FINAL PROJECT REPORT
DATA STRUCTURE

By Group 5 :

Dimas Rafif Arrizqi (24091397075)

Achmad Hakim ALJumadi (24091397099)

Hanifah Kurnia Fa'izah (24091397124)

INFORMATICS MANAGEMENT STUDY PROGRAM
FACULTY OF VOCATIONAL SURABAYA
STATE UNIVERSITY
2025

INTRODUCTION

With the rapid development of information and communication technology, the need for navigation systems and digital maps has increased significantly. Systems such as GPS (Global Positioning System) are not only used in mapping and transportation applications but have also become an integral part of various aspects of modern life, including logistics, vehicle management, delivery services, and travel planning. To understand the fundamentals of navigation systems like GPS, a solid grasp of graph data structures and shortest path search algorithms is essential. This laboratory exercise aims to provide a basic understanding of how a simple GPS can be built using graph concepts in computer science.

Graphs are a highly useful data structure for representing relationships between objects. In the context of navigation systems, cities are represented as nodes or vertices, while roads connecting cities are represented as edges linking these nodes. Each edge has a weight representing distance or travel time. By utilizing this graph structure, we can apply various graph algorithms to solve navigation problems, such as finding the fastest route between two cities or determining the optimal route to visit all cities once, as in the case of the Traveling Salesman Problem (TSP).

In this practicum, a graph is constructed with ten major cities located in the vicinity of Kuala Lumpur, Malaysia. Each city is represented as a vertex in the graph, and there are 30 edges depicting road connections between cities with varying travel time and distance weights based on three modes of transportation: car, motorcycle, and walking. Each mode of transportation has the same graph structure but different weights for each edge, reflecting the differences in travel time between modes.

To solve the navigation problem, Dijkstra's algorithm is used to determine the fastest route from one city to another based on the user's chosen mode of transportation. Dijkstra's algorithm is one of the most popular algorithms for finding the shortest path, working by gradually exploring the graph and updating the minimum distance to each node until the destination is reached. The implementation of this algorithm on a multimodal graph is highly relevant as it allows users to compare the effectiveness of different modes of transportation.

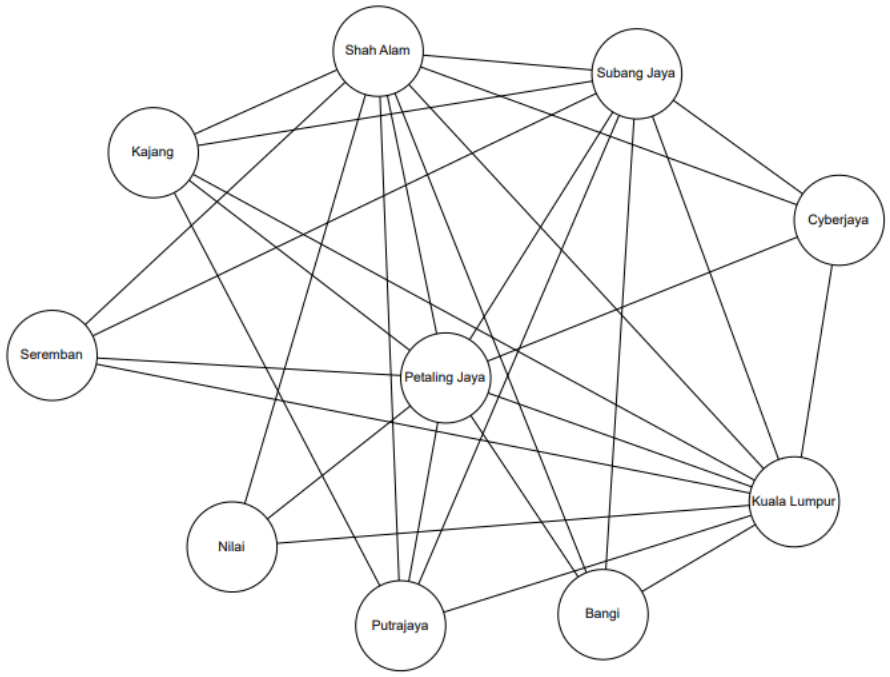
Additionally, to solve the optimal route problem that visits all cities once, the brute-force algorithm for the Traveling Salesman Problem (TSP) is also implemented. Although inefficient for a large number of cities, the brute-force approach allows for a thorough examination of all possible routes and provides definite results for a relatively small number of cities, as in this practical exercise. The results of this algorithm can be used to compare the total time and distance traveled by the optimal route, providing an overview of the overall efficiency of the journey.

Through this practicum, students not only learn about graph data structures and path-finding algorithms but also develop skills in Python programming, system testing, and documentation. This practical exercise is expected to serve as a foundation for developing more complex location-based applications in the future. The use of appropriate data structures and efficient algorithms is key to building reliable, fast, and accurate digital navigation systems.

This practicum also promotes an understanding of the importance of spatial data processing, route planning, and data-driven decision-making. With this approach, students can apply theoretical concepts in the form of real-world implementations that closely resemble modern navigation systems, such as Google Maps or Waze, but on a simpler and more manageable scale and complexity.

IMPLEMENTATION AND TESTING

Graph Representation



Structure Graph

```
import heapq
import itertools

#==== Struktur Graph ====
class MultiModeGraph:
    def __init__(self):
        self.vertices = set()
        self.edges = {}

    def tambah_kota(self, nama):
        self.vertices.add(nama)
        self.edges[nama] = []

    def tambah_jalan(self, dari, ke, waktu_mobil, waktu_motor, waktu_jalan,
                     jarak):
        bobot = {
            "mobil": {"waktu": waktu_mobil, "jarak": jarak},
            "motor": {"waktu": waktu_motor, "jarak": jarak},
            "jalan kaki": {"waktu": waktu_jalan, "jarak": jarak}
        }
        self.edges[dari].append((ke, bobot))
        self.edges[ke].append((dari, bobot))
```

Explanation:

- heapq is used for heapq.heappush() and heapq.heappop() operations in the implementation of the Dijkstra algorithm.
- itertools is used for the itertools.permutations() function in the brute-force implementation of the Traveling Salesman Problem (TSP), which generates all possible city visit sequences.
- tambah_kota is used to add a new city to the graph.
- tambah_jalan is used to add an edge between two cities with time and distance weights for each mode of transportation.

City & Distance Graph

```python
# ===== Graf Kota & Jarak =====
def graf_kota():
    kota_list = [
        "Kuala Lumpur", "Shah Alam", "Petaling Jaya", "Subang Jaya", "Kajang",
        "Putrajaya", "Cyberjaya", "Nilai", "Seremban", "Bangi"
    ]
    graf = MultiModeGraph()
    for kota in kota_list:
        graf.tambah_kota(kota)

    # Format: (dari, ke, waktu_mobil, waktu_motor, waktu_jalan, jarak)
    jalur = [
        ("Kuala Lumpur", "Petaling Jaya", 0.3, 0.35, 3.0, 14),
        ("Kuala Lumpur", "Shah Alam", 0.5, 0.6, 5.0, 25),
        ("Kuala Lumpur", "Subang Jaya", 0.45, 0.5, 4.5, 20),
        ("Kuala Lumpur", "Kajang", 0.6, 0.7, 6.0, 30),
        ("Kuala Lumpur", "Putrajaya", 0.75, 0.85, 7.5, 37),
        ("Kuala Lumpur", "Cyberjaya", 0.8, 0.9, 8.0, 40),
        ("Kuala Lumpur", "Nilai", 1.0, 1.1, 10.0, 50),
        ("Kuala Lumpur", "Seremban", 1.2, 1.3, 12.0, 60),
        ("Kuala Lumpur", "Bangi", 0.5, 1, 5.0, 25),
        ("Petaling Jaya", "Shah Alam", 0.4, 0.45, 4.0, 20),
        ("Petaling Jaya", "Subang Jaya", 0.3, 0.35, 3.0, 15),
        ("Petaling Jaya", "Kajang", 0.5, 0.6, 5.0, 25),
        ("Petaling Jaya", "Putrajaya", 0.7, 0.8, 7.0, 35),
        ("Petaling Jaya", "Cyberjaya", 0.75, 0.85, 7.5, 37),
        ("Petaling Jaya", "Nilai", 0.9, 1.0, 9.0, 45),
        ("Petaling Jaya", "Seremban", 1.1, 1.2, 11.0, 55),
        ("Petaling Jaya", "Bangi", 0.6, 0.7, 6.0, 30),
        ("Shah Alam", "Subang Jaya", 0.25, 0.3, 2.5, 12),
        ("Shah Alam", "Kajang", 0.6, 0.7, 6.0, 30),
        ("Shah Alam", "Putrajaya", 0.7, 0.8, 7.0, 35),
        ("Shah Alam", "Cyberjaya", 0.75, 0.85, 7.5, 37),
        ("Shah Alam", "Nilai", 0.9, 1.0, 9.0, 45),
        ("Shah Alam", "Seremban", 1.1, 1.2, 11.0, 55),
        ("Shah Alam", "Bangi", 0.6, 0.7, 6.0, 30),
        ("Subang Jaya", "Kajang", 0.5, 0.6, 5.0, 25),
        ("Putrajaya", "Subang Jaya", 0.6, 0.7, 6.0, 30),
        ("Subang Jaya", "Cyberjaya", 0.65, 0.75, 6.5, 32),
        ("Subang Jaya", "Nilai", 0.8, 0.9, 8.0, 40),
        ("Subang Jaya", "Seremban", 1.0, 1.1, 10.0, 50),
        ("Kajang", "Putrajaya", 0.4, 0.5, 4.0, 20)
    ]


    for data in jalur:
        graf.tambah_jalan(*data)

    return graf, kota_list
```

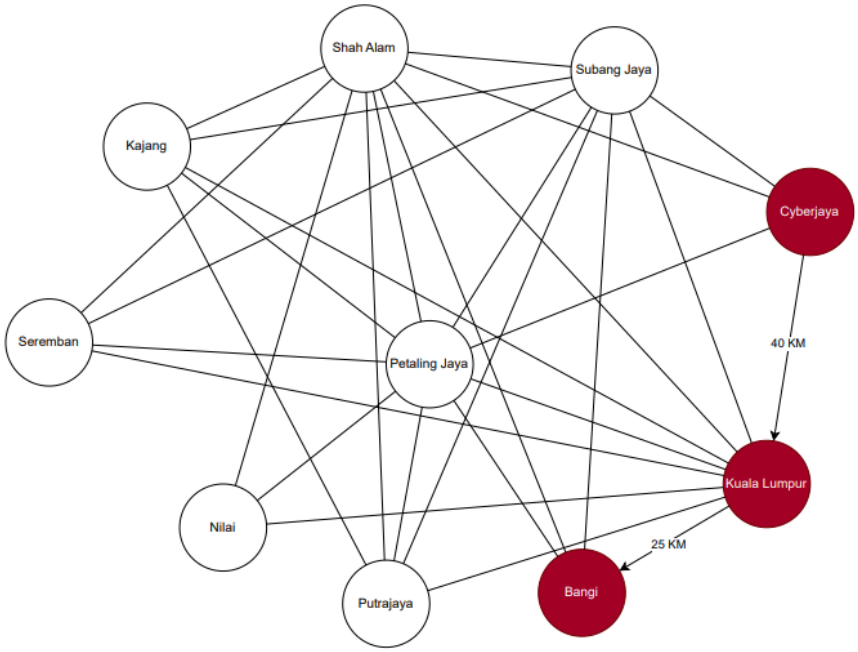
Explanation:

- List of Cities
  - There are 10 cities represented: Kuala Lumpur, Shah Alam, Petaling Jaya, Subang Jaya, Kajang, Putrajaya, Cyberjaya, Nilai, Seremban, and Bangi.
- Path Data
  - Each path connects two cities with the format: (from, to, time_car, time_motorcycle, time_walk, distance)
  - time_car: Travel time by car (in hours)
  - time_motor: Travel time by motorcycle (in hours)
  - time_walk: Travel time on foot (in hours)
  - distance: Distance between cities (in km)

Illustration of the graphs

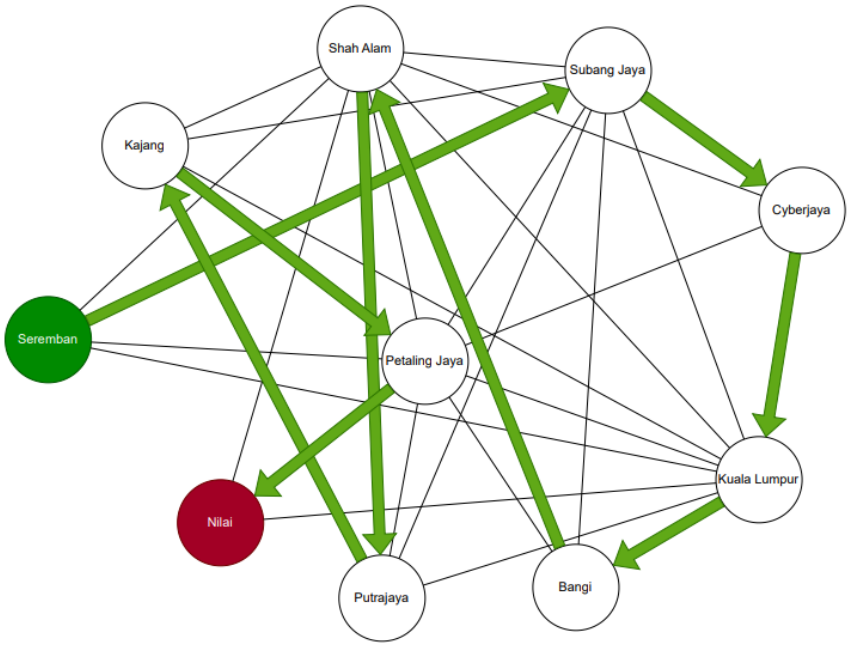
Illustration of a graph from the home city to the destination city

Cyberjaya -> Kuala Lumpur -> Bangi



This graph shows that the input is from the city of Cyberjaya to the destination city of Bangi.

Seremban –> Subang Jaya –> Cyberjaya –> Kuala Lumpur –> Bangi –> Shah Alam –> Putrajaya –> Kajang –> Petaling Jaya –> Nilai



This graph explains the TSP (Traveling Salesman Problem) program that is used to show the route to each city.

Dijkstra & TSP results

A. Dijkstra

```python
def dijkstra(self, awal, tujuan, moda):
    antrian = [(0, 0, awal, [])]  # Urutan input(total_waktu, total_jarak, kota, jalur)
    sudah_dikunjungi = set()

    while antrian:
        waktu, jarak, kota, jalur = heapq.heappop(antrian)
        if kota in sudah_dikunjungi:
            continue
        sudah_dikunjungi.add(kota)
        jalur = jalur + [kota]

        if kota == tujuan:
            return waktu, jarak, jalur

        for tetangga, bobot in self.edges[kota]:
            if tetangga not in sudah_dikunjungi:
                waktu_tambah = bobot[moda]["waktu"]
                jarak_tambah = bobot[moda]["jarak"]
                heapq.heappush(antrian, (waktu + waktu_tambah, jarak + jarak_tambah, tetangga, jalur))

    return float('inf'), float('inf'), []
```

Explanation:

- Initialization
  - Start from the origin city with travel time 0 and distance 0.
  - Store all cities to be visited in a priority queue (heap) based on travel time.
- Node Expansion
  - Take the city with the shortest travel time from the queue.
  - If it is the destination, return the route.
  - Otherwise, expand all the neighbors of that city.
- Distance Update
  - For each neighbor, calculate the new time and distance from the origin city.
  - If the travel time to a neighbor is faster than previously known, update the time and distance values.
- Termination
  - If the queue is empty and the destination has not been found, there is no route.

B. Traveling Salesman Problem(TSP)

```python
def tsp_brute_force(self, moda):
    kota_list = list(self.vertices)
    min_waktu = float('inf')
    min_jarak = float('inf')
    rute_terbaik = None

    for urutan in itertools.permutations(kota_list):
        total_waktu = 0
        total_jarak = 0
        valid = True
        for i in range(len(urutan) - 1):
            tetangga_dict = {n: w for n, w in self.edges[urutan[i]]}
            if urutan[i + 1] in tetangga_dict:
                total_waktu += tetangga_dict[urutan[i + 1]][moda]["waktu"]
                total_jarak += tetangga_dict[urutan[i + 1]][moda]["jarak"]
            else:
                valid = False
                break
        if valid and total_waktu < min_waktu:
            min_waktu = total_waktu
            min_jarak = total_jarak
            rute_terbaik = urutan

    return min_waktu, min_jarak, rute_terbaik
```

Explanation:

- Generate Permutation:
  - Create all possible sequences of city visits (permutations).
- Calculate Total Weight
  - For each permutation, calculate the total time and distance of following that sequence.
  - If any route between consecutive cities is invalid, ignore the permutation.
- Choose the Best Route
  - Compare all valid permutations.
  - Choose the one with the shortest total time.

Main Program

```python
# ===== Program Utama =====
def main():
    graf, kota_list = graf_kota()

    print("==== RUTE TERCEPAT MENGGUNAKAN METODE DIJKSTRA ====")
    print("Daftar Kota Yang Tersedia:", ', '.join(kota_list))

    moda = input("Silakan pilih moda transportasi (mobil, motor, atau jalan kaki): ").strip().lower()
    asal = input("Tulis nama kota asal anda: ").strip().title()
    tujuan = input("Tulis nama kota tujuan anda: ").strip().title()

    if asal not in kota_list or tujuan not in kota_list:
        print("Nama kota yang dimasukkan tidak tersedia.")
        return

    if moda not in ["mobil", "motor", "jalan kaki"]:
        print("Jenis transportasi yang dipilih tidak tersedia.")
        return

    waktu, jarak, jalur = graf.dijkstra(asal, tujuan, moda)
    if jalur:
        print(f"\nRute tercepat dari {asal} ke {tujuan} menggunakan {moda}:")
        print(" -> ".join(jalur))
        print(f"Waktu tempuh: {waktu:.2f} jam")
        print(f"Jarak tempuh: {jarak:.2f} km\n")
    else:
        print("Tidak ada rute yang valid.\n")

    print("=== RUTE TSP TERBAIK (Mengunjungi semua kota) ===")
    waktu_tsp, jarak_tsp, rute_tsp = graf.tsp_brute_force(moda)
    if rute_tsp:
        print(f"Rute TSP: {' -> '.join(rute_tsp)}")
        print(f"Total waktu perjalanan: {waktu_tsp:.2f} jam")
        print(f"Total jarak perjalanan: {jarak_tsp:.2f} km\n")
    else:
        print("Tidak ada rute TSP yang valid ditemukan.\n")

if __name__ == "__main__":
    main()
```

Explanation:

a. Initializing the Graph and City List
   The graph_city() function returns:
   - graph: A graph object containing the relationships between cities along with their distances and travel times.
   - city_list: A list of available city names.

b. User Input Mode of Transportation
   - The user is asked to select a mode of transportation (car, motorcycle, or walking).
   - Input is normalized to lowercase (lower()).
     Origin & Destination Cities
   - The user enters the names of the origin and destination cities.
   - Input is normalized to title case (title()) for consistency.

c. Input Validation

   Available Cities?

   - If the origin/destination city is not in the city_list, the program stops with an error message.

   Valid Transportation Mode?

   - If the mode is not available, the program stops with an error message.

d. Fastest Route Search (Dijkstra)
   - time, distance, route = graph.dijkstra(origin, destination, mode)
   - The Dijkstra algorithm is used to find the fastest route based on the transportation mode.
   - Results:

     time: Total travel time (hours).

     distance: Total travel distance (km).

     route: Sequence of cities passed through.

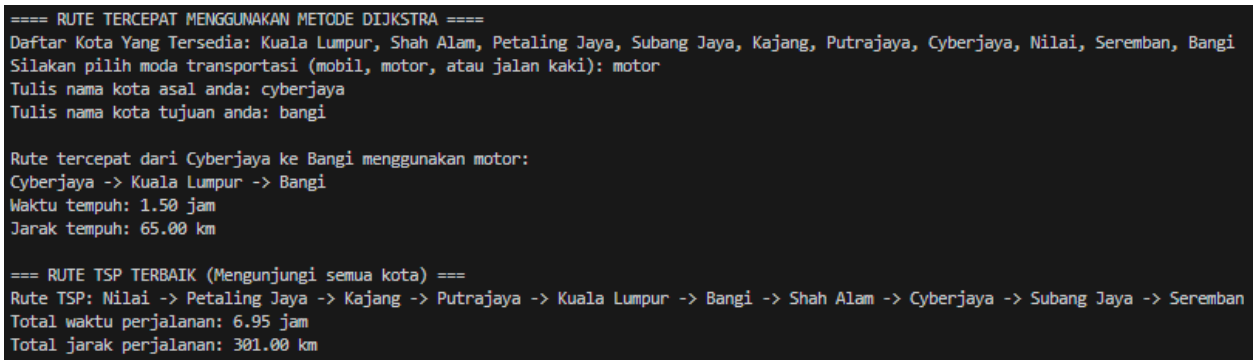
e. TSP Route Search (Traveling Salesman Problem)
   - tsp_time, tsp_distance, tsp_route = graph.tsp_brute_force(mode)
   - Using the Brute Force approach to find the shortest route that visits all cities.
   - Results:

     tsp_time: Total travel time (hours).

     tsp_distance: Total travel distance (km).

     route_tsp: Sequence of visits to all cities.

Screenshot of program output



Explanation:

Fastest Route using Dijkstra:

Using Dijkstra's method, the results show that the route through Kuala Lumpur is the optimal route, with a travel time of 1.50 hours and a distance of 65.00 km. This algorithm considers the weight (time or distance) between cities based on the selected mode of transportation (motorcycle), then chooses the path with the smallest total weight. This output reflects travel efficiency by prioritizing the fastest time.

TSP (Traveling Salesman Problem) Route:

Using the TSP route by visiting all available cities in one cycle, starting from Nilai and ending in Seremban. The total travel time is 6.95 hours with a distance of 301.00 km. The TSP algorithm is used to find the shortest route that passes through each city exactly once. This output provides a complete travel overview with estimated time and distance, useful for logistics planning or tours.

CONCLUSION AND ATTACHMENT

- CONCLUSION

  This practicum effectively demonstrates the application of graph data structures and algorithms in building a simple GPS navigation system. By representing cities as nodes and roads as weighted edges, Dijkstra's algorithm efficiently finds the shortest path between two cities based on the chosen mode of transportation (car, motorcycle, or walking). The implementation highlights how theoretical concepts like graph traversal and priority queues translate into practical solutions for real-world problems, such as route optimization in navigation systems. Additionally, the inclusion of multimodal weights allows users to compare travel times and distances, emphasizing the adaptability of graph-based models to diverse scenarios.

  The brute-force approach to solving the Traveling Salesman Problem (TSP) further reinforces the importance of algorithmic efficiency, especially when dealing with combinatorial complexity. While the brute-force method guarantees an optimal solution for small graphs (e.g., 10 cities), its impracticality for larger datasets underscores the need for heuristic or approximate algorithms in real-world applications. This exercise provides valuable insights into the trade-offs between computational accuracy and performance, preparing students to tackle more advanced optimization challenges in fields like logistics and urban planning.

  Overall, this project underscores the significance of graph data structures and algorithmic problem-solving in the development of location-based services. Students gain hands-on experience in implementing Dijkstra's algorithm and TSP, while also understanding their limitations and practical considerations. The project serves as a foundational step toward more complex navigation systems, emphasizing the role of efficient algorithms and data-driven decision-making in modern technology.

- LINK ATTACHMENT

  LINK VIDEO YOUTUBE: https://youtu.be/FfqTkjyYdPo

  LINK CODE GITHUB: https://github.com/Dimas-RA/Struktur-data