LAPORAN PRAKTIKUM

STRUKTUR DATA

MODUL KE-08

GRAPH DALAM PYTHON

Disusun Oleh:

Nama : Oktario Mufti Yudha

NPM : 2320506044

Kelas : 04 (Empat)

Program Studi S1 Teknologi Informasi

Fakultas Teknik, Universitas Tidar

Genap 2023/2024

- **Tujuan Praktikum**

Adapun tujuan praktikum ini sebagai berikut :

a. Mahasiswa akan belajar tentang konsep dasar Graph, pada bahasa pemrograman Python.
b. Mahasiswa akan mempelajari cara Representasi dari Struktur Data Graph, Transpose Graph, BFS dan DFS, Siklus Graf, Minimum Spanning Tree.

- **Dasar Teori**

a. Struktur Data Graf adalah kumpulan simpul yang terhubung oleh tepi. Digunakan untuk merepresentasikan hubungan antara entitas yang berbeda. Algoritma graf adalah metode yang digunakan untuk memanipulasi dan menganalisis graf, memecahkan berbagai masalah seperti menemukan lintasan terpendek atau mendeteksi siklus.
b. Graf adalah struktur data non-linear yang terdiri dari simpul dan tepi. Simpul kadang-kadang juga disebut sebagai simpul dan tepi adalah garis atau lengkungan yang menghubungkan dua simpul dalam graf. Lebih formal sebuah Graf terdiri dari kumpulan simpul (V) dan kumpulan tepi (E). Graf tersebut dilambangkan dengan G(V, E).

- **Hasil dan Pembahasan**

  Adjacency matrix

```python
# adjacency matrix
class Graph:
    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        self.matrix = [[0] * num_vertices for _ in range(num_vertices)]

    def add_edge(self, u, v):
        self.matrix[u][v] = 1
        self.matrix[v][u] = 1

    def display(self):
        for row in self.matrix:
            print(row)

#contoh penggunaan
g = Graph(5)
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(2, 4)

print("Adjacency Matrix")
g.display()
```

```
0.0s
Adjacency Matrix
[0, 1, 1, 0, 0]
[1, 0, 0, 1, 0]
[1, 0, 0, 0, 1]
[0, 1, 0, 0, 0]
[0, 0, 1, 0, 0]
```

(Gambar 3.1)

- class Graph: Mendefinisikan kelas Graph.
- def __init__(self, num_vertices): Ini adalah konstruktor untuk kelas Graph. Ini mengambil jumlah simpul sebagai argumen.
- self.num_vertices = num_vertices Menyimpan jumlah simpul ke variabel instance.
- self.matrix = [[0] * num_vertices for _ in range(num_vertices)] Membuat matriks adjacensi dengan semua nilai awal 0.
- def add_edge(self, u, v): Ini adalah metode untuk menambahkan tepi antara simpul u dan v.
- self.matrix[u][v] = 1 dan self.matrix[v][u] = 1 Menandai keberadaan tepi antara simpul u dan v dalam matriks adjacensi.
- def display(self): Ini adalah metode untuk mencetak matriks adjacensi.
- for row in self.matrix: print(row) Mencetak setiap baris dalam matriks adjacensi.
- g = Graph(5) Membuat objek Graph dengan 5 simpul.
- g.add_edge(0, 1), g.add_edge(0, 2), g.add_edge(1, 3), dan g.add_edge(2, 4) Menambahkan tepi antara simpul yang sesuai.
- print("Adjacency Matrix") Mencetak string "Adjacency Matrix".

- g.display() Memanggil metode display pada objek Graph untuk mencetak matriks adjacensi.

Adjacency list

(Gambar 3.2)

- class Graph: Mendefinisikan kelas Graph.
- def __init__(self): Ini adalah konstruktor untuk kelas Graph. Ini menginisialisasi daftar adjacensi sebagai kamus kosong.
- def add_edge(self, u, v): Ini adalah metode untuk menambahkan tepi antara simpul u dan v.
- if u not in self.adj_list: dan if v not in self.adj_list: Memeriksa apakah simpul u dan v sudah ada dalam daftar adjacensi. Jika tidak, mereka ditambahkan.
- self.adj_list[u].append(v) dan self.adj_list[v].append(u) Menambahkan simpul lainnya ke daftar adjacensi dari masing-masing simpul.
- def display(self): Ini adalah metode untuk mencetak daftar adjacensi.
- for vertex in self.adj_list: Iterasi melalui setiap simpul dalam daftar adjacensi.
- print(vertex, "->", "->".join(map(str, self.adj_list[vertex])))

Mencetak simpul dan semua simpul yang terhubung dengannya.

- g = Graph() Membuat objek Graph.
- g.add_edge(0, 1), g.add_edge(0, 2), g.add_edge(1, 3), dan g.add_edge(2, 4) Menambahkan tepi antara simpul yang sesuai.
- print("Adjacency List") Mencetak string "Adjacency List".
- g.display() Memanggil metode display pada objek Graph untuk mencetak daftar adjacensi.

Transpose Graph

```
# Transpose Graph

# function to add an edge form vertex source to vertex dest
def addEdge(adj, src, dest):
    adj[src].append(dest)

# function to print adjacency list of a graph
def displayGraph(adj, v):
    for i in range(v):
        print(i, "⟶", end = "")
        for j in range(len(adj[i])):
            print(adj[i][j], end = " ")
        print()

# function to get the transpose of the graph
# and that of transpose graph
def transposeGraph(adj, transpose, v):
    for i in range(v):
        for j in range(len(adj[i])):
            addEdge(transpose, adj[i][j], i)

# driver code
if __name__ == "__main__":
    v = 5
    adj = [[] for i in range(v)]
    addEdge(adj, 0, 1)
    addEdge(adj, 0, 4)
    addEdge(adj, 0, 3)
    addEdge(adj, 2, 0)
    addEdge(adj, 3, 2)
    addEdge(adj, 4, 1)
    addEdge(adj, 4, 3)
```



```
    # finding transpose of the graph
    transpose = [[] for i in range(v)]
    transposeGraph(adj, transpose, v)

    # displaying adjacency list of the graph
    displayGraph(transpose, v)
```

✓ 0.0s

```
0 ⟶2
1 ⟶0 4
2 ⟶3
3 ⟶0 4
4 ⟶0
```

(Gambar 3.3)

- def addEdge(adj, src, dest): Fungsi ini menambahkan tepi dari simpul sumber ke simpul tujuan dalam graf yang direpresentasikan sebagai daftar adjacensi.
- adj[src].append(dest) Menambahkan simpul tujuan ke daftar

adjacensi dari simpul sumber.

- def displayGraph(adj, v): Fungsi ini mencetak daftar adjacensi dari graf.
- for i in range(v): Iterasi melalui setiap simpul dalam graf.
- print(i, "-->", end = "") Mencetak simpul dan panah yang menunjukkan ke simpul lain.
- for j in range(len(adj[i])): Iterasi melalui setiap simpul yang terhubung dengan simpul saat ini.
- print(adj[i][j], end = " ") Mencetak simpul yang terhubung.
- def transposeGraph(adj, transpose, v): Fungsi ini mendapatkan transpose dari graf.
- for i in range(v): Iterasi melalui setiap simpul dalam graf.
- for j in range(len(adj[i])): Iterasi melalui setiap simpul yang terhubung dengan simpul saat ini.
- addEdge(transpose, adj[i][j], i) Menambahkan tepi dari simpul yang terhubung ke simpul saat ini dalam graf transpose.
- if __name__ == "__main__": Memastikan bahwa kode dijalankan hanya jika file ini dijalankan sebagai program utama, bukan sebagai modul.
- v = 5 Menetapkan jumlah simpul dalam graf.
- adj = [[] for i in range(v)] Membuat daftar adjacensi untuk graf.
- addEdge(adj, 0, 1) dan seterusnya Menambahkan tepi antara simpul yang sesuai.
- transpose = [[] for i in range(v)] Membuat daftar adjacensi untuk graf transpose.
- transposeGraph(adj, transpose, v) Mendapatkan transpose dari graf.
- displayGraph(transpose, v) Mencetak daftar adjacensi dari graf transpose.

Breadth First Search

```python
# Breadth First Search

from collections import deque

# function to perform breadth first search on graph
# represented using adjacency list
def bfs(adjList, startNode, visited):
    # create a queue for BFS
    q = deque()
    visited[startNode] = True
    q.append(startNode)

    while q:
        currentNode = q.popleft()
        print(currentNode, end = " ")

        for neighbour in adjList[currentNode]:
            if not visited[neighbour]:
                visited[neighbour] = True
                q.append(neighbour)

# function to add an edge to graph
def addEdge(adjList, u, v):
    adjList[u].append(v)
```

```python
def main():
    vertices = 5
    adjList = [[] for _ in range(vertices)]

    addEdge(adjList, 0, 1)
    addEdge(adjList, 0, 2)
    addEdge(adjList, 1, 3)
    addEdge(adjList, 1, 4)
    addEdge(adjList, 2, 4)

    visited = [False] * vertices

    print("BFS starting from vertex 0:", end = "")
    bfs(adjList, 0, visited)

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

✓ 0.0s

```
BFS starting from vertex 0:0 1 2 3 4
```

(Gambar 3.4)

- from collections import deque Impor deque dari modul collections. Deque adalah struktur data yang dapat digunakan untuk menambah dan menghapus elemen dari kedua ujungnya.
- def bfs(adjList, startNode, visited): Fungsi ini melakukan pencarian lebar pertama (BFS) pada graf.
- q = deque() Membuat deque untuk BFS.
- visited[startNode] = True Menandai simpul awal sebagai dikunjungi.
- q.append(startNode) Menambahkan simpul awal ke deque.

- while q: Melakukan loop selama deque tidak kosong.
- currentNode = q.popleft() Menghapus simpul dari deque dan menyimpannya dalam variabel currentNode.
- print(currentNode, end = " ") Mencetak simpul saat ini.
- for neighbour in adjList[currentNode]: Melakukan loop melalui setiap tetangga dari simpul saat ini.
- if not visited[neighbour]: Jika tetangga belum dikunjungi, tandai sebagai dikunjungi dan tambahkan ke deque.
- def addEdge(adjList, u, v): Fungsi ini menambahkan tepi ke graf.
- adjList[u].append(v) Menambahkan simpul v ke daftar tetangga dari simpul u.
- def main(): Fungsi utama program.
- vertices = 5 Menetapkan jumlah simpul dalam graf.
- adjList = [[] for _ in range(vertices)] Membuat daftar adjacensi untuk graf.
- addEdge(adjList, 0, 1) dan seterusnya Menambahkan tepi antara simpul yang sesuai.
- visited = [False] * vertices Membuat daftar untuk melacak simpul yang telah dikunjungi.
- print("BFS starting from vertex 0:", end = "") Mencetak string "BFS starting from vertex 0:".
- bfs(adjList, 0, visited) Melakukan BFS pada graf.
- if __name__ == "__main__": Memastikan bahwa kode dijalankan hanya jika file ini dijalankan sebagai program utama, bukan sebagai modul.
- main() Memanggil fungsi utama program.

Depth First Search

```python
# Depth First Search
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFSUtil(self, v, visited):
        visited.add(v)
        print(v, end = " ")

        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    def DFS(self, v):
        visited = set()
        self.DFSUtil(v, visited)
```

```python
# Driver code
if __name__ == "__main__":
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)

    print("following is DFS from (starting from vertex 2)")
    g.DFS(2)
```

✓ 0.0s

```
following is DFS from (starting from vertex 2)
2 0 1 3
```

(Gambar 3.5)

- from collections import defaultdict Impor defaultdict dari modul collections. Defaultdict adalah tipe kamus yang menyediakan nilai default untuk kunci yang belum ada.
- class Graph: Mendefinisikan kelas Graph.
- def __init__(self): Ini adalah konstruktor untuk kelas Graph. Ini menginisialisasi graf sebagai defaultdict dengan list sebagai nilai default.
- def addEdge(self, u, v): Fungsi ini menambahkan tepi dari simpul u ke v dalam graf.
- self.graph[u].append(v) Menambahkan simpul v ke daftar tetangga dari simpul u.
- def DFSUtil(self, v, visited): Fungsi utilitas untuk melakukan pencarian dalam kedalaman (DFS) pada graf.
- visited.add(v) Menambahkan simpul saat ini ke set simpul yang telah dikunjungi.

- print(v, end = " ") Mencetak simpul saat ini.
- for neighbour in self.graph[v]: Melakukan loop melalui setiap tetangga dari simpul saat ini.
- if neighbour not in visited: Jika tetangga belum dikunjungi, lakukan DFS pada tetangga tersebut.
- def DFS(self, v): Fungsi ini melakukan DFS pada graf.
- visited = set() Membuat set untuk melacak simpul yang telah dikunjungi.
- self.DFSUtil(v, visited) Memanggil fungsi utilitas DFS.
- if __name__ == "__main__": Memastikan bahwa kode dijalankan hanya jika file ini dijalankan sebagai program utama, bukan sebagai modul.
- g = Graph() Membuat objek Graph.
- g.addEdge(0, 1) dan seterusnya Menambahkan tepi antara simpul yang sesuai.
- print("following is DFS from (starting from vertex 2)") Mencetak string "following is DFS from (starting from vertex 2)".
- g.DFS(2) Melakukan DFS pada graf, dimulai dari simpul 2.

Detect cycle in a directed graph

```python
# detect cycle in a directed graph

from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def isCyclicUtil(self, v, visited, recStack):
        visited[v] = True
        recStack[v] = True

        for neighbour in self.graph[v]:
            if visited[neighbour] == False:
                if self.isCyclicUtil(neighbour, visited, recStack) == True:
                    return True
            elif recStack[neighbour] == True:
                return True

        recStack[v] = False
        return False

    def isCyclic(self):
        visited = [False] * self.V
        recStack = [False] * self.V

        for node in range(self.V):
            if visited[node] == False:
                if self.isCyclicUtil(node, visited, recStack) == True:
                    return True
        return False
```

```python
# Driver code
if __name__ == "__main__":
    g = Graph(4)
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)

    if g.isCyclic() == 1:
        print("Graph contains cycle")
    else:
        print("Graph doesn't contain cycle")
```

```
✓ 0.0s
Graph contains cycle
```

(Gambar 3.6)

- from collections import defaultdict Impor defaultdict dari modul collections. Defaultdict adalah tipe kamus yang menyediakan nilai default untuk kunci yang belum ada.
- class Graph: Mendefinisikan kelas Graph.
- def __init__(self, vertices): Ini adalah konstruktor untuk kelas Graph. Ini menginisialisasi graf sebagai defaultdict dengan list sebagai nilai default dan jumlah simpul.
- def addEdge(self, u, v): Fungsi ini menambahkan tepi dari simpul u ke v dalam graf.
- self.graph[u].append(v) Menambahkan simpul v ke daftar tetangga dari simpul u.
- def isCyclicUtil(self, v, visited, recStack): Fungsi utilitas untuk memeriksa apakah ada siklus dalam graf.
- visited[v] = True dan recStack[v] = True Menandai simpul saat ini sebagai dikunjungi dan menambahkannya ke stack rekursif.

- for neighbour in self.graph[v]: Melakukan loop melalui setiap tetangga dari simpul saat ini.
- if visited[neighbour] == False: Jika tetangga belum dikunjungi, lakukan pemeriksaan siklus pada tetangga tersebut.
- elif recStack[neighbour] == True: Jika tetangga ada dalam stack rekursif, itu berarti ada siklus.
- recStack[v] = False Menghapus simpul saat ini dari stack rekursif.
- def isCyclic(self): Fungsi ini memeriksa apakah ada siklus dalam graf.
- visited = [False] * self.V dan recStack = [False] * self.V Membuat daftar dan stack untuk melacak simpul yang telah dikunjungi.
- for node in range(self.V): Melakukan loop melalui setiap simpul dalam graf.
- if visited[node] == False: Jika simpul belum dikunjungi, lakukan pemeriksaan siklus pada simpul tersebut.
- if __name__ == "__main__": Memastikan bahwa kode dijalankan hanya jika file ini dijalankan sebagai program utama, bukan sebagai modul.
- g = Graph(4) Membuat objek Graph dengan 4 simpul.
- g.addEdge(0, 1) dan seterusnya Menambahkan tepi antara simpul yang sesuai.
- if g.isCyclic() == 1: Jika ada siklus dalam graf, cetak "Graph contains cycle". Jika tidak, cetak "Graph doesn't contain cycle".

Detect cycle in an undirected graph

```python
# detect cycle in an undirected graph

from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)

    def addEdge(self, v, w):
        self.graph[v].append(w)
        self.graph[w].append(v)

    def isCyclicUtil(self, v, visited, parent):
        visited[v] = True

        for i in self.graph[v]:
            if visited[i] == False:
                if (self.isCyclicUtil(i, visited, v)):
                    return True
            elif parent != i:
                return True
        return False

    def isCyclic(self):
        visited = [False] * self.V

        for i in range(self.V):
            if visited[i] == False:
                if (self.isCyclicUtil
                    (i, visited, -1)) == True:
                    return True
        return False
```

```python
# Driver code
g = Graph(5)
g.addEdge(1, 0)
g.addEdge(0, 2)
g.addEdge(2, 0)
g.addEdge(0, 3)
g.addEdge(3, 4)

if g.isCyclic():
    print("Graph contains cycle")
else:
    print("Graph doesn't contain cycle")
g1 = Graph(3)
g1.addEdge(0, 1)
g1.addEdge(1, 2)

if g1.isCyclic():
    print("Graph contains cycle")
else:
    print("Graph doesn't contain cycle")
```

✓ 0.0s

```
Graph contains cycle
Graph doesn't contain cycle
```

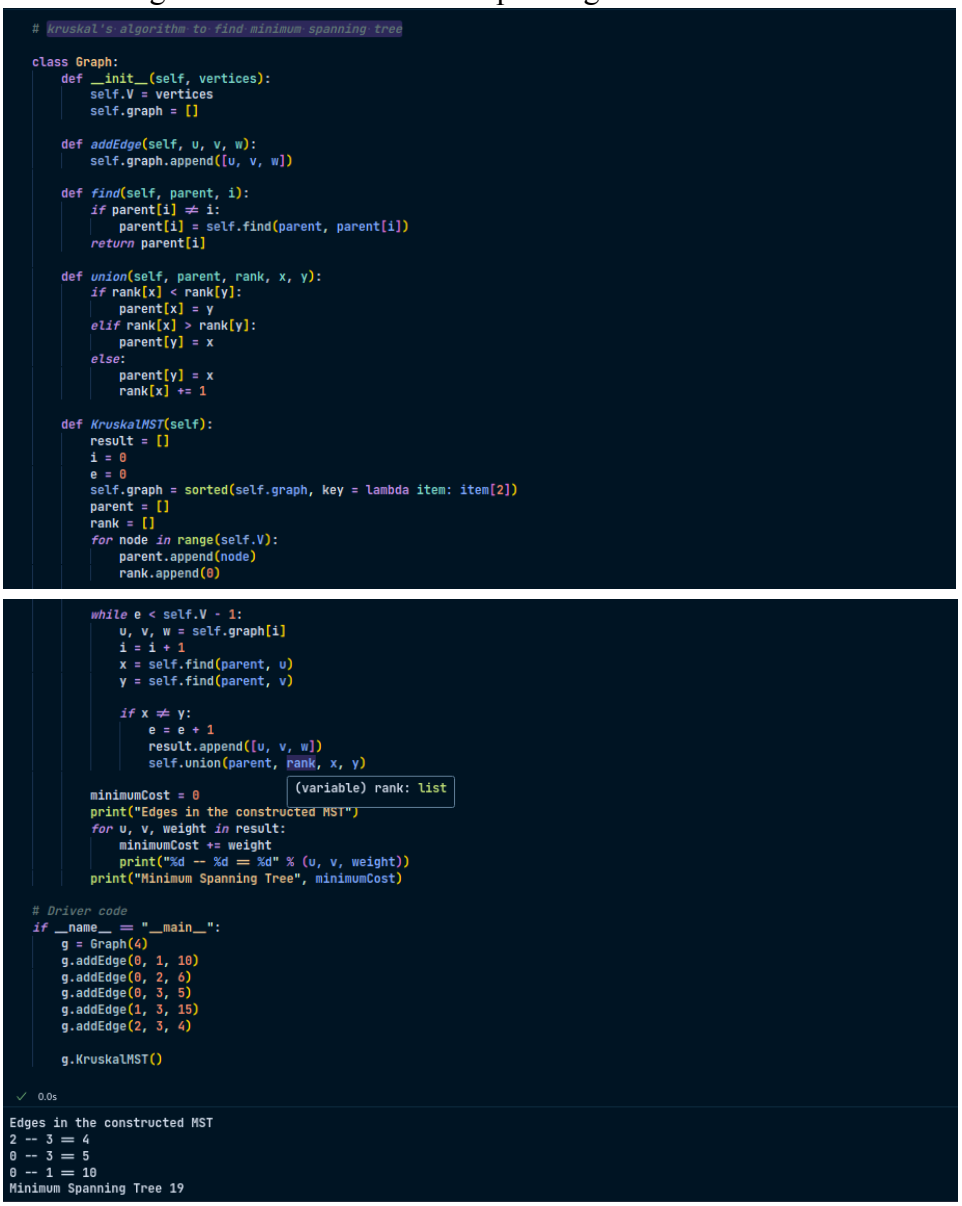
(Gambar 3.7)

- from collections import defaultdict Impor defaultdict dari modul collections. Defaultdict adalah tipe kamus yang menyediakan nilai default untuk kunci yang belum ada.
- class Graph: Mendefinisikan kelas Graph.
- def __init__(self, vertices): Ini adalah konstruktor untuk kelas Graph. Ini menginisialisasi graf sebagai defaultdict dengan list sebagai nilai default dan jumlah simpul.
- def addEdge(self, v, w): Fungsi ini menambahkan tepi antara simpul v dan w dalam graf.
- self.graph[v].append(w) dan self.graph[w].append(v) Menambahkan simpul lainnya ke daftar tetangga dari masing-masing simpul.
- def isCyclicUtil(self, v, visited, parent): Fungsi utilitas untuk

memeriksa apakah ada siklus dalam graf.

- visited[v] = True Menandai simpul saat ini sebagai dikunjungi.
- for i in self.graph[v]: Melakukan loop melalui setiap tetangga dari simpul saat ini.
- if visited[i] == False: Jika tetangga belum dikunjungi, lakukan pemeriksaan siklus pada tetangga tersebut.
- elif parent != i: Jika tetangga adalah simpul yang telah dikunjungi dan bukan orang tua dari simpul saat ini, itu berarti ada siklus.
- def isCyclic(self): Fungsi ini memeriksa apakah ada siklus dalam graf.
- visited = [False] * self.V Membuat daftar untuk melacak simpul yang telah dikunjungi.
- for i in range(self.V): Melakukan loop melalui setiap simpul dalam graf.
- if visited[i] == False: Jika simpul belum dikunjungi, lakukan pemeriksaan siklus pada simpul tersebut.
- g = Graph(5) Membuat objek Graph dengan 5 simpul.
- g.addEdge(1, 0) dan seterusnya Menambahkan tepi antara simpul yang sesuai.
- if g.isCyclic(): Jika ada siklus dalam graf, cetak "Graph contains cycle". Jika tidak, cetak "Graph doesn't contain cycle".
- g1 = Graph(3) Membuat objek Graph lain dengan 3 simpul.
- g1.addEdge(0, 1) dan seterusnya Menambahkan tepi antara simpul yang sesuai.
- if g1.isCyclic(): Jika ada siklus dalam graf, cetak "Graph contains cycle". Jika tidak, cetak "Graph doesn't contain cycle".

Kruskal's algorithm to find minimum spanning tree



```python
# kruskal's algorithm to find minimum spanning tree

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] != i:
            parent[i] = self.find(parent, parent[i])
        return parent[i]

    def union(self, parent, rank, x, y):
        if rank[x] < rank[y]:
            parent[x] = y
        elif rank[x] > rank[y]:
            parent[y] = x
        else:
            parent[y] = x
            rank[x] += 1

    def KruskalMST(self):
        result = []
        i = 0
        e = 0
        self.graph = sorted(self.graph, key = lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)

        minimumCost = 0
        print("Edges in the constructed MST")
        for u, v, weight in result:
            minimumCost += weight
            print("%d -- %d == %d" % (u, v, weight))
        print("Minimum Spanning Tree", minimumCost)

# Driver code
if __name__ == "__main__":
    g = Graph(4)
    g.addEdge(0, 1, 10)
    g.addEdge(0, 2, 6)
    g.addEdge(0, 3, 5)
    g.addEdge(1, 3, 15)
    g.addEdge(2, 3, 4)

    g.KruskalMST()
```

```
✓ 0.0s
Edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Spanning Tree 19
```

(Gambar 3.8)

- class Graph: Mendefinisikan kelas Graph.
- def __init__(self, vertices): Ini adalah konstruktor untuk kelas Graph. Ini menginisialisasi jumlah simpul dan graf sebagai list kosong.
- def addEdge(self, u, v, w): Fungsi ini menambahkan tepi antara simpul u dan v dengan bobot w ke graf.
- self.graph.append([u, v, w]) Menambahkan tepi ke graf.
- def find(self, parent, i): Fungsi ini mencari representasi set dari

elemen i.

- if parent[i] != i: Jika i bukan representan dari setnya sendiri, cari representan setnya.
- def union(self, parent, rank, x, y): Fungsi ini menggabungkan dua set x dan y.
- if rank[x] < rank[y]: Jika rank x lebih kecil dari y, buat y sebagai parent dari x.
- elif rank[x] > rank[y]: Jika rank y lebih kecil dari x, buat x sebagai parent dari y.
- else: Jika rank x dan y sama, buat salah satunya sebagai parent dan tingkatkan ranknya.
- def KruskalMST(self): Fungsi ini mengimplementasikan algoritma Kruskal untuk mencari Minimum Spanning Tree (MST) dari graf.
- self.graph = sorted(self.graph, key = lambda item: item[2]) Mengurutkan semua tepi graf berdasarkan bobotnya.
- for node in range(self.V): Melakukan loop melalui setiap simpul dalam graf dan membuat set sendiri untuk setiap simpul.
- while e < self.V - 1: Melakukan loop sampai jumlah tepi dalam MST menjadi V-1.
- u, v, w = self.graph[i] Mengambil tepi dengan bobot minimum yang belum diproses.
- x = self.find(parent, u) dan y = self.find(parent, v) Mencari representan set dari kedua simpul u dan v.
- if x != y: Jika mereka bukan bagian dari set yang sama, tambahkan tepi ke hasil dan gabungkan set mereka.
- for u, v, weight in result: Melakukan loop melalui setiap tepi dalam MST dan mencetaknya.
- if __name__ == "__main__": Memastikan bahwa kode dijalankan hanya jika file ini dijalankan sebagai program utama, bukan sebagai modul.
- g = Graph(4) Membuat objek Graph dengan 4 simpul.
- g.addEdge(0, 1, 10) dan seterusnya Menambahkan tepi antara simpul yang sesuai dengan bobot yang sesuai.
- g.KruskalMST() Mencari dan mencetak MST dari graf.

- **Latihan**

```python
# Latihan
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def topologicalSortUtil(self, v, visited, stack):
        visited[v] = True
        for i in self.graph[v]:
            if visited[i] == False:
                self.topologicalSortUtil(i, visited, stack)
        stack.insert(0, v)

    def topologicalSort(self):
        visited = [False]*self.V
        stack =[]
        for i in range(self.V):
            if visited[i] == False:
                self.topologicalSortUtil(i, visited, stack)
        print(stack)

if __name__ == "__main__":
    g = Graph(6)
    g.addEdge(5, 2)
    g.addEdge(5, 0)
    g.addEdge(4, 0)
    g.addEdge(4, 1)
    g.addEdge(2, 3)
    g.addEdge(3, 1)
    g.topologicalSort()
```

```
0.0s
[5, 4, 2, 3, 1, 0]
```

(Gambar 4.1)

- from collections import defaultdict Impor defaultdict dari modul collections. Defaultdict adalah tipe kamus yang menyediakan nilai default untuk kunci yang belum ada.
- class Graph: Mendefinisikan kelas Graph.
- def __init__(self, vertices): Ini adalah konstruktor untuk kelas Graph. Ini menginisialisasi graf sebagai defaultdict dengan list sebagai nilai default dan jumlah simpul.
- def addEdge(self, u, v): Fungsi ini menambahkan tepi dari simpul u ke v dalam graf.
- self.graph[u].append(v) Menambahkan simpul v ke daftar tetangga dari simpul u.
- def topologicalSortUtil(self, v, visited, stack): Fungsi utilitas untuk melakukan pengurutan topologis pada graf.
- visited[v] = True Menandai simpul saat ini sebagai dikunjungi.
- for i in self.graph[v]: Melakukan loop melalui setiap tetangga dari simpul saat ini.
- if visited[i] == False: Jika tetangga belum dikunjungi, lakukan pengurutan topologis pada tetangga tersebut.

- stack.insert(0, v) Menambahkan simpul saat ini ke awal stack.
- def topologicalSort(self): Fungsi ini melakukan pengurutan topologis pada graf.
- visited = [False]*self.V Membuat daftar untuk melacak simpul yang telah dikunjungi.
- stack =[] Membuat stack untuk menyimpan hasil pengurutan topologis.
- for i in range(self.V): Melakukan loop melalui setiap simpul dalam graf.
- if visited[i] == False: Jika simpul belum dikunjungi, lakukan pengurutan topologis pada simpul tersebut.
- print(stack) Mencetak hasil pengurutan topologis.
- if __name__ == "__main__": Memastikan bahwa kode dijalankan hanya jika file ini dijalankan sebagai program utama, bukan sebagai modul.
- g = Graph(6) Membuat objek Graph dengan 6 simpul.
- g.addEdge(5, 2) dan seterusnya Menambahkan tepi antara simpul yang sesuai.
- g.topologicalSort() Melakukan pengurutan topologis pada graf dan mencetak hasilnya.

- **Kesimpulan**

  Graph adalah struktur data yang terdiri dari simpul dan tepi yang menghubungkan simpul-simpul tersebut. Graf dapat digunakan untuk merepresentasikan banyak jenis data dalam dunia nyata, seperti jaringan sosial, web, jaringan transportasi, dan lainnya.

  Berikut adalah beberapa poin penting yang saya dapat pada praktikum kali ini:

1. Representasi Graf: Graf dapat direpresentasikan dalam berbagai cara, termasuk matriks adjacensi dan daftar adjacensi. Dalam kode di atas, graf direpresentasikan sebagai daftar adjacensi menggunakan defaultdict dari modul collections.

2. Penambahan Tepi: Tepi dapat ditambahkan ke graf dengan menambahkan simpul tujuan ke daftar tetangga dari simpul sumber.

3. Pencarian Graf: Ada berbagai algoritma untuk melakukan pencarian dalam graf, termasuk Breadth-First Search (BFS) dan Depth-First Search (DFS). BFS menggunakan deque untuk melacak simpul yang harus dikunjungi, sementara DFS menggunakan rekursi atau stack.

4. Pemeriksaan Siklus: Untuk memeriksa apakah ada siklus dalam graf, kita dapat menggunakan variasi dari algoritma DFS di mana kita melacak simpul yang sedang kita kunjungi dan melihat apakah kita menemui simpul yang sudah kita kunjungi sebelumnya.

5. Pengurutan Topologis: Pengurutan topologis adalah pengurutan simpul dalam graf berarah sehingga untuk setiap tepi berarah dari simpul u ke v, u datang sebelum v dalam pengurutan. Pengurutan topologis digunakan dalam penjadwalan tugas, analisis kode sumber, dan banyak aplikasi lainnya.

6. Minimum Spanning Tree: Minimum Spanning Tree (MST) adalah subgraf dari graf berbobot yang mencakup semua simpul dengan jumlah bobot yang minimal. Algoritma Kruskal adalah salah satu algoritma untuk mencari MST.