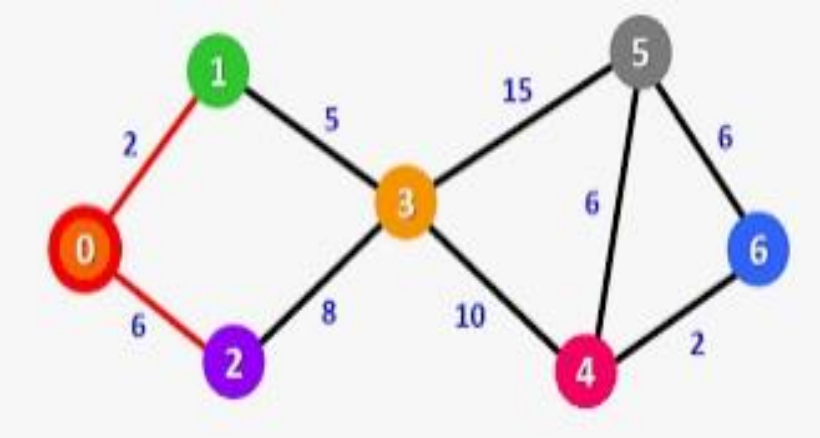
NAMA : Aliffian Akbar H
NIM : G.211.22.0122

Tugas Akhir Praktikum

Algoritma dari Graph yang dipilih :

C. Dijkstra shortest path 2



Algoritma Dijkstra, (sesuai penemunya Edsger Dijkstra), adalah sebuah algoritma yang dipakai dalam memecahkan permasalahan jarak terpendek (shortest path problem) untuk sebuah graf berarah (directed graph). Algoritma Dijkstra bekerja dengan membuat jalur ke satu simpul optimal pada setiap langkah. Jadi pada langkah ke n, setidaknya ada n node yang sudah diketahui jalur terpendeknya.

Mencari Jarak terpendek dan jalur terpendek dari graph diatas

V	0	1	2	3	4	5	6
0	⓪	2	6	∞	∞	∞	∞
1	0	②	6	7	∞	∞	∞
2	0	2	⑥	7	∞	∞	∞
3	0	2	6	⑦	17	22	∞
4	0	2	6	7	⑰	22	19
6	0	2	6	7	17	22	⑲

Hasilnya :

Jarak terpendek dari simpul 0 ke simpul 6 = 19

Jalur terpendek dari simpul 0 ke simpul 6 = 0 - 1 - 3 - 4 – 6

Membuat coding dari graph Dijkstra shortest path 2
Link : https://github.com/aliffianbr/Kuliah/tree/main/strukturData/Praktikum8

Penjelasanya :

```
import heapq
```

Baris pertama adalah impor modul heapq. Modul ini menyediakan fungsi heap queue (priority queue) yang akan digunakan untuk mengelola antrian prioritas pada algoritma Dijkstra.

```
def dijkstra(graph, start, end):
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]
```

-Fungsi dijkstra diinisialisasi dengan mengambil input berupa graf (graph), simpul awal (start), dan simpul tujuan (end).
-distances adalah kamus yang menyimpan jarak terpendek dari simpul awal ke simpul tertentu.
-Jarak awal untuk semua simpul diatur menjadi tak terbatas kecuali simpul awal yang diatur menjadi 0.
-priority_queue adalah heap yang digunakan untuk menyimpan pasangan (jarak, simpul).

```
    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        if current_node == end:
            break
        if current_distance > distances[current_node]:
            continue
```

-Iterasi dilakukan selama heap tidak kosong.
-heapq.heappop(priority_queue) mengambil simpul dengan jarak terpendek dari heap.
-Jika simpul yang diambil adalah simpul tujuan, hentikan iterasi.
-Jika jarak terkini lebih besar dari jarak yang telah dihitung sebelumnya, abaikan simpul ini dan lanjutkan ke simpul berikutnya.

```
for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
```

-Iterasi dilakukan untuk setiap tetangga dari simpul saat ini.
-Hitung jarak baru ke tetangga.
-Jika jarak baru lebih kecil dari jarak yang telah dihitung sebelumnya ke tetangga, perbarui jarak tersebut dan masukkan tetangga ke dalam heap dengan jarak yang baru.

```
  return distances[end]
```

Kembalikan jarak terpendek dari simpul awal ke simpul tujuan setelah keluar dari loop.

```
def dijkstra_path(graph, start, end):
   distances = {node: float('infinity') for node in graph}
   distances[start] = 0
  priority_queue = [(0, start, [])]
```

Fungsi dijkstra_path hampir sama dengan dijkstra, tetapi ada penambahan argumen current_path yang digunakan untuk menyimpan jalur yang telah diambil.

```
   while priority_queue:
      current_distance, current_node, current_path =
heapq.heappop(priority_queue)
      if current_node == end:
         return current_path + [end]
      if current_distance > distances[current_node]:
         continue
```

Iterasi dilakukan dengan menambahkan current_path ke parameter fungsi.
Jika simpul saat ini adalah simpul tujuan, kembalikan jalur yang telah diambil.

```
for neighbor, weight in graph[current_node].items():
     distance = current_distance + weight
       if distance < distances[neighbor]:
          distances[neighbor] = distance
            heapq.heappush(priority_queue, (distance, neighbor, current_path +
[current_node]))
```

Iterasi dilakukan seperti di dijkstra, tetapi kali ini jalur yang telah diambil juga diperbarui.

```
return []
```

Jika tidak ada jalur yang ditemukan, kembalikan daftar kosong.

```
graph = {
   0: {1: 2, 2: 6},
   1: {0: 2, 3: 5},
   2: {0: 6, 3: 8},
   3: {1: 5, 2: 8, 4: 10, 5: 15},
   4: {3: 10, 5: 6, 6: 2},
   5: {3: 15, 4: 6, 6: 6},
   6: {4: 2, 5: 6}
}
```

Inisialisasi graf berbobot yang akan diuji menggunakan algoritma Dijkstra.

```
start_node = 0
end_node = 6
```

Penetapan simpul awal (start_node) dan simpul tujuan (end_node) untuk

`shortest_distance = dijkstra(graph, start_node, end_node)`: Memanggil fungsi dijkstra untuk mencari jarak terpendek.

`shortest_path = dijkstra_path(graph, start_node, end_node)`: Memanggil fungsi dijkstra_path untuk mencari jalur terpendek.

Blok kondisional untuk mencetak output sesuai dengan hasil pencarian jalur terpendek:

Jika shortest_path tidak kosong, mencetak jalur terpendek dan jarak terpendek.
Jika shortest_path kosong, mencetak pesan bahwa tidak ada jalur yang ditemukan.