LAPORAN PRAKTIKUM

STRUKTUR DATA

MODUL KE-10

ALGORITMA DJIKSTRA

Disusun Oleh:

Nama	**:** Oktario Mufti Yudha
NPM	**:** 2320506044
Kelas	**:** 04 (Empat)

Program Studi S1 Teknologi Informasi

Fakultas Teknik, Universitas Tidar

Genap 2023/2024

I. Tujuan Praktikum

1. Mahasiswa mampu memahami Algoritma Djikstra pada python
2. Mahasiswa mampu menerapkan Algoritma Djikstra pada python

II. Dasar Teori

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. list.

III. Hasil dan Pembahasan

a. Minimum spaning tree

```python
# Minimum spaning tree
def min_distance(distance, visited):
    min_val = float('inf')
    min_index = -1

    for i in range(len(distance)):
        if distance[i] < min_val and i not in visited:
            min_val = distance[i]
            min_index = i
    return min_index

def djikstra_algorithm(graph, start_node):
    num_nodes = len(graph)
    distance = [float('inf')] * num_nodes
    visited = []
    distance[start_node] = 0

    for i in range(num_nodes):
        current_node = min_distance(distance, visited)
        visited.append(current_node)
        for j in range(num_nodes):
            if graph[current_node][j] != 0:
                new_distance = distance[current_node] + graph[current_node][j]
                if new_distance < distance[j]:
                    distance[j] = new_distance
    return distance
```

```python
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],
         [14,0,2,0,9,10]]

shortest_distance = djikstra_algorithm(graph, 0)
print(shortest_distance)
```

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

Gambar 3.1: Minimum Spaning Tree

1. Fungsi min_distance(distance, visited): Fungsi ini digunakan untuk mencari node dengan jarak minimum yang belum dikunjungi. Fungsi ini menerima dua parameter, yaitu distance yang berisi jarak dari node awal ke node lainnya dan visited yang berisi node-node yang sudah dikunjungi.
2. Fungsi djikstra_algorithm(graph, start_node): Fungsi ini merupakan implementasi dari algoritma Dijkstra. Fungsi ini menerima dua parameter, yaitu graph yang berisi representasi graf dalam bentuk matriks adjacency dan start_node yang merupakan node awal.
3. num_nodes = len(graph): Menghitung jumlah node dalam graf.
4. distance = [float('inf')] * num_nodes: Membuat list distance dengan panjang sama dengan jumlah node dan mengisi setiap elemennya dengan

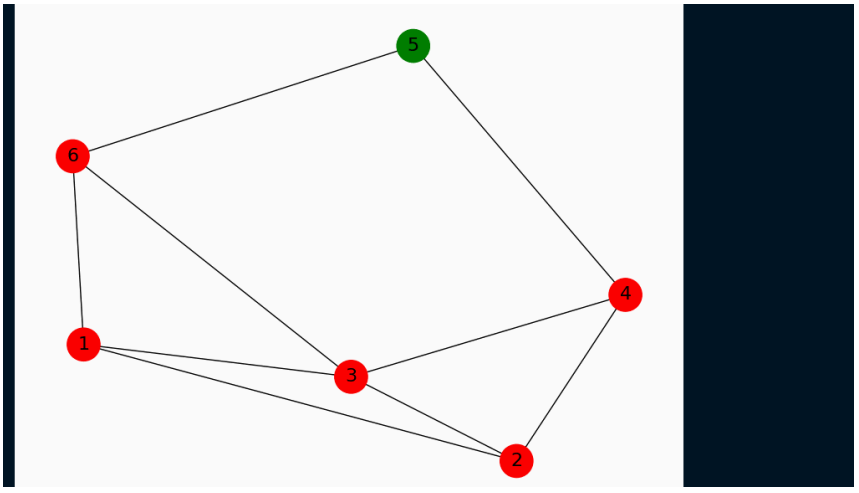
nilai tak hingga. Ini merepresentasikan bahwa jarak awal dari node awal ke node lainnya adalah tak hingga.

5. visited = []: Membuat list visited untuk menyimpan node-node yang sudah dikunjungi.
6. distance[start_node] = 0: Mengatur jarak dari node awal ke dirinya sendiri menjadi 0.
7. Loop for i in range(num_nodes): Melakukan iterasi sebanyak jumlah node.
8. current_node = min_distance(distance, visited): Menentukan node dengan jarak minimum yang belum dikunjungi.
9. visited.append(current_node): Menambahkan node yang baru dikunjungi ke dalam list visited.
10. Loop for j in range(num_nodes): Melakukan iterasi sebanyak jumlah node untuk memeriksa setiap node lainnya.
11. if graph[current_node][j] != 0: Jika ada edge antara current_node dan node j.
12. new_distance = distance[current_node] + graph[current_node][j]: Menghitung jarak baru jika kita pergi dari current_node ke node j.
13. if new_distance < distance[j]: Jika jarak baru lebih kecil dari jarak sebelumnya, maka update jarak tersebut.
14. return distance: Mengembalikan list distance yang berisi jarak terpendek dari node awal ke setiap node lainnya.
15. graph = [[0,7,9,0,0,14],...,[14,0,2,0,9,10]]: Membuat graf dalam bentuk matriks adjacency.
16. shortest_distance = djikstra_algorithm(graph, 0): Menjalankan algoritma Dijkstra pada graf dengan node awal adalah 0.
17. print(shortest_distance): Mencetak jarak terpendek dari node awal ke setiap node lainnya.

b. Menggunakan library matplotlib networkx imageio

```python
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()

def animate_djikstra(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}
    distance = {node: float('inf') for node in graph.nodes}
    distance[start_node] = 0
    pq = [(0, start_node)]
```

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

        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 < distance[neighbor]:
                distance[neighbor] = new_distance
                heapq.heappush(pq, (new_distance, neighbor))

    images = []
    for i in range(frame_id):
        images.append(imageio.imread(f'frames/frame_{i:03d}.png'))
    imageio.mimsave('djikstra.gif', images, duration=5)
    shutil.rmtree('frames')

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)

animate_djikstra(G, 1)
from IPython.display import Image
Image(filename='djikstra.gif')
```



Gambar 3.1: Menggunakan library

1. Fungsi draw_graph(G, node_colors, edge_colors, pos, frame_id): Fungsi ini digunakan untuk menggambar graf dan menyimpannya sebagai gambar PNG. Fungsi ini menerima lima parameter, yaitu G yang merupakan graf, node_colors yang merupakan warna dari setiap node, edge_colors yang

merupakan warna dari setiap edge, pos yang merupakan posisi dari setiap node, dan frame_id yang merupakan nomor frame.

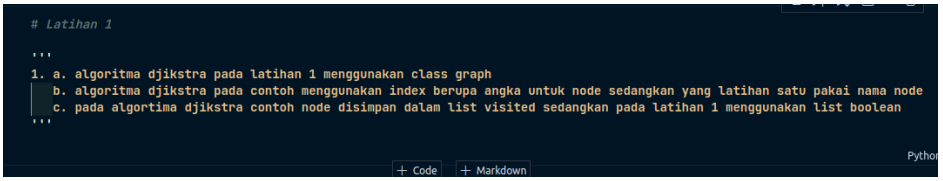
2. Fungsi animate_djikstra(graph, start_node): Fungsi ini merupakan implementasi dari algoritma Dijkstra yang diubah sedikit untuk membuat animasi. Fungsi ini menerima dua parameter, yaitu graph yang berisi representasi graf dan start_node yang merupakan node awal.
3. os.makedirs('frames', exist_ok=True): Membuat direktori bernama 'frames' untuk menyimpan setiap frame dari animasi.
4. pos = nx.spring_layout(graph, seed=42): Mengatur posisi dari setiap node dalam graf menggunakan layout spring.
5. visited = {node: False for node in graph.nodes}: Membuat dictionary visited untuk menyimpan status kunjungan setiap node.
6. distance = {node: float('inf') for node in graph.nodes}: Membuat dictionary distance untuk menyimpan jarak dari node awal ke setiap node lainnya.
7. pq = [(0, start_node)]: Membuat priority queue pq dan memasukkan node awal dengan jarak 0.
8. Loop while pq: Melakukan iterasi selama pq tidak kosong.
9. current_distance, current_node = heapq.heappop(pq): Mengambil node dengan jarak minimum dari pq.
10. if visited[current_node]: Jika current_node sudah dikunjungi, maka lanjutkan ke iterasi berikutnya.
11. node_colors = ['green' if node == current_node else 'red' if visited[node] else 'gray' for node in graph.nodes]: Mengatur warna dari setiap node. Jika node adalah current_node, maka warnanya hijau. Jika node sudah dikunjungi, maka warnanya merah. Jika tidak, maka warnanya abu-abu.
12. edge_colors = ['black'for edge in graph.edges]: Mengatur warna dari setiap edge menjadi hitam.
13. draw_graph(graph, node_colors, edge_colors, pos, frame_id): Menggambar graf dengan warna node dan edge yang sudah ditentukan.

14. for neighbor, edge_weight in graph[current_node].items(): Melakukan iterasi untuk setiap tetangga dari current_node.
15. new_distance = current_distance + edge_weight['weight']: Menghitung jarak baru jika kita pergi dari current_node ke neighbor.
16. if not visited[neighbor] and new_distance < distance[neighbor]: Jika neighbor belum dikunjungi dan jarak baru lebih kecil dari jarak sebelumnya, maka update jarak tersebut dan masukkan neighbor ke dalam pq.
17. images = []: Membuat list images untuk menyimpan setiap frame dari animasi.
18. imageio.mimsave('djikstra.gif', images, duration=5): Membuat file GIF dari setiap frame dan menyimpannya dengan nama 'djikstra.gif'.
19. shutil.rmtree('frames'): Menghapus direktori 'frames' dan semua isinya.
20. G = nx.Graph(): Membuat objek graf G.
21. 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)]): Menambahkan edge dan bobotnya ke dalam graf G.
22. animate_djikstra(G, 1): Menjalankan fungsi animate_djikstra pada graf G dengan node awal adalah 1.
23. Image(filename='djikstra.gif'): Menampilkan animasi yang sudah dibuat.

IV. Latihan

a. Latihan 1

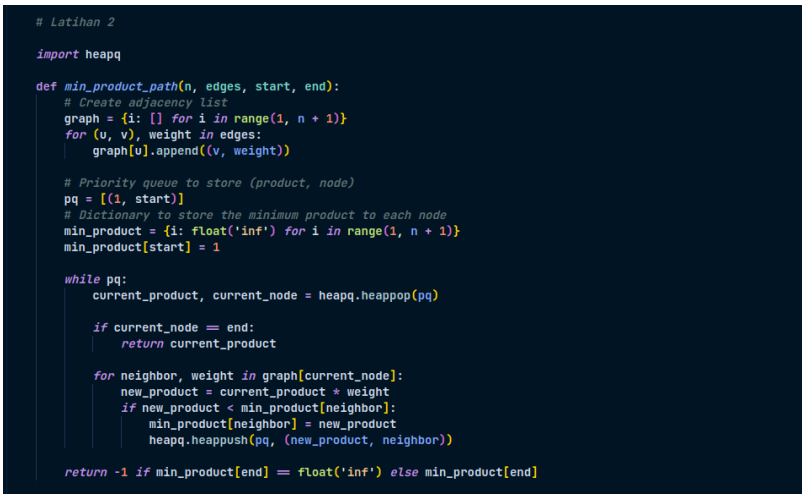
Latihan 1 menjawab soal yang ada di modul yaitu mengenai perbedaan antara algoritma djikstra yang di sediakan dan juga algoritma yang ada pada code sebelumnya

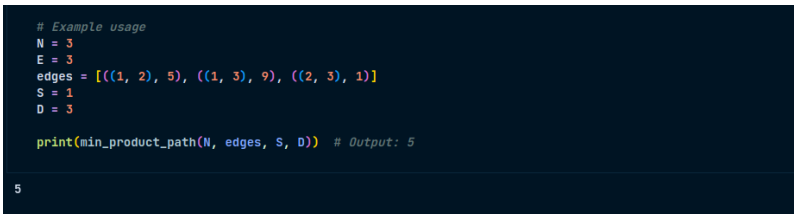


Gambar 4.1: Latihan1

b. 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.





Gambar 4.3: Latihan 2

1. Fungsi min_product_path(n, edges, start, end): Fungsi ini digunakan untuk mencari jalur dengan produk terkecil dari node awal ke node akhir. Fungsi ini menerima empat parameter, yaitu n yang merupakan jumlah node, edges yang merupakan list berisi pasangan node dan bobotnya, start yang merupakan node awal, dan end yang merupakan node akhir.
2. graph = {i: [] for i in range(1, n + 1)}: Membuat dictionary graph untuk menyimpan adjacency list dari setiap node.
3. Loop for (u, v), weight in edges: Melakukan iterasi untuk setiap edge dan bobotnya.
4. graph[u].append((v, weight)): Menambahkan node v dan bobotnya ke dalam adjacency list dari node u.
5. pq = [(1, start)]: Membuat priority queue pq dan memasukkan node awal dengan produk 1.
6. min_product = {i: float('inf') for i in range(1, n + 1)}: Membuat dictionary min_product untuk menyimpan produk terkecil ke setiap node.
7. Loop while pq: Melakukan iterasi selama pq tidak kosong.
8. current_product, current_node = heapq.heappop(pq): Mengambil node dengan produk terkecil dari pq.
9. if current_node == end: Jika current_node adalah node akhir, maka kembalikan current_product.
10. Loop for neighbor, weight in graph[current_node]: Melakukan iterasi untuk setiap tetangga dari current_node.
11. new_product = current_product * weight: Menghitung produk baru jika kita pergi dari current_node ke neighbor.
12. if new_product < min_product[neighbor]: Jika produk baru lebih kecil dari produk sebelumnya, maka update produk tersebut dan masukkan neighbor ke dalam pq.
13. return -1 if min_product[end] == float('inf') else min_product[end]: Jika produk terkecil ke node akhir adalah tak hingga, maka kembalikan -1. Jika tidak, kembalikan produk terkecil tersebut.

14. N = 3, E = 3, edges = [((1, 2), 5), ((1, 3), 9), ((2, 3), 1)], S = 1, D = 3: Mendefinisikan jumlah node, jumlah edge, list edge, node awal, dan node akhir.
15. print(min_product_path(N, edges, S, D)): Menjalankan fungsi min_product_path dan mencetak produk terkecil dari node awal ke node akhir.

V. Kesimpulan

Praktikum ini memberikan pemahaman yang mendalam tentang konsep dan implementasi Algoritma Dijkstra dalam bahasa Python. Algoritma Dijkstra digunakan untuk menemukan jalur terpendek antara dua titik dalam graf berarah dan berbobot positif. Melalui langkah-langkah iteratif dan penggunaan antrian prioritas, algoritma ini secara efisien melacak dan memperbarui jarak terpendek dari simpul awal ke setiap simpul lainnya. Praktikum ini mencakup implementasi Algoritma Dijkstra menggunakan struktur data seperti list dan dictionary untuk merepresentasikan graf. Fungsi seperti min_distance digunakan untuk menentukan node dengan jarak minimum yang belum dikunjungi, sedangkan djikstra_algorithm menjalankan algoritma utamanya. Selain itu, praktikum ini juga menyertakan visualisasi graf menggunakan library seperti matplotlib dan networkx, yang membantu dalam memahami proses algoritma secara lebih intuitif. Secara keseluruhan, praktikum ini berhasil memberikan wawasan yang baik tentang penggunaan dan implementasi Algoritma Dijkstra dalam pemrograman.