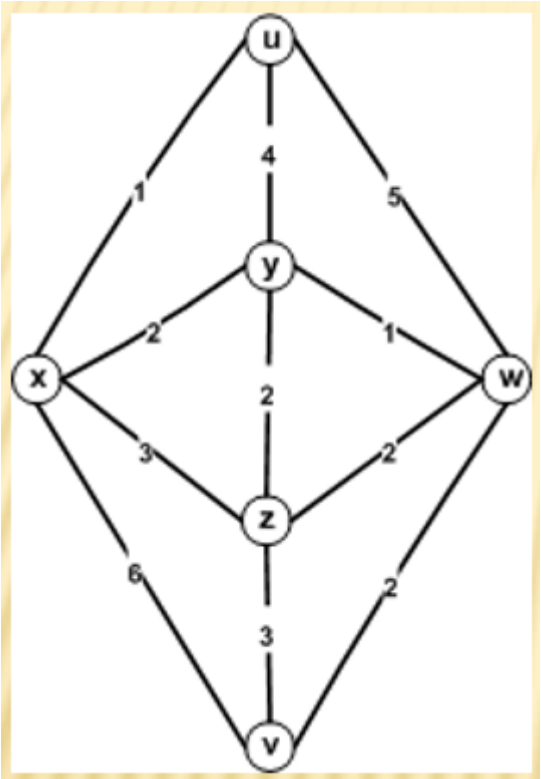
TUGAS MINGGUAN 3

Nama : Muhammad Daffa Rizky Sutrisno

NRP : 5025231207

Graph :



1. Mencari minimum-weight tour (optimal tour). Anda dapat menggunakan algoritma Fleury untuk menyelesaikan masalah pada graph di bawah kiri.

Jwb :

Algoritma Fleury adalah algoritma untuk menemukan Eulerian Path (jalur Euler) atau Eulerian Circuit (siklus Euler) pada sebuah graph. Eulerian Path adalah lintasan yang melewati setiap edge dalam sebuah graph tepat satu kali, sedangkan Eulerian Circuit adalah lintasan yang kembali ke titik awal.

Syarat :

a. Eulerian Circuit (Siklus Euler): Semua vertex harus memiliki derajat genap.
b. Eulerian Path (Jalur Euler): Hanya ada dua vertex yang berderajat ganjil (dua vertex dengan derajat ganjil ini akan menjadi titik awal dan akhir jalur Euler).

Langkah-langkah Algoritma Fleury :

1. Periksa syarat Eulerian Path atau Eulerian Circuit:
    - Jika semua vertex memiliki derajat genap, graph memiliki Eulerian Circuit.
    - Jika hanya ada dua vertex berderajat ganjil, graph memiliki Eulerian Path.
    - Jika lebih dari dua vertex berderajat ganjil, maka graph tidak memiliki Eulerian Path atau Eulerian Circuit.
2. Pilih titik awal:
    - Jika Eulerian Circuit: Anda dapat mulai dari vertex mana saja.
    - Jika Eulerian Path: Mulai dari salah satu dari dua vertex berderajat ganjil.
3. Telusuri graph:
    - Dari vertex awal, pilih edge mana yang akan dilalui. Jika ada beberapa pilihan, hindari memilih jembatan (edge yang jika dihapus akan memisahkan graph menjadi dua bagian) kecuali itu satu-satunya pilihan.

    Algoritma Fleury menghindari jembatan kecuali diperlukan karena ingin memastikan graph tetap terhubung saat menyusuri edge.

4. Hapus edge yang dilalui:
    - Setelah edge dilalui, hapus edge tersebut dari graph sehingga tidak digunakan lagi. Kemudian lanjutkan ke vertex yang terhubung oleh edge tersebut.
5. Ulangi hingga semua edge dilalui:
    - Ulangi proses pemilihan edge hingga semua edge telah dilalui tepat satu kali.
6. Selesai:
    - Jika semua edge sudah dilalui, maka Anda telah menemukan Eulerian Path atau Eulerian Circuit. Jika Anda kembali ke titik awal, maka hasilnya adalah Eulerian Circuit.

Pada soal terdapat tepat 2 vertex berderajat ganjil maka algoritma fleury dapat digunakan.

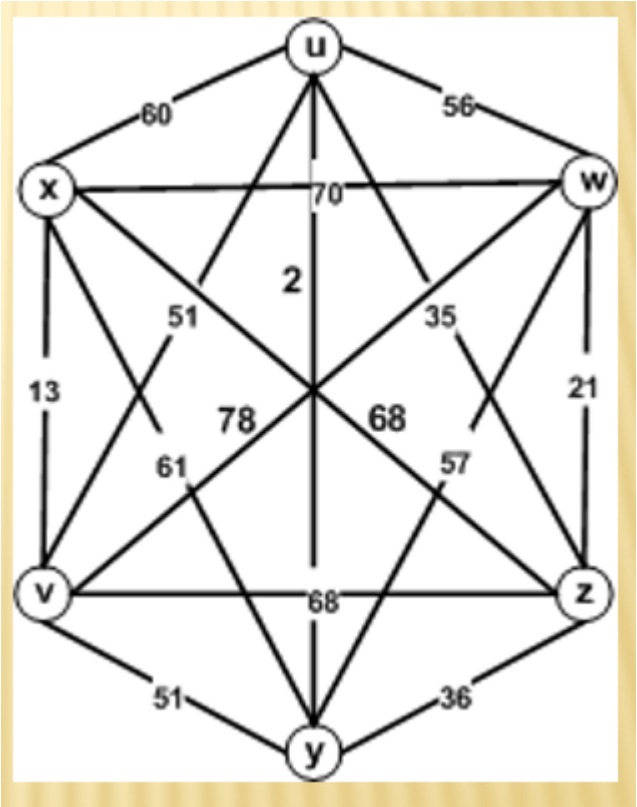
Program :

```python
class Graph:

    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)
        self.Time = 0

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

    def rmvEdge(self, u, v):
        for index, (key, _) in enumerate(self.graph[u]):
            if key == v:
                self.graph[u].pop(index)
        for index, (key, _) in enumerate(self.graph[v]):
            if key == u:
                self.graph[v].pop(index)

    def DFSCount(self, v, visited):
        count = 1
        visited[v] = True
        for i, _ in self.graph[v]:
            if visited[i] == False:
                count = count + self.DFSCount(i, visited)
        return count

    def isValidNextEdge(self, u, v):
        if len(self.graph[u]) == 1:
            return True
        else:
            visited = [False] * (self.V)
            count1 = self.DFSCount(u, visited)

            self.rmvEdge(u, v)
            visited = [False] * (self.V)
            count2 = self.DFSCount(u, visited)

            self.addEdge(u, v, 0)  # Menambahkan dummy weight 0

            return False if count1 > count2 else True
```

Output :

```
Eulerian Trail:
U-X (1) X-Y (2) Y-U (4) U-W (5) W-Y (1) Y-Z (2) Z-X (3) X-V (6) V-Z (3) Z-W (2) W-V (2)
Total Weight of Eulerian Trail: 31
```

Graph :



2. Temukan salah satu algoritma yang dapat digunakan untuk menyelesaikan permasalahan TSP pada graph di atas kanan.

Jwb :

TSP adalah masalah yang termasuk dalam kategori NP-Hard, yang berarti tidak ada algoritma eksak yang dapat menyelesaikan masalah ini secara efisien untuk graph besar. Namun, ada beberapa pendekatan heuristik dan aproksimatif yang dapat digunakan untuk memberikan solusi mendekati optimal.

Berikut adalah salah satu algoritma yang dapat digunakan untuk menyelesaikan TSP:

1. Algoritma Nearest Neighbor (NN)

Algoritma Nearest Neighbor (NN) adalah algoritma heuristik sederhana yang digunakan untuk menyelesaikan Travelling Salesman Problem (TSP). Meskipun tidak menjamin solusi optimal, algoritma ini memberikan solusi yang mendekati optimal dengan waktu komputasi yang cepat, sehingga cocok digunakan pada kasus-kasus dengan jumlah kota yang besar. Algoritma Nearest Neighbor beroperasi dengan prinsip keserakahan (greedy), di mana pada setiap langkahnya, kota terdekat yang belum dikunjungi akan dipilih untuk dikunjungi berikutnya. Algoritma ini sangat intuitif karena mengikuti logika “pilih yang terdekat dulu”.

Langkah-langkah Algoritma NN :

1. Pilih kota awal: Pilih sebuah kota untuk memulai perjalanan. Algoritma dapat dimulai dari kota mana saja, meskipun dalam implementasi praktis, pemilihan kota awal secara acak atau berdasarkan urutan tertentu umum dilakukan.
2. Pilih kota terdekat: Dari kota yang saat ini dikunjungi, cari kota terdekat (dengan bobot/biaya/jarak terkecil) yang belum dikunjungi.
3. Kunjungi kota tersebut: Bergerak ke kota terdekat yang dipilih, dan tandai kota tersebut sebagai sudah dikunjungi.
4. Ulangi langkah ini sampai semua kota dikunjungi: Teruskan memilih kota terdekat yang belum dikunjungi sampai semua kota telah dikunjungi.
5. Kembali ke kota awal: Setelah semua kota dikunjungi, kembali ke kota awal untuk menyelesaikan perjalanan.

Algoritma Nearest Neighbor adalah metode heuristik yang cepat dan sederhana untuk menyelesaikan Travelling Salesman Problem. Meskipun hasilnya tidak selalu optimal, algoritma ini sering digunakan pada kasus di mana kecepatan lebih penting daripada akurasi absolut, atau sebagai solusi awal untuk pendekatan yang lebih kompleks seperti algoritma genetika atau simulated annealing.

Program :

```python
# Fungsi untuk menghitung rute dengan algoritma Nearest Neighbor
def nearest_neighbor(start_index):
    n = len(vertex)
    visited = [False] * n  # List untuk menandai kota yang sudah dikunjungi
    visited[start_index] = True  # Tandai kota awal sebagai sudah dikunjungi
    route = [vertex[start_index]]  # Simpan rute perjalanan dimulai dari kota awal
    total_distance = 0  # Jarak total perjalanan

    current_city = start_index

    for _ in range(n - 1):
        nearest_city = -1
        nearest_distance = float('inf')

        # Cari kota terdekat yang belum dikunjungi
        for i in range(n):
            if not visited[i] and distances[current_city][i] < nearest_distance:
                nearest_city = i
                nearest_distance = distances[current_city][i]

        # Kunjungi kota terdekat
        visited[nearest_city] = True
        route.append(vertex[nearest_city])
        total_distance += nearest_distance
        current_city = nearest_city

    # Kembali ke kota awal
    total_distance += distances[current_city][start_index]
    route.append(vertex[start_index])

    return route, total_distance

# Pilih kota awal (misalnya mulai dari 'U' yaitu index 0)
start_index = 0  # U adalah index ke-0
route, total_distance = nearest_neighbor(start_index)

# Tampilkan hasil rute dan total jarak
print("Rute perjalanan:", " -> ".join(route))
print("Total jarak:", total_distance)
```

Output :

```
Rute perjalanan: U -> Y -> Z -> W -> X -> V -> U
Total jarak: 193
```