# JAWABAN UJIAN AKHIR SEMESTER (UAS)

# ALGORITMA ANALISIS

Dosen Pengampu :

**Dr. Achmad Hindasyah, M.Si**


**OLEH**

**ASEP RIDWAN HIDAYAT**

**231012050036**

**TI 01MKME001 REGULAR C**

**PROGRAM STUDI MAGISTER TEKNIK INFORMATIKA**

**UNIVERSITAS PAMULANG**

**TANGERANG SELATAN**


**2024**

**SOAL NO. 1**

1. Pemakaian ruang multi fungsi seperti ditabelkan di bawah. Lakukan simulasi untuk mendapatkan urutan aktifitas yang bisa memberikan keuntungan maksimum

| $X_i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $P_i$ | 50 | 75 | 25 | 30 | 60 | 45 | 25 |
| $D_i$ | 12 | 13 | 11 | 12 | 13 | 9 | 10 |

## Jawaban No 1

**A. Algoritma Seleksi aktifitas**

1. Langkah pertama urutkan aktivitas berdasarkan Profit terbesar dari waktu finis terkecil (tercepat) hingga terlama
2. Pilih aktifitas pertama dari array yang sudah diurutkan
3. Eksekusi aktifitas berikutnya yang masih ada dalam array tersisa
    a. Pilih aktifitas yang mempunyai waktu start lebih besar daripada waktu finish aktifitas sebelumnya
    b. kondisi lain ditolak
4. Ulangi langkah 3 hingga semua aktifitas dieksekusi

**B. Implementasi Analitik**

Berikut ini implementasinya, Terdapat 7 job sequence dari table diatas, urutkan aktifitas yang memberikan profit maksimum

Xi = aktifitas ke-i

Pi = Profit ke-i

Di = Durasi dari aktifitas

1. Urutkan berdasarkan profit

| Xi | 2 | 5 | 1 | 6 | 4 | 7 | 3 |
|---|---|---|---|---|---|---|---|
| Pi | 75 | 60 | 50 | 45 | 30 | 25 | 25 |
| Di | 13 | 13 | 12 | 9 | 12 | 10 | 11 |

2. Urutkan sesuai urutan profit dan waktu

| Aktifitas (durasi ke-) | Profit |
|---|---|
| $X_2(13)$ | P = 75 |
| $X_2(13)$ + $X_5(13)$ | P = 75 + 60 = 135 |
| X2(13) + $X_5(13)$ + $X_6(9)$ | P = 75+60+45 = 180 |
| | |

Atau bisa juga digambarkan dengan tabel sebagai berikut:

| Aktivitas | Profit (Pi) | Waktu Penyelesaian (Di) | Waktu Mulai | Waktu Selesai |
|---|---|---|---|---|
| 2 | 75 | 13 | 1 | 13 |
| 5 | 60 | 13 | 14 | 26 |
| 6 | 45 | 9 | 27 | 35 |
| 1 | 50 | 12 | 36 | 47 |
| 4 | 30 | 12 | 48 | 59 |
| 7 | 25 | 10 | 60 | 69 |
| 3 | 25 | 11 | Tidak bisa dijadwalkan | Tidak bisa dijadwalkan |

**Jadi Urutan maksimum didapat [2, 5,6,], Profit maksimum adalah 180**

Note:
Aktifitas 6 dipilih terlebih dahulu dibanding aktifitas 1 karena memiliki profit besar (45) dan membutuhkan waktu penyelesaian yang lebih singkat (9). Aktivitas 1, 4, 7, dan 3 tidak dapat dimulai sebelum aktivitas 6 selesai karena akan bertabrakan dengan waktu penyelesaian aktivitas 6 yang sudah dijadwalkan.

## C. Berikut pemograman menggunakan Bahasa python

```
import matplotlib.pyplot as plt
import numpy as np

def maximize_profit(n, activities):
    Pi = [activities[x][0] for x in range(1, n + 1)]
    Di = [activities[x][1] for x in range(1, n + 1)]

    activities_with_values = list(zip(range(1, n + 1), Pi, Di))
    activities_with_values.sort(key=lambda x: x[1] / x[2], reverse=True)
```

```
    max_profit = 0
    current_time = 0
    chosen_activities = []
    cumulative_profit = []

    for activity in activities_with_values:
        if current_time + activity[2] <= 50:
            chosen_activities.append(activity[0])
            max_profit += activity[1]
            current_time += activity[2]
            cumulative_profit.append(max_profit)
        else:
            break

    return max_profit, chosen_activities, cumulative_profit

# Input jumlah jenis aktivitas dan nilai Pi serta Di
def get_input():
    try:
        n = int(input("Masukkan jumlah jenis aktivitas: "))
        activities = {}

        for x in range(1, n + 1):
            p = int(input(f"Masukkan nilai Pi untuk jenis aktivitas {x}: "))
            d = int(input(f"Masukkan nilai Di untuk jenis aktivitas {x}: "))
            activities[x] = (p, d)

        return n, activities

    except ValueError:
        print("Masukkan hanya angka untuk jumlah jenis aktivitas, Pi, dan
Di.")
        return None, None

# Contoh penggunaan:
n, activities = get_input()

if n is not None and activities is not None:
    max_profit, chosen_activities, cumulative_profit = maximize_profit(n,
activities)

    print(f"Urutan aktivitas untuk keuntungan maksimum:
{chosen_activities}")
    print(f"Keuntungan maksimum yang dapat diperoleh: {max_profit}")
```

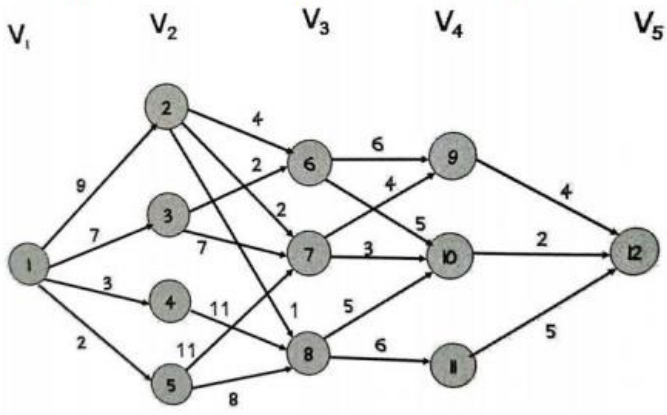
```
    # Plotting cumulative profit over time or activities
    plt.figure(figsize=(10, 6))
    plt.plot(np.arange(len(cumulative_profit)) + 1, cumulative_profit,
marker='o', linestyle='-', color='b', label='Cumulative Profit')
    plt.xlabel('Number of Activities')
    plt.ylabel('Cumulative Profit')
    plt.title('Cumulative Profit over Selected Activities')
    plt.xticks(np.arange(len(cumulative_profit)) + 1)
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()
```

**D. Berikut Output dari ekskusi pemogramannya**

```
Masukkan jumlah jenis aktivitas: 7
Masukkan nilai Pi untuk jenis aktivitas 1: 50
Masukkan nilai Di untuk jenis aktivitas 1: 12
Masukkan nilai Pi untuk jenis aktivitas 2: 75
Masukkan nilai Di untuk jenis aktivitas 2: 13
Masukkan nilai Pi untuk jenis aktivitas 3: 25
Masukkan nilai Di untuk jenis aktivitas 3: 11
Masukkan nilai Pi untuk jenis aktivitas 4: 30
Masukkan nilai Di untuk jenis aktivitas 4: 12
Masukkan nilai Pi untuk jenis aktivitas 5: 60
Masukkan nilai Di untuk jenis aktivitas 5: 13
Masukkan nilai Pi untuk jenis aktivitas 6: 45
Masukkan nilai Di untuk jenis aktivitas 6: 9
Masukkan nilai Pi untuk jenis aktivitas 7: 25
Masukkan nilai Di untuk jenis aktivitas 7: 10
Urutan aktivitas untuk keuntungan maksimum: [2, 5, 6]
Keuntungan maksimum yang dapat diperoleh: [180]
```

**SOAL NO. 2**

2. Dengan menggunakan metoda forward dan metoda backward dapatkan jalur terpendek dari grafik multistage di bawah ini :




## Jawaban No 2

### A. ALGORITMA

**1. Algoritma metoda forward (menghitung jarak ke depan)**

1) Hitung lintasan path dari suatu simpul ke tujuan
   Rumus cost(I,j) = min {c(j,k)+cost(i+1,k)}
2) Perhitungan dimulai dari node k-2

**Perhitungan analitic:**

| | | | | |
|---|---|---|---|---|
| V4 | | | | |
| Cost (4,9) | = | c (9,12) | = | 4 |
| Cost (4,10) | = | c (10,12) | = | 2 {minimal} |
| Cost (4,11) | = | c (11,12) | = | 5 |
| V3 | | | | |
| Cost (3,6) | =<br><br>=<br>= | Min {c (6,9) +cost (4,9) ; c(6,10) + cost(4,10)}<br>Min { 6+4;5+2}<br>Min {10;7} | = | 7 |
| Cost (3,7) | = | Min {c (7,9) +cost (4,9) ; c(7,10) + cost(4,10)} | = | 7 |

| | | | | |
|---|---|---|---|---|
| | =<br>= | Min { 4+4;3+2}<br>Min {8;7} | | |
| Cost (3,8) | =<br><br>=<br>= | Min {c (8,10) +cost (4,10) ; c(8,11) + cost(4,11)}<br>Min { 5+2;6+5}<br>Min {7;11} | = | 7 |
| V2 | | | | |
| Cost (2,2) | =<br><br>=<br>= | Min {c(2,6) +cost (3,6) ; c(2,7) + cost(3,7);c(2,8)+cost(3,8) }<br>Min { 4+7;2+7;1+7}<br>Min {11;9,8} | = | 8 {min} |
| Cost (2,3) | =<br><br>=<br>= | Min {c (3,6) +cost (3,6) ; c(3,7) + cost(3,7) }<br>Min { 2+7;7+7}<br>Min {9;14} | = | 9 |
| Cost (2,4) | =<br><br>= | Min {c (4,8) +cost (3,8) }<br>Min { 11+7}<br>Min {18} | = | 18 |
| Cost (2,5) | =<br><br>= | Min {c (5,7) +cost (3,7) ; c(5,8) + cost(3,8) }<br>Min { 11+7;8+7}<br>Min {18;15} | = | 15 |
| V1 | | | | |
| Cost (1,1) | =<br><br>=<br>= | Min {c (1,2) +cost (2,2); c(1,3) + cost(2,3); c(1,4) + cost(2,4); c(1,5) + cost(2,5)}<br>Min { 9+8;7+9;3+18;2+15}<br>Min {17;16;21;17} | = | 16 |

**Jadi jalur terpendek nya yaitu : 1-2-6-10-12 dengan panjang lintasan 33**

**2. Algoritma Metoda Backward (Menghitung jarak kebelakang)**

1. Hitung lintasan path dari suatu simpul ke tujuan
   Rumus : bcost(I,j) = min {bc(i-1,1)+cost(1,j)}
2. Perhitungan dimulai dari node-node distage 3

**Perhitungan analitic**

| V2 | | | | |
|---|---|---|---|---|
| bcost b(2,2) | = | C(1,2) | = | 9 |
| bcost (2,3) | = | C(1,3) | = | 7 |
| bcost (2,4) | = | C(1,4) | = | 3 |
| bcost (2,5) | = | C(1,5) | = | 2 {min} |

| | | | | |
|---|---|---|---|---|
| V3 | | | | |
| bCost (3,6) | =<br><br>=<br>= | Min {c (2,6) +bcost (2,2) ; c(2,7) + bcost(2,2);c(2,8)+bcost(2,2)}<br>Min { 4+9;2+9;1+9}<br>Min {13;11;20} | = | 11 |
| bCost (3,7) | =<br><br>= | Min {c (3,6) +bcost (2,3) ; c(3,7) + bcost(2,3) }<br>Min { 2+7;7+7}<br>Min {9;14} | = | 9 {min} |
| bCost (3,8) | =<br><br>= | Min {c (3,8) +bcost (2,4) ; c(3,8) + bcost(2,5) }<br>Min { 11+3;8+2}<br>Min {14;10} | = | 10 |
| V4 | | | | |
| bCost (4,9) | =<br><br>= | Min {c (9,6) +bcost (3,6) ; c(9,7) + bcost(3,7) }<br>Min { 6+11;4+9}<br>Min {17;13} | = | 13 {min} |
| bCost (4,10) | =<br><br>=<br>= | Min {c (10,7) +bcost (3,6) ; c(10,7) + bcost(3,7);c(10,8)+bcos(3,8) }<br>Min { 5+11;3+9;5+10}<br>Min {16;14;15} | = | 14 |
| bCost (4,11) | =<br><br>=<br>= | Min {c (11,8) +bcost (3,8) }<br>Min { 6+10}<br>Min {16} | = | 16 |
| V1 | | | | |
| bCost (1,12) | =<br><br>=<br>= | Min {c (12,9) +bcost (4,9); c(12,10) + bcost(4,10); c(12,11) + bcost(4,11) }<br>Min { 4+13;2+14;5+16}<br>Min {17;19;21} | = | 17 |

**Jadi jalur terpendek nya yaitu : 12 -10- 7-5-1 dengan panjang lintasan 41**

**B. Pemograman dengan Python**

```
import networkx as nx

def forward_multistage_graph(graph, stages, start):
    # Initialize the cost to reach each node and the paths
    cost = {node: float('inf') for stage in stages for node in stage}
    cost[start] = 0
    path = {node: [] for stage in stages for node in stage}
    path[start] = [start]

    # Iterate over each stage
    for stage in stages[:-1]:
        for node in stage:
```

```
            for neighbor, weight in graph[node]:
                if cost[node] + weight < cost[neighbor]:
                    cost[neighbor] = cost[node] + weight
                    path[neighbor] = path[node] + [neighbor]

    # Extract the minimum cost to reach the final stage nodes and their paths
    last_stage = stages[-1]
    min_cost = float('inf')
    best_path = []
    for node in last_stage:
        if cost[node] < min_cost:
            min_cost = cost[node]
            best_path = path[node]

    return min_cost, best_path

def backward_multistage_graph(graph, stages, start, end):
    # Initialize the cost to reach each node and the paths
    cost = {node: float('inf') for stage in stages for node in stage}
    cost[end] = 0
    path = {node: [] for stage in stages for node in stage}
    path[end] = [end]

    # Create a reversed graph with all nodes
    reversed_graph = {node: [] for node in graph}

    # Populate the reversed graph with reversed edges
    for node in graph:
        for neighbor, weight in graph[node]:
            reversed_graph.setdefault(neighbor, []).append((node, weight))

    # Reverse the stages
    reversed_stages = stages[::-1]

    # Iterate over each stage (in reversed order)
    for stage in reversed_stages[:-1]:
        for node in stage:
            for neighbor, weight in reversed_graph.get(node, []):
                if cost[node] + weight < cost[neighbor]:
                    cost[neighbor] = cost[node] + weight
                    path[neighbor] = path[node] + [neighbor]

    # Extract the minimum cost to reach the initial stage nodes and their paths
    first_stage = reversed_stages[-1]
    min_cost = float('inf')
```

```
    best_path = []
    for node in first_stage:
        if cost[node] < min_cost:
            min_cost = cost[node]
            best_path = path[node]

    return min_cost, best_path

# Define the graph and stages (as per your example)
graph = {
    1: [(2, 9), (3, 7), (4, 3), (5, 2)],
    2: [(6, 4), (6, 2), (7,2),(7,7),(7,11),(8, 1),(8,11),(8,8)],
    3: [(6, 2), (7, 7)],
    4: [(8, 11)],
    5: [(7, 11), (8, 8)],
    6: [(9, 6), (10, 5)],
    7: [(9, 4), (10, 3)],
    8: [(10, 5), (11, 6)],
    9: [(12, 4)],
    10: [(12, 2)],
    11: [(12, 5)],
    12: []  # End node without outgoing edges
}

stages = [
    [1],    # v1
    [2, 3, 4, 5],  # v2
    [6, 7, 8],  # v3
    [9, 10, 11],  # v4
    [12]  # v5
]

# Calculate shortest path using forward method
start_node = 1
min_cost_forward, best_path_forward = forward_multistage_graph(graph, stages,
start_node)

print("Shortest Path (Forward Method):")
print("Cost:", min_cost_forward)
print("Path:", best_path_forward)

# Calculate shortest path using backward method
end_node = 12
min_cost_backward, best_path_backward = backward_multistage_graph(graph,
stages, start_node, end_node)
```

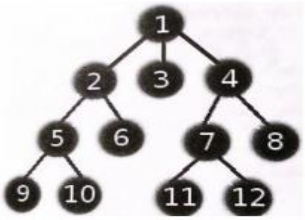
```
print("\nShortest Path (Backward Method):")
print("Cost:", min_cost_backward)
print("Path:", best_path_backward)
```

**C. Output Pemograman**

```
Shortest Path (Forward Method):
Cost: [1, 2, 6, 10, 12]
Path: [33]

Shortest Path (Backward Method):
Cost: 16
Path: [12, 10, 7, 2, 1]
```

## SOAL NO 3

3. Perhatikan grafik di bawah ini, start di ambil dari simpul 1. Dapatkan urutan (himpunan) simpul keluaran jika digunkan algoritma
   - Level order tarversal
   - Pre order tarversal
   - In order tarversal
   - Post order tarversal




**Jawaban No 3**

**A. ALGORITMA DAN ANALITIC**

**1. LEVEL ORDER TRAVERSAL**

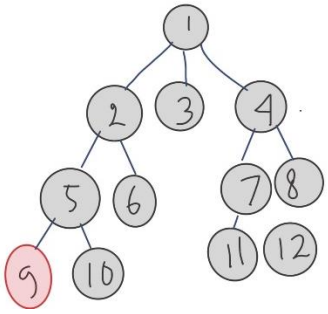
**A. Algoritma Level Order Traversal**

1. Nyatakan dua daftar kosong : **open list** dan **closed list**
2. Tambahkan simpul awal pada **open list**
3. Selama **open list** tidak kosong maka :
   a. Keluarkan simpul pertama dari open list

b. Cek apakah yang dikeluarkan simpul tujuan atau bukan?
   i. Jika ya, stop pencarian dan keluar dan tambahkan simpul pada closed list dan kembalikan nilai closed list,
   ii. jika bukan simpul tujuan kerjakan Langkah c
4. eksplorasi simpul dilevel lebih tinggi dari simpul yang dikeluarkan
5. tambahkan disekeliling simpul akhir dari **open list,** dan tambahkan simpul yang dikeluarkan pada **closed list**

**B. Level Order Traversal sebagai berikut**

1. Langkah 1
   - Open list : 1
   - Closed list :<empty>
2. Langkah 2
   - Open list : 2,3,4
   - Closed list : 1
3. Langkah 3
   - Open list 5,6,7,8
   - Closed list : 1,2,3,4
4. Langkah 4
   - Open list 9,10,11,12
   - Closed list : 1,2,3,4,5,6,7,8
5. Langkah 5
   Simpul 9 dieksplorasi dan ternyata **simpul tujuan**, maka pelacakan berhenti sampai disini dan diperoleh




Dengan metode order traversal lintasan pelacakan yang dilalui berada pada daftar close list yaitu : **1,2,3,4,5,6,7,8, 9**

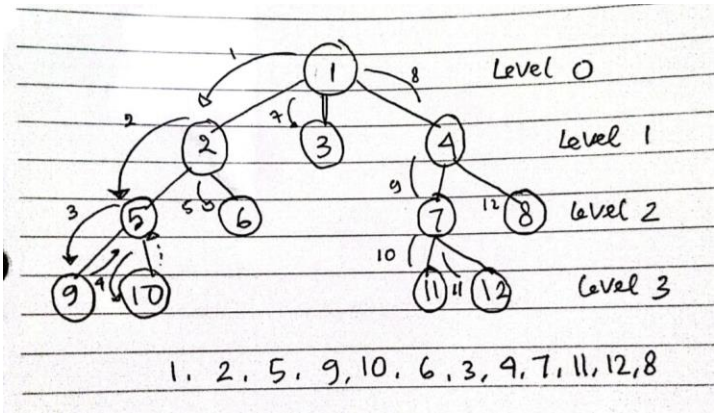
## 2. PREORDER TRAVERSAL

### A. Algoritma Level Order Traversal

1. Nyatakan dua daftar kosong : **open list** dan **closed list**
2. Tambahkan simpul awal pada **open list**
3. Selama **open list** tidak kosong maka :

a. Keluarkan simpul pertama dari open list
b. Cek apakah yang dikeluarkan simpul tujuan atau bukan?
   - Jika ya, stop pencarian dan keluar dan tambahkan simpul pada closed list dan kembalikan nilai closed list,
   - Jika bukan simpul tujuan kerjakan Langkah c
4. eksplorasi simpul dilevel lebih tinggi dari simpul yang dikeluarkan
5. tambahkan disekeliling simpul akhir dari **open list,** dan tambahkan simpul yang dikeluarkan pada **closed list**

**B. Preorder traversal sebagai berikut**



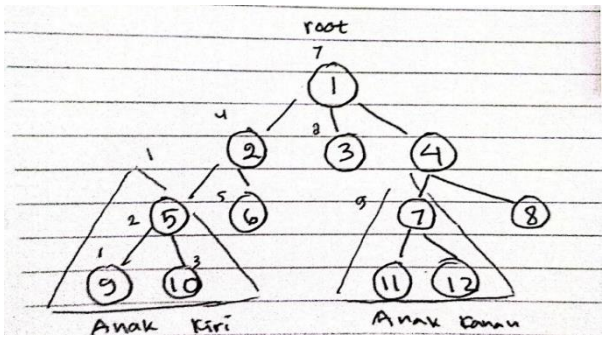

**Jadi : 1, 2, 5, 9, 10, 6, 3, 4, 7, 11, 12, 8**

**3. IN ORDER TRAVERSAL**

**A. Algoritma In Order Traveersa**

1. Eksplorasi simpul anak kiri
2. Eksplorasi simpul induk
3. Eksplorasi simpul kanan

**B. Berikut analitic in order traversaal**

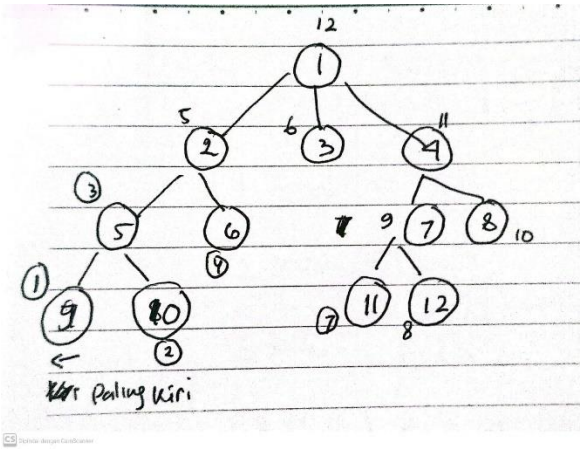



**Jadi : 9, 5, 10, 2, 6, 1, 3**

## 4. POST ORDER TRAVERSAL

### A. Algoritma Post Order Traversal

1. Lintasi  semua eksternal kiti dimulai paling kiri dan ikuti simpul internal diatasnya
2. Lintasi semua simpul kanan mulai dari simpul paling kiri dan ikuti simpul internal diatasnya




**Jadi :  [9, 10, 5, 6, 2, 3, 11, 12, 7, 8, 4, 1]**

### B. Pemograman dengan Python

```
import networkx as nx
import matplotlib.pyplot as plt

# Definisi graf menggunakan dictionary
graph = {
    1: [2, 3, 4],
    2: [5, 6],
    5: [9, 10],
    4: [7, 8],
    7: [11, 12],
    # Nodes with no outgoing edges are omitted since they won't affect
traversal.
}

# Membuat objek graf
G = nx.DiGraph(graph)

# Mengatur posisi node secara manual untuk visualisasi yang lebih rapi
pos = {
    1: (0, 0),
    2: (-1, -1),
    3: (0, -1),
```

```
    4: (1, -1),
    5: (-2, -2),
    6: (-1, -2),
    7: (1, -2),
    8: (2, -2),
    9: (-3, -3),
    10: (-2, -3),
    11: (1, -3),
    12: (2, -3),
}

# Inisialisasi level order traversal menggunakan BFS
def level_order_traversal(start):
    if start not in graph:
        return []

    queue = [start]
    result = []
    visited = set(queue)

    while queue:
        node = queue.pop(0)
        result.append(node)
        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

    return result

# Pre order traversal
def pre_order_traversal(node):
    if node is None:
        return []

    result = []
    stack = [node]

    while stack:
        current = stack.pop()
        result.append(current)
        children = graph.get(current, [])
        stack.extend(reversed(children))  # Reverse to maintain left-to-
right order
```

```
    return result

# In order traversal
def in_order_traversal(node):
    if node is None:
        return []

    result = []
    stack = []
    current = node

    while stack or current:
        while current:
            stack.append(current)
            current = graph.get(current, [])[0] if graph.get(current) else
None  # Go leftmost

        current = stack.pop()
        result.append(current)
        current = graph.get(current, [])[1] if graph.get(current) and
len(graph.get(current)) > 1 else None  # Go right

    return result

# Post order traversal
def post_order_traversal(node):
    if node is None:
        return []

    result = []
    stack = [node]

    while stack:
        current = stack.pop()
        result.append(current)
        children = graph.get(current, [])
        stack.extend(children)  # Extend with children to process them next

    return result[::-1]  # Reverse the result to get post-order

# Calculate and print traversals
print("Level Order Traversal:", level_order_traversal(1))
print("Pre Order Traversal:", pre_order_traversal(1))
print("In Order Traversal:", in_order_traversal(1))
print("Post Order Traversal:", post_order_traversal(1))
```

```
# Draw the graph for visualization
plt.figure(figsize=(8, 8))
nx.draw(G, pos, with_labels=True, node_size=2000, node_color='lightblue',
font_size=12, font_weight='bold', arrows=True)
plt.title('Traversal Graph')
plt.show()
```

## C. Output Pemograman Python

```
Level Order Traversal: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Pre Order Traversal: [1, 2, 5, 9, 10, 6, 3, 4, 7, 11, 12, 8]
In Order Traversal: [9, 5, 10, 2, 6, 1, 3]
Post Order Traversal: [9, 10, 5, 6, 2, 3, 11, 12, 7, 8, 4, 1]
```

## Soal No 4

4. Lakukan simulasi untuk menentukan jumlah warna yang dibutuhkan untuk peta Jawa Tengah berikut :


**Jawaban No 4**

### A. Algoritma Graph Coloring

1. Tentukan jumlah warna
2. Eksplorasi simpul pertama dan beri warna pertama yang ditentukan
3. Eksplorasi simpul kedua yang berdekatan dan beri warna kedua
4. Kerjakan untuk simpul-simpul sisanya, dan beri warna
   - Jika warna sama dengan warna simpul berdekatan ditolak
   - Jika warna berbeda diterima
5. Ulangi Langkah 3 sampai semua simpul diberi warna

### B. Pemograman dengan Python

```
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import math

# Definisikan wilayah dan atributnya dengan variabel
wilayah_1 = 'batang (1)'
wilayah_2 = 'tegal (2)'
wilayah_3 = 'purwerejo (3)'
wilayah_4 = 'west java (4)'
wilayah_5 = 'demak (5)'
wilayah_6 = 'cilacap (6)'
wilayah_7 = 'Berebes (7)'
```

```
wilayah_8 = 'klaten (8)'
wilayah_9 = 'sragen (9)'
wilayah_10 = 'wonogiri (10)'
wilayah_11 = 'Rembang(11)'
wilayah_12 = 'pemalang (12)'
wilayah_13 = 'magelang (13)'
wilayah_14 = 'kudus (14)'
wilayah_15 = 'Sukoharjo (15)'
wilayah_16 = 'semarang (16)'
wilayah_17 = 'Karang Anyar (17)'
wilayah_18 = '18'
wilayah_19 = 'purbalingga (19)'
wilayah_20 = 'pekalongan (20)'
wilayah_21 = 'yogyakarta (21)'
# wilayah_22 = '22'
wilayah_23 = 'semarang (23)'
wilayah_24 = 'Blora (24)'
wilayah_25 = 'salatiga (25)'
wilayah_26 = 'Banyumas (26)'
wilayah_27 = 'temanggung (27)'
wilayah_28 = 'tegal pekal(28)'
wilayah_29 = 'kendal (29)'
wilayah_30 = 'Jepara (30)'
wilayah_31 = 'kebumen (31)'
wilayah_32 = 'Grobogan (32)'
wilayah_33 = '33'
wilayah_34 = '34'
# wilayah_35 = '35'

# Definisikan semua wilayah sebagai daftar
wilayahs = [
    wilayah_1, wilayah_2, wilayah_3, wilayah_4,wilayah_5, wilayah_6,
wilayah_7, wilayah_8, wilayah_9, wilayah_10,
    wilayah_11, wilayah_12, wilayah_13, wilayah_14, wilayah_15,
wilayah_16, wilayah_17, wilayah_18, wilayah_19, wilayah_20,
    wilayah_21,    wilayah_23, wilayah_24, wilayah_25, wilayah_26,
wilayah_27, wilayah_28, wilayah_29, wilayah_30,
    wilayah_31, wilayah_32, wilayah_33, wilayah_34 #,
wilayah_35  #   wilayah_22,
]

# Bangun grafik menggunakan networkx
G = nx.Graph()

# Tambahkan semua wilayah sebagai node
```

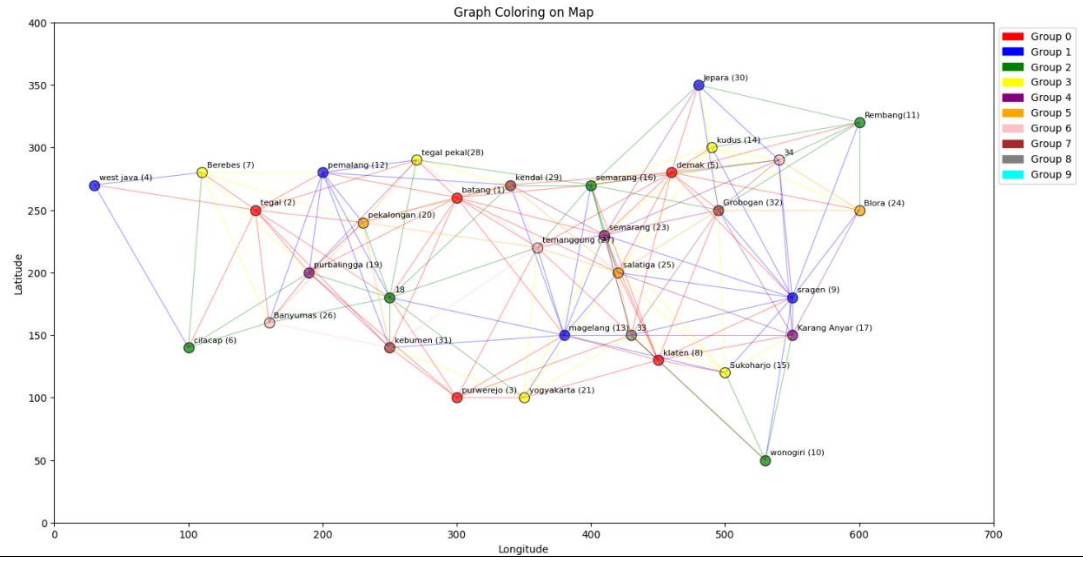
```
G.add_nodes_from(wilayahs)

# Tentukan posisi node (koordinat untuk plotting)
positions = {
    wilayah_1: (300, 260), wilayah_2: (150, 250), wilayah_3: (300,
100),    wilayah_4: (30, 270),
    wilayah_5: (460, 280), wilayah_6: (100, 140), wilayah_7: (110, 280),
wilayah_8: (450, 130),
    wilayah_9: (550, 180), wilayah_10: (530, 50), wilayah_11: (600,
320), wilayah_12: (200, 280),
    wilayah_13: (380, 150), wilayah_14: (490, 300), wilayah_15: (500,
120), wilayah_16: (400, 270),
    wilayah_17: (550, 150), wilayah_18: (250, 180), wilayah_19: (190,
200), wilayah_20: (230, 240),
    wilayah_21: (350, 100),    #  wilayah_22: (660, 110),
    wilayah_23: (410, 230), wilayah_24: (600, 250),    wilayah_25: (420,
200), wilayah_26: (160, 160), wilayah_27: (360, 220), wilayah_28: (270,
290),
    wilayah_29: (340, 270), wilayah_30: (480, 350), wilayah_31: (250,
140), wilayah_32: (495, 250),
    wilayah_33: (430, 150), wilayah_34: (540, 290)#, wilayah_35: (140,
130)
}

# Fungsi untuk menghitung jarak Euclidean antara dua titik
def euclidean_distance(pos1, pos2):
    return math.sqrt((pos1[0] - pos2[0])**2 + (pos1[1] - pos2[1])**2)

# Tentukan ambang batas jarak (misalnya 150)
distance_threshold = 150

# Tambahkan edge antara node yang jaraknya kurang dari ambang batas
for node1 in wilayahs:
    for node2 in wilayahs:
        if node1 != node2 and euclidean_distance(positions[node1],
positions[node2]) < distance_threshold:
            G.add_edge(node1, node2)

# Algoritma pewarnaan graf greedy
def greedy_graph_coloring(graph):
    color_map = {}
    for node in graph.nodes():
        available_colors = set(range(len(graph.nodes())))
        for neighbor in graph.neighbors(node):
            if neighbor in color_map:
```

```
                available_colors.discard(color_map[neighbor])
        color_map[node] = min(available_colors)
    return color_map

# Panggil fungsi pewarnaan graf
color_map = greedy_graph_coloring(G)

# Tentukan warna untuk setiap kelompok berdasarkan hasil pewarnaan graf
color_palette = [
    'red', 'blue', 'green', 'yellow', 'purple', 'orange', 'pink',
'brown', 'gray', 'cyan'
]

# Fungsi untuk plot grafik pada peta
def plot_graph_on_map(positions, color_map):
    # Initialize figure and axis
    fig, ax = plt.subplots(figsize=(12, 10))

    # Tambahkan gambar peta di latar belakang plot
    # map_img = plt.imread("peta.png")
    # ax.imshow(map_img, extent=[0, 700, 0, 400])

    # Gambar node (wilayah) dengan warna berdasarkan hasil pewarnaan
graf
    for node, pos in positions.items():
        color = color_palette[color_map[node] % len(color_palette)]
        ax.scatter(pos[0], pos[1], s=100, edgecolors='k',
facecolors=color, alpha=0.7)
        # Tambahkan nama wilayah di posisi node
        ax.annotate(node, xy=pos, xytext=(5, 5), textcoords='offset
points', fontsize=8, color='black')

    # Gambar edge
    for edge in G.edges():
        color = color_palette[color_map[edge[0]] % len(color_palette)]
        ax.plot([positions[edge[0]][0], positions[edge[1]][0]],
[positions[edge[0]][1], positions[edge[1]][1]], '-', color=color,
alpha=0.4, linewidth=0.8)

    # Buat legenda untuk kelompok warna
    legend_patches = []
    for idx, color in enumerate(color_palette):
        legend_patches.append(mpatches.Patch(color=color, label=f'Group
{idx}'))
```

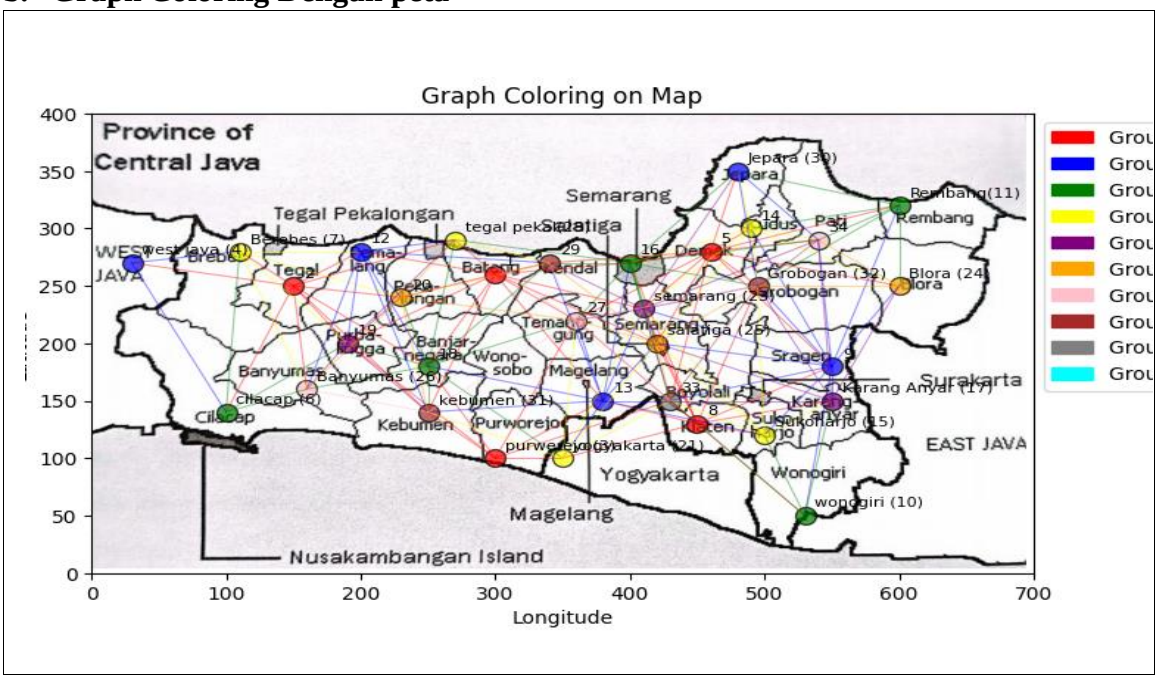
```
    # Tambahkan legenda ke plot
    ax.legend(handles=legend_patches, loc='upper left',
bbox_to_anchor=(1, 1))

    # Atur batas dan label sumbu
    ax.set_xlim(0, 700)
    ax.set_ylim(0, 400)
    ax.set_xlabel('Longitude')
    ax.set_ylabel('Latitude')
    ax.set_title('Graph Coloring on Map')

    # Tampilkan grafik
    plt.tight_layout()
    plt.show()

# Panggil fungsi untuk menampilkan grafik pada peta
plot_graph_on_map(positions, color_map)
```

## C. Output pemograman Pyhthon

### a. Graph Coloring Tanpa peta

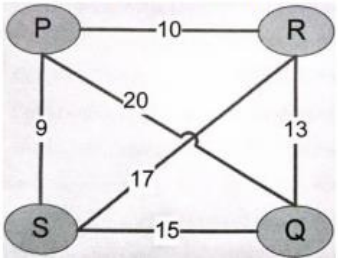


**b. Graph Coloring Dengan peta**




**SOAL NO 5**

5. Seorang sales harus memasarkan produknya di lima kota yang berbeda. Jarak antar kota seperti tertera pada gambar. Pilih lintasa terpendek menggunakan algoritma depth fist searh !

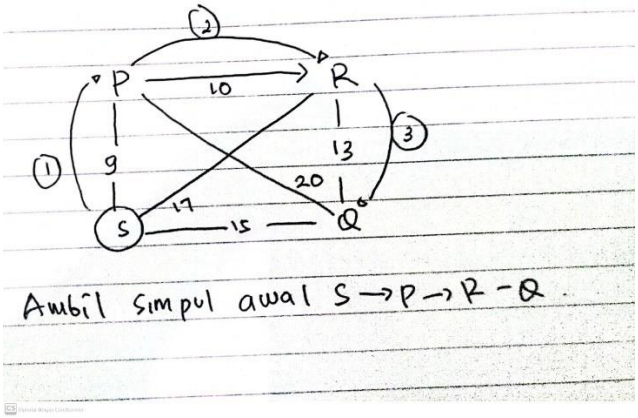



**Jawaban No. 5**

**A. Algoritma Depth First Search**

1. Mulai dari Node Akar atau Node Awal
2. Tandai node awal sebagai dikunjungi.
3. Jelajahi Tetangga yang Belum Dikunjungi
4. Ulangi langkah 2 dan 3 untuk node tetangga.
5. Jika semua tetangga dari node saat ini telah dikunjungi, kembali ke node sebelumnya (backtrack) dan jelajahi tetangga lainnya yang belum dikunjungi

**B. Analitic nya**




**Jadi lintasan terpendek : S-P-R-Q**

## C. Pemograman Menggunakan Python

```
def dfs(graph, node, visited, path):
    # Tandai node sebagai dikunjungi
    visited[node] = True
    path.append(node)

    # Telusuri semua tetangga node yang belum dikunjungi
    for next_node in graph[node]:
        if not visited[next_node]:
            dfs(graph, next_node, visited, path)

    return path

# Membangun graf dari gambar
graph = {
    "S": ["P", "Q"],
    "P": ["R"],
    "Q": [],
    "R": []
}

# Inisialisasi variabel
start = "S"
visited = {node: False for node in graph}  # Semua node
diinisialisasi sebagai belum dikunjungi
path = []

# Menjalankan DFS
path = dfs(graph, start, visited, path)
```

```
# Menampilkan hasil
print("Hasil DFS:", path)
```

**D. Output pemograman dengan Python**

```
Hasil DFS: ['S', 'P', 'R', 'Q']
```

Alhamdulilah, Terimakasih