Nama : Ifham Syafwan Fikri

NIM : 24/545184/PA/23161

Github : https://github.com/syafwan-ux/PenugasanASD

Dosen Pengampu : Guntur Budi Herwanto, S.Kom., M.Cs.

PENUGASAN ALGORITMA DAN STRUKTUR DATA

GRAF SMART CITY

Dalam penyelesaian proyek ini, saya memilih untuk menggunakan **kota fiksi** sebagai referensi graf karena fleksibilitasnya dalam merancang struktur kota sesuai kebutuhan permasalahan. Dengan kota fiksi, saya dapat menentukan fasilitas-fasilitas, seperti rumah, kantor, dan tempat umum sebagai nilai dari tiap jenis node secara bebas serta memberikan jalur atau jalan yang dapat ditempuh (atau edges) antar lokasi fasilitas.

Untuk Graph Traversal, saya memilih menggunakan **algoritma Depth First Search (DFS)**. Menurut saya, karakteristiknya cocok untuk menjelajahi lebih dalam ke suatu jalur tertentu sebelum kembali dan mengeksplorasi rute lain. Berikut merupakan adjacency list dari graf:

```
int[][] edges = {
    {0, 1, 45},   {0, 2, 32},
    {1, 3, 21},   {1, 4, 78},
    {2, 5, 44},   {2, 6, 67},
    {3, 7, 15},   {4, 7, 38},
    {4, 8, 59},   {5, 9, 27},
    {5, 10, 33},  {6, 10, 12},
    {7, 11, 48},  {8, 11, 63},
    {8, 12, 29},  {9, 13, 71},
    {10, 13, 54}, {10, 14, 82},
    {11, 14, 36}, {12, 13, 41},
    {12, 14, 65}, {0, 13, 90},
    {1, 12, 83},  {3, 14, 74},
    {6, 9, 53}
};
```

0 ↔ 1 (45), 2 (32), 13 (90)

1 ↔ 0 (45), 3 (21), 4 (78), 12 (83)

2 ↔ 0 (32), 5 (44), 6 (67)

3 ↔ 1 (21), 7 (15), 14 (74)

4 ↔ 1 (78), 7 (38), 8 (59)

5 ↔ 2 (44), 9 (27), 10 (33)

6 ↔ 2 (67), 10 (12), 9 (53)

7 ↔ 3 (15), 4 (38), 11 (48)

8 ↔ 4 (59), 11 (63), 12 (29)

9 ↔ 5 (27), 6 (53), 13 (71)

10 ↔ 5 (33), 6 (12), 13 (54), 14 (82)

11 ↔ 7 (48), 8 (63), 14 (36)

12 ↔ 8 (29), 13 (41), 14 (65), 1 (83)

13 ↔ 9 (71), 10 (54), 12 (41), 0 (90)

14 ↔ 10 (82), 11 (36), 12 (65), 3 (74)

Langkah pertama yang dilakukan adalah menerapkan algoritma Depth First Search (DFS). DFS digunakan untuk menelusuri graf yang telah dibentuk berdasarkan *nodes* dan *edge*s. Algoritma ini bekerja dengan menjelajahi node tertentu sedalam mungkin sebelum kembali (*backtracking*) untuk mengeksplorasi cabang lain yang belum dikunjungi dengan menyimpannya di dalam stack. Berikut merupakan algoritma DFS yang digunakan:

- Saat metode dalam file DFS.java, yaitu dfsUtil dipanggil untuk node awal, alamat eksekusi fungsi tersebut disimpan di stack.
- Fungsi menandai node saat ini sebagai sudah dikunjungi dan mulai mengunjungi tetangga-tetangganya.
- Untuk setiap tetangga yang belum dikunjungi, fungsi dfsUtil dipanggil secara rekursif. Setiap pemanggilan rekursif ini menambahkan frame baru ke stack, menyimpan status eksekusi fungsi saat ini.
- Jika sebuah node tidak memiliki tetangga yang belum dikunjungi, fungsi dfsUtil selesai dan frame tersebut dihapus dari stack, kembali ke fungsi pemanggil sebelumnya.
- Proses ini berlanjut hingga semua node yang dapat dijangkau telah dikunjungi dan semua frame di stack telah dihapus. Salah satu contoh hasil traversal dari DFS (dimulai dari node 0) adalah:

```
DFS traversal mulai dari node 0:
[0, 1, 3, 7, 4, 8, 11, 14, 10, 5, 2, 6, 9, 13, 12]
Total bobot traversal: 668
```

Langkah kedua adalah penerapan algoritma Dijkstra. Dijkstra digunakan untuk menghitung *jalur terpendek* antara satu node asal ke semua node lain dalam graf berbobot tidak negatif yang telah dibentuk sebelumnya. Algoritma ini digunakan seperti untuk menentukan rute tercepat dari titik pusat ke berbagai fasilitas penting.

Berikut merupakan implementasi algoritma Dijkstra yang digunakan:

- Graf direpresentasikan menggunakan adjacency list, yaitu setiap node memiliki daftar tetangga beserta bobot sisi yang menghubungkannya. Dalam Metode addEdge di file Dijkstra.java, metode menambahkan sisi dua arah antara dua node karena graf tidak berarah
- Metode dijkstra menerima node awal sebagai parameter dan menginisialisasi jarak ke semua node dengan nilai maksimum (tak hingga), kecuali node awal yang jaraknya 0. Kemudian, Priority queue digunakan untuk memilih node dengan jarak terkecil yang belum diproses.
- Untuk setiap tetangga node yang sedang diproses, algoritma memperbarui jarak jika ditemukan jalur yang lebih pendek melalui node tersebut. Setelah semua node diproses, jarak terpendek ke setiap node dicetak. Berikut merupakan hasil perhitungan Dijkstra (dengan node 0 sebagai titik awal):

```
Jarak terpendek dari node 0:
Ke node 0 jarak: 0
Ke node 1 jarak: 45
Ke node 2 jarak: 32
Ke node 3 jarak: 66
Ke node 4 jarak: 119
Ke node 5 jarak: 76
Ke node 6 jarak: 99
Ke node 7 jarak: 81
Ke node 8 jarak: 157
Ke node 9 jarak: 103
Ke node 10 jarak: 109
Ke node 11 jarak: 129
Ke node 12 jarak: 128
Ke node 13 jarak: 90
Ke node 14 jarak: 140
```

Langkah ketiga adalah penerapan Minimum Spanning Tree (MST) menggunakan algoritma Kruskal. Algoritma Kruskal bertujuan untuk membentuk koneksi dengan total bobot minimum yang menghubungkan seluruh node dalam graf tanpa membentuk siklus. Algoritma ini bekerja dengan mengurutkan semua edge berdasarkan bobot terkecil, lalu menambahkan edge satu per satu ke dalam MST selama tidak membentuk siklus, hingga seluruh node terhubung. Berikut adalah algoritma Kruskal beserta hasil MST yang terbentuk.

- Algoritma mengumpulkan semua sisi dalam sebuah list dan mengurutkannya berdasarkan bobot dari yang terkecil.
- Struktur data Disjoint Set (Union-Find) digunakan untuk mendeteksi siklus dengan mengelompokkan node-node yang sudah terhubung.
- Algoritma memilih sisi dengan bobot terkecil yang tidak membentuk siklus dan menambahkannya ke MST.
- Proses ini berlanjut hingga MST mencakup semua node.
- Hasil MST dan total bobotnya dicetak. Berikut hasil MST (dalam bentuk daftar edge yang terpilih):

```
Edge-edge dalam MST:
6 - 10 : 12
3 - 7 : 15
1 - 3 : 21
5 - 9 : 27
8 - 12 : 29
0 - 2 : 32
5 - 10 : 33
11 - 14 : 36
4 - 7 : 38
12 - 13 : 41
2 - 5 : 44
0 - 1 : 45
7 - 11 : 48
10 - 13 : 54
Total bobot MST: 475
```