

Nama: Dewa Bagus Putu Arya Dhananjaya

NPM: 10122362

Kelas: 3KA21

Implementasi algoritma Dijkstra untuk mencari shortest-path

Deskripsi

Pemerintah kota ingin mengoptimalkan sistem transportasi dengan mencari jalur tercepat antara dua lokasi berdasarkan kondisi lalu lintas. Setiap jalan memiliki bobot berdasarkan waktu tempuh rata-rata akibat kepadatan lalu lintas. Tujuan utama adalah mengurangi waktu perjalanan dan meningkatkan efisiensi rute transportasi. Untuk menyelesaikan masalah ini, kita menggunakan **algoritma Dijkstra** untuk menemukan jalur terpendek dari satu titik ke titik lain di jaringan jalan kota.

Spesifikasi

- **Directed**
 - **Weighted (berdasarkan waktu tempuh dalam menit)**
 - **Kemungkinan Dense**
 - **Karakteristik masalah sesuai dengan tipe: Shortest-Path-Problem**
-

Implementasi Algoritma Dijkstra dalam Python

Berikut adalah implementasi algoritma Dijkstra untuk mencari jalur terpendek berdasarkan waktu tempuh dalam jaringan transportasi perkotaan:

```

import heapq

# Definisikan graf (rute jalan dan waktu tempuh dalam menit)
city_graph = {
    'Terminal': {'A': 8, 'B': 12},
    'A': {'C': 6, 'D': 10},
    'B': {'D': 5, 'E': 15},
    'C': {'F': 7},
    'D': {'F': 3, 'G': 9},
    'E': {'G': 4},
    'F': {'H': 5},
    'G': {'H': 2},
    'H': {}
}

# Implementasi algoritma Dijkstra
def dijkstra(graph, start, end):
    # Inisialisasi jarak ke semua node dengan nilai tak hingga
    distances = {node: float('inf') for node in graph}
    distances[start] = 0 # Jarak node awal ke dirinya sendiri = 0
    priority_queue = [(0, start)] # (jarak, node)
    predecessors = {} # Menyimpan jalur yang ditempuh

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        # Jika mencapai tujuan, rekonstruksi jalur
        if current_node == end:
            path = []
            while current_node:
                path.append(current_node)
                current_node = predecessors.get(current_node)
            return distances[end], list(reversed(path))

        # Jika jarak saat ini lebih besar dari yang diketahui, lewati
        if current_distance > distances[current_node]:
            continue

        # Periksa semua tetangga
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight


            # Jika menemukan jalur lebih pendek
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                predecessors[neighbor] = current_node
                heapq.heappush(priority_queue, (distance, neighbor))

    return float('inf'), [] # Jika tidak ada jalur

# Contoh penggunaan
start_location = 'Terminal'
destination = 'H'
shortest_time, path = dijkstra(city_graph, start_location, destination)

# Output hasil
print(f"Jalur tercepat dari {start_location} ke {destination}: {path}")
print(f"Total waktu tempuh: {shortest_time} menit")

```

 Jalur tercepat dari Terminal ke H: ['Terminal', 'B', 'D', 'F', 'H']
 Total waktu tempuh: 25 menit

Kodingan Python:

```
import heapq

# Definisikan graf (rute jalan dan waktu tempuh dalam menit)
city_graph = {
    'Terminal': {'A': 8, 'B': 12},
    'A': {'C': 6, 'D': 10},
    'B': {'D': 5, 'E': 15},
    'C': {'F': 7},
    'D': {'F': 3, 'G': 9},
    'E': {'G': 4},
    'F': {'H': 5},
    'G': {'H': 2},
    'H': {}
}

# Implementasi algoritma Dijkstra
def dijkstra(graph, start, end):
    # Inisialisasi jarak ke semua node dengan nilai tak hingga
    distances = {node: float('inf') for node in graph}
    distances[start] = 0 # Jarak node awal ke dirinya sendiri = 0
    priority_queue = [(0, start)] # (jarak, node)
    predecessors = {} # Menyimpan jalur yang ditempuh

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        # Jika mencapai tujuan, rekonstruksi jalur
        if current_node == end:
```

```

    path = []

    while current_node:
        path.append(current_node)
        current_node = predecessors.get(current_node)
    return distances[end], list(reversed(path))

# Jika jarak saat ini lebih besar dari yang diketahui, lewati
if current_distance > distances[current_node]:
    continue

# Periksa semua tetangga
for neighbor, weight in graph[current_node].items():
    distance = current_distance + weight

# Jika menemukan jalur lebih pendek
if distance < distances[neighbor]:
    distances[neighbor] = distance
    predecessors[neighbor] = current_node
    heapq.heappush(priority_queue, (distance, neighbor))

return float('inf'), [] # Jika tidak ada jalur

# Contoh penggunaan
start_location = 'Terminal'
destination = 'H'
shortest_time, path = dijkstra(city_graph, start_location, destination)

# Output hasil
print(f"Jalur tercepat dari {start_location} ke {destination}: {path}")
print(f"Total waktu tempuh: {shortest_time} menit")

```

Output:

Jalur tercepat dari Terminal ke H: ['Terminal', 'B', 'D', 'F', 'H']

Total waktu tempuh: 25 menit