


Nama : Mutiara Novianti Rambe NIM : 064002300029	 Algoritma dan Pemrograman Dasar	Modul 14 Nama Dosen: 1. Abdul Rochman 2. Anung B. Ariwibowo
Hari/Tanggal: Jumat/24 Mei 2024		Nama Aslab: 1. Nathanael W. (064002100020) 2. Adrian Alfajri (064002200009)

MODUL 14 : Graph

Deskripsi Modul : Memahami dan menerapkan ilmu struktur data dan algoritma untuk menyelesaikan masalah yang disajikan dengan menggunakan program berbasis bahasa Python.

No.	Elemen Kompetensi	Indikator Kinerja	Halaman
1.	Mampu memahami dan mengimplementasikan Graph pada Python	Membuat dan memahami sebuah program yang menerapkan struktur data Graph.	

TEORI SINGKAT

Algoritma Depth First Search (DFS):

DFS adalah algoritma penelusuran yang mulai dari node awal, kemudian menelusuri sepanjang cabang yang belum dikunjungi sebelum kembali ke node sebelumnya untuk menelusuri cabang lain. DFS menggunakan struktur data stack (tumpukan) secara implisit melalui rekursi atau secara eksplisit.

Algoritma Breadth First Search (BFS):

BFS adalah algoritma penelusuran yang mulai dari node awal, kemudian menelusuri semua tetangga pada tingkat yang sama sebelum melanjutkan ke tingkat berikutnya. BFS menggunakan struktur data queue (antrian) dan sering digunakan untuk mencari jalur terpendek dalam graf tak berbobot.

Algoritma Dijkstra:

Dijkstra adalah algoritma untuk menemukan jalur terpendek dari satu node ke node lain dalam sebuah graf berbobot. Algoritma ini berfungsi dengan menandai jarak terpendek dari node awal ke node lainnya dan memperbarui jarak berdasarkan bobot edge. Dijkstra menggunakan

struktur data priority queue (antrian prioritas) untuk memilih node dengan jarak terpendek yang belum diproses.

DAFTAR PERTANYAAN

1. Jelaskan perbedaan antara DFS dan BFS dari segi pendekatan dan aplikasi nyata!
2. Mengapa algoritma Dijkstra tidak cocok digunakan untuk graf dengan edge berbobot negatif?
3. Jelaskan bagaimana cara mengoptimalkan representasi graf yang sangat besar dan jarang menggunakan adjacency list, dan bagaimana ini mempengaruhi performa algoritma DFS dan BFS.

JAWABAN

- 1.
- 2.
- 3.

LAB SETUP

Hal yang harus disiapkan dan dilakukan oleh praktikan untuk menjalankan praktikum modul ini, antara lain:

1. Menyiapkan IDE untuk membangun program python (Spyder, Sublime, VSCode, dll);
2. Python sudah terinstal dan dapat berjalan dengan baik di laptop masing-masing;
3. Menyimpan semua dokumentasi hasil praktikum pada laporan yang sudah disediakan.

ELEMEN KOMPETENSI I

Deskripsi : Mampu membuat program tentang graph sesuai perintah yang ada.

Kompetensi Dasar : Membuat program yang mengimplementasikan dua algoritma dalam pencarian jalur pada direct graph.

LATIHAN 1

Mengimplementasikan algoritma Depth First Search (DFS) pada directed graph.

1. Buat fungsi untuk menambahkan vertex ke dalam graph.

Code Clip:

```
def tambah_vertex(self, v):  
    if v not in self.graph:  
        # Tambahkan vertex dengan daftar vertex lain kosong
```

2. Buat fungsi untuk menambahkan edge antara dua vertex.

Code Clip:

```
def tambah_edge(self, u, v):  
    if u not in self.graph:  
        # Tambahkan edge dari u ke v
```

3. Buat fungsi untuk menghapus vertex dari graph.

Code Clip:

```
def hapus_vertex(self, v):  
    if v in self.graph:  
        # Hapus vertex 'v' dari graph  
        for k in self.graph:  
            if v in self.graph[k]:  
                # Hapus vertex dari daftar tetangga
```

4. Buat fungsi untuk menghapus edge antara dua vertex.

Code Clip:

```
def hapus_edge(self, u, v):  
    if u in self.graph and v in self.graph[u]:  
        # Hapus edge dari u ke v
```

5. Buat fungsi utilitas untuk melakukan DFS secara rekursif

Code Clip:

```
def dfs_util(self, v, visited):  
    visited.add(v)  
    # Cetak vertex yang saat ini sedang dikunjungi  
    for neighbor in self.graph.get(v, []):  
        # Jika tetangga belum dikunjungi, lakukan sesuatu yang rekursif  
        if neighbor not in visited:  
            # Panggil fungsi ini lagi dengan tetangga sebagai vertex baru
```

6. Buat fungsi untuk memulai DFS dari vertex tertentu.

Code Clip:

```
def dfs(self, start):
    # Inisialisasi struktur untuk melacak node yang telah dikunjungi
    print(f"\nPenelusuran DFS dimulai dari vertex {start}: ", end='')
    # Panggil fungsi utilitas DFS dengan vertex awal
    print("\n")
```

7. Buat fungsi untuk menampilkan graph dalam bentuk adjacency.

Code Clip:

```
def tampilkan(self):
    print("\nRepresentasi Graf (Adjacency List):")
    # Iterasi melalui setiap node dalam graf
    # Cetak node dan daftar tetangganya
    print()
```

Preview Output (Latihan 1)

```
=== Operasi Graph (DFS) ===
1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar
=====
Masukkan pilihan: 1
Masukkan vertex: A
```

Tambahkan A s/d D

```
=== Operasi Graph (DFS) ===
1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar
=====
Masukkan pilihan: 2
Masukkan vertex awal: A
Masukkan vertex akhir: B
```

Tambahkan beberapa edge

```
=== Operasi Graph (DFS) ===
1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar
=====
Masukkan pilihan: 6
Masukkan vertex awal untuk DFS: C

Penelusuran DFS dimulai dari vertex C: -> C -> B -> D
```

```
=== Operasi Graph (DFS) ===
1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar
=====
Masukkan pilihan: 5

Representasi Graph (Adjacency list):
A -> ['B', 'C']
B -> ['D']
C -> ['B']
D -> ['B']
E -> ['C']
```

Source Code

```
class Graph:
    def __init__(self):
        self.graph = {}

    def tambah_vertex(self, v):
        if v not in self.graph:
            self.graph[v] = []
            print(f"Vertex {v} ditambahkan.")

    def tambah_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)
        print(f"Edge dari {u} ke {v} ditambahkan.")

    def hapus_vertex(self, v):
        if v in self.graph:
            del self.graph[v]
            for k in self.graph:
                if v in self.graph[k]:
                    self.graph[k].remove(v)
```

```

        print(f"Vertex {v} dihapus.")

    def hapus_edge(self, u, v):
        if u in self.graph and v in self.graph[u]:
            self.graph[u].remove(v)
        if v in self.graph and u in self.graph[v]:
            self.graph[v].remove(u)
        print(f"Edge dari {u} ke {v} dihapus.")

    def dfs_util(self, v, visited):
        visited.add(v)
        print(v, end=' ')
        for neighbor in self.graph.get(v, []):
            if neighbor not in visited:
                self.dfs_util(neighbor, visited)

    def dfs(self, start):
        visited = set()
        print(f"\nPenelusuran DFS dimulai dari vertex {start}: ", end="")
        self.dfs_util(start, visited)
        print("\n")

    def tampilkan(self):
        print("\nRepresentasi Graf (Adjacency List):")
        for vertex in self.graph:
            print(f"{vertex} -> {self.graph[vertex]}")
        print()

def main():
    g = Graph()
    while True:
        print("==== Operasi Graph (DFS) =====")
        print("1. Tambah Vertex")
        print("2. Tambah Edge")
        print("3. Hapus Edge")
        print("4. Hapus Vertex")
        print("5. Tampilkan Graf")
        print("6. Lakukan DFS")
        print("7. Keluar")
        pilihan = input("Masukkan pilihan: ")

        if pilihan == '1':
            v = input("Masukkan vertex: ")
            g.tambah_vertex(v)
        elif pilihan == '2':
            u = input("Masukkan vertex awal: ")

```

```
        v = input("Masukkan vertex akhir: ")
        g.tambah_edge(u, v)
    elif pilihan == '3':
        u = input("Masukkan vertex awal: ")
        v = input("Masukkan vertex akhir: ")
        g.hapus_edge(u, v)
    elif pilihan == '4':
        v = input("Masukkan vertex: ")
        g.hapus_vertex(v)
    elif pilihan == '5':
        g.tampilkan()
    elif pilihan == '6':
        start = input("Masukkan vertex awal untuk DFS: ")
        g.dfs(start)
    elif pilihan == '7':
        break
    else:
        print("Pilihan tidak valid. Silakan coba lagi.")

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


Screenshot

==== Operasi Graph (DFS) ====

1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar

Masukkan pilihan: 1
Masukkan vertex: A
Vertex A ditambahkan.

==== Operasi Graph (DFS) ====

1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar

Masukkan pilihan: 1
Masukkan vertex: B
Vertex B ditambahkan.

==== Operasi Graph (DFS) ====

1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar

Masukkan pilihan: 1
Masukkan vertex: C
Vertex C ditambahkan.

==== Operasi Graph (DFS) ====

1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar

Masukkan pilihan: 1
Masukkan vertex: D
Vertex D ditambahkan.

==== Operasi Graph (DFS) ====

1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar

Masukkan pilihan: 2
Masukkan vertex awal: A
Masukkan vertex akhir: C

Edge dari A ke C ditambahkan.

==== Operasi Graph (DFS) ====

1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar

Masukkan pilihan: 2
Masukkan vertex awal: C

Masukkan vertex akhir: B

Edge dari C ke B ditambahkan.

==== Operasi Graph (DFS) ====

1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar

Masukkan pilihan: 2

Masukkan vertex awal: B

Masukkan vertex akhir: D

Edge dari B ke D ditambahkan.

==== Operasi Graph (DFS) ====

1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf
6. Lakukan DFS
7. Keluar

Masukkan pilihan: 6

Masukkan vertex awal untuk DFS: A

Penelusuran DFS dimulai dari vertex A: A C B D

==== Operasi Graph (DFS) ====

1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graf

6. Lakukan DFS

7. Keluar

Masukkan pilihan: 5

Representasi Graf (Adjacency List):

A -> ['C']

B -> ['C', 'D']

C -> ['A', 'B']

D -> ['B']

LATIHAN 2

mengimplementasikan algoritma Breadth First Search (BFS) pada directed graph.

1. Melanjutkan latihan 1, Tambahkan fungsi untuk melakukan BFS.

Code Clip:

```
from collections import deque
# Mengapa mengimpor deque?
# Deque adalah struktur data antrian yang efisien untuk operasi
# penambahan dan penghapusan di kedua ujung.
```

2. Inisialisasi struktur data yang diperlukan untuk BFS

Code Clip:

```
def bfs(self, start):
    #Inisialisasi struktur data
    queue = deque([start])
    visited.add(start)
    print(f"\nPenelusuran BFS dimulai dari vertex {start}: ", end='')
```

3. Lakukan penelusuran BFS dengan menggunakan *queue* dan proses vertex satu per satu.

Code Clip:

```
while queue:
    # Ambil elemen pertama dari antrian
    print(f"-> {vertex}", end=' ')

    # Proses semua tetangga dari vertex saat ini
    for neighbor in self.graph.get(vertex, []):
        if neighbor not in visited:
            # Tambahkan tetangga ke kumpulan yang dikunjungi
            # Masukkan tetangga ke antrian untuk diproses nanti
```

Preview Output

```

=== Operasi Graph (BFS) ===
1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graph
6. Lakukan BFS
7. Keluar
=====
Masukkan pilihan Anda: 5

Representasi Graph (Adjacency List):
A -> ['B', 'C']
B -> []
C -> []
D -> ['C']
E -> ['A']

```

```

=== Operasi Graph (BFS) ===
1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graph
6. Lakukan BFS
7. Keluar
=====
Masukkan pilihan Anda: 6
Masukkan vertex awal untuk BFS: A

Penelusuran BFS dimulai dari vertex A: -> A -> B -> C

```

Source Code

```

from collections import deque

class Graph:
    def __init__(self):
        self.graph = {}

    def tambah_vertex(self, v):
        if v not in self.graph:
            self.graph[v] = []
            print(f"Vertex {v} ditambahkan.")

    def tambah_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)
        print(f"Edge dari {u} ke {v} ditambahkan.")

```

```

def hapus_vertex(self, v):
    if v in self.graph:
        del self.graph[v]
        for k in self.graph:
            if v in self.graph[k]:
                self.graph[k].remove(v)
        print(f"Vertex {v} dihapus.")

def hapus_edge(self, u, v):
    if u in self.graph and v in self.graph[u]:
        self.graph[u].remove(v)
    if v in self.graph and u in self.graph[v]:
        self.graph[v].remove(u)
    print(f"Edge dari {u} ke {v} dihapus.")

def bfs(self, start):
    visited = set()
    queue = deque([start])
    visited.add(start)
    print(f"\nPenelusuran BFS dimulai dari vertex {start}: ", end="")
    while queue:
        vertex = queue.popleft()
        print(f"-> {vertex}", end=' ')
        for neighbor in self.graph.get(vertex, []):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    print()

def tampilkan(self):
    print("\nRepresentasi Graf (Adjacency List):")
    for vertex in self.graph:
        print(f"{vertex} -> {self.graph[vertex]}")
    print()

def main():
    g = Graph()
    while True:
        print("==== Operasi Graph (BFS) =====")
        print("1. Tambah Vertex")
        print("2. Tambah Edge")
        print("3. Hapus Edge")
        print("4. Hapus Vertex")
        print("5. Tampilkan Graph")
        print("6. Lakukan BFS")

```

```
print("7. Keluar")
pilihan = input("Masukkan pilihan: ")

if pilihan == '1':
    v = input("Masukkan vertex: ")
    g.tambah_vertex(v)
elif pilihan == '2':
    u = input("Masukkan vertex awal: ")
    v = input("Masukkan vertex akhir: ")
    g.tambah_edge(u, v)
elif pilihan == '3':
    u = input("Masukkan vertex awal: ")
    v = input("Masukkan vertex akhir: ")
    g.hapus_edge(u, v)
elif pilihan == '4':
    v = input("Masukkan vertex: ")
    g.hapus_vertex(v)
elif pilihan == '5':
    g.tampilkan()
elif pilihan == '6':
    start = input("Masukkan vertex awal untuk BFS: ")
    g.bfs(start)
elif pilihan == '7':
    break
else:
    print("Pilihan tidak valid. Silakan coba lagi.")

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

Screenshot

```
==== Operasi Graph (BFS) ====
```

1. Tambah Vertex
 2. Tambah Edge
 3. Hapus Edge
 4. Hapus Vertex
 5. Tampilkan Graph
 6. Lakukan BFS
 7. Keluar
- Masukkan pilihan: 5

```
Representasi Graf (Adjacency List):
```

```
A -> ['B', 'C', 'D']  
B -> ['A']  
C -> ['A', 'E', 'D']  
D -> ['A', 'C']  
E -> ['C']
```

```
==== Operasi Graph (BFS) ====
```

1. Tambah Vertex
2. Tambah Edge
3. Hapus Edge
4. Hapus Vertex
5. Tampilkan Graph
6. Lakukan BFS
7. Keluar

Masukkan pilihan: 6

Masukkan vertex awal untuk BFS: C

```
Penelusuran BFS dimulai dari vertex C: -> C -> A -> E -> D -> B
```

ELEMEN KOMPETENSI II

Deskripsi : Mampu membuat program tentang graph sesuai perintah yang ada.

Kompetensi Dasar : Membuat program yang mengimplementasikan algoritma dijkstra dalam pencarian jalur pada direct graph.

LATIHAN 2

mengimplementasikan algoritma Dijkstra untuk menemukan jalur terpendek pada weighted directed graph

1. Melanjutkan latihan 1 dan 2, Tambahkan fungsi untuk menjalankan Algoritma Dijkstra.
Code Clip:

```
import heapq

# Mengapa mengimpor heapq?
# Heapq menyediakan struktur data priority queue yang efisien
# untuk operasi penambahan dan penghapusan elemen.

def dijkstra(self, start):
    # Inisialisasi struktur data
    # buat priority queue(pq) yang diinisialisasi dengan (0, start).
    distances = {vertex: float('infinity') for vertex in self.graph}
    distances[start] = 0
    # buat visited untuk melacak node yang telah dikunjungi.

    while pq:
        current_distance, current_vertex = heapq.heappop(pq)
        # Cek apakah vertex sudah dikunjungi
        visited.add(current_vertex)

        # Proses semua tetangga dari vertex saat ini
        for neighbor, weight in self.graph.get(current_vertex, {}).items():
            if distance < distances[neighbor]:
                # Perbarui jarak jika ditemukan jarak yang lebih pendek
```

Preview Output

```
Menampilkan Graph:
A : {'B': 1, 'C': 4}
B : {'D': 2}
C : {'E': 5}
D : {'E': 1}
E : {'F': 3}
F : {'C': 2}

Menjalankan Dijkstra:
Jarak terpendek dari simpul A: {'A': 0, 'B': 1, 'C': 4, 'D': 3, 'E': 4, 'F': 7}
```

Source Code

Screenshot

KESIMPULAN

Buatlah kesimpulan hasil praktikum modul ini!!! (MINIMAL 3 BARIS)

CEKLIST

1. Memahami dan mengimplementasikan algoritma Depth First Search (DFS) (✓)
2. Memahami dan mengimplementasikan algoritma Breadth First Search (BFS) (✓)
3. Memahami dan mengimplementasikan algoritma Dijkstra untuk pencarian jalur terpendek ()

REFERENSI

<https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>
<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/?ref=lbp>
<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/?ref=lbp>
<https://www.programiz.com/dsa/graph-dfs>
<https://www.programiz.com/dsa/graph-bfs>
<https://www.javatpoint.com/dijkstras-algorithm>