DAFTAR PUSTAKA

Anggraeni, D. T. (2020). Peramalan Harga Saham Menggunakan Metode Autoregressive dan *Web Scrapping* pada Indeks Saham Lq45 dengan *Python*. *Rabit: Jurnal Teknologi Dan Sistem Informasi Univrab*, *5*(2), 138-145.

Buhaerah, B., Busrah, Z., & Sanjaya, H. (2022). Teori Graf dan Aplikasinya.

Dalem, I. B. G. W. A. (2018). Penerapan Algoritma A*(Star) Menggunakan *Graph* untuk Menghitung Jarak Terpendek. *Jurnal RESISTOR (Rekayasa Sistem Komputer)*, *1*(1), 41-47.

Daniel, F., & Taneo, P. N. (2019). *Teori Graf*. Deepublish.

Friaswanto, M., Lisangan, E. A., & Sumarta, S. C. (2021). The Simulation of Traffic Signal Preemption using GPS and Dijkstra Algorithm for Emergency Fire Handling at Makassar City Fire Service. *International Journal of Applied Sciences and Smart Technologies*, *3*(2), 185-202.

Harahap, M. K., & Khairina, N. (2017). Pencarian Jalur Terpendek dengan Algoritma Dijkstra. *Sinkron: Jurnal dan Penelitian Teknik Informatika*, *2*(2), 18-23.

Hasmawati. (2020). *Bahan Ajar Teori Graf.* Makassar: Universitas Hasanuddin.

Hasmawati. (2023). *Pengantar Teori dan Jenis-Jenis graf.* Unhas Press.

Jannah, M., Narwen, N., & Ginting, B. (2019). Mencari Minimum Spanning Tree dengan Konstren. Jurnal Matematika UNAND, 7(4), 22-26.

Luthfita, D., & Aripin, S. (2022). Implementasi Algoritma A* dalam Menentukan Tarif Minimum Berdasarkan Jarak Terpendek Rute Armada Taksi Bandara. *Journal of Informatics Management and Information Technology*, *2*(1), 43-47.

Mayadi, M., & Azhar, R. (2019). Perbandingan Perhitungan Manual dengan Algoritma A-*Star* dalam Pencarian Jalur Terpendek untuk Pengiriman Pesanan Dodol Khas Lombok. *Jurnal Informatika dan Rekayasa Elektronik*, *2*(2), 27-34.

Purnama, D. I. (2019). Analisis Komponen Utama pada Data Potensi Kecamatan di Kota Palu Sebelum Bencana Gempa Bumi dan Tsunami 28 September 2018. *Jurnal Matematika, Statistika dan Komputasi*, *16*(1), 25-32.

R. Munir. (2003). *Matematika Diskrit.* Bandung: Informatika.

Sarapang, H. T., Rogi, O. H., & Hanny, P. (2019). Analisis Kerentanan Bencana Tsunami di Kota Palu. *Spasial*, *6*(2), 432-439.

Yogaswara, G. (2013). *Pembangunan Aplikasi Permainan Jalan Pintas* (Doctoral dissertation, Universitas Komputer Indonesia).

Yulia, R., Sari, I. P., Syafi'i, M., & Hasibuan, L. H. (2022). Rute Evakuasi Tsunami Menggunakan Algoritma *Floyd Warshall* (Studi Kasus di Lubuk Buaya, Padang). UNEJ e-Proceeding, 383-391.

Lampiran 1: *Syntax* dan *Output* Jalur Optimal dari Titik *A* ke Titik *Y*

Syntax

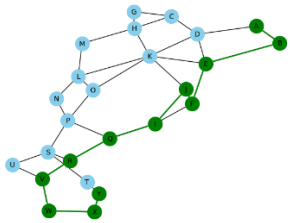
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242,
'O': 224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik A ke titik Y:
['A', 'B', 'E',
'F', 'J', 'I', 'Q',
'R', 'V', 'W', 'X',
'Y']
Total jarak: 1473
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan
titik awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] =
current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
```

```
dengan total cost
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi A* untuk mendapatkan
jalur terpendek
start_node = 'A'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak:
{shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 2: *Syntax* dan *Output* Jalur Optimal dari Titik *B* ke Titik *Y*

Syntax

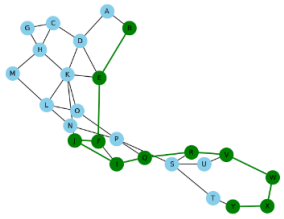
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242,
'O': 224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik B ke titik Y:
['B', 'E', 'F',
'J', 'I', 'Q', 'R',
'V', 'W', 'X', 'Y']
 Total jarak: 1378
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan
titik awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] =
current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
```

```
dengan total cost
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi B* untuk mendapatkan
jalur terpendek
start_node = 'B'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak:
{shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 3: *Syntax* dan *Output*Jalur Optimal dari Titik *C* ke Titik *Y*

Syntax

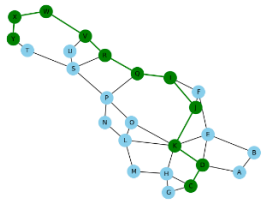
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242,
'O': 224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik C ke titik Y:
['C', 'D', 'K',
'J', 'I', 'Q', 'R',
'V', 'W', 'X', 'Y']
Total jarak: 1523
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan
titik awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] =
current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
```

```
dengan total cost
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi D* untuk mendapatkan
jalur terpendek
start_node = 'C'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak:
{shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 4: *Syntax* dan *Output* Jalur Optimal dari Titik *D* ke Titik *Y*

Syntax

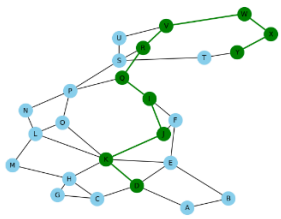
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242, 'O':
224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur terpendek
dari titik D ke
titik Y:
['D', 'K', 'J',
'I', 'Q', 'R', 'V',
'W', 'X', 'Y']
Total jarak: 1366
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan titik
awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] = current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
dengan total cost
```

```
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi D* untuk mendapatkan
jalur terpendek
start_node = 'D'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak: {shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 5: *Syntax* dan *Output* Jalur Optimal dari Titik *E* ke Titik *Y*

Syntax

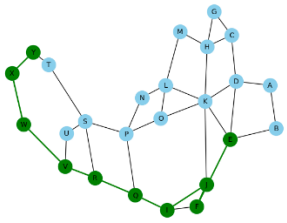
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242,
'O': 224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik E ke titik Y:
['E', 'F', 'J',
'I', 'Q', 'R', 'V',
'W', 'X', 'Y']
 Total jarak: 1324
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan
titik awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] =
current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
```

```
dengan total cost
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi E* untuk mendapatkan
jalur terpendek
start_node = 'E'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak:
{shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 6: *Syntax* dan *Output* Jalur Optimal dari Titik *F* ke Titik *Y*

Syntax

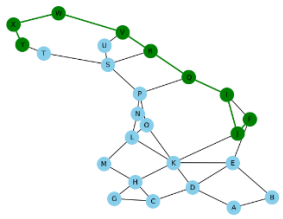
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242,
'O': 224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik F ke titik Y:
['F', 'J', 'I',
'Q', 'R', 'V', 'W',
'X', 'Y']
 Total jarak: 1257
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan
titik awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] =
current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
```

```
dengan total cost
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi F* untuk mendapatkan
jalur terpendek
start_node = 'F'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak:
{shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 7: *Syntax* dan *Output* Jalur Optimal dari Titik *G* ke Titik *Y*

Syntax

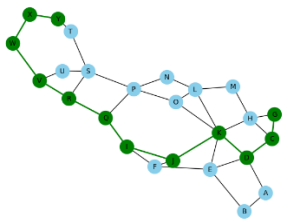
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242,
'O': 224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik G ke titik Y:
['G', 'C', 'D',
'K', 'J', 'I', 'Q',
'R', 'V', 'W', 'X',
'Y']
Total jarak: 1616
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan
titik awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] =
current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
```

```
dengan total cost
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi G* untuk mendapatkan
jalur terpendek
start_node = 'G'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak:
{shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 8: *Syntax* dan *Output* Jalur Optimal dari Titik *H* ke Titik *Y*

Syntax

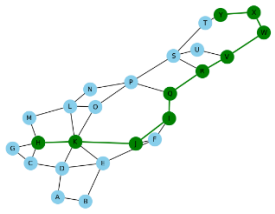
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242, 'O':
224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik H ke titik
Y:
['H', 'K', 'J',
'I', 'Q', 'R',
'V', 'W', 'X',
'Y']
 Total jarak: 1417
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan titik
awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] = current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
dengan total cost
```

```
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi H* untuk mendapatkan
jalur terpendek
start_node = 'H'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak: {shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 9: *Syntax* dan *Output* Jalur Optimal dari Titik *I* ke Titik *Y*

Syntax

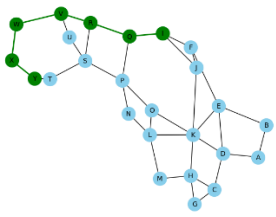
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242, 'O':
224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik I ke titik
Y:
['I', 'Q', 'R',
'V', 'W', 'X',
'Y']
 Total jarak: 1074
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan titik
awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] = current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
dengan total cost
```

```
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi I* untuk mendapatkan
jalur terpendek
start_node = 'I'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak: {shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 10: *Syntax* dan *Output* Jalur Optimal dari Titik *J* ke Titik *Y*

Syntax

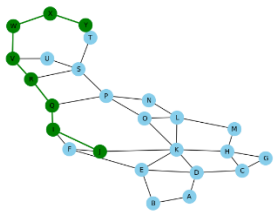
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242, 'O':
224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur terpendek
dari titik J ke
titik Y:
['J', 'I', 'Q',
'R', 'V', 'W',
'X', 'Y']
Total jarak: 1115
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan titik
awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] = current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
dengan total cost
```

```
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi J* untuk mendapatkan
jalur terpendek
start_node = 'J'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak: {shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 11: *Syntax* dan *Output* Jalur Optimal dari Titik *K* ke Titik *Y*

Syntax

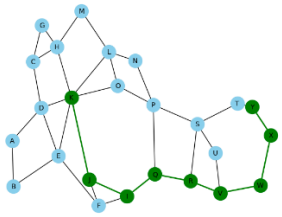
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242,
'O': 224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik K ke titik Y:
['K', 'J', 'I',
'Q', 'R', 'V', 'W',
'X', 'Y']
 Total jarak: 1174
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan
titik awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] =
current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
```

```
dengan total cost
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi K* untuk mendapatkan
jalur terpendek
start_node = 'K'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak:
{shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 12: *Syntax* dan *Output* Jalur Optimal dari Titik *L* ke Titik *Y*

Syntax

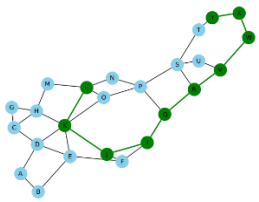
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46, 'K':
192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242, 'O':
224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik L ke titik
Y:
['L', 'K', 'J',
'I', 'Q', 'R',
'V', 'W', 'X',
'Y']
Total jarak: 1249
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W': 305,
'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan titik
awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil dari
priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] + cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] = current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
dengan total cost
                heapq.heappush(queue, (f,
neighbor))
```

```
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi L* untuk mendapatkan
jalur terpendek
start_node = 'L'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak: {shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path, node_color='green',
node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 13: *Syntax* dan *Output* Jalur Optimal dari Titik *M* ke Titik *Y*

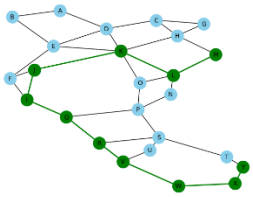
Syntax	*Output*

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

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46, 'K':
192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242, 'O':
224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

```
Jalur optimal
dari titik M ke
titik Y:
['M', 'L', 'K',
'J', 'I', 'Q',
'R', 'V', 'W',
'X', 'Y']
Total jarak: 1341
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W': 305,
'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan titik
awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil dari
priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] + cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] = current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
dengan total cost
                heapq.heappush(queue, (f,
neighbor))
```

```
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi M* untuk mendapatkan
jalur terpendek
start_node = 'M'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak: {shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path, node_color='green',
node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 14: *Syntax* dan *Output* Jalur Optimal dari Titik *N* ke Titik *Y*

Syntax

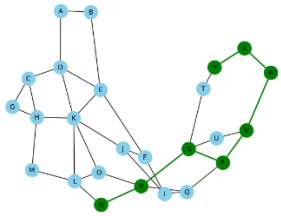
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242, 'O':
224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik N ke titik Y:
['N', 'P', 'S',
'R', 'V', 'W', 'X',
'Y']
Total jarak: 1133
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan titik
awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] = current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
dengan total cost
```

```
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi N* untuk mendapatkan
jalur terpendek
start_node = 'N'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak: {shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```

Lampiran 15: *Syntax* dan *Output* Jalur Optimal dari Titik *O* ke Titik *Y*

Syntax

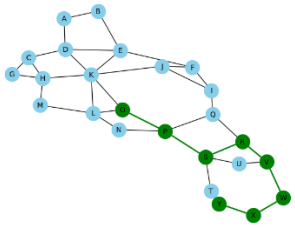
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Inisialisasi graf
graph = {
    'A': {'B': 95, 'D': 120},
    'B': {'A': 95, 'E': 54},
    'C': {'D': 157, 'G': 93, 'H': 136},
    'D': {'A': 120, 'C': 157, 'E': 46,
'K': 192},
    'E': {'B': 54, 'D': 46, 'F': 67, 'K':
156},
    'F': {'E': 67, 'I': 202, 'J': 142},
    'G': {'C': 93, 'H': 326},
    'H': {'C': 136, 'G': 326, 'K': 243,
'M': 107},
    'I': {'F': 202, 'J': 41, 'Q': 270},
    'J': {'F': 142, 'I': 41, 'K': 59},
    'K': {'D': 192, 'E': 156, 'H': 243,
'J': 59, 'L': 75, 'O': 101},
    'L': {'K': 75, 'M': 92, 'N': 242,
'O': 224},
    'M': {'H': 107, 'L': 92},
    'N': {'L': 242, 'P': 200},
    'O': {'K': 101, 'L': 224, 'P': 174},
    'P': {'N': 200, 'O': 174, 'Q': 138,
'S': 167},
    'Q': {'I': 270, 'P': 138, 'R': 155},
    'R': {'Q': 155, 'S': 117, 'V': 244},
    'S': {'P': 167, 'R': 117, 'T': 120,
'U': 253},
    'T': {'S': 120, 'Y': 668},
    'U': {'S': 253, 'V': 141},
    'V': {'R': 244, 'U': 141, 'W': 78},
    'W': {'V': 78, 'X': 126},
    'X': {'W': 126, 'Y': 201},
    'Y': {'T': 668, 'X': 201},
}

# Inisialisasi fungsi heuristik
heuristic = {
    'A': 1250, 'B': 1230, 'C': 1240, 'D':
```

Output

```
Jalur optimal dari
titik O ke titik Y:
['O', 'P', 'S', 'R',
'V', 'W', 'X', 'Y']
Total jarak: 1107
```



```
1200, 'E': 1200, 'F': 1190,
    'G': 1180, 'H': 1150, 'I': 1040, 'J':
1040, 'K': 1040, 'L': 1040,
    'M': 1060, 'N': 985, 'O': 947, 'P':
787, 'Q': 775, 'R': 622, 'S': 621,
    'T': 640, 'U': 371, 'V': 378, 'W':
305, 'X': 193, 'Y': 0,
}

def astar(graph, start, goal, heuristic):
# Inisialisasi jarak awal dan parent
    g = {start: 0}
    parent = {start: None}
# Inisialisasi priority queue dengan
titik awal dan total cost
    queue = [(heuristic[start], start)]
# Selama priority queue tidak kosong
    while queue:
# Pop titik dengan total cost terkecil
dari priority queue
        _, current = heapq.heappop(queue)
# Jika titik ini adalah titik tujuan,
pencarian selesai
        if current == goal:
            break
# Untuk setiap tetangga dari titik saat
ini
        for neighbor, cost in
graph[current].items():
# Hitung jarak baru dari titik awal ke
tetangga melalui titik saat ini
            tentative_g = g[current] +
cost
# Jika jarak baru lebih pendek dari jarak
sebelumnya
            if neighbor not in g or
tentative_g < g[neighbor]:
# Perbarui jarak dan parent dari tetangga
                g[neighbor] = tentative_g
                parent[neighbor] =
current
# Hitung total cost dari titik awal ke
tetangga melalui titik saat ini
                f = tentative_g +
heuristic[neighbor]
# Tambahkan tetangga ke priority queue
```

```
dengan total cost
                heapq.heappush(queue, (f,
neighbor))
# Rekonstruksi jalur dari titik awal ke
titik tujuan
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path, g[goal]
# Memanggil fungsi O* untuk mendapatkan
jalur terpendek
start_node = 'O'
goal_node = 'Y'
shortest_path, shortest_distance =
astar(graph, start_node, goal_node,
heuristic)
print(f"Jalur optimal dari titik
{start_node} ke titik {goal_node}:")
print(shortest_path)
print(f"Total jarak:
{shortest_distance}")
# Visualisasi graf
G = nx.Graph()
for node in graph:
    for neighbor, weight in
graph[node].items():
        G.add_edge(node, neighbor,
weight=weight)
pos = nx.spring_layout(G)  # Menentukan
posisi node dengan spring layout
nx.draw(G, pos, with_labels=True,
node_size=400, node_color='skyblue',
font_size=10)
nx.draw_networkx_nodes(G, pos,
nodelist=shortest_path,
node_color='green', node_size=400)
nx.draw_networkx_edges(G, pos,
edgelist=[(shortest_path[i],
shortest_path[i + 1]) for i in
range(len(shortest_path) - 1)],
edge_color='green', width=2)
# Menampilkan gambar graf
plt.show()
```