MODUL PERKULIAHAN

TFC251 – Praktikum Struktur Data

Algoritma Djikstra

Penyusun Modul	:	Suamanda Ika Novichasari, M.Kom
Minggu/Pertemuan	:	13
Sub-CPMK/Tujuan Pembelajaran	:	1. Mahasiswa mampu menerapkan algoritma djikstra pada kasus minimum spaning tree menggunakan bahasa pemrograman python
Pokok Bahasan	:	1. Algoritma Djikstra 2. Minimum Spaning Tree

Program Studi Teknologi Informasi (S1)
Fakultas Teknik
Universitas Tidar
Tahun 2024

Materi 10
ALGORITMA DJIKSTRA DAN ALGORITMA GREEDY

Petunjuk Praktikum :

- Cobalah semua contoh penyelesaian kasus dengan kode program yang terdapat pada semua sub bab dalam modul ini menggunakan google colab.
- Dokumentasikan kegiatan dalam bentuk laporan studi kasus sesuai template laporan praktikum yang telah ditentukan.

Dalam komputasi dan teori graf, Minimum Spanning Tree (MST) adalah konsep penting yang digunakan untuk mengoptimalkan jaringan yang terhubung. Pada graf yang terhubung dan memiliki bobot, terdapat banyak pohon rentang (spanning tree), namun MST adalah pohon yang memiliki total bobot paling rendah dibandingkan dengan semua pohon rentang lainnya. MST sering digunakan dalam berbagai bidang, seperti desain jaringan, perencanaan infrastruktur, dan bahkan dalam pengembangan algoritma genetika.

Masalah Knapsack adalah salah satu masalah optimasi klasik dalam ilmu komputer dan teori keputusan. Masalah Knapsack sering digunakan dalam konteks seperti alokasi sumber daya, pemilihan portofolio investasi, dan pengemasan barang. Algoritma dinamis dan teknik pemrograman lainnya digunakan untuk menemukan solusi optimal atau mendekati optimal untuk masalah ini, menjadikannya topik yang menarik dan penting dalam studi algoritma dan optimasi.

10.1 Algoritma Djikstra

Algoritma Dijkstra adalah algoritma pencarian jalur terpendek yang populer dalam teori graf, ilmu komputer, dan perencanaan transportasi. Algoritma ini ditemukan oleh ilmuwan komputer asal Belanda, Edsger W. Dijkstra, pada tahun 1956 saat ia bekerja pada solusi masalah rute di Mathematical Centre di Amsterdam. Sejak saat itu, algoritma Dijkstra telah diadopsi di berbagai bidang untuk menyelesaikan masalah optimasi yang melibatkan pencarian jalur terpendek antara dua titik pada

sebuah graf. Keindahan pendekatan pemrograman dinamis ini terletak tidak hanya pada kesederhanaannya tetapi juga pada efisiensinya, yang membuatnya menjadi alat penting untuk berbagai aplikasi seperti sistem navigasi, protokol perutean jaringan, dan bahkan algoritma media sosial.

Intuisi di balik Algoritma Dijkstra didasarkan pada prinsip mengunjungi semua simpul tetangga dari simpul awal sambil melacak jarak terkecil dari simpul awal sejauh ini. Algoritma ini beroperasi dengan mengikuti langkah-langkah berikut:

- Buat array yang menyimpan jarak setiap simpul dari simpul awal. Awalnya, atur jarak ini ke tak terhingga untuk semua simpul kecuali simpul awal yang diatur ke 0.
- Buat antrian prioritas (heap) dan masukkan simpul awal dengan jaraknya yang bernilai 0.
- Selama masih ada simpul yang tersisa di antrian prioritas, pilih simpul dengan jarak terkecil yang tercatat dari simpul awal dan kunjungi simpul-simpul tetangganya.
- Untuk setiap simpul tetangga, periksa apakah sudah dikunjungi atau belum. Jika belum dikunjungi, hitung jarak sementara dengan menambahkan bobotnya ke jarak terkecil yang ditemukan sejauh ini untuk simpul induknya (simpul awal dalam kasus simpul tingkat pertama).
- Jika jarak sementara ini lebih kecil dari nilai yang tercatat sebelumnya (jika ada), perbarui dalam array 'jarak'.
- Akhirnya, tambahkan simpul yang telah dikunjungi dengan jarak yang diperbarui ke antrian prioritas kita dan ulangi langkah ke-3 sampai kita mencapai tujuan atau semua simpul habis.

Dengan iterasi pada semua simpul tetangga, dapat dipastikan semua jalur yang mungkin telah tereksplorasi untuk menentukan mana yang memiliki total biaya (jarak) terpendek. Struktur data antrian prioritas digunakan untuk melacak simpul mana yang perlu dikunjungi selanjutnya secara efisien alih-alih memindai setiap simpul dalam setiap iterasi.

Kelebihan Hashing	Kekurangan Hashing
• Menjamin menemukan jalur terpendek dalam graf berbobot dengan bobot tepi non-negatif. • Mudah dipahami dan diimplementasikan.	• Tidak berfungsi dengan benar jika ada bobot tepi negatif karena mengasumsikan semua bobot tepi non-negatif.

• Dapat diterapkan dalam berbagai aplikasi, seperti navigasi GPS, sistem penjualan tiket pesawat, dan protokol perutean jaringan. • Dapat menghitung jarak dari satu node ke semua node lainnya dalam graf secara efisien.	• Tidak cocok untuk graf dengan ribuan atau jutaan node karena kompleksitas waktunya adalah O(N^2), di mana N adalah jumlah node. • Implementasinya mungkin memerlukan struktur data antrian prioritas, yang bisa lebih rumit dibandingkan dengan struktur data lain dalam beberapa bahasa pemrograman atau lingkungan.

10.2 Minimum Spaning Tree

Langkah-langkah implementasi Algoritma Djikstra untuk pencarian jalur terpendek ke dalam bahasa pemrograman python adalah sebagai berikut :

a. mendefinisikan fungsi untuk mencari node dengan jarak terkecil yang belum dikunjungi.

```
[ ]  def min_distance(distances, visited):
         # Initialize minimum distance for next node
         min_val = float('inf')
         min_index = -1

         # Loop through all nodes to find minimum distance
         for i in range(len(distances)):
             if distances[i] < min_val and i not in visited:
                 min_val = distances[i]
                 min_index = i

         return min_index
```

b. Mendefinisikan fungsi untuk mengimplementasikan algoritma djikstra

```python
def dijkstra_algorithm(graph, start_node):

    # Get total number of nodes in the graph
    num_nodes = len(graph)

    # Initialize distance and visited arrays
    distances = [float('inf')] * num_nodes
    visited = []

    # Set distance at starting node to 0 and add to visited list
    distances[start_node] = 0

    # Loop through all nodes to find shortest path to each node
    for i in range(num_nodes):

        # Find minimum distance node that has not been visited yet
        current_node = min_distance(distances, visited)

        # Add current_node to list of visited nodes
        visited.append(current_node)

        # Loop through all neighboring nodes of current_node
        for j in range(num_nodes):

            # Check if there is an edge from current_node to neighbor
            if graph[current_node][j] != 0:

                # Calculate the distance from start_node to neighbor,
                # passing through current_node
                new_distance = distances[current_node] + graph[current_node][j]

                # Update the distance if it is less than previous recorded value
                if new_distance < distances[j]:
                    distances[j] = new_distance

    # Return the list of the shortest distances to all nodes
    return distances
```

c. Membuat graf 2D sebagai contoh penerapan

```python
# Example graph represented as a 2D array
graph = [[0, 7, 9, 0, 0, 14],
         [7, 0, 10, 15, 0, 0],
         [9, 10, 0, 11, 0, 2],
         [0, 15, 11, 0, 6, 0],
         [0, 0, 0, 6, 0 ,9],
         [14, 0 ,2 ,0 ,9 ,8 ,10]]

# Call Dijkstra's algorithm to find shortest paths from node 'A'
# (index of 'A' in the array is 0)
shortest_distances = dijkstra_algorithm(graph, 0)

# Print the resulting shortest distances
print(shortest_distances)
```

```
[0, 7, 9, 20, 20, 11]
```

Kemudian untuk melihat visualisasi dari graf tersebut dapat dilakukan dengan cara berikut:

a. Install library matplotlib networkx imageio

```
pip install matplotlib networkx imageio
```

```
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (3.7.1)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (3.3)
Requirement already satisfied: imageio in /usr/local/lib/python3.10/dist-packages (2.31.6)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.2.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (4.51.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.5)
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.25.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (24.0)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)
```

b. Kemudian impor libraby dan menyiapkan fungsi untuk menggambar graf

```
[ ] import networkx as nx
    import matplotlib.pyplot as plt
    import imageio
    import os
    import shutil
    import heapq
```

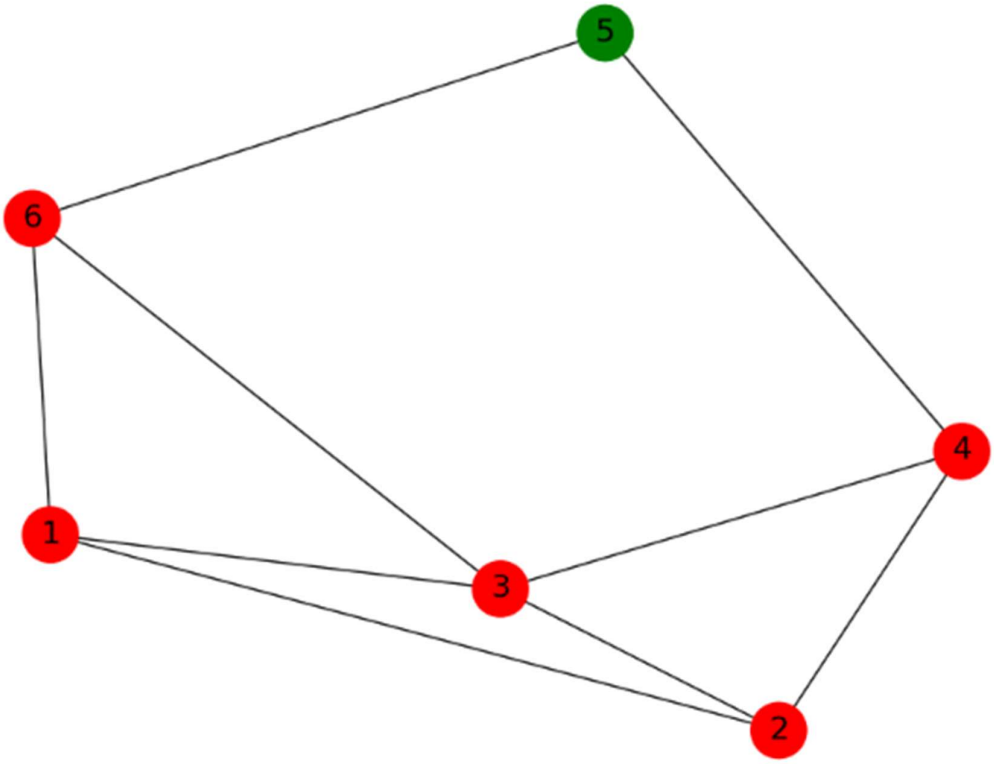
```
[ ] def draw_graph(G, node_colors, edge_colors, pos, frame_id):
        plt.figure(figsize=(8, 6))
        nx.draw(G, pos, node_color=node_colors, edge_color=edge_colors, with_labels=True, node_size=800, font_size=16)
        plt.savefig(f"frames/frame_{frame_id:03d}.png")
        plt.close()
```

c. Selanjutnya, implementasikan algoritma djikstra seperti yang telah dijelaskan sebelumnya dan menganimasikan prosesnya:

```
def animate_dijkstra(graph, start_node):
    os.makedirs("frames", exist_ok=True)
    frame_id = 0
    pos = nx.spring_layout(graph, seed=42)
    visited = {node: False for node in graph.nodes}
    distances = {node: float("inf") for node in graph.nodes}
    distances[start_node] = 0
    pq = [(0, start_node)]

    while pq:
        current_distance, current_node = heapq.heappop(pq)
        if visited[current_node]:
            continue
        visited[current_node] = True

        # Draw the graph at this step
        node_colors = ["green" if node == current_node else ("red" if visited[node] else "gray") for node in graph.nodes]
        edge_colors = ["black" for edge in graph.edges]
        draw_graph(graph, node_colors, edge_colors, pos, frame_id)
        frame_id += 1

        for neighbor, edge_weight in graph[current_node].items():
            new_distance = current_distance + edge_weight["weight"]
            if not visited[neighbor] and new_distance < distances[neighbor]:
                distances[neighbor] = new_distance
                heapq.heappush(pq, (new_distance, neighbor))

    # Generate the animated GIF
    images = []
    for i in range(frame_id):
        images.append(imageio.imread(f"frames/frame_{i:03d}.png"))
    imageio.mimsave("dijkstra.gif", images, duration=1)

    # Clean up the frames folder
    shutil.rmtree("frames")
```

d. Kemudian buat graf dan tampilkan

```
# Create a weighted graph
G = nx.Graph()
G.add_weighted_edges_from([(1, 2, 7), (1, 3, 9), (1, 6, 14), (2, 3, 10), (2, 4, 15),
                           (3, 4, 11), (3, 6, 2), (4, 5, 6), (5, 6, 9)])

# Run the animation for the graph
animate_dijkstra(G, 1)

# Display the resulting GIF (This will work in Jupyter Notebook)
from IPython.display import Image
Image(filename="dijkstra.gif")
```



Latihan 1

Jelaskan apa saja perbedaan dari penerapan algoritma djikstra berikut ini dibandingkan dengan yang ada pada halaman 4-5 diatas !

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
            self.adj_matrix[v][u] = weight  # For undirected graph

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data

    def dijkstra(self, start_vertex_data):
        start_vertex = self.vertex_data.index(start_vertex_data)
        distances = [float('inf')] * self.size
        distances[start_vertex] = 0
        visited = [False] * self.size
        for _ in range(self.size):
            min_distance = float('inf')
            u = None
            for i in range(self.size):
                if not visited[i] and distances[i] < min_distance:
                    min_distance = distances[i]
                    u = i
            if u is None:
                break
            visited[u] = True
            for v in range(self.size):
                if self.adj_matrix[u][v] != 0 and not visited[v]:
                    alt = distances[u] + self.adj_matrix[u][v]
                    if alt < distances[v]:
                        distances[v] = alt
        return distances

g = Graph(7)

g.add_vertex_data(0, 'A')
g.add_vertex_data(1, 'B')
g.add_vertex_data(2, 'C')
g.add_vertex_data(3, 'D')
g.add_vertex_data(4, 'E')
g.add_vertex_data(5, 'F')
g.add_vertex_data(6, 'G')

g.add_edge(3, 0, 4)  # D - A, weight 5
g.add_edge(3, 4, 2)  # D - E, weight 2
g.add_edge(0, 2, 3)  # A - C, weight 3
g.add_edge(0, 4, 4)  # A - E, weight 4
g.add_edge(4, 2, 4)  # E - C, weight 4
g.add_edge(4, 6, 5)  # E - G, weight 5
g.add_edge(2, 5, 5)  # C - F, weight 5
g.add_edge(2, 1, 2)  # C - B, weight 2
g.add_edge(1, 5, 2)  # B - F, weight 2
g.add_edge(6, 5, 5)  # G - F, weight 5

# Dijkstra's algorithm from D to all vertices
print("Dijkstra's Algorithm starting from vertex D:\n")
distances = g.dijkstra('D')
for i, d in enumerate(distances):
    print(f"Shortest distance from D to {g.vertex_data[i]}: {d}")
```

Latihan 2

Diberikan sebuah graf berarah dengan N node dan E sisi, di mana setiap sisi memiliki bobot > 1. Diberikan juga node sumber S dan tujuan D. Tugasnya adalah menemukan jalur dengan hasil kali bobot sisi minimum dari S ke D. Jika tidak ada jalur dari S ke D, cetak -1.

Contoh:

- Masukan: N = 3, E = 3, Tepi = {{{1, 2}, 5}, {{1, 3}, 9}, {{2, 3}, 1}}, S = 1, dan D = 3

  Keluaran: 5

  Jalur dengan hasil kali tepi terkecil adalah 1->2->3

  dengan hasil kali 5*1 = 5.

- Masukan: N = 3, E = 3, Tepi = {{{3, 2}, 5}, {{3, 3}, 9}, {{3, 3}, 1}}, S = 1, dan D = 3

  Keluaran: -1

Latihan 3

Diberikan graf tak berarah dengan N simpul (diberi nomor dari 0 sampai N-1) di mana setiap simpul dapat dicapai dari simpul lainnya dan tidak ada sisi yang sejajar. Juga diberikan array bilangan bulat edge[] di mana setiap elemen berbentuk {u, v, t} yang mewakili tepi antara u dan v dan memerlukan waktu t untuk berpindah melalui tepi tersebut. Tugasnya adalah mencari jumlah cara untuk mencapai node 0 ke node N-1 dalam waktu minimum.

Contoh:

- Masukan : N = 7, M = 10, tepi = [[0, 6, 7], [0, 1, 2], [1, 2, 3], [1, 3, 3], [6, 3, 3], [3, 5, 1], [6, 5, 1], [2, 5, 1], [0, 4, 5], [4, 6, 2]]
  Keluaran: 4
  Penjelasan: Keempat cara untuk sampai ke sana dalam 7 menit adalah:
  0->6
  0->4->6
  0->1->2->5->6
  0->1->3->5->6

- Masukan: N = 6, M = 8, tepian = [[0, 5, 8], [0, 2, 2], [0, 1, 1], [1, 3, 3], [1, 2, 3], [2, 5, 6], [3, 4, 2], [4, 5, 2]]
  Output: 3
  Penjelasan: Tiga cara untuk sampai ke sana dalam 8 menit adalah:
  0->5
  0-> 2->5
  0->1->3->4->5