Lab: AI & ML Lab (23A31403)

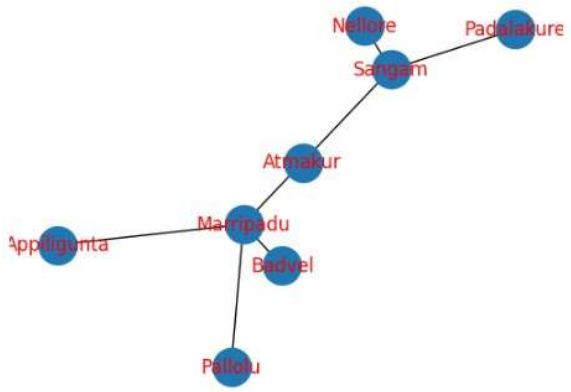
College: Andhra
Engineering College

Program:

```
import networkx as nx
import matplotlib.pyplot as plt
dir(nx)
print(nx.Graph.__doc__)

G = nx.Graph()
G.add_weighted_edges_from([ ('Atmakur', 'Marripadu', 43), ('Atmakur', 'Sangam', 17),
('Marripadu', 'Appiligunta', 10), ('Marripadu', 'Badvel', 51), ('Marripadu', 'Pallolu', 5),
('Sangam', 'Padalakure', 20), ('Sangam', 'Nellore', 60)], color="red")

G
plt.axis('off')
nx.draw_networkx(G,node_size=600,font_color='white')
```



3. Write a Program to Implement Breadth First Search using Python.

Program using Predefined Function Call

```
bfs_nodes = list(nx.bfs_tree(G, source='Atmakur'))
print("BFS Traversal:", bfs_nodes)
```

Output:

BFS Traversal: ['Atmakur', 'Marripadu', 'Sangam', 'Appiligunta', 'Badvel', 'Pallolu', 'Padalakure', 'Nellore']

Program using User Denied Function Call

```
visited = []

def dfs(visited, graph, node):
   if node not in visited:
      print(f"{node} has been visited")
      visited.append(node)
      for neighbor in graph[node]:
         visited = dfs(visited, graph, neighbor)

   return visited

dfs(visited, G, 'Atmakur')
```

Output:

```
Atmakur has been visited
Marripadu has been visited
Appiligunta has been visited
Badvel has been visited
Pallolu has been visited
Sangam has been visited
Padalakure has been visited
Nellore has been visited
['Atmakur',
 'Marripadu',
 'Appiligunta',
 'Badvel',
 'Pallolu',
 'Sangam',
 'Padalakure',
 'Nellore']
```

5. Write a Program to Implement Depth First Search using Python.

Pre Fun()

```
dfs_nodes = list(nx.dfs_tree(G, source='Atmakur'))
print("DFS Traversal:", dfs_nodes)
```

Output:

```
DFS Traversal: ['Atmakur', 'Marripadu', 'Appiligunta', 'Badvel', 'Pallolu', 'Sangam',
'Padalakure', 'Nellore']
```

User Fun() :

```
visited = []

def dfs(visited, graph, node):
    if node not in visited:
        print(f"{node} has been visited")
        visited.append(node)
        for neighbor in graph[node]:
            visited = dfs(visited, graph, neighbor)

    return visited

dfs(visited, G, 'Atmakur')
```

Output:

```
Atmakur has been visited
Marripadu has been visited
Appiligunta has been visited
Badvel has been visited
Pallolu has been visited
Sangam has been visited
Padalakure has been visited
Nellore has been visited
['Atmakur',
 'Marripadu',
 'Appiligunta',
 'Badvel',
 'Pallolu',
 'Sangam',
 'Padalakure',
 'Nellore']
```

4. Write a program to implement Best First Searching Algorithm

Program

```
from queue import PriorityQueue

def best_first_search(graph, start, goal, h):
    pq = PriorityQueue()
    pq.put((h.get(start, 0), start))
    visited = set()

    while not pq.empty():
        _, node = pq.get()
        print("Visiting:", node)
```

```
        if node == goal:
            return "Goal Reached"

        visited.add(node)

        for neighbor in graph.neighbors(node):
            if neighbor not in visited:
                pq.put((h.get(neighbor, 0), neighbor))

    return "Goal Not Found"
print("Best First Search Result:",best_first_search(G, 'Atmakur', 'Badvel', heuristic))
```

Output:

```
Visiting: Atmakur
Visiting: Marripadu
Visiting: Appiligunta
Visiting: Badvel
Best First Search Result: Goal Reached
```

6. Write a program to implement the Heuristic Search

Program

```
heuristic = {
    'Atmakur': 60,
    'Marripadu': 45,
    'Sangam': 50,
    'Appiligunta': 30,
    'Badvel': 25,
    'Pallolu': 20,
    'Padalakure': 15,
    'Nellore': 0
}

from queue import PriorityQueue

def heuristic_search(graph, start, goal, h):
    pq = PriorityQueue()
    pq.put((h[start], start))
    visited = set()

    while not pq.empty():
        _, node = pq.get()
        print("Visiting:", node)

        if node == goal:
            return "Goal reached"
```

```
        visited.add(node)

        for neighbor in graph.neighbors(node):
            if neighbor not in visited:
                pq.put((h[neighbor], neighbor))

    return "Goal not found"

print(heuristic_search(G, 'Atmakur', 'Nellore', heuristic))
```

Output:

```
Visiting: Atmakur
Visiting: Marripadu
Visiting: Pallolu
Visiting: Badvel
Visiting: Appiligunta
Visiting: Sangam
Visiting: Nellore
Goal reached
```

7. Write a python program to implement A* and AO* algorithm. (Ex: find the shortest path)

Program A*

```
astar_path = nx.astar_path(
    G,
    'Atmakur',
    'Nellore',
    heuristic=lambda n, goal=None: heuristic[n],
    weight='weight'
)

print("A* Path:", astar_path)
print("Total Cost:", nx.path_weight(G, astar_path, weight='weight'))
```

Output:

```
A* Path: ['Atmakur', 'Sangam', 'Nellore']
Total Cost: 77
```

Program AO*

```
import math

AO = nx.DiGraph()
```

```
# Add edges with weights from your towns data
edges = [
    ('Atmakur', 'Marripadu', 43),
    ('Atmakur', 'Sangam', 17),
    ('Marripadu', 'Appiligunta', 10),
    ('Marripadu', 'Badvel', 51),
    ('Marripadu', 'Pallolu', 5),
    ('Sangam', 'Padalakure', 20),
    ('Sangam', 'Nellore', 60)
]

for u, v, w in edges:
    AO.add_edge(u, v, weight=w)


heuristic_ao = {
    'Atmakur': 60,
    'Marripadu': 45,
    'Sangam': 50,
    'Appiligunta': 30,
    'Badvel': 25,
    'Pallolu': 20,
    'Padalakure': 15,
    'Nellore': 0
}

def ao_star(node):
    if node not in AO or AO.out_degree(node) == 0:
        return heuristic_ao[node]

    costs = []
    for child in AO.successors(node):
        cost = AO[node][child]['weight'] + ao_star(child)
        costs.append(cost)

    return min(costs)

print("AO* Minimum Cost from Nellore:", ao_star('Atmakur'))
```

Output

```
AO* Minimum Cost from Nellore: 52
```

Explanation

Graph edges (distances):
Atmakur → Marripadu (43)
Atmakur → Sangam (17)
Marripadu → Appiligunta (10)
Marripadu → Badvel (51)
Marripadu → Pallolu (5)
Sangam → Padalakure (20)
Sangam → Nellore (60)
Heuristic values (estimated distance to goal):
Atmakur: 60
Marripadu: 45
Sangam: 50
Appiligunta: 30
Badvel: 25
Pallolu: 20
Padalakure: 15
Nellore: 0

Step 1: AO* starts at Atmakur

- It looks at **successors**: 'Marripadu' and 'Sangam'
- Recursively calculates cost for each child

Step 2: Calculate cost via 'Marripadu'

- Cost = edge weight (Atmakur → Marripadu) + AO*(Marripadu)
- Edge weight = 43

AO*(Marripadu)

- Successors: 'Appiligunta', 'Badvel', 'Pallolu'

1. Marripadu → Appiligunta: 10 + heuristic(Appiligunta=30) = **10 + 30 = 40**
2. Marripadu → Badvel: 51 + heuristic(Badvel=25) = **51 + 25 = 76**
3. Marripadu → Pallolu: 5 + heuristic(Pallolu=20) = **5 + 20 = 25**

- AO* takes **minimum**: min(40, 76, 25) = 25
- Total cost via 'Marripadu' = 43 + 25 = **68**

Step 3: Calculate cost via 'Sangam'

- Cost = edge weight (Atmakur → Sangam) + AO*(Sangam)
- Edge weight = 17

AO*(Sangam)

- Successors: 'Padalakure', 'Nellore'

1. Sangam → Padalakure: 20 + heuristic(Padalakure=15) = **35**
2. Sangam → Nellore: 60 + heuristic(Nellore=0) = **60**

- AO* takes **minimum**: min(35, 60) = 35
- Total cost via 'Sangam' = 17 + 35 = **52**

Step 4: Choose minimum at root 'Atmakur'

- Costs:
    - Via Marripadu: 68
    - Via Sangam: 52

- AO* takes **minimum**: min(68, 52) = 52 ☑

☑ **Result**
AO* Minimum Cost from Atmakur: 52
Path implied (by AO*):
Atmakur → Sangam → Padalakure

- Total cost = 17 (Atmakur→Sangam) + 20 (Sangam→Padalakure) + 15 (heuristic of Padalakure) = 52

Note: Currently your code **returns only the cost**, not the path.
If you want, I can modify it to **also return the exact path**, so you can see ['Atmakur', 'Sangam', 'Padalakure'].