Проєкт “Побудова матриці досяжності графу за допомогою DFS та BFS”

Github проєкту

Виконали:

Денис Кучерук Руслан Куртвелієв Даніель Бонд

Опис Алгоритму

Постановка задачі

Здійснити дослідження швидкості виконання алгоритмів BFS (Breadth-First Search) та DFS (Depth-First Search) при створенні матриць досяжності за різних представлень графів (матриця/список суміжності). Проаналізувати та графічно продемонструвати як розмір та щільність графів корелює з часом виконання.

Вхідні та вихідні дані

Вхідні дані:

n	*кількість вершин у графі*
density	*щільність графа (%)*
method	*обраний метод обходу (BFS або DFS)*
rep_type	*тип представлення графа (matrix або list)*
iterations	*кількість ітерацій для кожного параметра*

Вихідні дані:

Матриця досяжності для кожної пари вершин.

Опис алгоритму псевдокодом (Джерело для BFS; Джерело для DFS)

BFS (Рекурсивне формулювання):

```
BFS(start_node, goal_node) {
  return BFS'({start_node}, ∅, goal_node);
}
BFS'(fringe, visited, goal_node) {
  if(fringe == ∅) {
    // цільовий вузол не знайдено
    return false;
  }
  if (goal_node ∈ fringe) {
    return true;
  }
  return BFS'({child | x ∈ fringe, child ∈ expand(x)} \ visited,
visited ∪ fringe, goal_node);
}
```

BFS (Ітеративне формулювання):

```
BFS(start_node, goal_node) {
 // спочатку список відвіданих вузлів порожній
 for(all nodes i) visited[i] = false;
 // починаючи з вузла-джерела
 queue.push(start_node);
 visited[start_node] = true;
 // поки черга не порожня
 while(! queue.empty() ) {
  // витягти перший елемент в черзі
  node = queue.pop();
  if(node == goal_node) {
    // перевірити, чи не є поточний вузол цільовим
    return true;
  }
  // всі наступники поточного вузла, ...
  foreach(child in expand(node)) {
   // ... які ще не були відвідані ...
   if(visited[child] == false) {
     // ... додати в кінець черги...
     queue.push(child);
     // ... і позначити як відвідані
     visited[child] = true;
   }
  }
 }
 // цільовий вузол недосяжний
 return false;
}
```

DFS (Рекурсивне формулювання):

```
def dfs(v):
    помітити v як відвідану
    обійти-в-прямому-порядку(v)
    для всіх ребер i інцидентних до v таких що i не відвідано
        dfs(i)
    обійти-в-зворотному-порядку(v)
```

DFS (Ітеративне формулювання):

```
def dfs(граф G):
  список L = порожній
  дерево T = порожнє
  обрати початкове ребро x
  пошук(x)
  поки L не порожній:
    видалити ребро (v, w) з голови L
    якщо w не відвідано:
      додати (v, w) до T
      пошук(w)

def пошук(вершина v):
  відвідати v
  для кожного ребра (v, w):
    додати ребро (v, w) на початок L
```

Теоретична оцінка складності

Алгоритм	Представлення графа	Часова складність	Просторова складність
BFS / DFS	Матриця суміжності	$O(n^2)$	$O(n^2)$
BFS / DFS	Список суміжності	$O(n + m)$	$O(n + m)$

де:

n — кількість вершин, m — кількість ребер.

Програмна реалізація

Програми написано мовою Python з використанням бібліотек *tqdm, multiprocessing, deque.* Основною структурою для представлення графа є двовимірний масив (n*n), який ми використовуємо як матрицю суміжності. Це допомагає нам швидше дізнатись про те, чи є ребро між двома вершинами (за O(1)), що є ключовим для вибраних алгоритмів. А для тестів реалізували перетворення з матриці в список.

Для алгоритму BFS ми розглядаємо кожну вершину як стартову (в різних процесах, які розділені індексом номерів вершин), і обходимо всі сусідні вершини по черзі. Черга зроблена через deque, де ми беремо кожну стартову вершину і позначаємо її, щоб потім видаляти найстаріші елементи черги через popleft(). Ну і також створений окремий масив visited, для того щоб ми не впадали в цикл (масив реалізований через булеан з індексацією до кожної вершини).

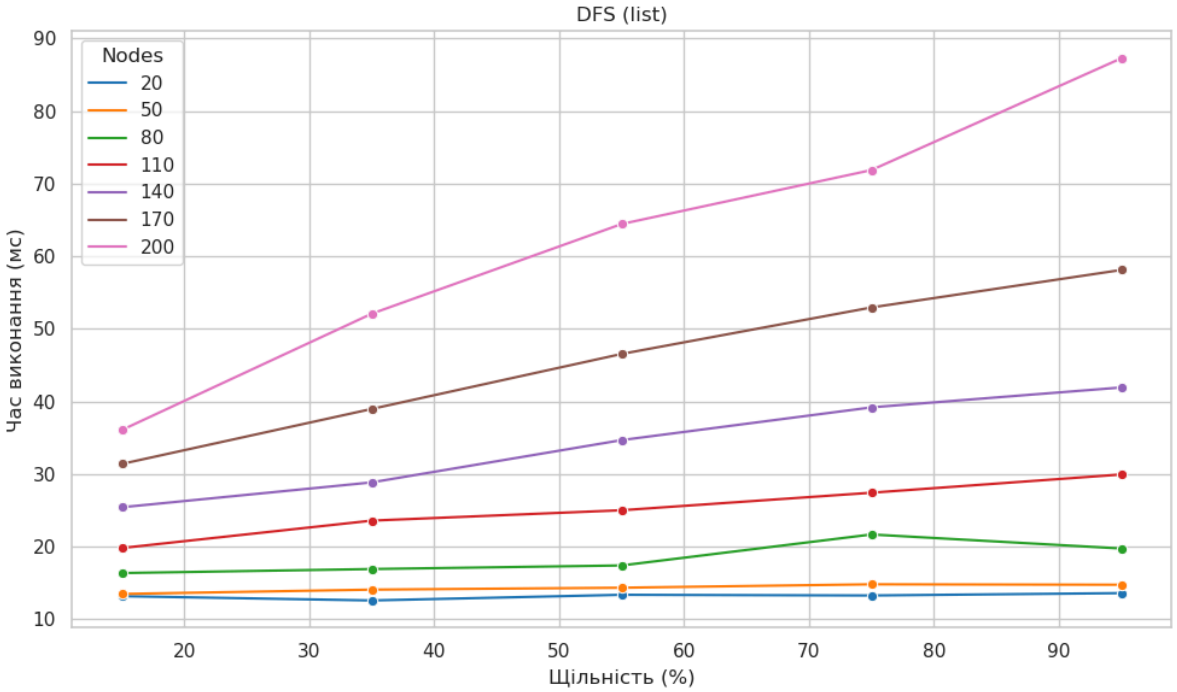
Для алгоритму DFS використано рекурсивний обхід, де ми ідемо до мінімальної вершини (в яку не входять ребра), а потім повертаємось назад. Це зроблено через окрему функцію explore, де з початку ми позначаємо стартову вершину як відвідану, потім знаходимо сусідів (що залежить від того, як ми подали, через матрицю чи список суміжності), і потім рекурсивно обходимо сусідів (що контролюється по факту через булеан, якщо сусід не відвіданий, до ми ідемо до нього. Далі просто повертаємось назад не викликаючи функцію explore (бектрекінг).

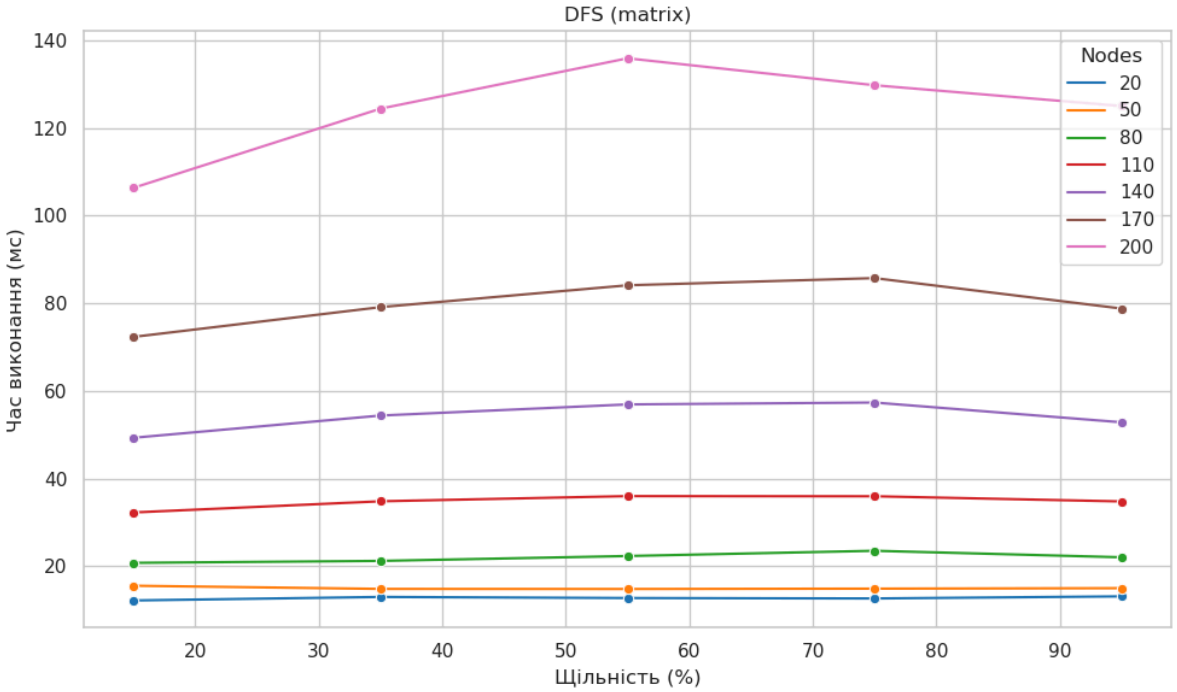
Для матриці досяжності ми створюємо різні задачі через tasks, де кожне завдання - обхід сусідів вершини графа. Паралелізація зроблена через multiprocessing.Pool для прискорення роботи на багатопотокових процесорах.

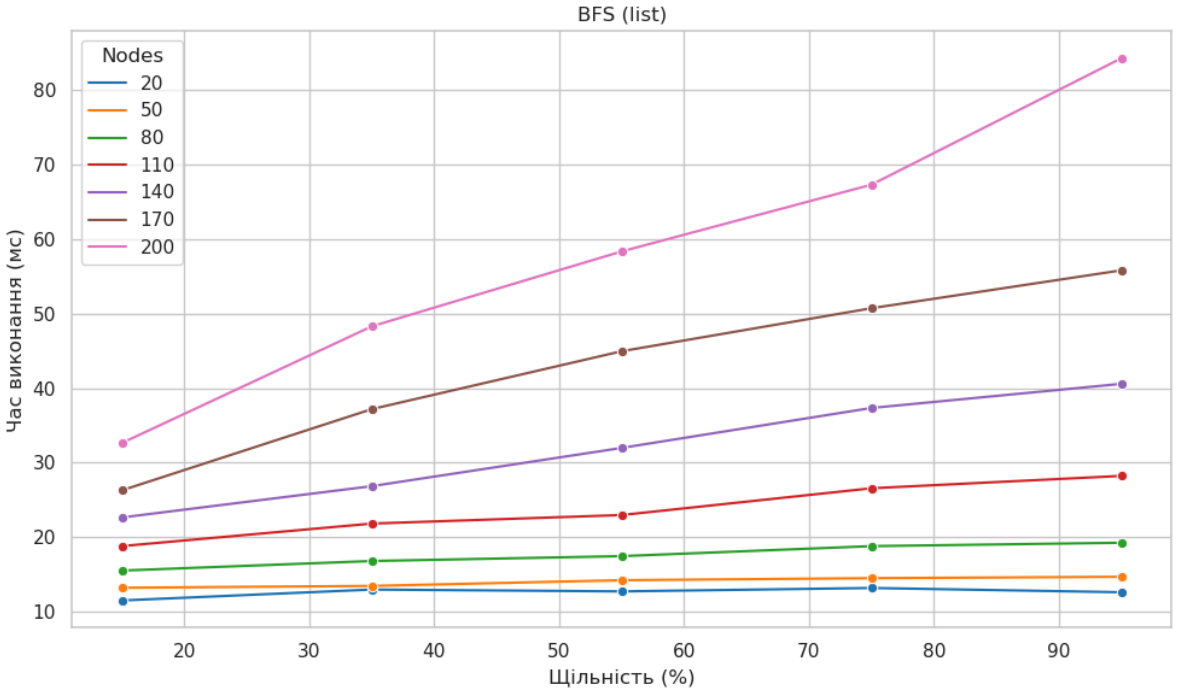
Експериментальна частина

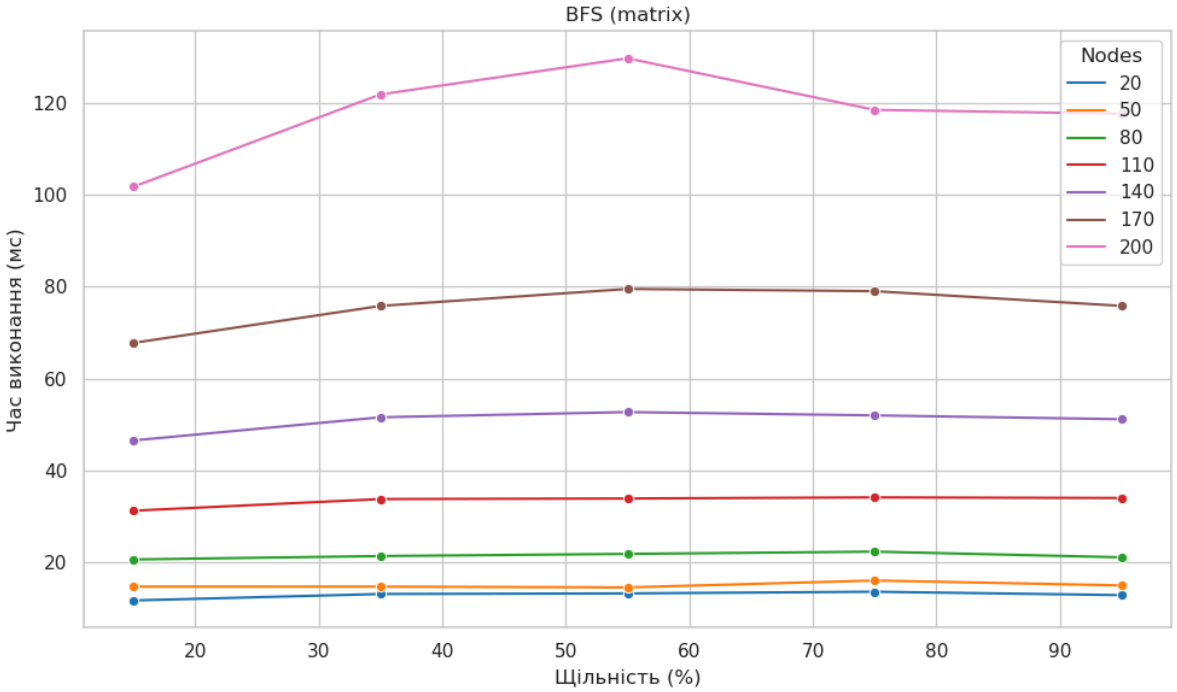
Параметри експериментів (схема)

Розмір	20	50	80	110	140	170	200
Щільність	15, 35, 55, 75, 95 - відсотків						
Ітерації	100						


DFS (list)
Nodes
20
50
80
110
140
170
200
Час виконання (мс)
Щільність (%)


DFS (matrix)
Nodes
20
50
80
110
140
170
200
Час виконання (мс)
Щільність (%)


BFS (list)
Nodes
20
50
80
110
140
170
200
Час виконання (мс)
Щільність (%)


BFS (matrix)
Nodes
20
50
80
110
140
170
200
Час виконання (мс)
Щільність (%)

Результати експериментів

Графіки та таблиці залежності сплинутого часу від розміру графів створені за параметрами експерименту. Таблиці збережені в форматі tsv, а графіки в png.

Аналіз результатів

- BFS зі списком суміжності працює найшвидше при розріджених графах
- DFS зі списками суміжності швидко працює на великих графах.
- Матриця суміжності сповільнює виконання алгоритмів за великої кількості вершин.

Порівняння форм представлення графу

Представлення графу	Розріджені графи	Щільні графи
Матриця суміжності	Повільно	Досить швидко
Список суміжності	Найефективніше	>щільність = <швидкість

Контрибуції виконавців проєкту

- Руслан - візіонер, “вайб-кодер”, теоретична частина роботи.
- Даніель - магістр ООП, критик, перевірка коду, оптимізація, створення звіту.
- Денис - царь обробки текстової інформації, оптимізація, створення звіту.

Висновки

- Продемонстрували роботу алгоритмів пошуку BFS та DFS

- Проаналізували залежність швидкості алгоритмів від розміру та щільності графів

- BFS за представлення списком суміжності та з розрідженими графами був найшвидшим. Чим більша щільність графу, тим менш доречним стає використання списку суміжності.