IMPLEMENTASI ALGORITMA

TSP (Traveling Salesman Problem)

Mata Kuliah :
Visual Programming

Materi Praktikum ke : 1

Nama :
Andrey Hartawan Suwardi
NIM :
0806022310021

Tanggal Praktikum :
18 September 2024

BAB I

PENDAHULUAN

1.1. Latar Belakang

Traveling salesman problem, atau TSP, adalah tantangan salesman untuk menemukan rute terpendek dan paling efisien untuk seseorang sesuai daftar tujuannya. Karl Menger, seorang ahli matematika dan ekonomi, pertama kali membuat TSP pada tahun 1930-an. Menggambarkan masalah yang dihadapi oleh pengirim surat dan banyak pengelana, Menger menyebutnya sebagai "masalah pengirim surat" (Technology, 2023).

Pada praktikum kali ini, diberikan sebuah gambaran flowchart yang dimana struktur dari flowchart tersebut menggunakan sistem logika TSP tersebut dan akan diterapkan dalam kode program bahasa dart.

1.2. Tujuan

Mengimplementasikan logika TSP menggunakan solusi Brute-Force, memberikan semua kemungkinan jalur tercepat dari Kota A ke kota lainnya dan seterusnya.

1.3. Tinjauan Pustaka

Mempelajari algoritma TSP dengan pendekatan Brute-Force juga dikenal dengan pendekatan Naive yaitu mengkalkulasi dan membandingkan semua kemungkinan permutasi dari rute untuk menentukan solusi rute terpendek. Jika menggunakan pendekatan ini, setiap kota perlu menghitung jumlah rute dan gambar serta membuat daftar semua rute yang memungkinkan. Menghitung jarak dari setiap rute dan kemudian memilih yang terpendek.

BAB II
ALAT DAN BAHAN

2.1. Alat

- Laptop
- Visual Studio Code

2.2. Bahan

- **Dart**

BAB III
PROSEDUR KERJA

- **Membuka Vscode**

  Buka Vscode yang sudah terinstall dengan Dart SDK

- **Mengetik kode dart dengan implementasi TSP**
  tsp.dart atau file nama yang lain
- **Membuat Class Vertice dan Class TSP**

```
class Vertice {
  String name;
  Map<String, double> neighbors;
  bool visited; // Menandakan apakah kota telah dikunjungi

  Vertice(this.name, this.neighbors) : visited = false; //
Inisialisasi visited dengan false

  @override
  String toString() => name;
}
class TSP {
  Map<String, Vertice> vertices;

  TSP(this.vertices);
}
```

- **Membuat semua fungsi yang diperlukan di kelas TSP**

```
// Menghitung total jarak berdasarkan jalur yang diberikan
 double calculateTotalDistance(List<String> path) {
   double totalDistance = 0;
   for (int i = 0; i < path.length - 1; i++) {
     if (vertices.containsKey(path[i]) &&
```

```
          vertices.containsKey(path[i + 1]) &&
          vertices[path[i]]!.neighbors.containsKey(path[i + 1])) {
        totalDistance += vertices[path[i]]!.neighbors[path[i + 1]]!;
      } else {
        print("Error: No connection between ${path[i]} and ${path[i +
1]}.");
        return double.infinity; // Kembalikan infinity jika tidak ada
koneksi
      }
    }
    totalDistance += vertices[path.last]!.neighbors[path.first]!; //
Kembali ke kota asal
    return totalDistance;
  }

  List<String> findShortestPath() {
    List<String> shortestPath = [];
    double shortestDistance = double.infinity;

    List<List<String>> allPermutations =
_permutations(vertices.keys.toList());

    for (var path in allPermutations) {
      // Reset visited status for each vertex
      _resetVisitedStatus();

      double distance = calculateTotalDistance(path);
      if (distance < shortestDistance) {
        shortestDistance = distance;
```

```
        shortestPath = path;
      }

      // Tandai kunjungan dan sambungkan ke vertex terdekat
      _markVisitsAndConnect(path);
    }

    print("Rizzest Path: ${shortestPath}");
    print("Total Distance: $shortestDistance");

    return shortestPath;
  }

  void _markVisitsAndConnect(List<String> path) {
    for (var current in path) { // 'current' adalah kota yang sedang diproses
      if (!vertices[current]!.visited) { // Cek apakah kota sudah dikunjungi
        vertices[current]!.visited = true; // Tandai node yang sudah dikunjungi
        print("Current City: $current");
        print("Visited: $current");

        // Cek untuk menghubungkan ke vertex terdekat yang belum dikunjungi
        _connectToNearestUnvisited(current);
      }
    }
```

```
    print("Visited nodes: ${vertices.values.where((v) =>
v.visited).map((v) => v.name).toList()}");
  }

  void _connectToNearestUnvisited(String current) {
    double shortestDistance = double.infinity;
    String? nearestUnvisited;

    for (var neighbor in vertices[current]!.neighbors.keys) {
      if (!vertices[neighbor]!.visited &&
          vertices[current]!.neighbors[neighbor]! < shortestDistance)
{
        shortestDistance = vertices[current]!.neighbors[neighbor]!;
        nearestUnvisited = neighbor;
      }
    }

    if (nearestUnvisited != null) {
      print("Connecting $current to nearest unvisited node:
$nearestUnvisited");
    }
  }

  // Goon helper untuk menghasilkan semua permutasi
  List<List<String>> _permutations(List<String> list) {
    if (list.isEmpty) return [[]];
    List<List<String>> result = [];
    for (int i = 0; i < list.length; i++) {
```

```
      var head = list[i];
      var tail = List.of(list)..removeAt(i);
      for (var perm in _permutations(tail)) {
        result.add([head] + perm);
      }
    }
    return result;
  }

  // Mengatur ulang status kunjungan semua vertices
  void _resetVisitedStatus() {
    for (var vertex in vertices.values) {
      vertex.visited = false; // Setiap vertex belum dikunjungi
    }
  }
}
```

- **Membuat isi list dari kota di void main()**

```
  Map<String, Vertice> cities = {
    'A': Vertice('A', {'B': 8, 'C': 3, 'D': 4, 'E': 10}),
    'B': Vertice('B', {'A': 8, 'C': 5, 'D': 2, 'E': 7}),
    'C': Vertice('C', {'A': 3, 'B': 5, 'D': 1, 'E': 6}),
    'D': Vertice('D', {'A': 4, 'B': 2, 'C': 1, 'E': 3}),
    'E': Vertice('E', {'A': 10, 'B': 7, 'C': 6, 'D': 3}),
  };
TSP tsp = TSP(cities);
  tsp.findShortestPath();
```

BAB IV

HASIL & PEMBAHASAN

- Class Vertice bertujuan untuk menyimpan kota, tetangga dengan jaraknya dan status kunjungan
- Class TSP memproses list yang disimpan oleh kelas vertice di Map dengan susunan Map<String,Vertice> vertices
- Class TSP dengan fungsi fungsi yang ada didalamnya untuk menjalankan Brute-Force solusi
- **Hasil jika dijalankan diterminal :**

  Connecting C to nearest unvisited node: A

  Current City: B

  Visited: B

  Connecting B to nearest unvisited node: A

  Current City: A

  Visited: A

  Visited nodes: [A, B, C, D, E]

  Rizzest Path: [A, B, D, E, C]

  Total Distance: 22.0

BAB V

KESIMPULAN

Menggunakan permasalahan TSP dari kota asal ke kota tetangga dan kembali lagi ke kota asal dengan solusi mengkalkulasi semua jarak dari 1 permutasi dan membandingkannya dengan kalkulasi permutasi yang lain, solusi brute-force telah membuat banyak pilihan rute yang dihasilkan dari setiap vertice(kota) dan tetangga-nya , maka dari itu solusi brute-force dalam pemograman dapat menyelesaikan permasalahan terkait rute terpendek.

DAFTAR PUSTAKA

Technology, B. V. (2023, January 24). *Traveling Salesman Problem: Definisi dan Implementasi*. Bhumi Varta Technology. https://bvarta.com/id/traveling-salesman-problem-definisi-dan-implementasi/

Amin, A. R., Ikhsan, M., & Wibisono, L. (2003). Traveling Salesman Problem. *Institut Teknologi Bandung*.