

LAPORAN PRAKTIKUM STRUKTUR DATA

MODUL KE-08

GRAPH DALAM PYTHON



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 konsep graph pada bahasa pemrograman python

II. Dasar Teori (minimal 3 halaman)

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.

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 $G(V, E)$.

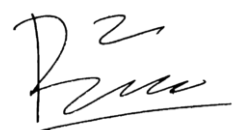
Komponen Graf:

- Simpul: Simpul adalah unit dasar dari graf. Kadang-kadang, simpul juga dikenal sebagai simpul atau simpul. Setiap simpul/simpul dapat diberi label atau tidak diberi label.
- Tepi: Tepi digambar atau digunakan untuk menghubungkan dua simpul graf. Ini bisa berupa pasangan terurut dari simpul dalam graf terarah. Tepi dapat menghubungkan dua simpul dalam cara apa pun. Tidak ada aturan. Kadang-kadang, tepi juga dikenal sebagai lengkungan. Setiap tepi dapat diberi label/tanpa label.

Berikut adalah operasi dasar pada graf:

- Penyisipan Simpul/Tepi dalam graf - Memasukkan simpul ke dalam graf.
- Penghapusan Simpul/Tepi dalam graf - Menghapus simpul dari graf.
- Pencarian pada Graf - Mencari entitas dalam graf.
- Traversal Graf - Melintasi semua simpul dalam graf.

Tanda Tangan

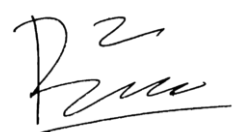


Kelebihan Graph	Kekurangan Graph
<ul style="list-style-type: none"> • Graf dapat digunakan untuk memodelkan dan menganalisis sistem dan hubungan yang kompleks. • Graf bermanfaat untuk memvisualisasikan dan memahami data. • Algoritma graf secara luas digunakan dalam ilmu komputer dan bidang lainnya, seperti analisis jaringan sosial, logistik, dan transportasi. • Graf dapat digunakan untuk merepresentasikan berbagai jenis data, termasuk jaringan sosial, jaringan jalan, dan internet 	<ul style="list-style-type: none"> • Graf yang besar dapat sulit untuk divisualisasikan dan dianalisis. • Algoritma graf dapat memakan banyak sumber daya komputasi, terutama untuk graf yang besar. • Interpretasi hasil graf dapat subjektif dan mungkin memerlukan pengetahuan spesifik domain. • Graf dapat rentan terhadap noise dan outliers, yang dapat memengaruhi akurasi hasil analisis.

Perbedaan Kunci Antara Graf dan Pohon:

- Siklus: Graf dapat mengandung siklus, sedangkan pohon tidak bisa.
- Konektivitas: Graf bisa tidak terhubung (misalnya, memiliki beberapa komponen), sedangkan pohon selalu terhubung.
- Hirarki: Pohon memiliki struktur hirarkis, dengan satu titik ditetapkan sebagai akar. Graf tidak memiliki struktur hirarkis ini.
- Aplikasi: Graf digunakan dalam berbagai aplikasi, seperti jaringan sosial, jaringan transportasi, dan ilmu komputer. Pohon sering digunakan dalam struktur data hirarkis, seperti sistem file dan dokumen XML.

Tanda Tangan



III. Hasil dan Pembahasan

3.1. Adjacency Matrix

```
# 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)

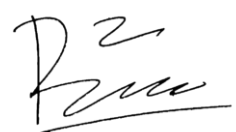
print("Adjacency Matrix")
g.display()
```

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 3.1.1)

1. Awal dari definisi kelas yang di sebut `Graph`, yang digunakan untuk merepresentasikan graf.
2. Metode khusus yang disebut saat objek `Graph` dibuat, `num_vertices` untuk parameter yang menentukan jumlah simpul dalam graf.
3. `self.num_vertices = num_vertices` berfungsi untuk menginisialisasi atribut `num_vertices` dari objek `Graph`.
4. Membuat matriks dengan `self.matrix` dengan ukuran `num_vertices x num_vertices`. Yang akan di gunakan untuk menyimpan informasi hubungan antar simpul.
5. Menambahkan tepian (edge) antara dua simpul `u` dan `v` dalam graf dengan `def add_edge(self, u, v):`.
6. Untuk menetapkan nilai 1 dalam posisi `(u, v)` dan `(v, u)` dalam matriks `self.matrix[u][v] = 1` dan `self.matrix[v][u] = 1` untuk menunjukkan terdapat tepian antara simpul `u` dan `v`.

Tanda Tangan



7. Dengan `def display(self):` untuk menampilkan matriks keberadjaan ke layar.
8. Program `for row in self.matrix:` Melakukan iterasi pada setiap baris dalam matriks.
9. `print(row)` akan mencetak setiap baris matriks.

3.2. Adjacency List

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

    def add_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)

    def display(self):
        for vertex in self.graph:
            print(vertex, "→", "→".join(map(str, self.graph[vertex])))

# Contoh Penggunaan
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(2, 4)

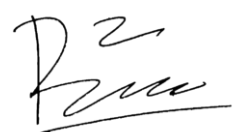
print("Adjacency List")
g.display()
```

Adjacency List
0 → 1→2
1 → 0→3
2 → 0→4
3 → 1
4 → 2

(Gambar 3.2.1)

1. Pertama membuat `class Graph:` dari definisi kelas yang disebut `Graph`, yang akan digunakan untuk merepresentasikan graf.
2. Melakukan metode khusus dengan `def __init__(self):` yang disebut saat objek `Graph` dibuat.
3. Membuat sebuah dictionary `self.graph = {}` yang akan berfungsi untuk menyimpan daftar keberadjaan setiap simpul dalam graf.
4. Menambahkan tepian (edge) antara dua simpul `u` dan `v` dalam graf dengan `def add_edge(self, u, v):`.

Tanda Tangan



5. Program akan memeriksa apakah simpul `u` sudah ada dalam kamus `if u not in self.graph:` jika tidak, maka di buat entri baru untuk simpul `u`.
6. Lalu memeriksa apakah simpul `v` sudah ada dalam kamus `if v not in self.graph:` jika tidak, maka di buat entri baru untuk simpul `v`.
7. Selanjutnya menambahkan simpul `v` ke daftar keberadjadian simpul `u`, menunjukkan bahwa ada tepian dari `u` ke `v` dengan `self.graph[u].append(v)`.
8. Menambahkan simpul `u` ke dalam daftar keberadjadian simpul `v`, untuk menunjukkan bahwa ada tepian dari `v` ke `u` dengan `self.graph[v].append(u)`.
9. Menampilkan daftar keberadjadian ke layar dengan metode `def display(self):`.
10. Melakukan iterasi pada setiap simpul `for vertex in self.graph:` dalam kamus.
11. Terakhir mencetak simpul `print(vertex, "->", "->".join(map(str, self.graph[vertex])))` yang diikuti oleh panah (->) dan daftar keberadjadiannya.

3.3. Transpose Graf

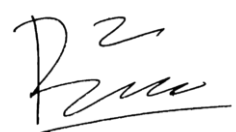
```
# function to add an edge from vortex source to vortex dest
def addEdge(adj, src, dest):
    adj[src].append(dest)

# 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 taking adj list of given graph and that of its transpose
def transposeGraph(adj, transpose, V):
    # traverse the adjacency list of given graph and for each edge (u, v) add an edge (v, u) in transpose
    for i in range(V):
        for j in range(len(adj[i])):
            addEdge(transpose, adj[i][j], i)
```

(Gambar 3.3.1)

Tanda Tangan



```
# 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 graph represented by adj
    transpose = [[] for i in range(V)]
    transposeGraph(adj, transpose, V)

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

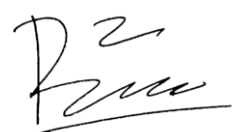
Python

```
0 -->2
1 -->04
2 -->3
3 -->04
4 -->0
```

(Gambar 3.3.2)

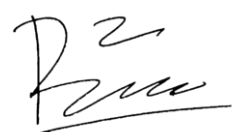
1. Program dimulai dengan `def addEdge(adj, src, dest):` yang berfungsi untuk mendefinisikan sebuah fungsi bernama `addEdge` yang akan menerima tiga parameter: `adj`, `src`, dan `dest`. Dimana fungsi ini untuk menambahkan sebuah tepian dari simpul `src` ke `dest` dalam daftar keberadjadian `adj`.
2. Menambahkan simpul `adj[src].append(dest)` untuk menambahkan `dest` ke daftar keberadjadian `src`, lalu menunjukkan adanya tepian simpul `src` ke `dest`.
3. Mendefinisikan sebuah fungsi dengan `def displayGraph(adj, V):` bernama `displayGraph` untuk menerima dua parameter: `adj` dan `V` ini berfungsi untuk menampilkan daftar keberadjadian graf.
4. Melakukan iterasi pada setiap simpul dalam graf dengan `for i in range(V):`.
5. `print(i, "--> ", end = "")` Mencetak nomor simpul yang diikuti oleh tanda panah (`-->`). Penggunaan `end = ""` akan memastikan cetakan berikutnya tidak pergi ke baris baru.
6. Melakukan iterasi `for j in range(len(adj[i])):` untuk setiap simpul yang berdekatan dengan simpul dalam daftar keberadjadian.

Tanda Tangan



7. Mencetak simpul yang berdekatan dengan simpul saat ini dengan metode ``print(adj[i][j], end = "")``.
8. ``print()`` berfungsi untuk mencetak baris baru setelah selesai mencetak semua simpul.
9. Mendefinisikan dengan ``def transposeGraph(adj, transpose, V):`` untuk sebuah fungsi bernama ``transposeGraph`` yang akan berfungsi menerima tiga parameter.
10. ``for i in range(V):`` Melakukan iterasi pada setiap simpul di dalam graf.
11. ``for j in range(len(adj[i])):`` Melakukan iterasi untuk setiap simpul yang berdekatan dengan simpul saat ini dalam daftar keberadjadian graf asli.
12. Memanggil fungsi ``addEdge`` berfungsi menambahkan tepian simpul yang dekat dengan simpul saat ini menuju simpul saat ini.
13. ``if __name__ == '__main__':`` adalah blok kode yang akan di jalankan saat skrip di eksekusi dengan langsung.
14. Menetapkan jumlah simpul dalam graf dengan ``V = 5``.
15. Membuat daftar kosong ``adj = [[] for i in range(V)]`` untuk menyimpan keberadjadian dari setiap simpul dalam graf.
16. Menambahkan beberapa tepian ke graf menggunakan fungsi ``addEdge``.
17. Membuat daftar kosong ``transpose = [[] for i in range(V)]`` untuk menyimpan keberadjadian transpose graf.
18. Dengan ``transposeGraph(adj, transpose, V)`` akan memanggil fungsi ``transposeGraph`` yang berfungsi untuk menghitung transpose dari graf.
19. Menampilkan daftar keberadjadian dari graf transpose ``displayGraph(transpose, V)``.

Tanda Tangan



3.4. Breadth First Search (BFS)

```
from collections import deque

def bfs(adjList, startNode, visited):
    # Create a queue for BFS
    q = deque()
    # Mark the current node as visited and enqueue it
    visited[startNode] = True
    q.append(startNode)
    # Iterate over the queue
    while q:
        # Dequeue a vertex from queue and print it
        currentNode = q.popleft()
        print(currentNode, end=" ")
        # Get all adjacent vertices of the dequeued vertex
        # If an adjacent has not been visited, then mark it visited and enqueue
        for neighbor in adjList[currentNode]:
            if not visited[neighbor]:
                visited[neighbor] = True
                q.append(neighbor)

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

def main():
    # Number of vertices in the graph
    vertices = 5
    # Adjacency list representation of the graph
    adjList = [[] for _ in range(vertices)]
    # Add edges to the graph
    addEdge(adjList, 0, 1)
    addEdge(adjList, 0, 2)
    addEdge(adjList, 1, 3)
    addEdge(adjList, 1, 4)
    addEdge(adjList, 2, 4)
    # Mark all the vertices as not visited
    visited = [False] * vertices
    # Perform BFS traversal starting from vertex 0
    print("Breadth First Traversal starting from vertex 0:", end=" ")
    bfs(adjList, 0, visited)

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

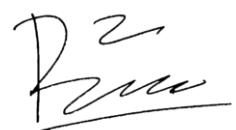
Python

Breadth First Traversal starting from vertex 0: 0 1 2 3 4

(Gambar 3.4.1)

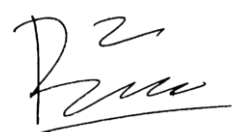
1. Pertama `from collections import deque` yang berfungsi untuk mengimpor kelas `deque` dari modul `collection` di gunakan sebagai antrian dalam algoritma BFS.
2. Selanjutnya `def bfs(adjList, startNode, visited):` adalah definisi untuk fungsi `bfs` yang menerima tiga parameter.
3. `q = deque()`: membuat objek untuk antrian `q` menggunakan `deque()`.
4. Menandai simpul awal `startNode` sebagai telah dikunjungi.
5. Memasukkan simpul awal ke dalam antrian dengan metode `q.append(startNode)`.

Tanda Tangan



6. Melakukan loop `while q:` selama antrian tidak kosong.
7. Menghapus dan mengembalikan elemen dengan `currentNode = q.popleft()` dari antrian, yang merupakan fungsi untuk simpul yang sedang di proses.
8. Mencetak simpul `print(currentNode, end=" ")` yang sedang diproses.
9. Melakukan iterasi pada semua tetangga `for neighbor in adjList[currentNode]:` dari simpul yang sedang di proses.
10. Memeriksa `if not visited[neighbor]:` apakah tetangga belum dikunjungi.
11. Menandai tetangga `visited[neighbor] = True` tersebut sebagai telah dikunjungi.
12. Memasukkan tetangga `q.append(neighbor)` yang belum dikunjungi ke dalam antrian.
13. Mendefinisikan dari fungsi `addEdge` yang di gunakan untuk menambahkan tepian baru ke daftar keberadjadian `adjList`.
14. Menetapkan jumlah simpul dalam graf dengan metode `vertices = 5`.
15. Membuat daftar kosong `adjList = [[] for _ in range(vertices)]` yang berfungsi untuk menyimpan daftar keberadjadian simpul dalam graf.
16. Selanjutnya menambahkan beberapa tepian ke graf menggunakan fungsi `addEdge`.
17. Membuat daftar `visited` yang menandai semua simpul belum dikunjungi.
18. Mencetak pesan untuk menandai bahwa pencarian BFS dimulai dari simpul 0 dengan metode `print("Breadth First Traversal starting from vertex 0:", end=" ")`.
19. Memanggil fungsi `bfs(adjList, 0, visited)` untuk melakukan pencarian BFS dari simpul 0.
20. Blok kode `if __name__ == "__main__":` yang akan di jalankan saat skrip di eksekusi secara langsung.

Tanda Tangan



21. Memanggil fungsi `main` untuk memulai eksekusi program dengan `main`.

3.5. Depth First Search (DFS)

```
from collections import defaultdict

class Graph:
    # Constructor
    def __init__(self):
        # Default dictionary to store graph
        self.graph = defaultdict(list)

    # Function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self, v, visited):
        # Mark the current node as visited and print it
        visited.add(v)
        print(v, end=' ')

        # Recur for all the vertices adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    # The function to do DFS traversal. It uses recursive DFSUtil()
    def DFS(self, v):
        # Create a set to store visited vertices
        visited = set()
        # Call the recursive helper function to print DFS traversal
        self.DFSUtil(v, visited)

# Driver's 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 Depth First Traversal (starting from vertex 2)")
    # Function call
    g.DFS(2)
```

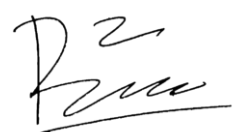
Python

Following is Depth First Traversal (starting from vertex 2)
2 0 1 3

(Gambar 3.5.1)

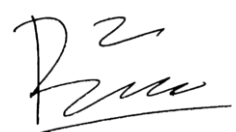
1. Mengimpor kelas `defaultdict` dari modul `collections`. `defaultdict` berfungsi di sini untuk membuat struktur data graf yang lebih mudah.
2. Membuat definisi `class Graph`, yang akan digunakan untuk merepresentasikan graf.
3. `def __init__(self):` berfungsi menjadi metode konstruktor kelas `Graph`.

Tanda Tangan



4. Membuat dictionary ``self.graph = defaultdict(list)`` yang menggunakan ``defaultdict`` berfungsi untuk menyimpan daftar keberadjadian dari setiap simpul.
5. Metode ``def addEdge(self, u, v):`` berfungsi untuk menambahkan tepian (edge) antara dua simpul ``u`` dan ``v`` dalam graf.
6. Menambahkan simpul ``v`` ke daftar keberadjadian simpul ``u`` dengan ``self.graph[u].append(v)``.
7. Metode utilitas ``def DFSUtil(self, v, visited):`` yang digunakan oleh DFS.
8. Untuk menandai simpul ``visited.add(v)`` saat ini sebagai telah dikunjungi.
9. ``print(v, end=' ')`` Mencetak simpul saat ini.
10. Melakukan iterasi ``for neighbour in self.graph[v]:`` untuk semua tetangga dari simpul saat ini dalam daftar keberadjadian.
11. Memeriksa ``if neighbour not in visited:`` apakah tetangga belum dikunjungi.
12. Memanggil ``self.DFSUtil(neighbour, visited)`` rekursif untuk menjelajahi tetangga yang belum di kunjungi.
13. Metode ``def DFS(self, v):`` untuk melakukan DFS traversal dari simpul ``v``.
14. Lalu membuat himpunan ``visited = set()`` untuk menyimpan simpul yang telah di kunjungi.
15. Memanggil ``self.DFSUtil(v, visited)`` metode utilitas DFS untuk memulai traversal dari simpul ``v``.
16. Dengan blok kode ``if __name__ == "__main__":`` yang akan di jalankan saat skrip di eksekusi secara langsung.
17. Membuat objek ``g`` dari kelas ``Graph``.
18. Menambahkan beberapa edge ke graf menggunakan metode ``addEdge``.

Tanda Tangan



19. Mencetak pesan untuk menandai pencarian DFS yang di mulai dari simpul dengan metode ``print("Following is Depth First Traversal (starting from vertex 2)")``.
20. Memanggil metode ``DFS`` untuk memulai DFS traversal dari simpul

3.6. Siklus Detect Cycle in a Directed Graph

```
# Detect cycle in an undirected 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):
        # Mark current node as visited and adds to recursion stack
        visited[v] = True
        recStack[v] = True

        # Recur for all neighbours if any neighbour is visited and in recStack then graph
        for neighbour in self.graph[v]:
            if not visited[neighbour]:
                if self.isCyclicUtil(neighbour, visited, recStack):
                    return True
            elif recStack[neighbour]:
                return True

        # The node needs to be popped from recursion stack before function ends
        recStack[v] = False
        return False

    def isCyclic(self):
        visited = [False] * self.V
        recStack = [False] * self.V
        for node in range(self.V):
            if not visited[node]:
                if self.isCyclicUtil(node, visited, recStack):
                    return True
        return False

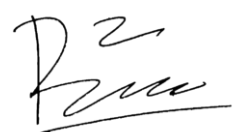
# Driver code
if __name__ == '__main__':
    g = Graph(4)
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)
    if g.isCyclic():
        print("Graph contains cycle")
    else:
        print("Graph doesn't contain cycle")

Graph contains cycle
```

(Gambar 3.6.1)

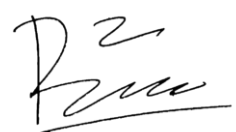
1. Mengimpor kelas ``from collections import defaultdict`` dari modul ``collections``, ``defaultdict`` digunakan membuat struktur graf.

Tanda Tangan



2. Mendefinisikan kelas dengan ``class Graph:``, yang akan di gunakan untuk merepresentasikan graf.
3. Selanjutnay ``def __init__(self, vertices):`` berfungsi untuk konstruktor kelas ``Graph`` yang akan menerima parameter ``vertices``.
4. Membuat dictionary ``self.graph = defaultdict(list)`` yang menggunakan ``defaultdict`` berfungsi untuk menyimpan daftar keberadjadian setiap simpul dalam graf.
5. Menyimpan jumlah simpul dalam atribut ``V`` dengan ``self.V = vertices``.
6. ``def addEdge(self, u, v):`` untuk menambahkan tepian antara dua simpul ``u`` dan ``v`` dalam graf.
7. Menambahkan simpul ``v`` ke daftar keberadjadian simpul ``u`` dengan ``self.graph[u].append(v)``.
8. Metode utilitas ``def isCyclicUtil(self, v, visited, recStack):`` yang di gunakan untuk mendeteksi siklus dengan cara menggunakan rekursif.
9. Menandai simpul saat ini dengan ``visited[v] = True`` dan ``recStack[v] = True`` sebagai telah di kunjungi dan menambahkannya ke dalam tumpukan rekursi.
10. Program ``def isCyclic(self):`` adalah metode untuk memeriksa apakah graf berisi siklus atau tidak.
11. Lalu jika ``isCyclicUtil`` mengembalikan ``True``, maka dalam graf berisi siklus dan metode mengembalikan ``True``.
12. Blok kode ``if __name__ == '__main__':`` yang akan dijalankan jika skrip di eksekusi secara langsung.
13. Selanjutnya menambahkan beberapa edge ke graf menggunakan metode ``addEdge``.
14. Memanggil metode ``isCyclic`` yang berfungsi untuk memeriksa apakah dalam graf berisi siklus atau tidak, dan mencetak hasilnya.

Tanda Tangan



3.7. Siklus Detect cycle in an undirected graph

```
# Detect cycle in a directed graph

from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[] for _ in range(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 not visited[neighbour] and self.isCyclicUtil(neighbour, visited, recStack):
                return True
            elif recStack[neighbour]:
                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 not visited[node]:
                if self.isCyclicUtil(node, visited, recStack):
                    return True
        return False

# 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)

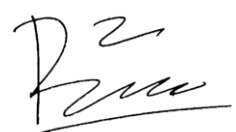
    if g.isCyclic():
        print("Graph contains cycle")
    else:
        print("Graph doesn't contain cycle")

Graph contains cycle
```

(Gambar 3.7.1)

1. Pertama mengimpor kelas `from collections import defaultdict` dari modul `collections`, pada `defaultdict` di gunakan untuk membuat struktur data graf yang lebih mudah.
2. Selanjutnya mendefinisi dari kelas `Graph`, yang akan digunakan untuk merepresentasikan graf dengan merode `class Graph:`.
3. Pada metode konstruktor kelas `Graph` yang menerima parameter `def __init__(self, vertices):`, yang akan menunjukkan jumlah simpul dalam graf.

Tanda Tangan



4. `def addEdge(self, u, v):` adalah metode yang berfungsi untuk menambahkan tepian antara simpul `u` dan `v` dalam graf.
5. Menambahkan simpul `v` ke daftar keberadjaan simpul `u` dengan metode `self.graph[u].append(v)`.
6. Metode utilitas `def isCyclicUtil(self, v, visited, recStack):` yang berfungsi untuk mendeteksi siklus menggunakan rekursif.
7. Menandai simpul `visited[v] = True` dan `recStack[v] = True` saat ini telah di kunjungi dan menambahkannya dalam tumpukan rekursi.
8. Metode `def isCyclic(self):` untuk memeriksa pada graf berisi siklus atau tidak.
9. Blok kode `if __name__ == '__main__':` akan berfungsi disaat skrip dijalankan secara langsung.
10. Menambahkan beberapa edge ke graf dengan metode `addEdge`.
11. Memanggil metode `isCyclic` yang berfungsi untuk memeriksa apakah graf berisi siklus.

3.8. Kruskal's algorithm to find Minimum Spanning Tree

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

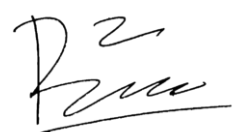
    # Function to add an edge to graph
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    # A utility function to find set of an element i (truly uses path compression tech)
    def find(self, parent, i):
        if parent[i] != i:
            # Reassignment of node's parent to root node as path compression requires
            parent[i] = self.find(parent, parent[i])
        return parent[i]

    # A function that does union of two sets of x and y (uses union by rank)
    def union(self, parent, rank, x, y):
        # Attach smaller rank tree under root of high rank tree (Union by Rank)
        if rank[x] < rank[y]:
            parent[x] = y
        elif rank[x] > rank[y]:
            parent[y] = x
        # If ranks are same, then make one as root and increment its rank by one
        else:
            parent[y] = x
            rank[x] += 1

    # Function to perform Kruskal's algorithm and print the Minimum Spanning Tree
    def KruskalMST(self):
        result = [] # This will store the resultant Minimum Spanning Tree
        i = 0 # An index variable, used for sorted edges
        e = 0 # An index variable, used for result[]
```

Tanda Tangan



(Gambar 3.8.1)

```
# Step 1: Sort all the edges in non-decreasing order of their weight.
self.graph = sorted(self.graph, key=lambda item: item[2])

parent = [i for i in range(self.V)] # Create V subsets with single elements
rank = [0] * self.V # To keep track of the rank of elements in each subset

# Number of edges to be taken is equal to V-1
while e < self.V - 1:
    # Step 2: Pick the smallest edge and increment the index for the next iteration
    u, v, w = self.graph[i]
    i = i + 1
    x = self.find(parent, u)
    y = self.find(parent, v)

    # If including this edge doesn't cause cycle, include it in result
    if x != y:
        e = e + 1
        result.append([u, v, w])
        self.union(parent, rank, x, y)
    # Else discard the edge

# Print the resultant Minimum Spanning Tree
print("Following are the edges in the constructed Minimum Spanning Tree")
for u, v, weight in result:
    print(f"{u} -- {v} = {weight}")

# Driver code
if __name__ == '__main__':
    g = Graph(4)
    g.addEdge(0, 1, 18)
    g.addEdge(0, 2, 6)
    g.addEdge(1, 3, 15)
    g.addEdge(2, 3, 4)
    g.KruskalMST()
```

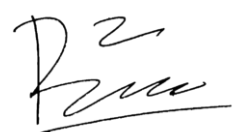
Python

Following are the edges in the constructed Minimum Spanning Tree
2 -- 3 = 4
0 -- 2 = 6
1 -- 3 = 15

(Gambar 3.8.2)

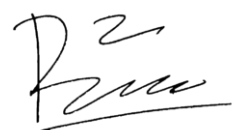
1. Pertama membuat `class Graph:` definisi dari kelas `Graph`, yang akan berfungsi untuk merepresentasikan graf.
2. Metode konstruktor `def __init__(self, vertices):` kelas `Graph` akan menerima parameter `vertices`, berfungsi menunjukkan jumlah simpul dalam graf.
3. Metode `def addEdge(self, u, v, w):` untuk menambahkan edge antara simpul `u` dan `v` dengan `w` ke dalam graf.
4. Selanjutnya metode utilitas `def find(self, parent, i):` yang berfungsi menemukan himpunan (set) dari elemen `i`.
5. Fungsi `def union(self, parent, rank, x, y):` adalah utilitas untuk melakukan penggabungan dua himpunan dengan teknik penggabungan berdasarkan peringkat.

Tanda Tangan



6. Selanjutnya pada ``def KruskalMST(self):`` akan melakukan algoritma Kruskal dan mencetak Minimum Spanning Tree.
7. Blok kode ``if __name__ == '__main__':`` akan dijalankan saat skrip di eksekusi secara langsung.
8. Membuat objek ``g`` dengan kelas ``Graph`` dari 4 simpul.
9. Menambahkan beberapa edge ke dalam graf dengan menggunakan metode ``addEdge``.
10. Memanggil metode ``KruskalMST`` yang akan mencetak MST dari graf.

Tanda Tangan

A handwritten signature in black ink, consisting of stylized cursive letters, is written over a white background within a rectangular box.

IV. Latihan

```
class Graph:
    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        self.adj = [[] for _ in range(num_vertices)]

    def add_edge(self, u, v):
        self.adj[u].append(v)

    def topological_sort_util(self, v, visited, stack):
        visited[v] = True

        for neighbor in self.adj[v]:
            if not visited[neighbor]:
                self.topological_sort_util(neighbor, visited, stack)

        stack.append(v)

    def topological_sort(self):
        visited = [False] * self.num_vertices
        stack = []

        for v in range(self.num_vertices):
            if not visited[v]:
                self.topological_sort_util(v, visited, stack)

        return stack[::-1]

if __name__ == "__main__":
    V = 5
    edges = [(0, 1), (0, 2), (1, 3), (2, 4), (3, 4)]

    graph = Graph(V)
    for u, v in edges:
        graph.add_edge(u, v)

    topological_order = graph.topological_sort()
    print("Topological Order:")
    for vertex in topological_order:
        print(vertex, end=" ")

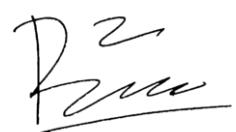
    Python
```

Topological Order:
0 2 1 3 4

(Gambar 4.1)

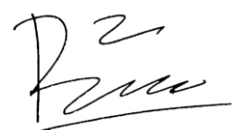
1. `class Graph:` berfungsi untuk definisi kelas `Graph`, yang akan digunakan untuk merepresentasikan graf.
2. Dengan metode `def __init__(self, num_vertices):` konstruktor kelas `Graph` akan menerima parameter `num_vertices`, yang menunjukkan jumlah simpul dalam graf.
3. Selanjutnya metode `def add_edge(self, u, v):` untuk menambahkan edge antara simpul `u` dan `v` ke dalam graf.

Tanda Tangan



4. ``def topological_sort_util(self, v, visited, stack):`` adalah metode utilitas yang berfungsi untuk melakukan topological sort dengan menggunakan pendekatan rekursif.
5. Metode ``def topological_sort(self):`` berfungsi untuk melakukan topological sort pada graf.
6. Blok kode ``if __name__ == "__main__":`` yang akan berjalan saat skrip di eksekusi secara langsung.
7. Menetapkan jumlah simpul dalam graf ``V = 5``.
8. Menetapkan tepian dengan ``edges = [(0, 1), (0, 2), (1, 3), (2, 4), (3, 4)]`` yang akan ditambahkan ke graf.
9. Membuat objek ``graph`` dari kelas ``Graph`` dengan 5 simpul.
10. Menambahkan edge ke graf menggunakan metode ``add_edge``.
11. Memanggil metode ``topological_sort`` untuk mendapatkan urutan topological dari graf.
12. Mencetak urutan topological hasilnya.

Tanda Tangan

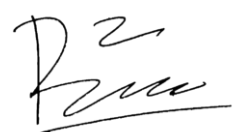


V. Kesimpulan

Program di atas akan membantu Anda lebih memahami penggunaan struktur data dan algoritma dalam konteks grafik. Program ini menggunakan kelas Graph dan metodenya untuk mengimplementasikan algoritma pengurutan topologi menggunakan pendekatan depth-first search (DFS). DFS digunakan untuk menjelajahi grafik dan menentukan urutan topologi simpulnya, dan struktur data seperti daftar dan daftar dua dimensi digunakan untuk menyimpan informasi tentang grafik dan perilaku algoritma.

Implementasi algoritma pengurutan topologi ini memberikan pemahaman yang lebih mendalam tentang konsep pengurutan topologi, dimana setiap sisi grafik menghubungkan dari node sebelumnya ke node berikutnya. Dengan memahami implementasi ini, siswa akan memperoleh pemahaman lebih dalam tentang struktur data grafik, algoritma pengurutan topologi, dan penerapan DFS untuk menyelesaikan masalah grafik. Lokakarya yang mencakup latihan-latihan ini merupakan langkah efektif untuk memperdalam pemahaman Anda tentang konsep-konsep ini.

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.