Міністерство освіти і науки України
Львівський національний університет імені Івана Франка
Факультет прикладної математики та інформатики
Кафедра програмування

Звіт
до лабораторної роботи №7
“Алгоритм Прима”

Підготував:
студент групи ПМІ-31
Процьків Назарій

Львів 2023

Завдання

Для зваженого зв'язного неорієнтованого графа G, використовуючи алгоритм Прима, з довільно заданої вершини a побудувати мінімальне кісткове дерево. Для різної розмірності графів та довільного вузла a порахувати час виконання програми без потоків та при заданих k потоках розпаралелення.

Теоретичні відомості

Граф — це структура, що складається з набору об’єктів, у якому деякі пари об’єктів у певному сенсі «пов’язані». Об’єкти відповідають математичним абстракціям, які називаються вершинами (також називаються вузлами або точками), а кожна з пов’язаних пар вершин називається ребром (також називається ланкою або лінією). Як правило, граф зображується у вигляді діаграми як набір точок або кіл для вершин, з’єднаних лініями або кривими для ребер. Графи є одним з об’єктів вивчення дискретної математики.

Графом G = (V, E) називають сукупність двох множин: скінченної непорожньої множини V вершин і скінченої множини E ребер, які з'єднують пари вершин. Ребра зображуються невпорядкованими парами вершин (u, v).

У графі можуть бути петлі — ребра, що починаються і закінчуються в одній вершині, а також повторювані ребра (кратні, або паралельні). Якщо в графі немає петель і кратних ребер, то такий граф називають простим. Якщо граф містить кратні ребра, то граф називають мультиграфом.

Ребра вважаються неорієнтованими в тому сенсі, що пари (u, v) та (v,u) вважаються одним і тим самим ребром.

Зваженим називають простий граф, кожному ребру e якого приписано дійсне число w(e). Це число називають вагою ребра e.

Хід роботи

Виконав цю лабораторну мовою програмування Python.

Реалізація:

Створив клас `GraphPoint:`

```python
class GraphPoint:
    def __init__(self, name):
        self.name = name
        self.passed = False
        self.weight = float('inf')
        self.connections = []
```

Створив клас `GraphConnection:`

```python
class GraphConnection:
    def __init__(self, first_point, second_point, weight):
        self.first_point = first_point
        self.second_point = second_point
        self.weight = weight
```

Створив клас `Graph:`

```python
class Graph:
    def __init__(self, point_num): ...

    1 usage
    def generate_points(self, point_num): ...

    1 usage
    def add_connections(self, new_point, point_num): ...

    4 usages
    @staticmethod
    def create_connection(first_point, second_point, weight): ...

    def __str__(self): ...
```

Він генерує рандомний граф, з допомогою методу `add_connections.`

Написав три функції:

```python
def prim_worker(args): ...


2 usages
def parallel_prim(graph, start, process_num): ...


2 usages
def sequential_prim(graph, start): ...
```

1. Допоміжна функція для паралельного алгоритму Прима.
2. Функція паралельного алгоритму Прима.
3. Функція послідовного алгоритму Прима.

Створив перший тест test1() щоб перевірити роботу алгоритму на малих графах:

```
def test1():
    size = 10
    graph = Graph(size)

    result_seq = sequential_prim(graph, 0)
    result_par = parallel_prim(graph, 0, 4)
    return result_seq == result_par
```

Тут створив граф на 10 вершин і порівняв чи результати роботи послідовного та паралельного алгоритмів збігаються.

Результат: True

Створив другий тест test2() щоб прогнати test1() багато разів і подивитись чи точно все працює правильно:

```
def test2():
    return all(test1() for i in range(1000))
```

Результат: True

Створив третій тест test3() щоб перевірити правильність роботи алгоритму на великих графах та визначити прискорення та ефективність:

```
def test3():
    size = 300
    graph = Graph(size)

    start_time = time.time()
    result_seq = sequential_prim(graph, 0)
    end_time = time.time()

    seq_time = end_time - start_time
    print(f"Sequential time: {seq_time:.4f}s")
    print(f"Result: {result_seq}")

    for i in range(2, 11):
        start_time = time.time()
        result_par = parallel_prim(graph, 0, i)
        end_time = time.time()
        duration = end_time - start_time
        acceleration = seq_time / duration
        print(f"{i} threads: {duration:.4f}s\t"
              f"Acceleration: {acceleration:.4f}\t"
              f"Efficiency: {(acceleration / i):.4f}\t"
              f"Correct: {result_par == result_seq}")
```

Результат:

```
Sequential time: 0.7030s

Result: 880

2 threads: 0.2491s  Acceleration: 2.8226    Efficiency: 1.4113  Correct: True

3 threads: 0.3430s  Acceleration: 2.0497    Efficiency: 0.6832  Correct: True

4 threads: 0.3665s  Acceleration: 1.9181    Efficiency: 0.4795  Correct: True

5 threads: 0.3861s  Acceleration: 1.8211    Efficiency: 0.3642  Correct: True

6 threads: 0.4408s  Acceleration: 1.5949    Efficiency: 0.2658  Correct: True

7 threads: 0.4962s  Acceleration: 1.4169    Efficiency: 0.2024  Correct: True

8 threads: 0.5655s  Acceleration: 1.2432    Efficiency: 0.1554  Correct: True

9 threads: 0.6294s  Acceleration: 1.1169    Efficiency: 0.1241  Correct: True

10 threads: 0.6880s Acceleration: 1.0219    Efficiency: 0.1022  Correct: True
```

В цьому тесті створив граф на 300 вершин. Виміряв час виконання послідовного алгоритму, вивів його на екран, далі в циклі для кількості потоків від 2 до 10 виміряв час виконання паралельного алгоритму, вивів прискорення та ефективність на екран. Для цієї лабораторної роботи вирішив для кожного потоку також вивести чи збігається його результат з результатом виконання послідовного алгоритму.

Висновок: під час виконання лабораторної роботи №7 написав програму для побудови мінімального кістякового дерева у зваженому неорієнтованому графі, використовуючи алгоритм Прима (послідовний та паралельний), обчислив прискорення та ефективність для різної кількості потоків та навчився аналізувати ці дані.