

REPRESENTASI GRAF PETA JAWA TIMUR DENGAN ALGORITMA DIJKSTRA DAN TSP

Mata Kuliah

Struktur Data

Disusun oleh :

Septiana Dwi Lestari (24091397113)

Fitrya Chalifatuz Z (24091397117)

Naufal Ihsan Awari (24091397131)



PROGRAM STUDI MANAJEMEN INFORMATIKA

FAKULTAS VOKASI

UNIVERSITAS NEGERI SURABAYA

2025

BAB 1

PENDAHULUAN

1.1 LATAR BELAKANG

Peta dan sistem navigasi berbasis algoritma merupakan salah satu aplikasi penting dalam ilmu komputer dan teknologi informasi, terutama dalam konteks optimasi rute perjalanan. Di era modern, kebutuhan akan perencanaan rute yang efisien semakin meningkat, baik untuk transportasi pribadi, logistik, maupun pariwisata. Jawa Timur, sebagai salah satu provinsi terpadat di Indonesia, memiliki kompleksitas jaringan transportasi yang melibatkan banyak kota besar dan kecil, yang menjadikannya subjek yang menarik untuk dianalisis menggunakan struktur data graf.

Tugas ini dirancang untuk mengimplementasikan representasi peta Jawa Timur dalam bentuk graf, di mana setiap kota dianggap sebagai *vertex* dan jalur antar kota sebagai *edge* dengan bobot berupa jarak dalam kilometer. Program ini mengintegrasikan dua algoritma optimasi rute yang terkenal:

- **Algoritma Dijkstra:** Digunakan untuk menemukan jalur terpendek antara dua kota tertentu, yang sangat berguna untuk navigasi point-to-point.
- **Traveling Salesman Problem (TSP):** Digunakan untuk menentukan urutan perjalanan yang mengunjungi semua kota dengan total jarak minimum, relevan untuk perencanaan perjalanan wisata atau distribusi barang.

Pemilihan Jawa Timur sebagai fokus studi didasarkan pada keberagaman geografis dan ekonomi wilayah ini, yang mencakup kota-kota seperti Surabaya (pusat ekonomi), Malang (pusat pendidikan), dan Banyuwangi (gerbang ke Bali). Dengan 10 kota yang dipilih, tugas ini bertujuan untuk mensimulasikan sistem GPS sederhana yang dapat diterapkan dalam konteks nyata dengan pengembangan lebih lanjut.

1.2 TUJUAN

Tujuan utama tugas ini adalah:

1. **Membangun Representasi Graf:** Membuat struktur data graf menggunakan *adjacency list* untuk merepresentasikan peta Jawa Timur dengan 10 kota dan 30 jalur antar kota.
2. **Menerapkan Algoritma Optimasi:**

- Mengimplementasikan algoritma Dijkstra untuk mencari jalur tercepat antara dua kota yang ditentukan pengguna.
 - Mengimplementasikan algoritma TSP dengan pendekatan *brute-force* untuk menemukan rute optimal yang mengunjungi semua kota tepat satu kali.
3. **Menguji dan Mendokumentasikan:** Menguji program dengan berbagai kombinasi kota asal dan tujuan, serta menyusun laporan yang mencakup penjelasan, hasil, dan visualisasi.
 4. **Visualisasi:** Menyediakan representasi visual dari graf untuk mempermudah pemahaman struktur peta dan jalur yang dihasilkan.

1.3 MANFAAT

1. Memberikan pemahaman praktis tentang struktur data graf dan algoritma pencarian jalur.
2. Menyediakan alat simulasi sederhana yang dapat dikembangkan menjadi prototipe sistem navigasi.
3. Mendukung pengambilan keputusan dalam perencanaan perjalanan atau distribusi di wilayah Jawa Timur.

1.4 RUANG LINGKUP

Program dikembangkan menggunakan bahasa pemrograman Python dengan pustaka pendukung:

1. `heapq` untuk implementasi *priority queue* dalam algoritma Dijkstra.
2. `itertools.permutations` untuk menghasilkan semua kemungkinan urutan kota dalam TSP.
3. `networkx` dan `matplotlib` untuk visualisasi graf. Graf dirancang sebagai *undirected graph* yang terhubung sepenuhnya, dengan 10 kota di Jawa Timur (Surabaya, Malang, Kediri, Blitar, Madiun, Jember, Banyuwangi, Pasuruan, Probolinggo, Lamongan) dan 30 jalur antar kota berdasarkan jarak perkiraan. Program ini berjalan dalam lingkungan konsol dengan input pengguna dan output berupa jalur serta jarak tempuh.

BAB II

STRUKTUR DAN ILUSTRASI GRAF

2.1 STRUKTUR GRAF

Vertex (Kota): Graf terdiri dari 10 kota di Jawa Timur, yaitu:

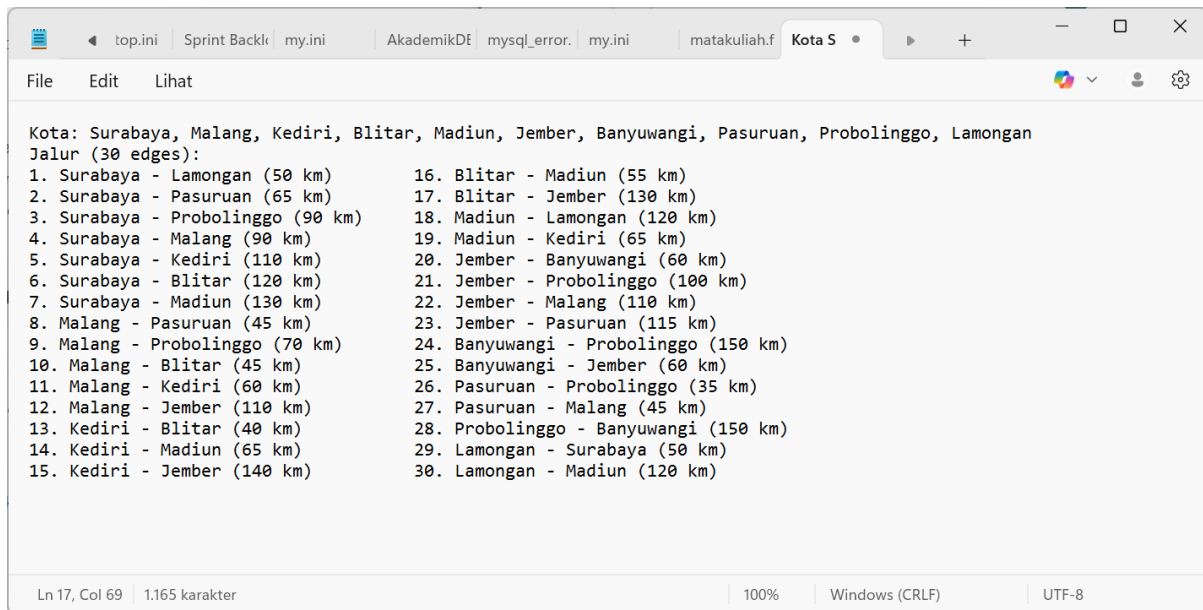
1. Surabaya
2. Malang
3. Kediri
4. Blitar
5. Madiun
6. Jember
7. Banyuwangi
8. Pasuruan
9. Probolinggo
10. Lamongan

Edge (Jalur): Terdapat 30 jalur antar kota dengan bobot berupa jarak dalam kilometer. Graf bersifat tidak berarah, sehingga setiap jalur dapat dilalui dua arah dengan jarak yang sama.

Representasi: Graf disimpan menggunakan adjacency list dalam kelas Graph, di mana setiap kota memiliki daftar tetangga beserta jaraknya. Struktur ini efisien untuk menyimpan dan mengakses informasi koneksi antar kota.

2.2 DAFTAR JALUR

Berikut adalah daftar lengkap 30 jalur antar kota beserta jaraknya (dalam kilometer):

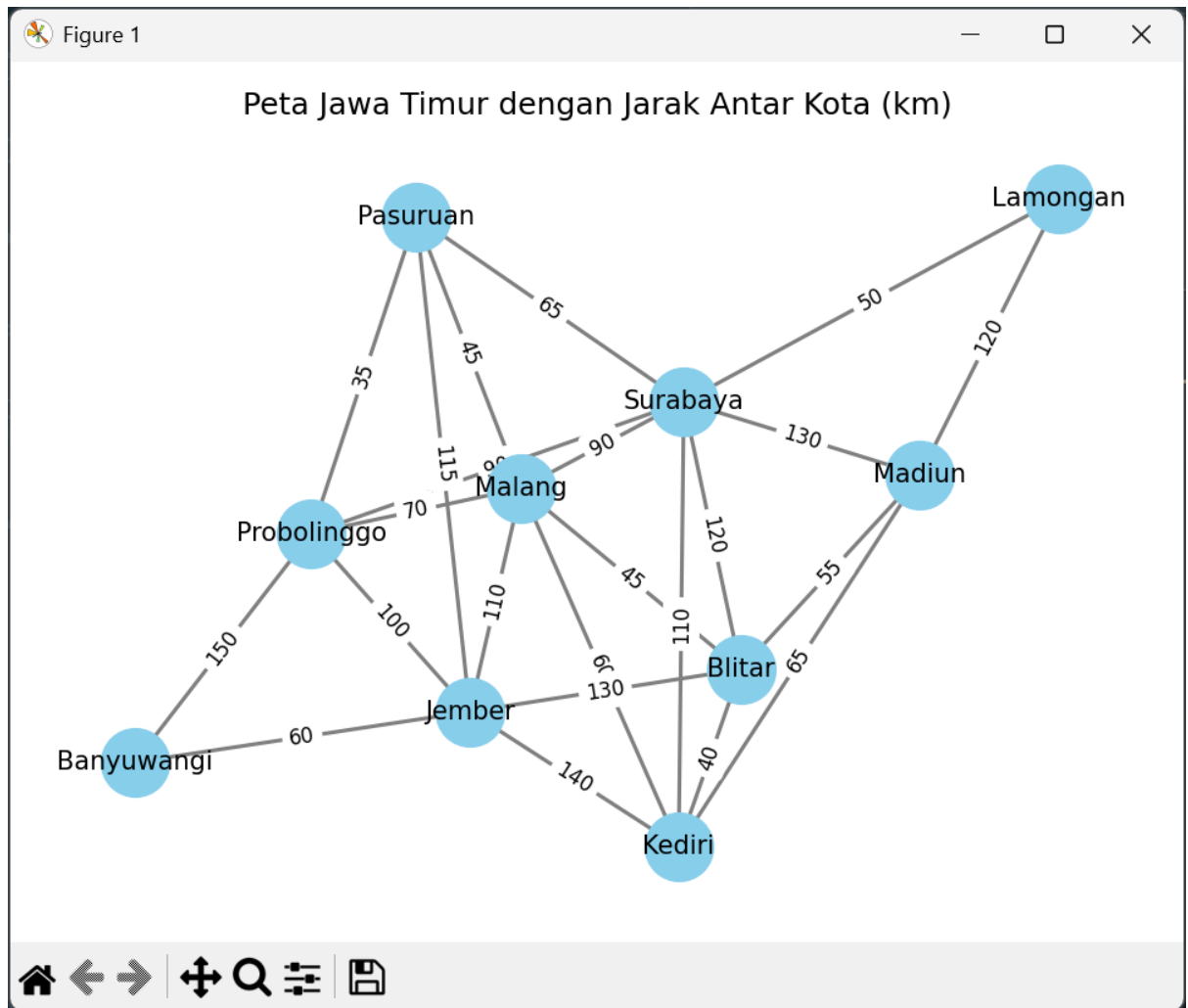


```
Kota: Surabaya, Malang, Kediri, Blitar, Madiun, Jember, Banyuwangi, Pasuruan, Probolinggo, Lamongan
Jalur (30 edges):
1. Surabaya - Lamongan (50 km)
2. Surabaya - Pasuruan (65 km)
3. Surabaya - Probolinggo (90 km)
4. Surabaya - Malang (90 km)
5. Surabaya - Kediri (110 km)
6. Surabaya - Blitar (120 km)
7. Surabaya - Madiun (130 km)
8. Malang - Pasuruan (45 km)
9. Malang - Probolinggo (70 km)
10. Malang - Blitar (45 km)
11. Malang - Kediri (60 km)
12. Malang - Jember (110 km)
13. Kediri - Blitar (40 km)
14. Kediri - Madiun (65 km)
15. Kediri - Jember (140 km)
16. Blitar - Madiun (55 km)
17. Blitar - Jember (130 km)
18. Madiun - Lamongan (120 km)
19. Madiun - Kediri (65 km)
20. Jember - Banyuwangi (60 km)
21. Jember - Probolinggo (100 km)
22. Jember - Malang (110 km)
23. Jember - Pasuruan (115 km)
24. Banyuwangi - Probolinggo (150 km)
25. Banyuwangi - Jember (60 km)
26. Pasuruan - Probolinggo (35 km)
27. Pasuruan - Malang (45 km)
28. Probolinggo - Banyuwangi (150 km)
29. Lamongan - Surabaya (50 km)
30. Lamongan - Madiun (120 km)
```

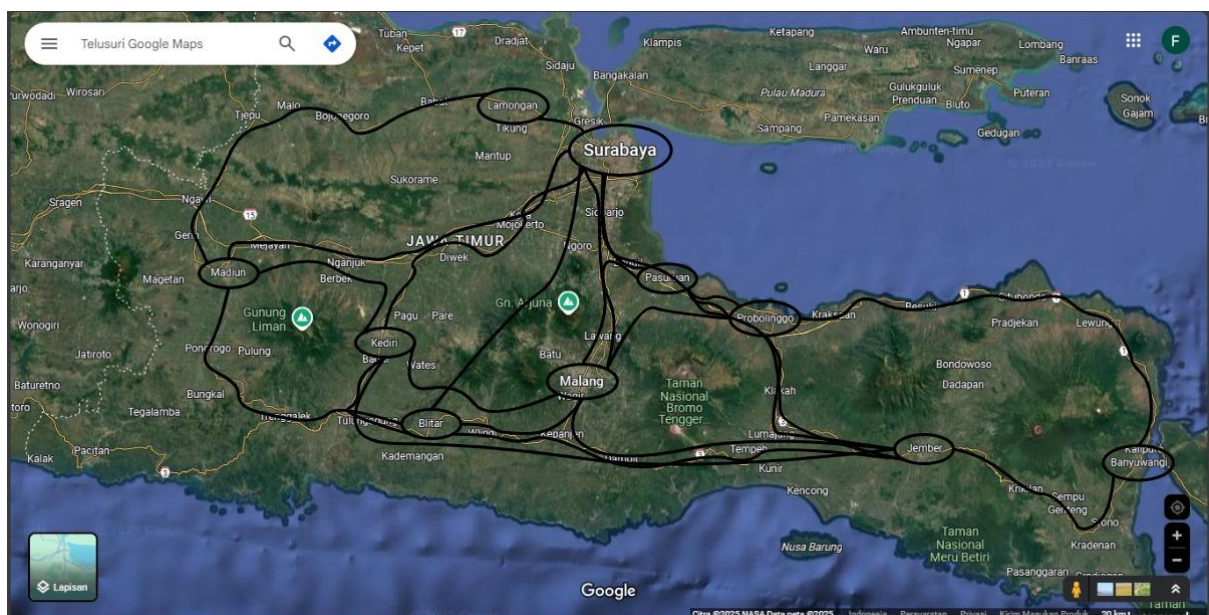
2.3 Ilustrasi Graf

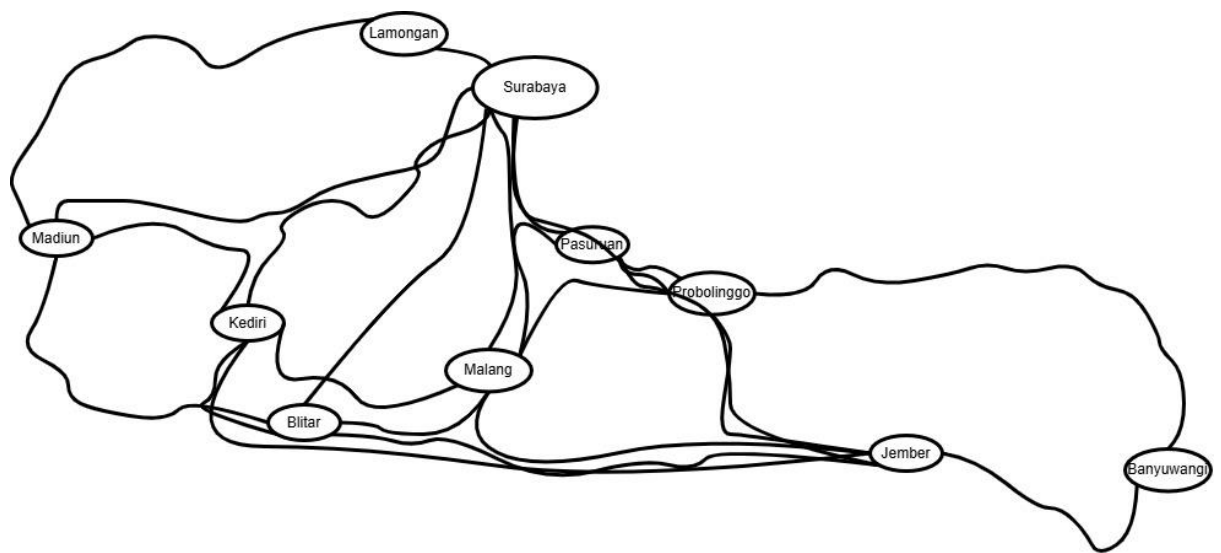
Gambar ilustrasi graf disajikan pada Gambar 1 di lampiran. Graf ini menunjukkan:

- **Node:** 10 kota di Jawa Timur, direpresentasikan sebagai lingkaran berwarna biru muda dengan label nama kota.
- **Edge:** 30 jalur antar kota, ditampilkan sebagai garis abu-abu dengan label jarak (km).
- **Metode Visualisasi:** Graf dibuat menggunakan pustaka `networkx` dan `matplotlib` dengan algoritma *spring layout*, yang secara otomatis menata posisi node agar tidak tumpang tindih dan memberikan tampilan yang estetik.
- **Karakteristik:** Graf ini adalah *connected graph*, artinya setiap kota dapat dijangkau dari kota lain melalui satu atau lebih jalur, yang memenuhi syarat tugas.



2.4 Peta Geografis





BAB III PENJELASAN ALGORITMA

3.1 ALGORITMA DIJKSTRA

3.1.1 Deskripsi

Algoritma Dijkstra adalah algoritma pencarian jalur terpendek pada graf berbobot non-negatif, yang dikembangkan oleh Edsger Dijkstra pada tahun 1956. Algoritma ini ideal untuk aplikasi navigasi seperti GPS.

3.1.2 Cara Kerja

1. Inisialisasi jarak ke semua kota sebagai tak hingga, kecuali kota asal (jarak = 0).
2. Gunakan *priority queue* untuk memilih kota dengan jarak terkecil yang belum diproses.
3. Perbarui jarak ke tetangga kota saat ini jika ditemukan jalur yang lebih pendek.
4. Simpan kota sebelumnya untuk merekonstruksi jalur.
5. Ulangi hingga kota tujuan tercapai atau semua kota diproses.

3.1.3 Implementasi

1. Fungsi `dijkstra(graph, start, end)` dalam kode menggunakan `heapq` untuk efisiensi *priority queue*.
2. Input: Nama kota asal dan tujuan dari pengguna.
3. Output: Daftar kota dalam jalur tercepat dan total jarak (km).

Kompleksitas: $O((V + E) \log V)$, di mana $V = 10$ (jumlah vertex) dan $E = 30$ (jumlah edge).

3.2 ALGORITMA TSP (BRUTE-FORCE)

3.2.1 Deskripsi

TSP adalah masalah optimasi kombinatorial yang bertujuan menemukan siklus Hamilton dengan total bobot minimum. Dalam tugas ini, pendekatan brute-force digunakan untuk menghitung semua kemungkinan urutan.

3.2.2 Cara Kerja

1. Ambil semua permutasi kota (kecuali kota awal) menggunakan `itertools.permutations`.
2. Hitung total jarak untuk setiap permutasi dengan menelusuri jalur antar kota.
3. Anggap jarak tak hingga jika tidak ada jalur langsung antar kota, membatalkan permutasi tersebut.
4. Pilih permutasi dengan total jarak terkecil.

3.2.3 Implementasi

1. Fungsi `tsp_bruteforce(graph, start)` mengembalikan urutan kota dan total jarak.
2. Input: Kota awal dari pengguna.
3. Output: Urutan kota dan total jarak tempuh.

Kompleksitas: $O(n!)$, di mana $n = 10$. Untuk 9 kota (setelah mengabaikan kota awal), ini menghasilkan $9! = 362.880$ perhitungan, yang masih dapat diterima untuk skala kecil.

BAB IV HASIL PENGUJIAN

4.1 PENGUJIAN ALGORITMA DIJKSTRA & TSP

```
Users > septi > OneDrive > Dokumen > STRUKDAT > maps.py > ...

# import library yang diperlukan
import heapq # untuk implementasi algoritma Dijkstra
import networkx as nx # untuk visualisasi graph
import matplotlib.pyplot as plt # untuk plotting graf
from itertools import permutations # untuk implementasi TSP

class Graph: # class untuk representasi graph
    def __init__(self): # konstruktor
        self.adj_list = {} # adjacency list untuk representasi graph

    def add_vertex(self, city): # fungsi untuk menambahkan vertex (kota)
        if city not in self.adj_list: # jika kota belum ada di graph
            self.adj_list[city] = {} # tambahkan kota ke adjacency list

    def add_edge(self, city1, city2, distance): # fungsi untuk menambahkan edge (jalur)
        self.adj_list[city1][city2] = distance # menambahkan jalur dari kota1 ke kota2 dengan jarak
        self.adj_list[city2][city1] = distance # menambahkan jalur dari kota2 ke kota1 dengan jarak yang sama

    def get_neighbors(self, city): # fungsi untuk mendapatkan tetangga suatu vertex (kota)
        return self.adj_list[city] # return dictionary tetangga suatu vertex (kota)

jawa_timur = Graph() # inisialisasi graph Jawa Timur

# menambahkan 10 vertex (kota) di Jawa Timur
cities = [
    "Surabaya", "Malang", "Kediri", "Blitar", "Madiun",
    "Jember", "Banyuwangi", "Pasuruan", "Probolinggo", "Lamongan"
]

for city in cities: # menambahkan vertex (kota) ke dalam graph
    jawa_timur.add_vertex(city)

# menambahkan edge (jalur) antar vertex (kota) dengan jarak yang telah ditentukan dalam km
edges = [
    ("Surabaya", "Lamongan", 50), ("Surabaya", "Pasuruan", 65),
    ("Surabaya", "Probolinggo", 90), ("Malang", "Pasuruan", 45),
    ("Malang", "Probolinggo", 70), ("Malang", "Blitar", 45),
    ("Malang", "Kediri", 60), ("Kediri", "Blitar", 40),
    ("Kediri", "Madiun", 65), ("Blitar", "Madiun", 55),
    ("Madiun", "Lamongan", 120), ("Jember", "Banyuwangi", 60),
    ("Jember", "Probolinggo", 100), ("Jember", "Malang", 110),
    ("Banyuwangi", "Probolinggo", 150), ("Pasuruan", "Probolinggo", 35),
    ("Lamongan", "Madiun", 120), ("Surabaya", "Madiun", 130),
    ("Malang", "Jember", 110), ("Kediri", "Jember", 140),
    ("Blitar", "Jember", 130), ("Probolinggo", "Banyuwangi", 150),
    ("Pasuruan", "Malang", 45), ("Surabaya", "Malang", 90),
    ("Kediri", "Surabaya", 110), ("Blitar", "Surabaya", 120),
    ("Madiun", "Kediri", 65), ("Jember", "Pasuruan", 115),
    ("Banyuwangi", "Jember", 60), ("Lamongan", "Surabaya", 50)
]

for edge in edges: # menambahkan edge (jalur) ke dalam graph
    jawa_timur.add_edge(*edge)
```

```

def dijkstra(graph, start, end): # fungsi untuk implementasi Algoritma Dijkstra
    distances = {city: float('infinity') for city in graph.adj_list} # inisialisasi jarak awal dengan nilai tak terhingga
    distances[start] = 0 # jarak awal dari kota start ke kota start adalah 0
    previous = {city: None for city in graph.adj_list} # inisialisasi kota sebelumnya dengan nilai None
    priority_queue = [(0, start)] # inisialisasi priority queue dengan jarak awal dan kota start

    while priority_queue: # Loop hingga priority queue kosong
        current_distance, current_city = heapq.heappop(priority_queue) # pop kota dengan jarak terkecil dari priority queue

        if current_city == end: # berhenti jika kota saat ini adalah kota tujuan
            break

        for neighbor, weight in graph.get_neighbors(current_city).items(): # Loop melalui tetangga kota saat ini
            distance = current_distance + weight # hitung jarak ke tetangga kota saat ini

            if distance < distances[neighbor]: # jika jarak baru lebih cepat dari jarak sebelumnya
                distances[neighbor] = distance # update jarak
                previous[neighbor] = current_city # update kota sebelumnya
                heapq.heappush(priority_queue, (distance, neighbor)) # tambahkan ke priority queue

    if distances[end] != float('infinity'): # jika jarak ke kota tujuan tidak tak terhingga
        path = [] # inisialisasi jalur dengan nilai kosong
        current = end # inisialisasi kota saat ini dengan kota tujuan
        while current is not None: # Loop hingga kota saat ini adalah kota start
            path.append(current) # tambahkan kota saat ini ke jalur
            current = previous[current] # update kota saat ini dengan kota sebelumnya
        path.reverse() # balik jalur
        return path, distances[end] # return jalur dan jarak ke kota tujuan
    else:
        return None, None # return None jika jarak ke kota tujuan tak terhingga

def tsp_bruteforce(graph, start): # fungsi untuk implementasi Algoritma TSP Brute Force
    cities = list(graph.adj_list.keys()) # inisialisasi kota dengan kunci dari adjacency list
    if start not in cities: # jika kota start tidak ada di kota
        return None, float('infinity') # return None dan jarak tak terhingga

    cities.remove(start) # hapus kota start dari kota
    shortest_path = None # inisialisasi jalur terpendek dengan nilai None
    min_distance = float('infinity') # inisialisasi jarak terpendek dengan nilai tak terhingga

    for perm in permutations(cities): # Loop melalui semua permutasi kota
        current_distance = 0 # inisialisasi jarak saat ini dengan nilai 0
        current_city = start # inisialisasi kota saat ini dengan kota start
        path = [start] # inisialisasi jalur dengan kota start
        valid_path = True # inisialisasi validitas jalur dengan nilai True

        for next_city in perm: # Loop melalui kota berikutnya
            if next_city in graph.adj_list[current_city]: # jika kota berikutnya ada di adjacency list kota saat ini
                current_distance += graph.adj_list[current_city][next_city] # tambahkan jarak ke kota berikutnya
                current_city = next_city # update kota saat ini dengan kota berikutnya
                path.append(next_city) # tambahkan kota berikutnya ke jalur
            else:
                valid_path = False # jika kota berikutnya tidak ada di adjacency list kota saat ini, berhenti
                break

        if valid_path and current_distance < min_distance: # jika jalur valid dan jarak saat ini lebih kecil dari jarak terpendek saat ini
            min_distance = current_distance # update jarak terpendek saat ini
            shortest_path = path

    return shortest_path, min_distance # return jalur terpendek dan jarak terpendek

```

```

def main(): # fungsi utama
    print("Kota di Jawa Timur:") # cetak kota di Jawa Timur
    for i, city in enumerate(cities, 1): # loop melalui kota di Jawa Timur
        print(f"{i}. {city}") # cetak kota dengan nomor urut

    ''' Algoritma Dijkstra '''
    city_mapping = {city.lower(): city for city in cities} # inialisasi mapping antar kota dalam bentuk kecil

    while True: # Loop hingga pengguna memilih untuk berhenti
        print("\n== Algoritma Dijkstra (Rute Tercepat Antar-Kota) ==") # cetak judul algoritma Dijkstra
        start_input = input("Masukkan kota asal (atau 'exit' untuk keluar): ").strip().lower() # input kota asal pengguna

        if start_input == 'exit': # jika pengguna memilih untuk keluar, berhenti
            break

        start = city_mapping.get(start_input) # inialisasi kota asal pengguna dengan kunci dari mapping
        if not start: # jika kota asal tidak ada di mapping
            print("Kota tidak ditemukan! Silakan coba lagi.") # cetak pesan kesalahan
            continue

        end_input = input("Masukkan kota tujuan: ").strip().lower() # input kota tujuan pengguna
        end = city_mapping.get(end_input) # inialisasi kota tujuan pengguna dengan kunci dari mapping
        if not end: # jika kota tujuan tidak ada di mapping
            print("Kota tujuan tidak ditemukan! Silakan coba lagi.") # cetak pesan kesalahan
            continue

        if start == end: # jika kota asal sama dengan kota tujuan
            print("Kota asal dan tujuan sama! Silakan pilih kota yang berbeda.") # cetak pesan kesalahan
            continue

        path, distance = dijkstra(jawa_timur, start, end) # jalankan algoritma Dijkstra untuk menemukan jalur terpendek antara kota awal dengan kota tujuan

        if path: # jika jalur terpendek ditemukan
            print(f"\nJalur tercepat dari {start} ke {end}:") # cetak judul jalur tercepat
            print(" -> ".join(path)) # cetak jalur tercepat
            print(f"Total jarak ditempuh: {distance} km") # cetak jarak total yang ditempuh
        else:
            print(f"\nTidak ditemukan rute yang valid dari {start} ke {end}!") # cetak pesan kesalahan

        lanjut = input("\nCari rute lain? (y/n): ").strip().lower() # input pengguna untuk melanjutkan atau berhenti
        if lanjut != 'y': # jika pengguna memilih untuk berhenti, berhenti
            break

```

```

''' Algoritma TSP '''
while True: # Loop hingga pengguna memilih untuk berhenti
    print("\n== Traveling Salesman Problem (TSP) ==") # cetak judul TSP
    start_input = input("Masukkan kota awal TSP (atau 'exit' untuk keluar): ").strip().lower() # input kota awal pengguna

    if start_input == 'exit': # jika pengguna memilih untuk keluar, berhenti
        break

    start_city = city_mapping.get(start_input) # inialisasi kota awal pengguna dengan kunci dari mapping
    if not start_city: # jika kota awal tidak ada di mapping
        print("Kota tidak ditemukan! Pilihan kota yang valid:") # cetak pesan kesalahan
        print(", ".join(cities)) # cetak kota yang valid
        continue

    path, distance = tsp_bruteforce(jawa_timur, start_city) # jalankan algoritma TSP dengan metode brute force

    if path: # jika jalur terpendek ditemukan
        print("\nRute TSP terbaik:") # cetak judul jalur TSP terbaik
        print(" -> ".join(path)) # cetak jalur TSP terbaik
        print(f"Total jarak ditempuh: {distance} km") # cetak jarak total yang ditempuh

        print("\nVisualisasi Graph:") # cetak judul visualisasi grafik
        for i in range(len(path)-1): # Loop untuk setiap kota di jalur TSP
            print(f"{path[i]} --{jawa_timur.adj_list[path[i]][path[i+1]]}km--> {path[i+1]}") # cetak jalur antara kota
    else:
        print("\nTidak ditemukan rute yang valid untuk mengunjungi semua kota!") # cetak pesan kesalahan
        print("Kemungkinan penyebab:") # cetak penyebab
        print("- Graph tidak terhubung sepenuhnya") # ada kota yang terisolasi
        print("- Beberapa kota tidak terhubung langsung") # tidak ada edge (jalur) langsung antar kota

    lanjut = input("\nHitung TSP lagi? (y/n): ").strip().lower() # input pengguna untuk melanjutkan atau berhenti
    if lanjut != 'y': # jika pengguna memilih untuk berhenti, berhenti
        break

```

```
def visualize_graph(graph): # fungsi untuk implementasi visualisasi graph
    G = nx.Graph() # membuat graph kosong atau graph tidak berarah

    for city in graph.adj_list: # loop melalui setiap kota
        for neighbor, distance in graph.adj_list[city].items(): # loop tetangga
            G.add_edge(city, neighbor, weight=distance) # tambahkan edge (jalur) dengan atribut jarak

    pos = nx.spring_layout(G, seed=42) # menentukan posisi node menggunakan algoritma spring layout

    nx.draw_networkx_nodes(G, pos, node_size=700, node_color='skyblue') # menggambar node dengan parameter

    nx.draw_networkx_edges(G, pos, width=1.5, edge_color='gray') # menggambar edge (jalur) dengan parameter

    nx.draw_networkx_labels(G, pos, font_size=10, font_family='sans-serif') # menggambar Label nama kota

    edge_labels = nx.get_edge_attributes(G, 'weight') # mengambil atribut jarak untuk ditampilkan di edge (jalur)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=8) # menggambar Label jarak di setiap edge (jalur)

    plt.title("Peta Jawa Timur dengan Jarak Antar Kota (km)") # menambahkan judul visualisasi
    plt.axis('off') # menghilangkan axis
    plt.tight_layout() # adjust layout
    plt.show() # menampilkan visualisasi

if __name__ == "__main__":
    visualize_graph(jawa_timur) # menjalankan visualisasi graph
    main() # menjalankan program utama
```

4.2 OUTPUT PENGUJIAN

```
PS C:\Users\septi> & C:/Users/septi/AppData/Local/Programs/Python/Python312/python.exe c:/Users/septi/OneDrive/Dokumen/STRUKDAT/maps.py
Kota di Jawa Timur:
1. Surabaya
2. Malang
3. Kediri
4. Blitar
5. Madiun
6. Jember
7. Banyuwangi
8. Pasuruan
9. Probolinggo
10. Lamongan

=== Algoritma Dijkstra (Rute Tercepat Antar-Kota) ===
Masukkan kota asal (atau 'exit' untuk keluar): Lamongan
Masukkan kota tujuan: Banyuwangi

Jalur tercepat dari Lamongan ke Banyuwangi:
Lamongan -> Surabaya -> Probolinggo -> Banyuwangi
Total jarak ditempuh: 290 km

Cari rute lain? (y/n): y

=== Algoritma Dijkstra (Rute Tercepat Antar-Kota) ===
Masukkan kota asal (atau 'exit' untuk keluar): Surabaya
Masukkan kota tujuan: Madiun

Jalur tercepat dari Surabaya ke Madiun:
Surabaya -> Madiun
Total jarak ditempuh: 130 km

Cari rute lain? (y/n): y

=== Algoritma Dijkstra (Rute Tercepat Antar-Kota) ===
Masukkan kota asal (atau 'exit' untuk keluar): Malang
Masukkan kota tujuan: Pasuruan

Jalur tercepat dari Malang ke Pasuruan:
Malang -> Pasuruan
Total jarak ditempuh: 45 km

Cari rute lain? (y/n): n
```

ANALISIS ALGORITMA DIJKSTRA:

1. Contoh Kasus 1:

- **Input:** Kota asal: Lamongan

Kota tujuan: Banyuwangi

- **Proses:** Algoritma Dijkstra menghitung seluruh kemungkinan jalur dari Lamongan ke Banyuwangi. Program memilih jalur dengan total bobot (jarak) paling kecil.
- **Output:** Jalur tercepat: Lamongan → Surabaya → Probolinggo → Banyuwangi. Total jarak tempuh: 290 km
- **Analisis:** Jalur ini menunjukkan bahwa meskipun ada beberapa kemungkinan jalur, rute melalui Probolinggo adalah yang paling efisien. Program berhasil menemukan rute optimal tanpa perlu memeriksa semua kombinasi secara brute-force.

2. Contoh Kasus 2:

- **Input:** Kota asal: Surabaya

Kota tujuan: Madiun

- **Proses:** Program mengevaluasi apakah ada jalur langsung antara kedua kota.
- **Output:** Jalur tercepat: Surabaya → Madiun. Total jarak tempuh: 130 km
- **Analisis:** Karena kedua kota saling terhubung langsung dengan bobot terkecil, program langsung mengembalikan rute tersebut. Hal ini menunjukkan efisiensi algoritma untuk kasus sederhana.

3. Contoh Kasus 3:

- **Input:** Kota asal: Malang

Kota tujuan: Pasuruan

- **Proses:** Algoritma menghitung kemungkinan rute antara kedua kota.
- **Output:** Jalur tercepat: Malang → Pasuruan. Total jarak tempuh: 45 km
- **Analisis:** Ini adalah rute pendek dan langsung, yang menunjukkan kemampuan algoritma Dijkstra dalam menangani koneksi antar kota yang berdekatan.

```

=== Traveling Salesman Problem (TSP) ===
Masukkan kota awal TSP (atau 'exit' untuk keluar): Probolinggo

Rute TSP terbaik:
Probolinggo -> Pasuruan -> Surabaya -> Lamongan -> Madiun -> Kediri -> Blitar -> Malang -> Jember -> Banyuwangi
Total jarak tempuh: 590 km

Visualisasi Graph:
Probolinggo --35km--> Pasuruan
Pasuruan --65km--> Surabaya
Surabaya --50km--> Lamongan
Lamongan --120km--> Madiun
Madiun --65km--> Kediri
Kediri --40km--> Blitar
Blitar --45km--> Malang
Malang --110km--> Jember
Jember --60km--> Banyuwangi

Hitung TSP lagi? (y/n): y

=== Traveling Salesman Problem (TSP) ===
Masukkan kota awal TSP (atau 'exit' untuk keluar): Jember

Rute TSP terbaik:
Jember -> Banyuwangi -> Probolinggo -> Pasuruan -> Malang -> Blitar -> Kediri -> Madiun -> Lamongan -> Surabaya
Total jarak tempuh: 610 km

Visualisasi Graph:
Jember --60km--> Banyuwangi
Banyuwangi --150km--> Probolinggo
Probolinggo --35km--> Pasuruan
Pasuruan --45km--> Malang
Malang --45km--> Blitar
Blitar --40km--> Kediri
Kediri --65km--> Madiun
Madiun --120km--> Lamongan
Lamongan --50km--> Surabaya

Hitung TSP lagi? (y/n): n
PS C:\Users\septi>

```

ANALISIS TRAVELING SALESMEN PROBLEM (TSP)

1. Contoh Kasus 1:

- **Input:** Kota awal: Probolinggo
- **Proses:** Program menghasilkan semua permutasi dari 9 kota lainnya, menghitung total jarak masing-masing rute, lalu memilih rute dengan total jarak minimum.
- **Output:**
 Rute terbaik:
 Probolinggo → Pasuruan → Surabaya → Lamongan → Madiun →
 Kediri → Blitar → Malang → Jember → Banyuwangi
 Total jarak: 590 km
- **Analisis:** Program berhasil menghasilkan rute efisien dengan kompleksitas perhitungan tinggi (karena $9! = 362.880$ kemungkinan). Rute ini

menunjukkan bahwa jalur tidak selalu linier secara geografis, tapi berdasarkan jarak minimum total.

2. Contoh Kasus 2:

- **Input:** Kota awal: Jember
- **Proses:** Sama seperti kasus sebelumnya, hanya berbeda titik awal.
- **Output:**

Rute terbaik:

Jember → Banyuwangi → Probolinggo → Pasuruan → Malang → Blitar
→ Kediri → Madiun → Lamongan → Surabaya

Total jarak: 610 km

- **Analisis:** Meskipun urutannya berbeda, program tetap menghasilkan rute dengan total jarak terkecil. Perbedaan 20 km dari kasus Probolinggo menunjukkan bahwa titik awal mempengaruhi total hasil dalam metode brute-force.

BAB V

KESIMPULAN

Tugas ini berhasil mengimplementasikan representasi graf peta Jawa Timur dengan 10 kota dan 30 jalur menggunakan adjacency list. Algoritma Dijkstra dan TSP brute-force berfungsi dengan baik untuk skala kecil, memberikan solusi jalur terpendek dan rute optimal berdasarkan jarak dalam kilometer. Visualisasi graf (Gambar 1) mendukung analisis dengan menampilkan jarak antar kota secara jelas, sementara peta geografis (akan menyusul) diharapkan melengkapi konteks nyata. Namun, keterbatasan TSP brute-force dengan kompleksitas $O(n!)$ (362.880 perhitungan untuk 9 kota) menunjukkan kebutuhan akan algoritma yang lebih efisien untuk skala besar. Untuk aplikasi lebih luas, pengembangan dapat mencakup integrasi data real-time seperti kondisi jalan, penggunaan algoritma TSP seperti Held-Karp, dan pembuatan antarmuka grafis. Hasil ini dapat menjadi dasar untuk pengembangan sistem navigasi sederhana di Jawa Timur, dengan potensi riset lebih lanjut dalam optimasi rute berbasis kecerdasan buatan.

LAMPIRAN

Github : https://github.com/fitryatkj3/UAS_StrukDat

YouTube : <https://youtu.be/Lk6ygYu71yk>