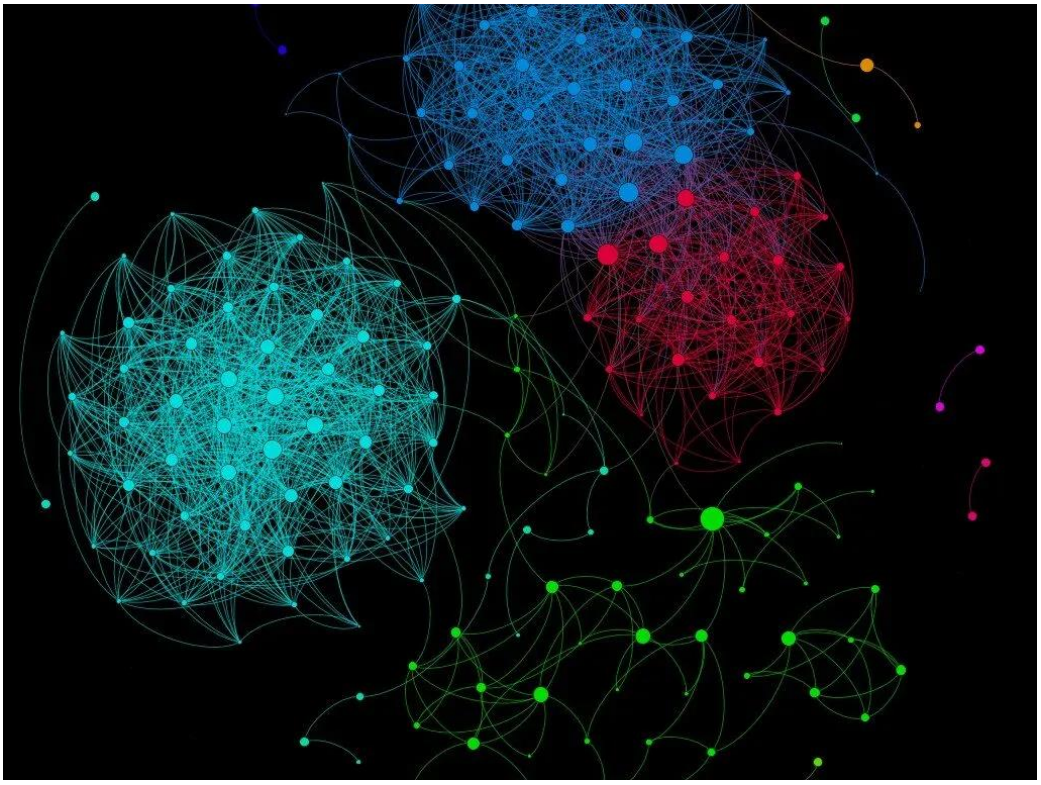


Побудова матриці досяжності графу за допомогою DFS та BFS

ОМЕЛЬЧЕНКО ЕМІР
СЕГЕДА МИХАЙЛО

Дискретна Математика | 21.04.2024

Зміст

Формальний опис алгоритму

2.1 Постановка проблеми

У теорії графів фундаментальною задачею є визначення досяжності графа, тобто встановлення того, чи існує шлях між двома вершинами. Для неорієнтованого графа **G = (V, E)**, де **V** - множина вершин, а **E** - множина ребер, метою є побудова матриці досяжності **R**. У цій матриці запис ***R[x][y] = 1*** вказує на існування шляху з вершини **x** у вершину **y**, а ***R[x][y] = 0*** означає, що такого шляху не існує.

Задача полягає в тому, щоб ефективно обчислити цю матрицю за допомогою двох класичних алгоритмів обходу графа: **Breadth-First Search (BFS)** та **Depth-First Search (DFS)**. Алгоритми повинні працювати з неорієнтованими графами і створювати матрицю, яка стисло викладає інформацію про зв'язність.

2.2 Вхідні та вихідні дані

Як для BFS, так і для DFS вхідні дані можуть бути надані у двох формах:

1) **BFS (приклад 1 та 2):**
    i) Вхідні дані (приклад 1): Матриця суміжності, що представляє граф G.
    ii) Вихід (випадок 1): Матриця досяжності R.

iii) Вхідні дані (приклад 2): Список суміжності графа G.
iv) Вихідні дані (приклад 2): Матриця досяжності R.

2) **DFS (приклади 3 та 4):**
   i) Вхідні дані (приклад 3): Матриця суміжності графа G.
   ii) Вихідні дані (приклад 3): Матриця досяжності R.
   iii) Вхідні дані (приклад 4): Список суміжності графа G.
   iv) Вихідні дані (приклад 4): Матриця досяжності R.

Вибір способу подання вхідних даних (матриця суміжності чи список) безпосередньо впливає на складність та ефективність алгоритмів, що і є предметом дослідження у наших експериментах.

2.3 Опис алгоритму псевдокодом

Алгоритми можна описати наступним чином:

I. BFS з матрицею суміжності (випадок 1):

```
function BuildReachabilityMatrixUsingBFS(adjacencyMatrix):
    let vertices = number of vertices in adjacencyMatrix
    let reachabilityMatrix be a matrix with dimensions [vertices][vertices] initial

    for each vertex i from 0 to vertices - 1:
        let queue be an empty queue
        let visited be an array with vertices elements, initialized to false
        enqueue i into queue
        set visited[i] to true

        while queue is not empty:
            let currentVertex be dequeued from queue
            for each neighbor in adjacencyMatrix[currentVertex]:
                if neighbor is not visited:
                    enqueue neighbor into queue
                    set visited[neighbor] to true
                    set reachabilityMatrix[i][neighbor] to 1
    return reachabilityMatrix
```

II. BFS зі списком суміжності (випадок 2): Подібно до випадку 1, але ітерація відбувається над списком суміжності, а не над матрицею.

III. DFS з матрицею суміжності (випадок 3):

```
function BuildReachabilityMatrixUsingDFS(adjacencyMatrix):
    let vertices = number of vertices in adjacencyMatrix
    let reachabilityMatrix be a matrix with dimensions [vertices][vertices] initia

    for each vertex i from 0 to vertices - 1:
        let stack be an empty stack
        let visited be an array with vertices elements, initialized to false
        push i onto stack
        set visited[i] to true

        while stack is not empty:
            let currentVertex be popped from stack
            for each neighbor in adjacencyMatrix[currentVertex]:
                if neighbor is not visited:
                    push neighbor onto stack
                    set visited[neighbor] to true
                    set reachabilityMatrix[i][neighbor] to 1
    return reachabilityMatrix
```

IV. DFS зі списком суміжності (випадок 4): Аналогічно до випадку 3, але ітерація відбувається над списком суміжності, а не над матрицею.

2.4 Теоретичні оцінки складності (Теоретичні оцінки складності)

I. **BFS (випадок 1 та 2):** Складність BFS дорівнює **O(V + E)**, де V - кількість вершин, а E - кількість ребер. При використанні списків суміжності цей алгоритм оптимально працює для розріджених графів, де E набагато менше **V²**. При використанні матриці суміжності складність залишається такою ж, але на практиці алгоритм може працювати повільніше через необхідність перевірки кожного елемента матриці.

II. **DFS (приклади 3 і 4):** DFS також демонструє складність **O(V + E)** за тих самих умов. Однак, він має тенденцію бути більш ефективним для пошуку шляхів у графах, де такі шляхи є довгими та звивистими, оскільки його підхід з поверненням назад добре підходить для таких сценаріїв.

Наведені значення складності є верхньою межею **(worst-case-scenario)**; фактичний час, що витрачається, може бути меншим залежно від структури графа та особливостей реалізації.

Коментарі щодо програмної реалізації

Під час розробки алгоритмів побудови матриці досяжності за допомогою BFS та DFS було прийнято декілька рішень для забезпечення ефективності та зрозумілості коду. Вибір структур і типів даних є критично важливим, оскільки

вони безпосередньо впливають на продуктивність і зручність використання алгоритмів.

Масив для матриць суміжності та досяжності: Масиви були обрані як основна структура даних для представлення матриць суміжності та досяжності через їх характеристики прямого доступу. Масив забезпечує час доступу O(1), що має вирішальне значення для частого доступу та оновлень, які вимагаються алгоритмами BFS та DFS. Цей постійний час доступу дозволяє швидко перевіряти та оновлювати ребра між вершинами та існування шляхів у матриці досяжності.

Список масивів для списків суміжності: Для представлення списків суміжності було використано список масивів. Цей вибір був мотивований необхідністю динамічної зміни розміру та можливістю ефективного перебору сусідів заданої вершини. На відміну від зв'язного списку, де пошук сусідів потребує O(n) часу, масив скорочує цей час до O(k), де k - степінь вершини, часто набагато менша за n - кількість вершин.

Структура класу Graph

Клас Graph структуровано так, щоб інкапсулювати представлення графа, будь то матриця суміжності або ж список, забезпечивши гнучкість. Закриті (private) властивості ***_adjacencyMatrix*** та ***_adjacencyList*** зберігають представлення, а методи доступу “розпаковують” їх у контрольований спосіб. Кількість вершин, що зберігається в ***_vertices***, впливає на створення матриць і списків, забезпечуючи основу для всіх інших методів.

Методи для ребер та сусідів:

Методи **AddEdge** та **RemoveEdge** керують зв'язками між вершинами, безпосередньо маніпулюючи структурою суміжності. Коли ребра додаються або видаляються, матриці суміжності та досяжності оновлюються, щоб негайно відобразити ці зміни, забезпечуючи узгодженість даних.

Перетворення між представленнями:

Було реалізовано можливість перетворення між матрицею суміжності та списком суміжності. Ця гнучкість дозволяє порівнювати вплив різних представлень на продуктивність алгоритмів, дотримуючись експериментальних цілей проекту.

Черги та стеки:

Вибір черги в BFS зумовлений характером алгоритму обходу рівня, який обробляє вершини пошарово. Черга природно підтримує це, дозволяючи обробляти вершини в порядку їх виявлення. І навпаки, стек був обраний для DFS, щоб полегшити зворотний шлях, який є основним компонентом обходу в глибину. Використовуючи стек, алгоритм природним чином слідує шляхом до своєї глибини, перш ніж розмотувати, щоб дослідити альтернативні шляхи.

Таким чином, структури і типи даних були ретельно підібрані, щоб відповідати вимогам і очікуваній поведінці алгоритмів. Вони були реалізовані таким чином,

щоб досягти балансу між ефективністю роботи та зрозумілістю програмного потоку, що дозволяє проводити точний та глибокий експериментальний аналіз.

Посилання на GitHub репозиторій

Посилання: https://github.com/WhatIsYourAlibi/Discrete-Math-Project

Опис експериментальної частини

4.1 Схема експериментів, вибір параметрів

Експериментальний компонент цього проекту був ретельно структурований для оцінки часової складності алгоритмів BFS та DFS при побудові матриць досяжності для графів різного розміру та щільності. Розміри графів варіювалися від 20 до 200 вершин, з кроком, що відображає масштаб, на якому зміни складності стають помітними. Експеримент розглядав графи з такою кількістю вершин: **sizes** = { **20, 40, 60, 80, 100, 120, 140, 160, 180, 200** }.

Також було обрано п'ять різних рівнів щільності - від розрідженої до щільної, щоб забезпечити широкий спектр типів графів: **densities** = { **0.1, 0.25, 0.4, 0.55, 0.7}.**

Щільність була обрана на основі ймовірності існування ребра між будь-якими двома вершинами. Вони були визначені як відношення кількості ребер до кількості можливих ребер, причому конкретні значення були визначені для того, щоб підкреслити продуктивність в розріджених, помірно щільних і щільних сценаріях.

Для кожної комбінації розміру та щільності було проведено **100 ітерацій,** щоб зменшити мінливість, спричинену різними технічними факторами, та забезпечити статистичну значущість. Кожен граф генерувався випадковим чином перед виконанням алгоритмів, щоб запобігти впливу процесу генерації графа на час роботи алгоритму.

4.2 Результати експериментів у вигляді таблиць

Результати були ретельно записані, зосереджуючись на середньому часі виконання обох алгоритмів для кожного розміру та щільності графа. Ці результати були зведені в таблиці для наочної візуалізації та порівняння. Таблиці містять стовпчики для розміру графа, щільності, середнього часу для BFS з матрицею суміжності, середнього часу для BFS зі списком суміжності, середнього часу для DFS з матрицею суміжності та середнього часу для DFS зі списком суміжності. Дані подані у файлі: “experiment_results.tsv” (у GitHub репозиторії).

4.3 Аналіз отриманих результатів

В результаті аналізу отриманих даних було виявлено декілька закономірностей. Для розріджених графів списки суміжності постійно перевершували матриці суміжності, що підтверджує теоретичний аналіз часової складності. Це пояснюється здатністю списків компактно представляти розріджені зв'язки, зменшуючи непотрібні обчислення.

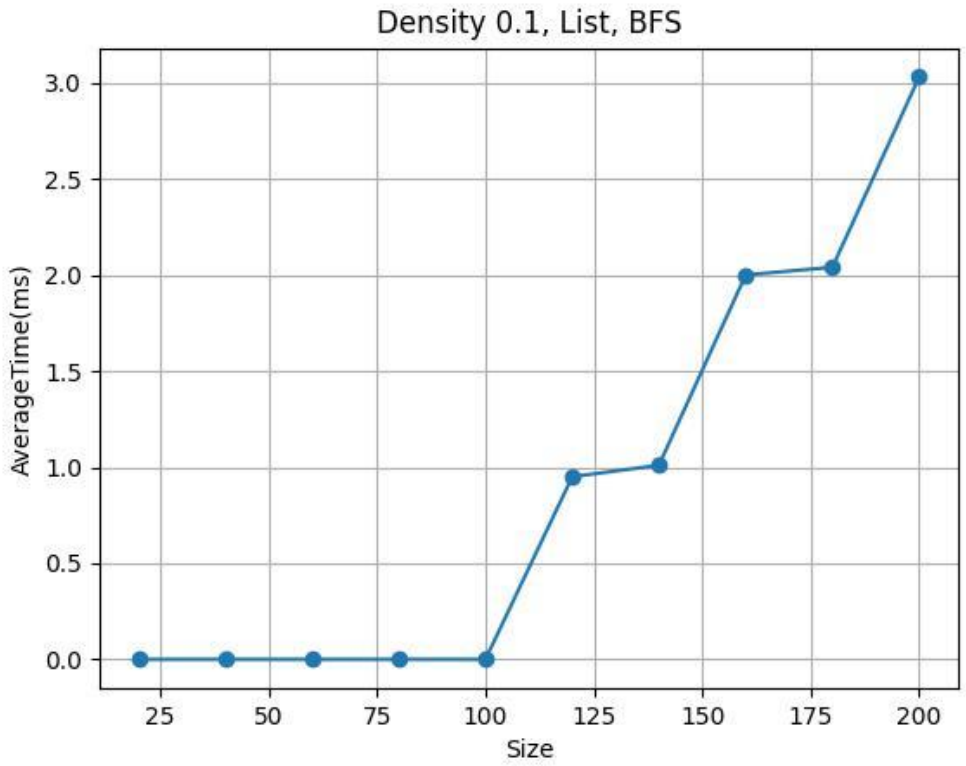
Зі збільшенням щільності графа розрив у продуктивності між цими двома способами представлення скорочується, причому матриці суміжності іноді пропонують конкурентну або вищу продуктивність, особливо в графах меншого розміру, де накладні витрати на перевірку кожного потенційного ребра стають менш значними порівняно з накладними витратами на обхід по списку.

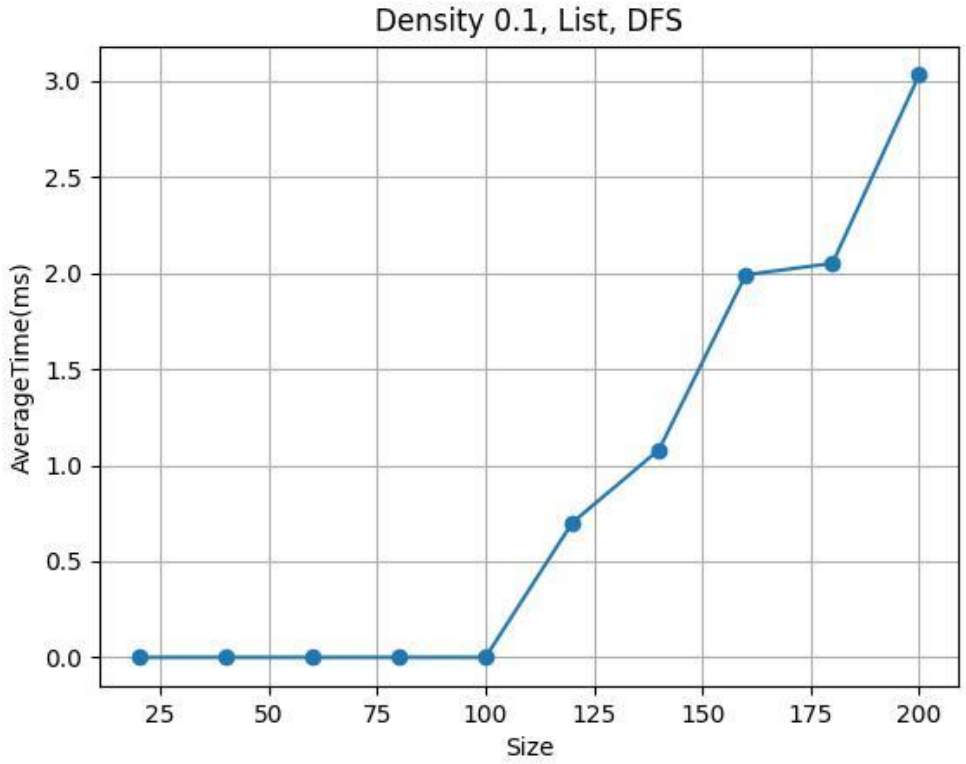
DFS показав помітну різницю в продуктивності порівняно з BFS при використанні списків суміжності на більш щільних графах, що, ймовірно, пов'язано з орієнтованим на глибину обходом, який може призвести до більш швидкого виявлення досяжності для певних структур графа.

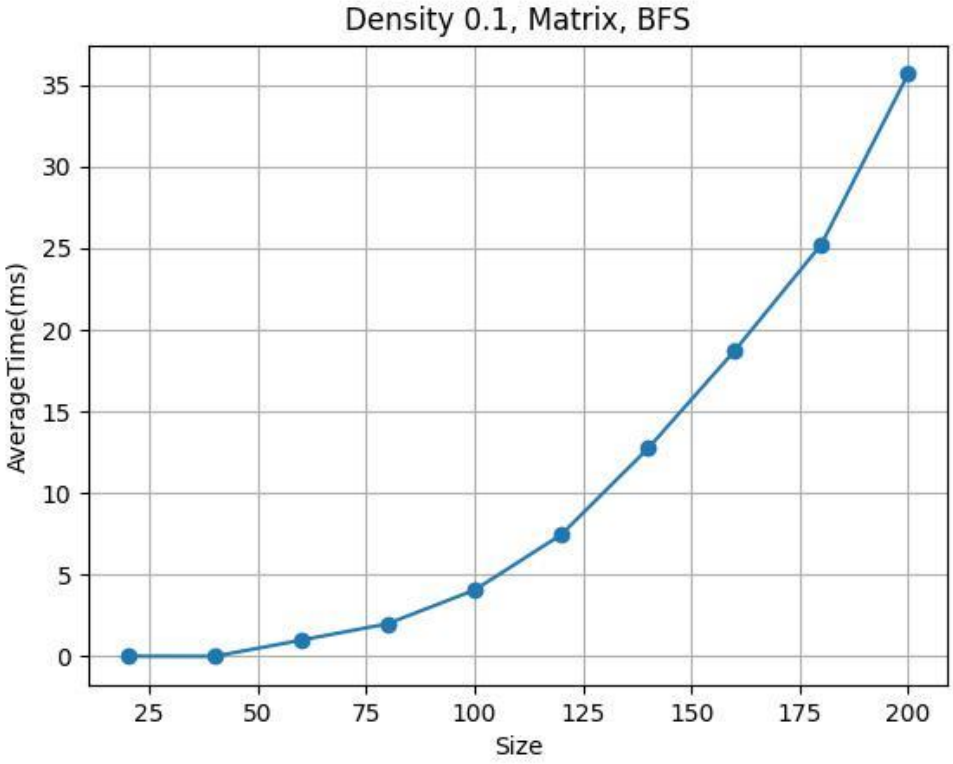
З точки зору початкової продуктивності, виміряний час узгоджувався з теоретичною складністю **O(V + E)** для обох алгоритмів. Однак на фактичну продуктивність впливали такі фактори, як мова реалізації, розміщення структури даних у пам'яті та кешування процесора, які не були враховані в теоретичній моделі.

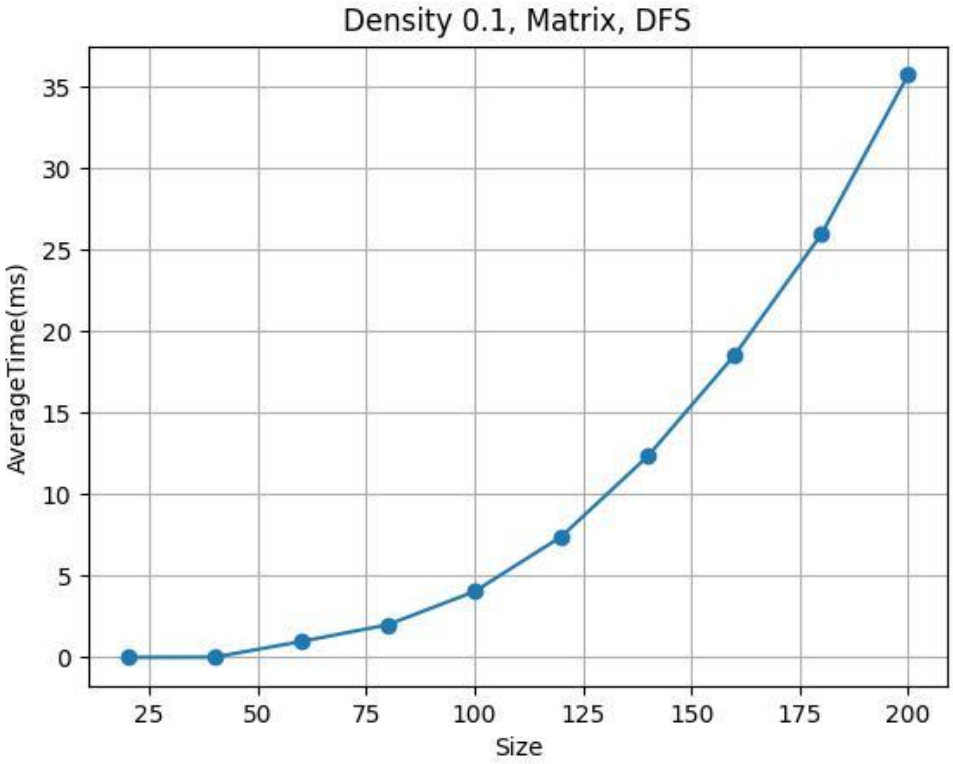
Цей аналіз демонструє необхідність врахування як теоретичної, так і емпіричної продуктивності при оцінці алгоритмів. Він також висвітлює вплив представлення даних на ефективність алгоритму, підкреслюючи важливість вибору відповідних структур даних на основі специфічних характеристик і вимог.

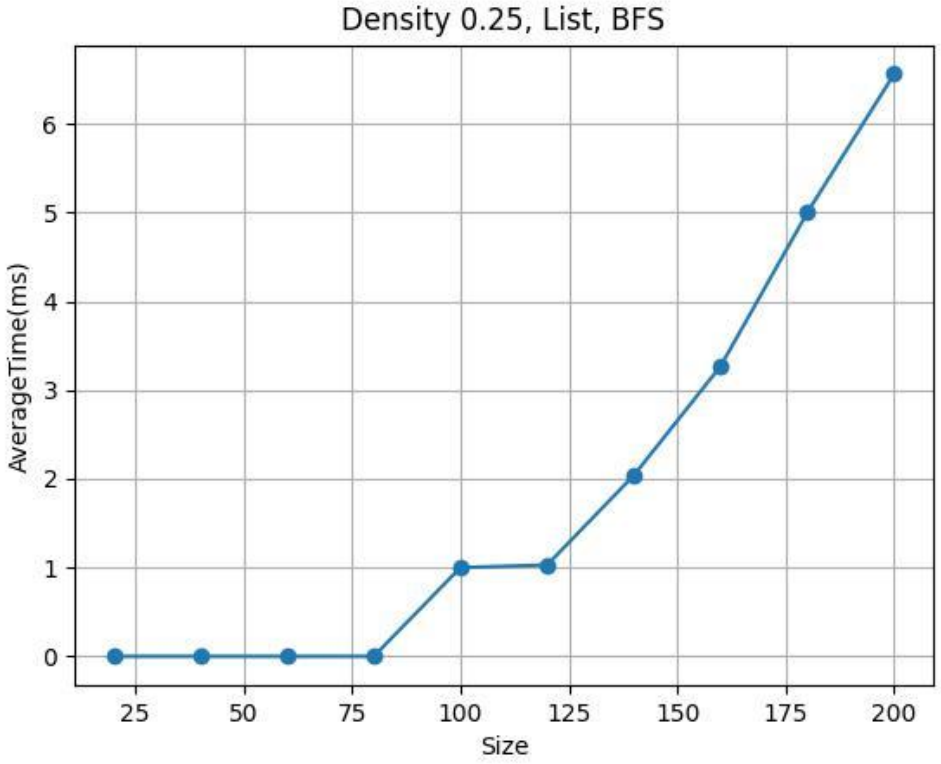
Графічне представлення отриманих результатів

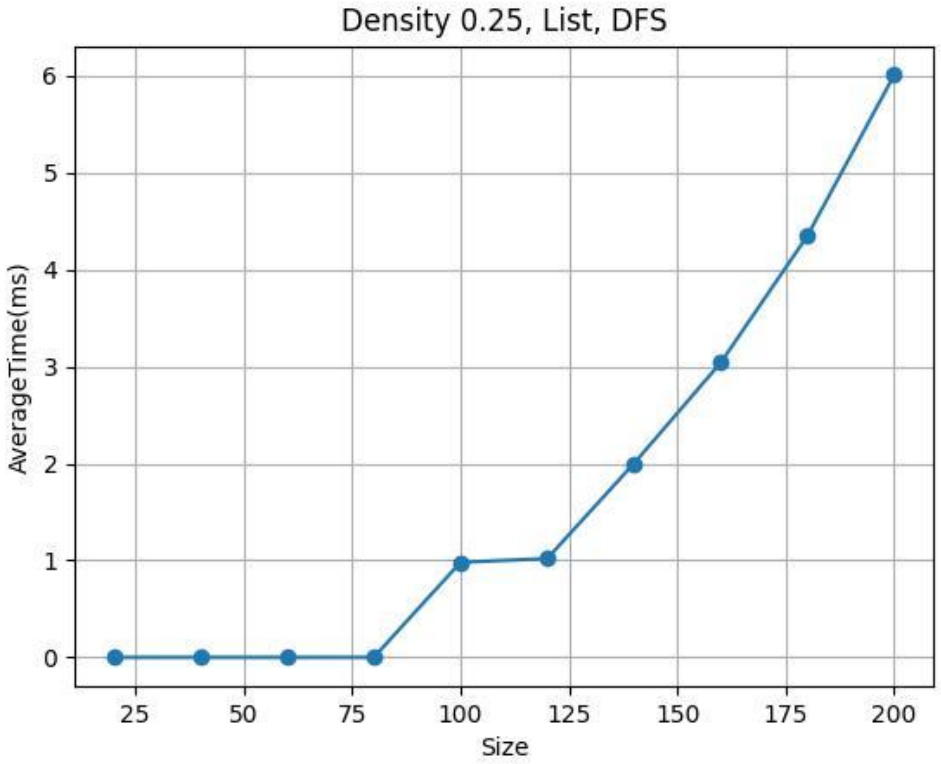


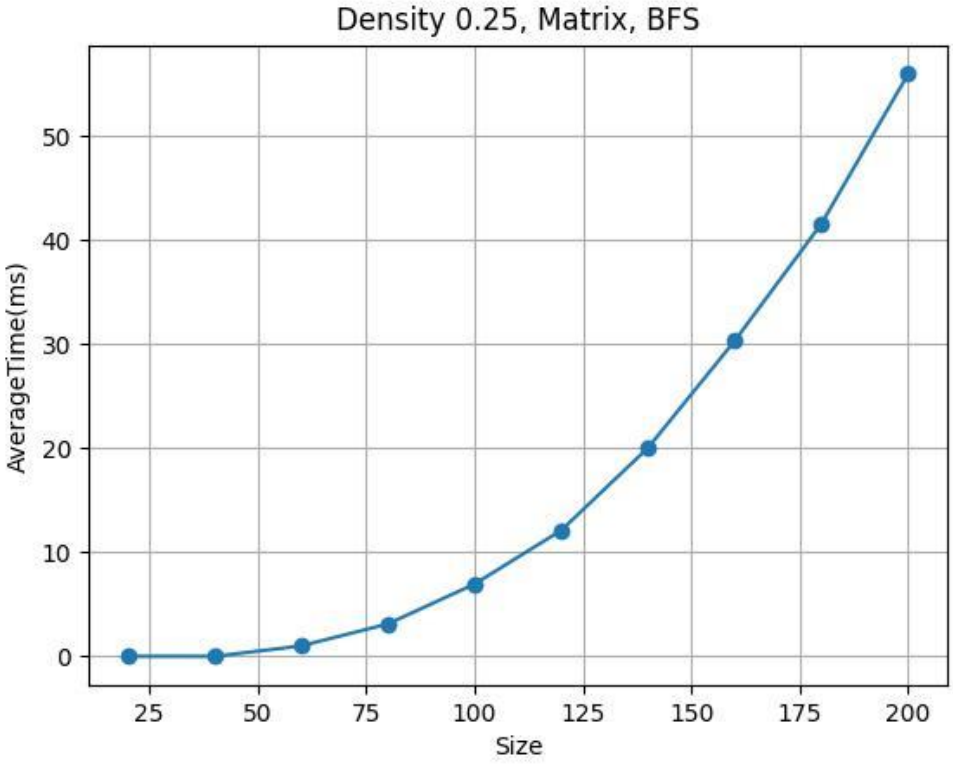


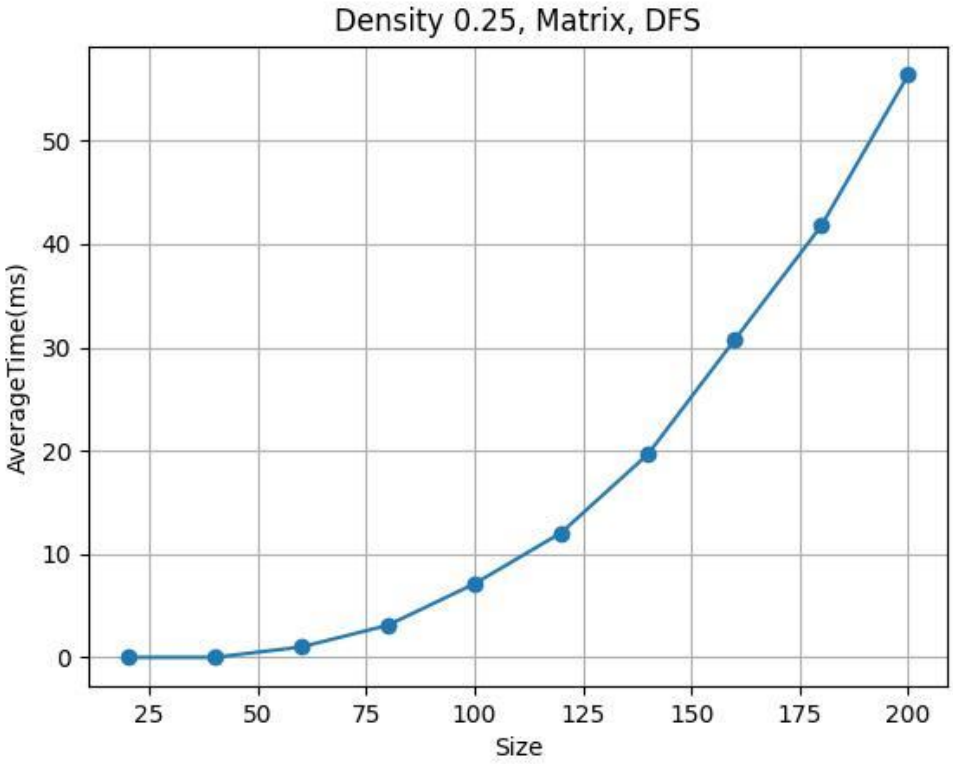

Density 0.1, Matrix, BFS
AverageTime(ms)
Size
0
5
10
15
20
25
30
35
25
50
75
100
125
150
175
200

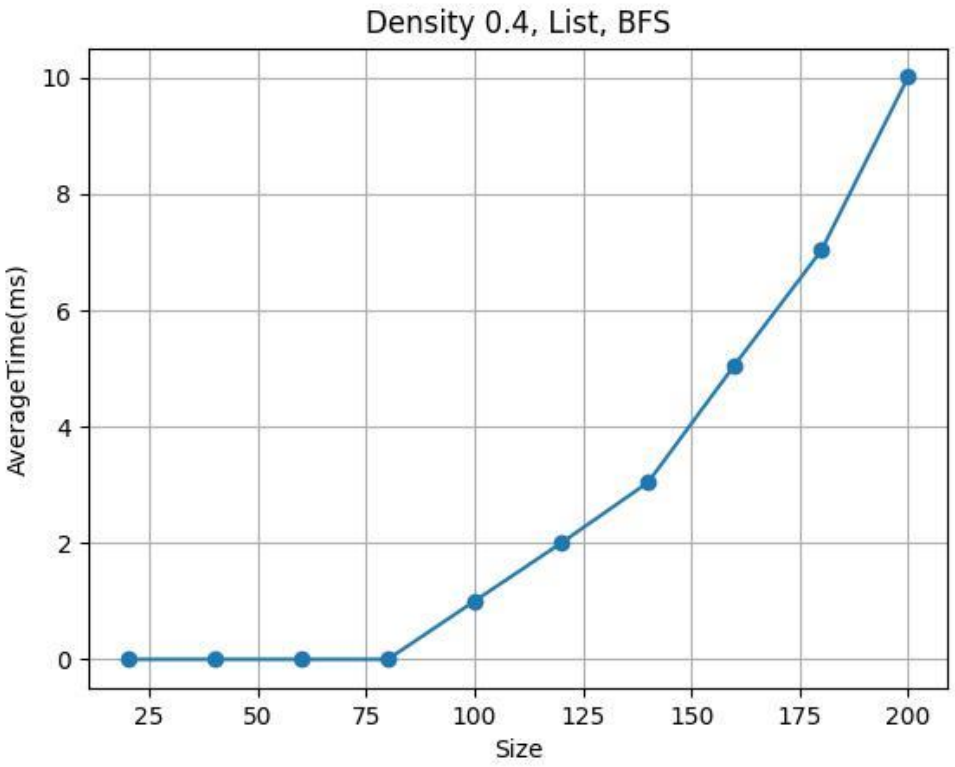

Density 0.1, Matrix, DFS
AverageTime(ms)
Size
0
5
10
15
20
25
30
35
25
50
75
100
125
150
175
200

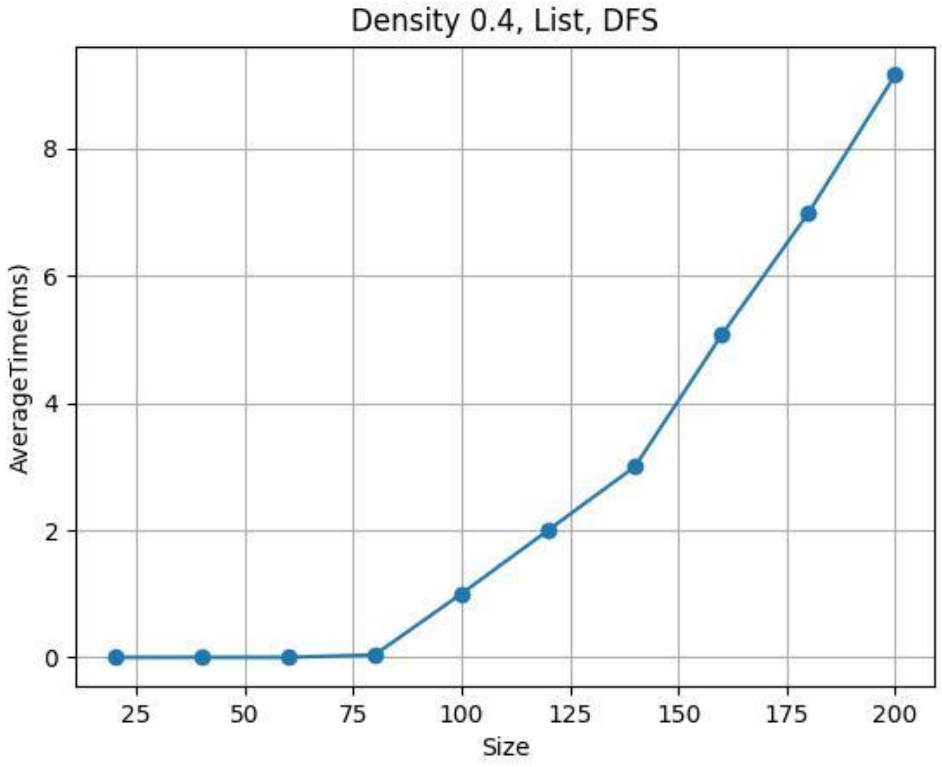

Density 0.25, List, BFS
AverageTime(ms)
Size
0
1
2
3
4
5
6
25
50
75
100
125
150
175
200

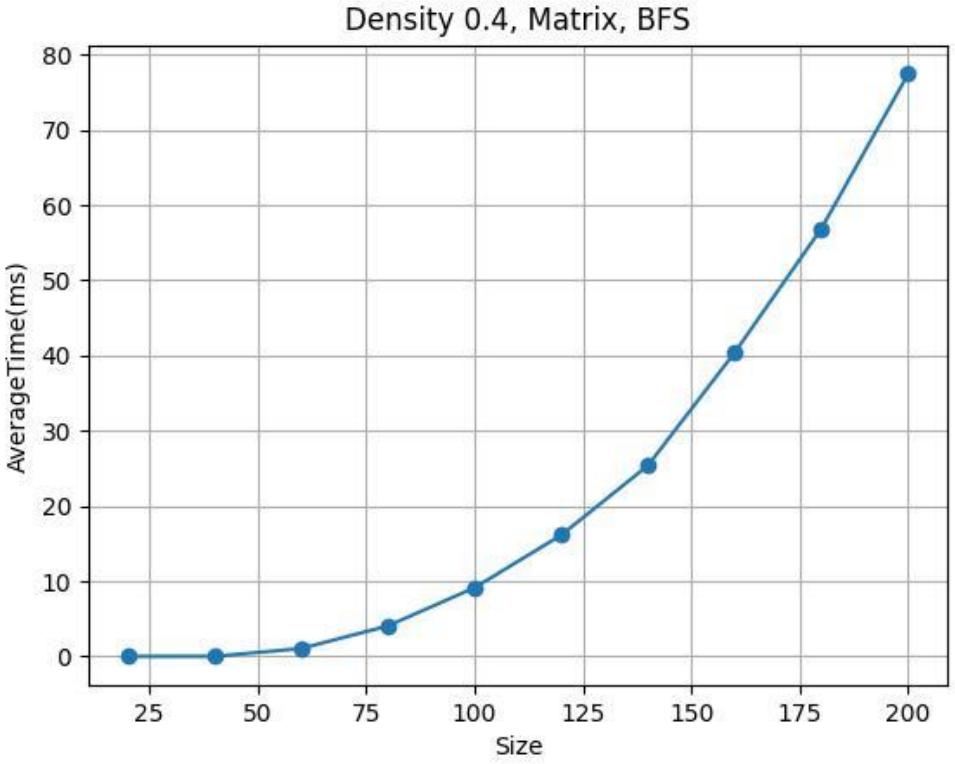

Density 0.25, List, DFS
AverageTime(ms)
Size
0
1
2
3
4
5
6
25
50
75
100
125
150
175
200

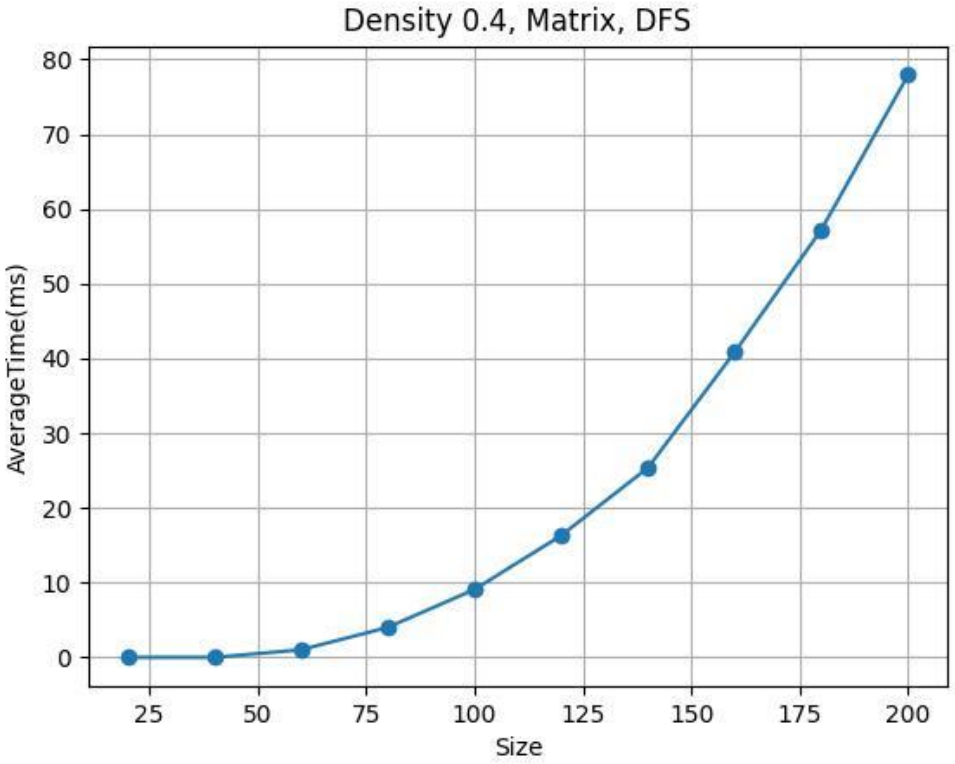

Density 0.25, Matrix, BFS
AverageTime(ms)
0
10
20
30
40
50
25
50
75
100
125
150
175
200
Size

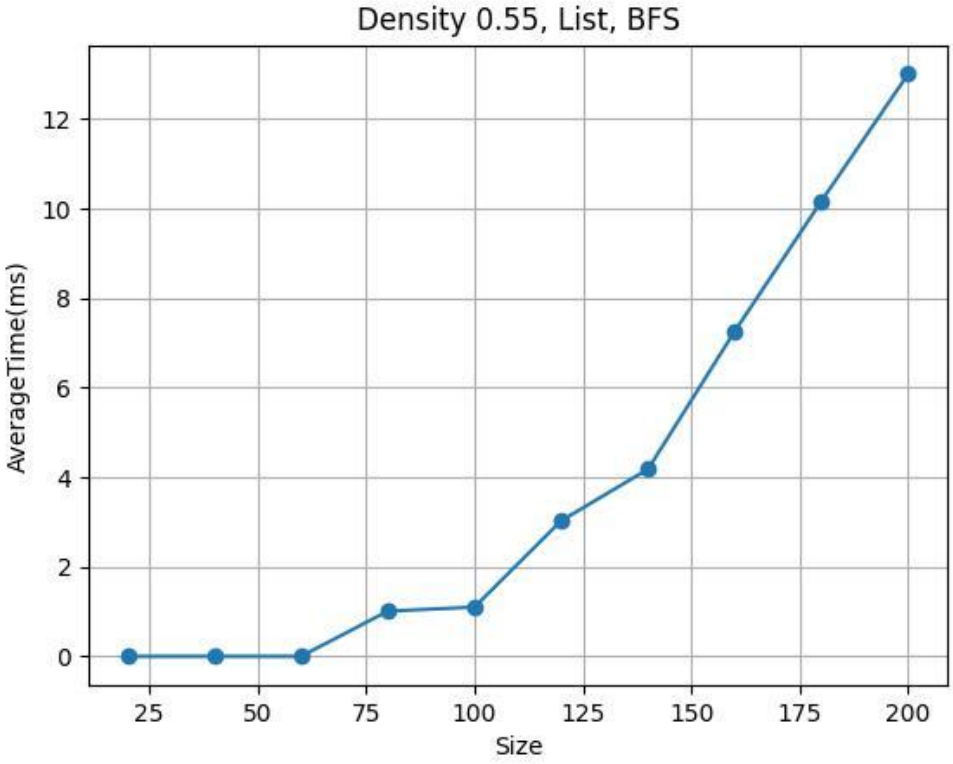

Density 0.25, Matrix, DFS
AverageTime(ms)
0
10
20
30
40
50
25
50
75
100
125
150
175
200
Size

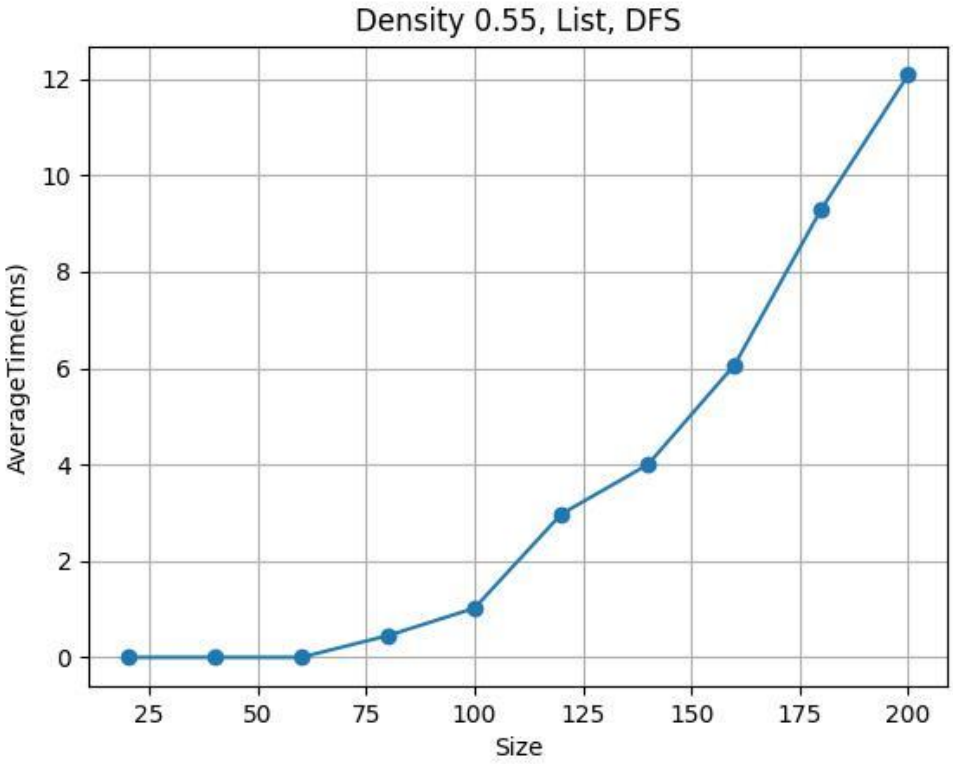

Density 0.4, List, BFS
AverageTime(ms)
Size
0
2
4
6
8
10
25
50
75
100
125
150
175
200

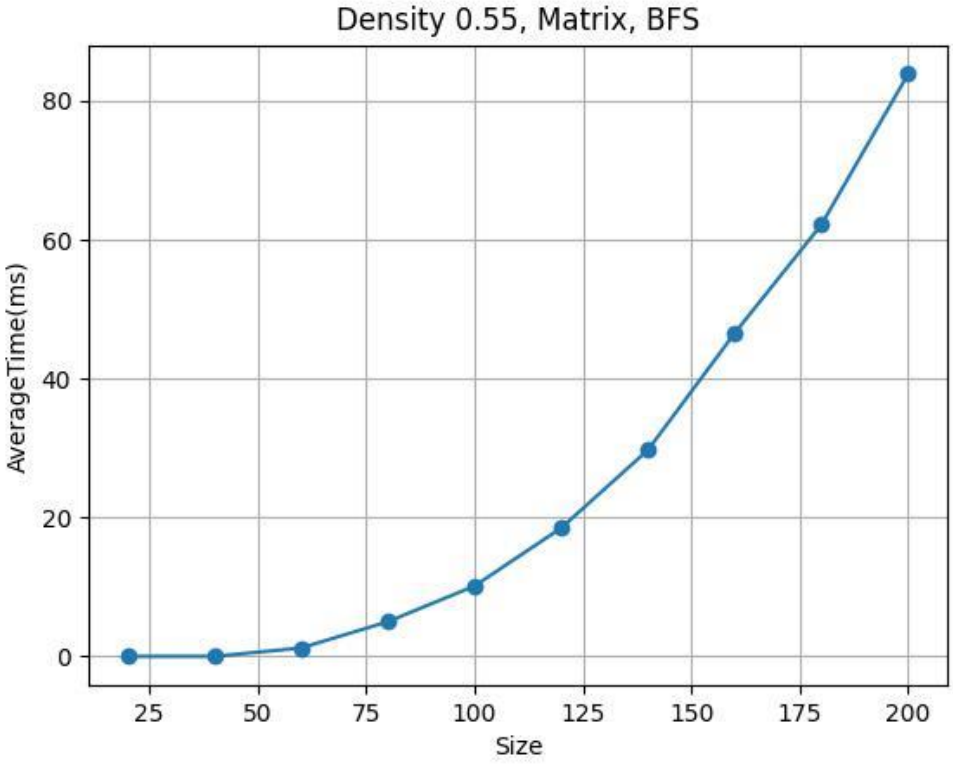

Density 0.4, List, DFS
AverageTime(ms)
Size
0
2
4
6
8
25
50
75
100
125
150
175
200

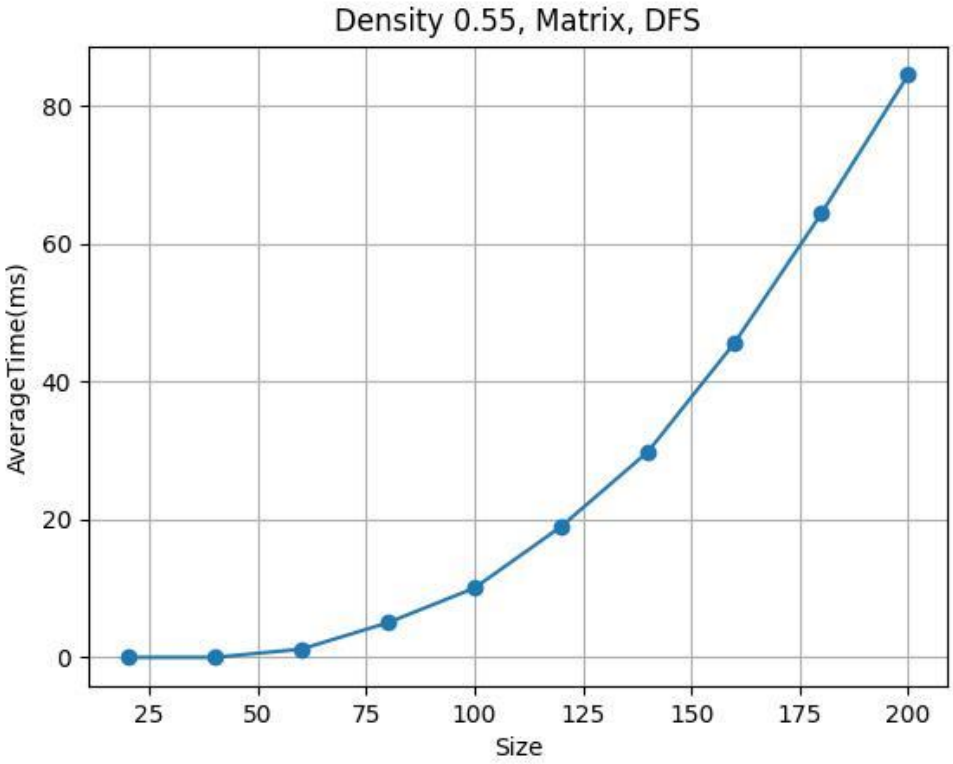

Density 0.4, Matrix, BFS
AverageTime(ms)
Size
0
10
20
30
40
50
60
70
80
25
50
75
100
125
150
175
200

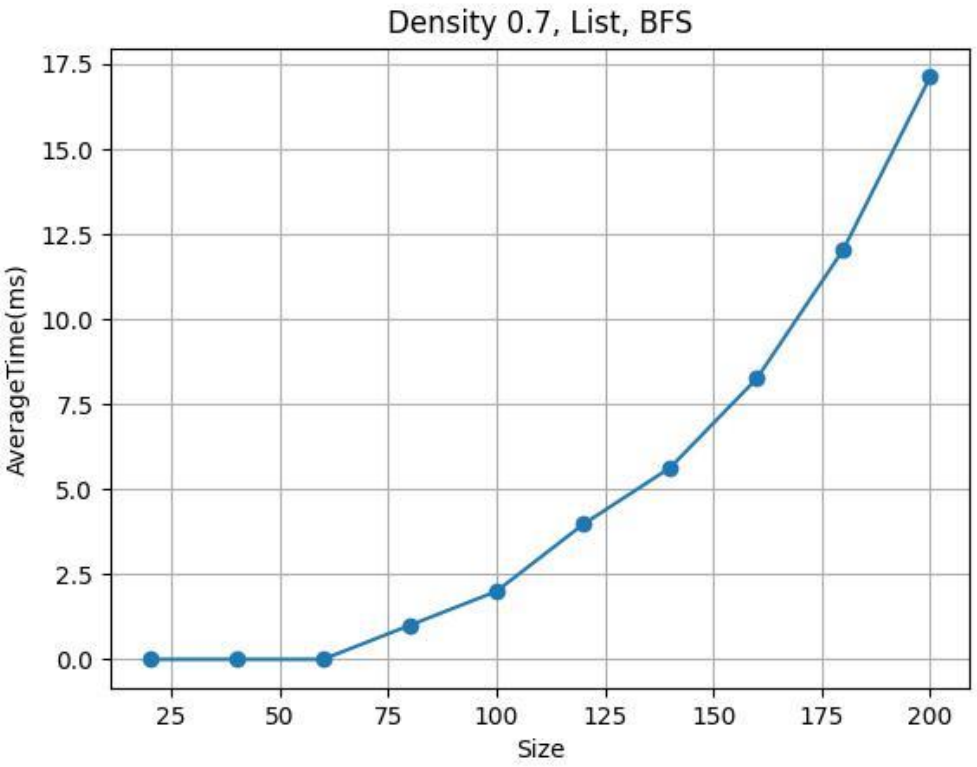

Density 0.4, Matrix, DFS
AverageTime(ms)
Size
0
10
20
30
40
50
60
70
80
25
50
75
100
125
150
175
200

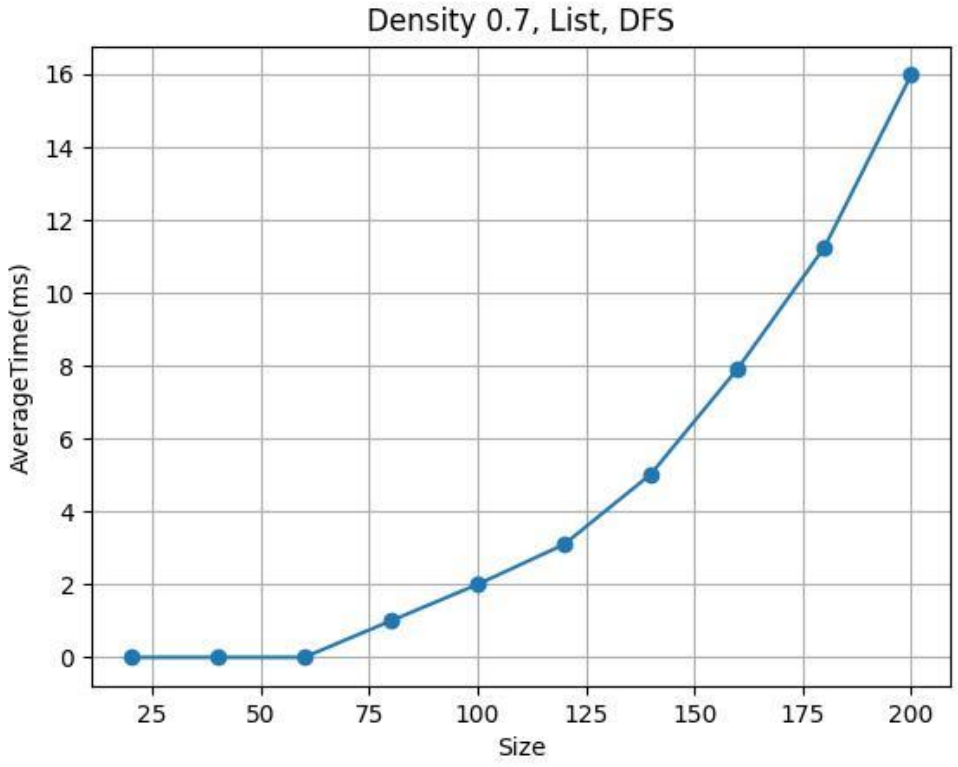

Density 0.55, List, BFS
AverageTime(ms)
Size
0
2
4
6
8
10
12
25
50
75
100
125
150
175
200

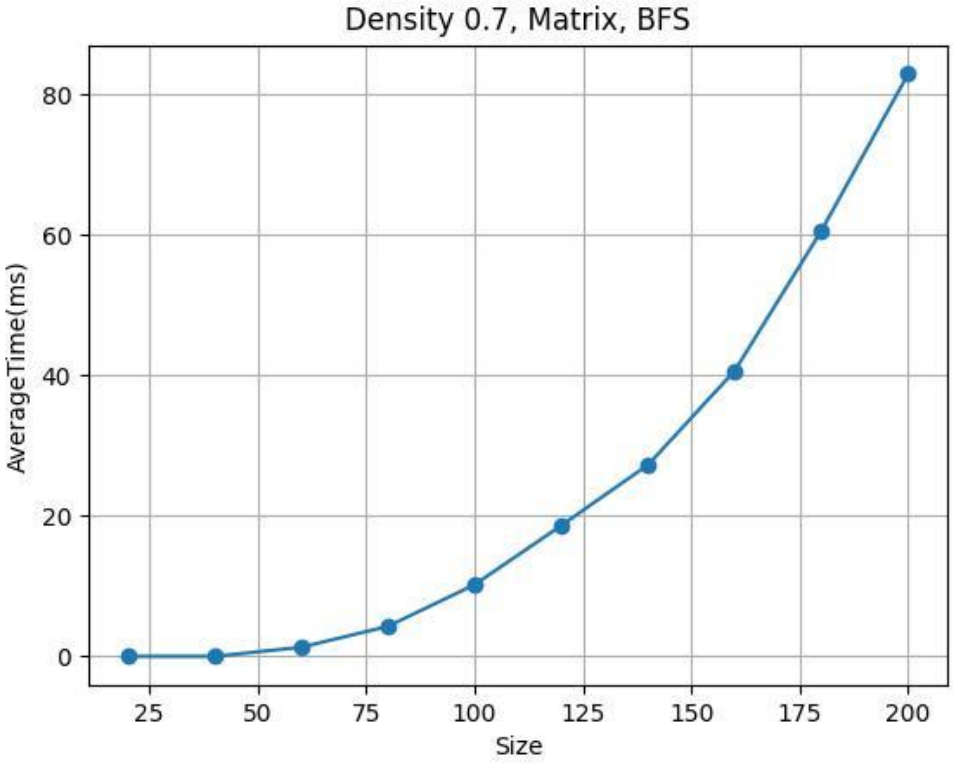

Density 0.55, List, DFS
AverageTime(ms)
Size
0
2
4
6
8
10
12
25
50
75
100
125
150
175
200

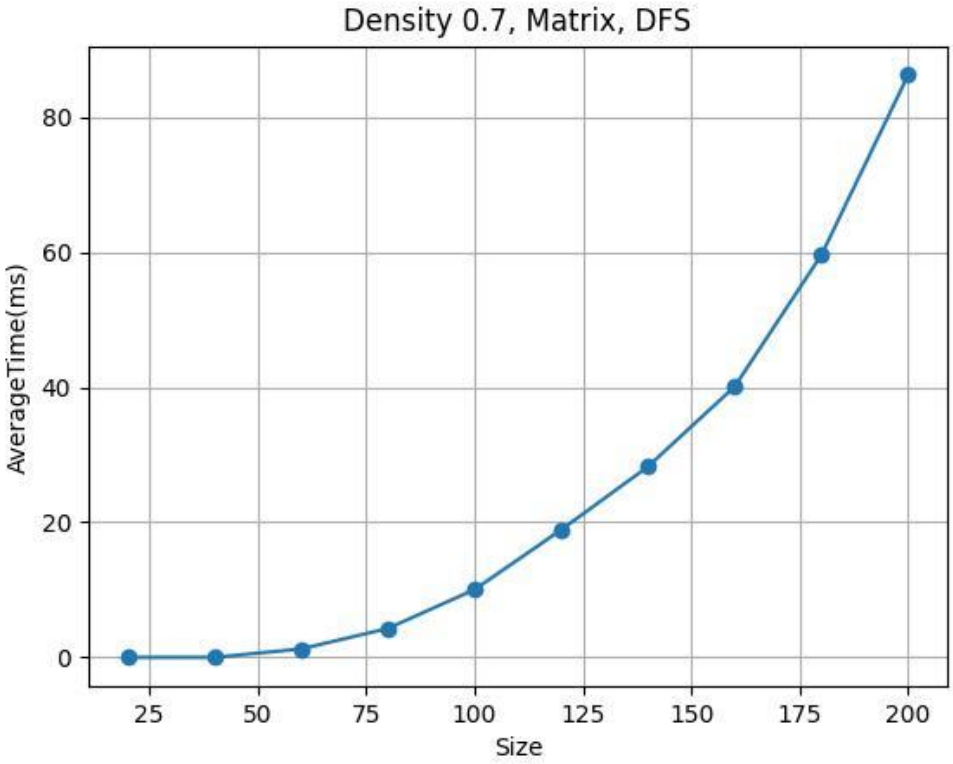

Density 0.55, Matrix, BFS
AverageTime(ms)
Size
0
20
40
60
80
25
50
75
100
125
150
175
200


Density 0.55, Matrix, DFS
AverageTime(ms)
Size
0
20
40
60
80
25
50
75
100
125
150
175
200


Density 0.7, List, BFS
AverageTime(ms)
Size
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
25
50
75
100
125
150
175
200


Density 0.7, List, DFS
AverageTime(ms)
Size
16
14
12
10
8
6
4
2
0
25
50
75
100
125
150
175
200


Density 0.7, Matrix, BFS
AverageTime(ms)
0
20
40
60
80
25
50
75
100
125
150
175
200
Size


Density 0.7, Matrix, DFS
AverageTime(ms)
0
20
40
60
80
25
50
75
100
125
150
175
200
Size

Розподіл роботи між членами команди

Омельченко Емір: Розробка та впровадження алгоритму

Емір зосередився на розробці та реалізації алгоритмів. Це включало написання початкових версій алгоритмів BFS і DFS, оптимізацію коду для підвищення продуктивності та забезпечення коректної роботи алгоритмів як для матриць суміжності, так і для списків.

Сегеда Михайло: Експериментальний дизайн та аналіз

Михайло відповідав за розробку експериментального фреймворку. Він розробив схему тестування, включаючи визначення розмірів і щільності графів, а також налаштування середовища для проведення численних ітерацій, необхідних для збору достатньої кількості даних. Після того, як дані були зібрані, Михайло провів аналіз, інтерпретував результати і порівняв їх з теоретичними очікуваннями.

Спільні зусилля: Тестування та документація

Обидва учасники тісно співпрацювали на етапі тестування. Вони разом працювали над визначенням граничних ситуацій для алгоритмів і перевіркою правильності результатів. Вони також розділили завдання документування проекту, де кожен учасник писав різні розділи звіту, а потім перевіряв роботу один одного на узгодженість і точність.

Розподіл праці був розроблений таким чином, щоб використовувати сильні сторони кожного учасника, забезпечуючи при цьому рівний внесок обох в основні аспекти проекту.

Висновки

По-перше, було підтверджено, що вибір структури даних суттєво впливає на ефективність графових алгоритмів. Список суміжності виявився більш ефективним для розріджених графів, тоді як матриця суміжності показала потенційні переваги в сценаріях щільних графів, особливо коли розмір графа був відносно невеликим. Це спостереження підкреслює необхідність враховувати природу даних графа при виборі алгоритму для практичного застосування.

По-друге, експериментальні результати проекту підтвердили теоретичну складність **O(V + E)** для алгоритмів BFS і DFS. Однак експерименти також висвітлили розбіжності, які можуть виникнути між теоретичними прогнозами і реальною продуктивністю, підкресливши важливість емпіричного тестування.

По-третє, реалізація обох алгоритмів в єдиному фреймворку продемонструвала універсальність BFS і DFS при застосуванні до різних представлень графів. Незважаючи на внутрішні відмінності між цими методами обходу, їх

адаптивність була очевидною, оскільки вони успішно застосовувалися як до матриць суміжності, так і до списків без значної модифікації основної логіки.

З точки зору командної роботи та управління проектами, проект проілюстрував цінність чіткої комунікації та справедливого розподілу завдань між членами команди.

Нарешті, проект сприяв глибшому розумінню алгоритмів обходу графів. Він забезпечив практичне розуміння їхньої поведінки в різних умовах.

Отримані результати не тільки збагатили наше розуміння цих алгоритмів, але й підкреслили важливість емпіричної перевірки в галузі аналізу алгоритмів.

Джерела

I. https://chat.openai.com/ - був використаний для оптимізації коду та в деяких випадках, як інструмент робити з документацією;
II. https://en.wikipedia.org/wiki/Breadth-first_search - література;
III. https://en.wikipedia.org/wiki/Depth-first_search - література;
IV. “**The algorithm design manual. Steven S.Skiena**” – література [V розділ].