

LAPORAN PRAKTIKUM STRUKTUR DATA

MODUL KE-10

ALGORITMA DIJKSTRA



Disusun Oleh:

Nama	Restu Wibisono
NPM	2340506061
Kelas	03 (Tiga)

Program Studi S1 Teknologi Informasi

Fakultas Teknik, Universitas Tidar

Genap 2023/2024

I. Tujuan Praktikum

Adapun tujuan praktikum ini sebagai berikut :

1. Mahasiswa mampu menerapkan algoritma djikstra pada kasus minimum spanning tree menggunakan bahasa pemrograman python

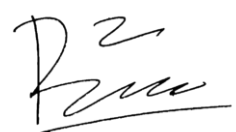
II. Dasar Teori

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.

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

Tanda Tangan

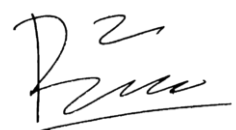


alat penting untuk berbagai aplikasi seperti sistem navigasi, protokol perutean jaringan, dan bahkan algoritma media sosial.

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.

Tanda Tangan

A handwritten signature in black ink, consisting of a stylized 'P' followed by a series of loops and a horizontal stroke at the bottom.

III. Hasil dan Pembahasan

1. Minimum Spaning Tree

```
# Minimum spanning 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

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]]

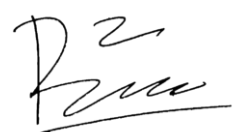
shortest_distance = djikstra_algorithm(graph, 0)
print(shortest_distance)
```

Python

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

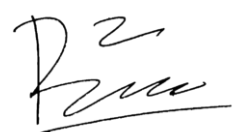
- Fungsi 'def min_distance(distance, visited):' ini digunakan untuk menemukan node dengan jarak terpendek yang belum dikunjungi.
- 'min_val = float('inf')' Variabel 'min_val' diinisialisasi dengan nilai tak terhingga untuk menampung jarak terpendek saat ini.
- 'min_index = -1' Variabel 'min_index' diinisialisasi dengan -1 untuk menandai indeks node dengan jarak terpendek.
- 'for i in range(len(distance)):' Melakukan loop melalui setiap elemen dalam array 'distance'.

Tanda Tangan



- 'if distance[i] < min_val and i not in visited:' Memeriksa apakah jarak ke node saat ini lebih kecil dari 'min_val' dan node tersebut belum dikunjungi.
- 'min_val = distance[i]' Jika ya, nilai 'min_val' diperbarui dengan jarak saat ini.
- 'min_index = i' Variabel 'min_index' diperbarui dengan indeks node saat ini.
- 'return min_index' Mengembalikan indeks dari node dengan jarak terpendek yang belum dikunjungi.
- 'def djikstra_algorithm(graph, start_node):' Fungsi ini mendefinisikan algoritma Dijkstra. Fungsi ini menerima dua parameter 'graph' (representasi graf dalam bentuk matriks berisi bobot setiap edge) dan 'start_node' (node awal dari pencarian jarak terpendek).
- 'num_nodes = len(graph)' Menghitung jumlah node dalam graf.
- 'distance = [float('inf')] * num_nodes' Inisialisasi array 'distance' dengan nilai tak terhingga sepanjang jumlah node dalam graf. Ini akan menyimpan jarak terpendek dari node awal ke setiap node lainnya.
- 'visited = []' Membuat daftar kosong untuk menyimpan node yang sudah dikunjungi.
- 'distance[start_node] = 0' Menandai node awal dengan jarak 0.
- 'for i in range(num_nodes):' Melakukan loop sebanyak jumlah node dalam graf.
- 'current_node = min_distance(distance, visited)' Mendapatkan node dengan jarak terpendek yang belum dikunjungi menggunakan fungsi 'min_distance'.
- 'visited.append(current_node)' Menambahkan node tersebut ke daftar node yang sudah dikunjungi.

Tanda Tangan



- 'for j in range(num_nodes):' Melakukan loop sebanyak jumlah node dalam graf lagi.
- 'if graph[current_node][j] != 0:' Memeriksa apakah terdapat edge antara 'current_node' dan node 'j'.
- 'new_distance = distance[current_node] + graph[current_node][j]' Menghitung jarak baru ke node 'j'.
- 'if new_distance < distance[j]:' Jika jarak baru lebih kecil dari jarak sebelumnya ke node 'j'.
- 'distance[j] = new_distance' Update jarak ke node 'j' dengan jarak baru.
- 'return distance' Mengembalikan array 'distance' yang berisi jarak terpendek dari node awal ke setiap node lainnya dalam graf.

2. Membuat Tampilan Gambar

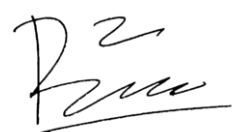
```
import networkx as nx
import matplotlib.pyplot as plt
import imageio
import os
import shutil
import heapq

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)
    plt.savefig(f'frames/frame_{frame_id:03d}.png')
    plt.close()

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}
    distance = {node: float('inf') for node in graph.nodes}
    distance[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
```

Tanda Tangan



```

node_colors = ['green' if node == current_node else 'red' if visited[node] else 'red']
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))

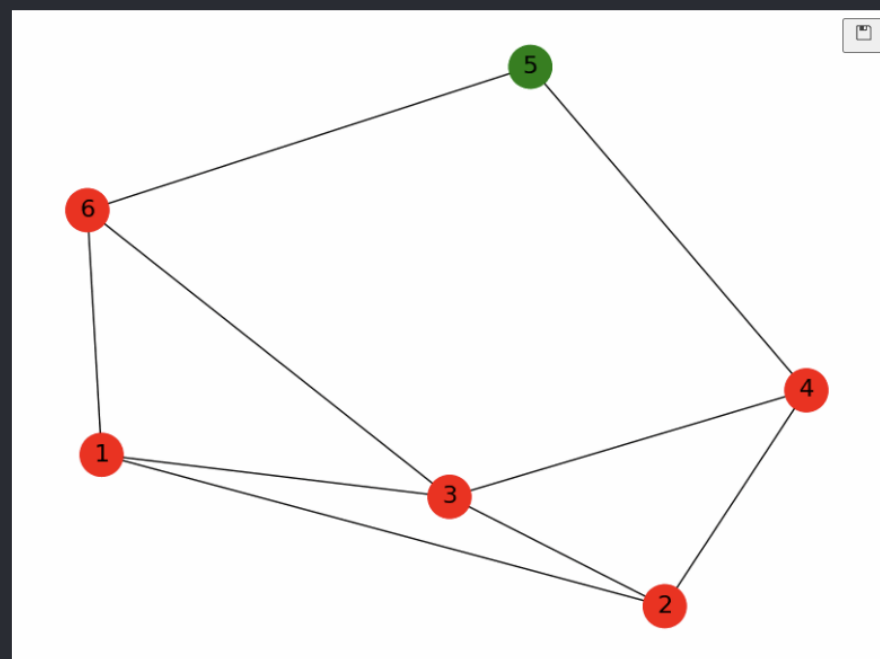
images = []
for i in range(frame_id):
    images.append(imageio.imread(f'frames/frame_{i:03d}.png'))
imageio.mimsave('djikstra.gif', images, duration=5)
shutil.rmtree('frames')

G = nx.Graph()
G.add_weighted_edges_from([(1, 2, 7), (1, 3, 9), (1, 6, 14), (2, 3, 10), (2, 4, 4), (3, 4, 3), (3, 5, 10), (4, 5, 6), (6, 5, 10)])

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

```

/tmp/ipykernel_11362/391035492.py:42: DeprecationWarning: Starting with ImageIO 2.9.0, the preferred API is `imageio.imread(f'frames/frame_{i:03d}.png')`

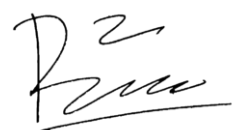


- 'import networkx as nx' Mengimpor pustaka NetworkX dan memberinya alias 'nx'.
- 'import matplotlib.pyplot as plt' Mengimpor pustaka Matplotlib dan memberinya alias 'plt'.
- 'import imageio' Mengimpor pustaka ImageIO.

Tanda Tangan

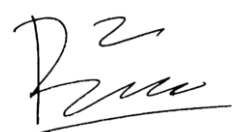
- `'import os'` Mengimpor modul OS untuk fungsi sistem operasi.
- `'import shutil'` Mengimpor modul shutil untuk operasi file.
- `'import heapq'` Mengimpor modul heapq, yang menyediakan implementasi dari algoritma antrian heap.
- `'def draw_graph(G, node_colors, edge_colors, pos, frame_id):'` Mendefinisikan fungsi bernama `'draw_graph'` yang mengambil graf `'G'`, warna node, warna edge, posisi `'pos'`, dan ID frame `'frame_id'` sebagai input.
- `'plt.figure(figsize=(8, 6))'` Membuat gambar baru dengan ukuran 8x6 inci menggunakan Matplotlib.
- `'nx.draw(G, pos, node_color=node_colors, edge_color=edge_colors, with_labels=True, node_size=800, font_size=16)'` Menggambar graf `'G'` menggunakan NetworkX dengan parameter yang ditentukan seperti warna node, warna edge, label, ukuran node, dan ukuran font.
- `'plt.savefig(f'frames/frame_{frame_id:03d}.png')'` Menyimpan gambar saat ini sebagai gambar PNG dengan nama file berdasarkan ID frame dalam direktori bernama `'frames'`.
- `'plt.close()'` Menutup plot Matplotlib saat ini.
- `'def animate_dijkstra(graph, start_node):'` Mendefinisikan fungsi bernama `'animate_dijkstra'` yang mengambil graf `'graph'` dan node awal `'start_node'` sebagai input.
- `'os.makedirs('frames', exist_ok=True)'` Membuat direktori bernama `'frames'` jika belum ada.
- `'frame_id = 0'` Menginisialisasi variabel `'frame_id'` menjadi 0.
- `'pos = nx.spring_layout(graph, seed=42)'` Menghasilkan posisi node menggunakan algoritma spring layout dari NetworkX dengan benih acak yang ditentukan.

Tanda Tangan



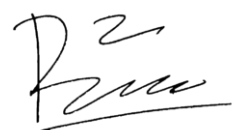
- `'visited = {node False for node in graph.nodes}'` Membuat kamus `'visited'` di mana setiap node awalnya ditandai sebagai belum dikunjungi.
- `'distance = {node float('inf') for node in graph.nodes}'` Membuat kamus `'distance'` di mana jarak ke setiap node awalnya diatur sebagai tak terhingga.
- `'distance[start_node] = 0'` Mengatur jarak ke node awal menjadi 0.
- `'pq = [(0, start_node)]'` Menginisialisasi antrian prioritas `'pq'` dengan tuple yang berisi jarak ke node awal dan node awal itu sendiri.
- `'while pq:'` Memulai loop while yang berlanjut selama antrian prioritas tidak kosong.
- `'current_distance, current_node = heapq.heappop(pq)'` Mengambil node dengan jarak terkecil dari antrian prioritas.
- `'if visited[current_node] continue'` Jika node saat ini sudah dikunjungi, lewati.
- `'visited[current_node] = True'` Tandai node saat ini sebagai sudah dikunjungi.
- `'node_colors = ['green' if node == current_node else 'red' if visited[node] else 'gray' for node in graph.nodes]'` Membuat warna untuk node berdasarkan statusnya (node saat ini, telah dikunjungi, belum dikunjungi).
- `'edge_colors = ['black' for edge in graph.edges]'` Membuat warna untuk edge (hitam dalam kasus ini).
- `'draw_graph(graph, node_colors, edge_colors, pos, frame_id)'` Menggambar keadaan graf saat ini menggunakan fungsi `'draw_graph'`.
- `'frame_id += 1'` Menambahkan ID frame.
- `'for neighbor, edge_weight in graph[current_node].items():'` Mengulangi melalui tetangga dari node saat ini.

Tanda Tangan



- `'new_distance = current_distance + edge_weight['weight']'`
Menghitung jarak baru ke tetangga dengan menambahkan bobot edge.
- `'if not visited[neighbor] and new_distance < distance[neighbor:]'`
Jika tetangga belum dikunjungi dan jarak baru lebih kecil dari jarak yang tercatat ke tetangga:
- Perbarui jarak ke tetangga.
- Dorong jarak baru dan tetangga ke antrian prioritas.
- Membuat daftar gambar dengan membaca file PNG yang disimpan dalam direktori 'frames'.
- Membuat animasi GIF dari daftar gambar dengan durasi 5 detik per frame.
- Hapus direktori 'frames'.
- Membuat graf kosong 'G' menggunakan NetworkX.
- Menambahkan edge berbobot ke graf.
- Memanggil fungsi 'animate_dijkstra' dengan graf yang dibuat 'G' dan node awal '1'.
- Mengimpor kelas 'Image' dari 'IPython.display'.
- Menampilkan file GIF yang dihasilkan bernama 'dijkstra.gif'.

Tanda Tangan



IV. Latihan

1. Latihan 1

```
# Latihan 1
...
a. Algoritma Dijkstra pada latihan 1 menggunakan kelas graph.
b. Algoritma Dijkstra pada contoh menggunakan indeks berupa angka untuk node, sedangkan pada latihan 1 menggunakan nama node.
c. Pada algoritma Dijkstra di contoh, node disimpan dalam daftar visited, sedangkan pada latihan 1 menggunakan daftar boolean.
...
```

2. Latihan 2

```
# Latihan 2

import heapq

def min_product_path(n, edges, start, end):
    # Create adjacency list
    graph = {i: [] for i in range(1, n + 1)}
    for (u, v), weight in edges:
        graph[u].append((v, weight))

    # Priority queue to store (product, node)
    pq = [(1, start)]
    # Dictionary to store the minimum product to each node
    min_product = {i: float('inf') for i in range(1, n + 1)}
    min_product[start] = 1

    while pq:
        current_product, current_node = heapq.heappop(pq)

        if current_node == end:
            return current_product

        for neighbor, weight in graph[current_node]:
            new_product = current_product * weight
            if new_product < min_product[neighbor]:
                min_product[neighbor] = new_product
                heapq.heappush(pq, (new_product, neighbor))

    return -1 if min_product[end] == float('inf') else min_product[end]

# Example usage
N = 3
E = 3
edges = [(1, 2), 5), ((1, 3), 9), ((2, 3), 1)]
S = 1
D = 3

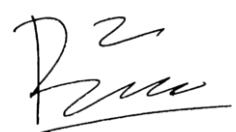
print(min_product_path(N, edges, S, D)) # Output: 5
```

Python

5

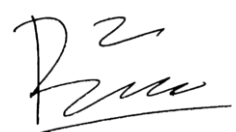
- 'import heapq' Ini mengimpor modul 'heapq', yang menyediakan struktur data heap (tumpukan prioritas), yang akan digunakan dalam program ini.
- 'def min_product_path(n, edges, start, end):' Ini mendefinisikan sebuah fungsi bernama 'min_product_path'.

Tanda Tangan



- `'graph = {i [] for i in range(1, n + 1)}'` Ini membuat kamus kosong yang akan merepresentasikan graf. Setiap node direpresentasikan oleh nomor uniknya dan setiap node awalnya terhubung dengan daftar kosong.
- `'for (u, v), weight in edges graph[u].append((v, weight))'` Ini mengisi kamus `'graph'` dengan sisi-sisi graf dan bobotnya dari daftar `'edges'`. Setiap entri dalam kamus `'graph'` adalah pasangan (node tujuan, bobot) yang terhubung ke node tertentu.
- `'pq = [(1, start)]'` Ini membuat sebuah antrian prioritas awal dengan satu elemen yang berisi tuple `'(1, start)'`. Ini mewakili node awal dengan produk minimum 1.
- `'min_product = {i float('inf') for i in range(1, n + 1)}'` Ini membuat kamus `'min_product'` dengan semua nilai awal tak terbatas (infinity), yang akan digunakan untuk melacak produk minimum ke setiap node.
- `'min_product[start] = 1'` Memperbarui nilai produk minimum untuk node awal menjadi 1.
- `'while pq:'` Ini memulai loop utama yang akan berjalan selama antrian prioritas tidak kosong.
- `'current_product, current_node = heapq.heappop(pq)'` Ini mengambil elemen dengan produk minimum dari antrian prioritas. Setelah diekstraksi, elemen ini diwakili oleh `'current_product'` dan `'current_node'`.
- `'if current_node == end return current_product'` Jika node saat ini adalah node akhir yang diinginkan, maka produk minimum langsung dikembalikan.
- `'for neighbor, weight in graph[current_node]:'` Loop ini mengiterasi melalui tetangga-tetangga dari node saat ini, serta bobot yang terhubung ke tetangga-tetangga tersebut.

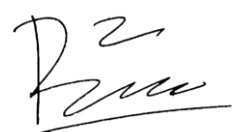
Tanda Tangan



- `'new_product = current_product * weight'` Ini menghitung produk baru dari node saat ini ke tetangga tertentu.
- `'if new_product < min_product[neighbor]:'` Ini memeriksa apakah produk baru lebih kecil dari produk minimum yang saat ini tercatat untuk tetangga tersebut.
- `'min_product[neighbor] = new_product'` Jika produk baru lebih kecil, maka produk minimum untuk tetangga tersebut diperbarui.
- `'heapq.heappush(pq, (new_product, neighbor))'` Node tetangga dan produk minimum baru yang sesuai ditempatkan kembali ke antrian prioritas.
- `'return -1 if min_product[end] == float('inf') else min_product[end]'` Setelah loop selesai, jika produk minimum untuk node akhir masih infinity, itu berarti tidak ada jalur yang tersedia. Dalam kasus ini, `'-1'` dikembalikan. Jika tidak, produk minimum untuk node akhir dikembalikan.
- `'N = 3', 'E = 3', 'edges = [((1, 2), 5), ((1, 3), 9), ((2, 3), 1)]', 'S = 1', 'D = 3'` Ini menentukan parameter untuk contoh yang akan diuji.
- `'print(min_product_path(N, edges, S, D))'` Ini mencetak hasil dari fungsi `'min_product_path'` dengan parameter yang telah ditentukan sebelumnya.
- Output yang diharapkan adalah `'5'`, yang merupakan produk minimum dari jalur dari node 1 ke node 3.

3. Latihan 3

Tanda Tangan



```

from collections import defaultdict, deque

def count_paths_with_min_time(N, M, edges):
    graph = defaultdict(list)
    for u, v, t in edges:
        graph[u].append((v, t))

    min_time = [float('inf')] * N
    min_time[0] = 0

    count_paths = [0] * N
    count_paths[0] = 1

    visited = set()
    visited.add(0)

    queue = deque()
    queue.append(0)

    while queue:
        node = queue.popleft()
        visited.remove(node)

        for neighbor, time in graph[node]:
            if min_time[node] + time <= min_time[neighbor]:
                if min_time[node] + time < min_time[neighbor]:
                    min_time[neighbor] = min_time[node] + time
                    count_paths[neighbor] = count_paths[node]
                else:
                    count_paths[neighbor] += count_paths[node]

            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

    return count_paths[N-1]

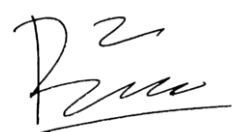
# Example usage:
N = 7
M = 10
edges = [[0, 6, 7], [0, 1, 2], [1, 2, 3], [1, 3, 3], [6, 3, 3], [3, 5, 1], [6, 5, 1], [2, 5, 1]]
print(count_paths_with_min_time(N, M, edges)) # Output: 4

N = 6
M = 8
edges = [[0, 5, 8], [0, 2, 2], [0, 1, 1], [1, 3, 3], [1, 2, 3], [2, 5, 6], [3, 4, 2], [4, 5, 1]]
print(count_paths_with_min_time(N, M, edges)) # Output: 3

```

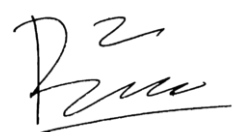
- 'from collections import defaultdict, deque' Mengimpor modul 'defaultdict' dan 'deque' dari pustaka 'collections'. 'defaultdict' digunakan untuk membuat kamus dengan nilai default untuk setiap kunci yang belum ada, sedangkan 'deque' adalah struktur data antrian yang efisien.
- 'def count_paths_with_min_time(N, M, edges):' Mendefinisikan sebuah fungsi bernama 'count_paths_with_min_time' yang mengambil tiga argumen, yaitu 'N' (jumlah simpul), 'M' (jumlah tepian), dan 'edges' (daftar tepian dalam graf).

Tanda Tangan



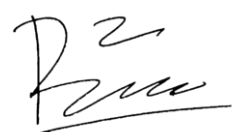
- `'graph = defaultdict(list)'` Membuat kamus default bernama `'graph'` untuk merepresentasikan graf. Setiap kunci dalam kamus akan merepresentasikan simpul, dan nilainya akan berupa daftar pasangan simpul tetangga dan waktu yang diperlukan untuk mencapai mereka.
- `'for u, v, t in edges:'` Iterasi melalui setiap tepian dalam daftar `'edges'`.
- `'graph[u].append((v, t))'` Menambahkan simpul tetangga `'v'` dan waktu `'t'` ke daftar tetangga dari simpul `'u'` dalam graf.
- `'min_time = [float('inf')] * N'` Membuat daftar `'min_time'` dengan panjang `'N'` (jumlah simpul) yang diinisialisasi dengan waktu tak terhingga untuk setiap simpul.
- `'min_time[0] = 0'` Menetapkan waktu minimum untuk simpul awal (simpul 0) menjadi 0.
- `'count_paths = [0] * N'` Membuat daftar `'count_paths'` dengan panjang `'N'` yang diinisialisasi dengan nilai 0 untuk setiap simpul.
- `'count_paths[0] = 1'` Menetapkan jumlah jalur yang mungkin dari simpul awal (simpul 0) menjadi 1.
- `'visited = set()'` Membuat himpunan kosong `'visited'` untuk melacak simpul yang telah dikunjungi.
- `'visited.add(0)'` Menambahkan simpul awal ke dalam himpunan `'visited'`.
- `'queue = deque()'` Membuat antrian kosong `'queue'` untuk menjelajahi simpul-simpul dalam graf secara BFS (Breadth-First Search).
- `'queue.append(0)'` Menambahkan simpul awal ke dalam antrian.
- `'while queue:'` Memulai loop while, akan terus berjalan selama antrian tidak kosong.
- `'node = queue.popleft()'` Mengambil simpul pertama dari antrian.
- `'visited.remove(node)'` Menghapus simpul yang sedang diproses dari himpunan `'visited'`.

Tanda Tangan



- ‘for neighbor, time in graph[node]:’ Iterasi melalui setiap tetangga dari simpul yang sedang diproses dan waktu yang diperlukan untuk mencapainya.
- ‘if min_time[node] + time <= min_time[neighbor]:’ Memeriksa apakah waktu minimum untuk mencapai tetangga melalui simpul saat ini lebih kecil dari waktu minimum sebelumnya.
- ‘if min_time[node] + time < min_time[neighbor]:’ Memeriksa apakah waktu minimum untuk mencapai tetangga melalui simpul saat ini lebih kecil dari waktu minimum sebelumnya.
- ‘min_time[neighbor] = min_time[node] + time’ Memperbarui waktu minimum untuk mencapai tetangga melalui simpul saat ini.
- ‘count_paths[neighbor] = count_paths[node]’ Memperbarui jumlah jalur yang mungkin ke tetangga dengan jumlah jalur dari simpul saat ini.
- ‘else:’ Jika waktu minimum untuk mencapai tetangga melalui simpul saat ini sama dengan waktu minimum sebelumnya.
- ‘count_paths[neighbor] += count_paths[node]’ Menambahkan jumlah jalur dari simpul saat ini ke jumlah jalur yang sudah ada ke tetangga.
- ‘if neighbor not in visited:’ Memeriksa apakah tetangga belum dikunjungi sebelumnya.
- ‘queue.append(neighbor)’ Menambahkan tetangga ke dalam antrian.
- ‘visited.add(neighbor)’ Menambahkan tetangga ke dalam himpunan ‘visited’.
- ‘return count_paths[N-1]’ Mengembalikan jumlah jalur yang mungkin untuk mencapai simpul terakhir (simpul ‘N-1’).

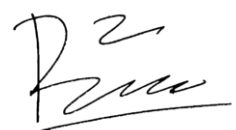
Tanda Tangan



V. Kesimpulan

Kesimpulan yang dapat diambil dalam uji coba praktikum dan lainnya.

Tanda Tangan

A handwritten signature in black ink, consisting of a stylized 'D' followed by a series of loops and a horizontal line at the bottom.

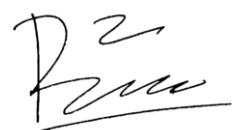
VI. Referensi

Minimal dari 3 sumber yang berbeda dari buku/jurnal (dilarang mengambil dari sumber website/wikipedia/blog)

Ketentuan pengumpulan laporan praktikum:

1. Laporan Diketik dengan ukuran paper A
2. Margins laporan Atas 2 cm Kiri 3 cm Kanan 2 cm Bawah 2 cm
3. Font Times New Roman ukuran 12.
4. Spasi 1,5.
5. Tidak boleh menggunakan garis tepi.
6. BAB ditulis dengan huru kapital.
7. Keterangan gambar di tulis di bawah gambar.
8. Laporan dikumpulkan paling lambat di hari praktikum minggu selanjutnya, jika terlambat diberi pengurangan nilai.
9. Copas laporan orang lain diberi pengurangan nilai.

Tanda Tangan

A handwritten signature in black ink, consisting of a stylized 'P' followed by a series of loops and a horizontal line at the end.