4. Advanced Search Techniques: AO Algorithm for Decision Making in Network Security

Use Case: Develop a system for detecting and responding to cybersecurity threats in a network.

Objective: Implement the AO* algorithm to construct a decision tree that identifies potential threats and suggests the best sequence of actions to mitigate them. The system should dynamically adjust based on new threat information.

Step 1: Import Libraries

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

- **Purpose**: Load essential libraries:
  - numpy for any mathematical operations.
  - networkx for creating and manipulating the decision tree as a graph.
  - matplotlib for visualizing the graph.

Step 2: Define the AO* Algorithm Class

```
class AOStar:
  def __init__(self, graph, start_node, heuristics):
    # Initialize the class with the given graph, start node, and heuristic values.
```

- **Purpose**: Create a class AOStar to encapsulate the AO* algorithm. This class handles:
  - **Expanding Nodes**: Decides which nodes (threat response actions) to consider next.
  - **Computing Costs**: Calculates the cost for both AND/OR paths based on heuristic values.
  - **Updating Heuristics**: Adjusts heuristic values dynamically to guide decision-making.
  - *AO Recursive Search**: The main search logic that recursively builds an optimal path by exploring nodes with the lowest cost.
- **Key Methods:**
  - **expand_node**: Expands nodes to access their children in the graph.
  - **compute_and_or_cost**: Calculates the cost of choosing between AND/OR nodes, helping the algorithm decide the best action.

- **update_heuristics**: Updates heuristic values at each step to ensure the algorithm considers the most relevant threats.
- **ao_star_search**: Recursively searches the graph to find the optimal solution path.

Step 3: Initialize Graph with Threats and Response Actions

```
graph = {
    'Start': ['Identify Threat', 'Evaluate Impact'],
    'Identify Threat': ['Analyze Traffic', 'Scan Logs'],
    'Evaluate Impact': ['Low Impact', 'High Impact'],
    ...
}
```

- **Purpose**: Define the decision tree (as a graph) with:
  - **Nodes**: Representing threats and potential response actions.
  - **Edges**: Showing relationships and potential decisions (from identifying threats to evaluating their impact and mitigating).
  - **Leaf Nodes**: Nodes without children, which represent terminal actions (e.g., Isolate System or Analyze Traffic).

This structure allows the AO* algorithm to traverse the network and evaluate the best response sequence.

Step 4: Define Initial Heuristic Values

```
heuristics = {
    'Start': 5,
    'Identify Threat': 3,
    'Evaluate Impact': 3,
    ...
}
```

- **Purpose**: Assign initial heuristic values to each node.
  - Heuristics represent the estimated cost or threat level, guiding the AO* algorithm.
  - Lower heuristic values indicate a more desirable action from a cost perspective, leading the algorithm to prefer lower-cost paths.

Step 5: Initialize and Run the AO* Algorithm

```
ao_star = AOStar(graph=graph, start_node='Start', heuristics=heuristics)
optimal_cost = ao_star.ao_star_search('Start')
```

- **Purpose**: Instantiate the AO* algorithm with the graph and heuristics and start the search.
  - **Result**: The AO* algorithm computes the optimal solution path, updating solution_graph to contain nodes in the best path from Start.
  - The final heuristic values reflect the updated costs for nodes based on the selected optimal path.

Step 6: Display Results

```
print("Optimal Solution Path:")
print(ao_star.solution_graph)
print("Heuristic Values after AO* Search:")
print(ao_star.heuristics)
print("Optimal Cost from Start:", optimal_cost)
```

- **Purpose**: Print out the results.
  - **Optimal Solution Path**: Displays the nodes included in the optimal path identified by the AO* algorithm.
  - **Updated Heuristics**: Shows the final heuristic values, representing updated costs after applying AO*.
  - **Optimal Cost**: The total minimum cost to mitigate the threat, starting from Start.

Step 7: Visualize the Decision Tree with Optimal Path Highlighted

```
def visualize_graph(graph, solution_graph, heuristics):
    ...
visualize_graph(graph, ao_star.solution_graph, ao_star.heuristics)
```

Visualization Steps:

1. **Define the Directed Graph**: Use nx.DiGraph() to set up a directed graph, representing the decision tree.
2. **Add Edges**: Populate G with edges based on parent-child relationships in the graph.
3. **Define Node Colors**:
   - Nodes on the optimal solution path are colored green.

- Non-solution nodes are colored blue.

4. **Add Labels**:
   - Labels include each node’s name and heuristic value, helping to interpret costs visually.
5. **Configure Layout and Display**:
   - A spring_layout spreads nodes evenly, making the decision tree readable.
   - Nodes, edges, and labels are drawn with matplotlib, and the visualization is displayed using plt.show().

In [1]:
```python
# Install and import necessary libraries
import numpy as np
import pandas as pd
import networkx as nx  # For graph representation of the decision tree
import matplotlib.pyplot as plt
```

In [2]:
```python
# Define the AO* Algorithm class
class AOStar:
    def __init__(self, graph, start_node, heuristics):
        self.graph = graph
        self.start_node = start_node
        self.heuristics = heuristics  # Heuristic values for each node
        self.solution_graph = {}  # Store optimal solution path

    def expand_node(self, node):
        """Expand a node by considering all children (AND/OR nodes)"""
        if node not in self.graph:
            return None
        children = self.graph[node]
        return children

    def compute_and_or_cost(self, node, children):
        """Calculate cost for AND-OR nodes based on children nodes' cost
s"""
        if children is None:
            return float('inf')

        and_cost = sum(self.heuristics[child] for child in children)
        or_cost = min(self.heuristics[child] for child in children)

        # Return the lower of AND/OR costs to select optimal path
        return min(and_cost, or_cost)

    def update_heuristics(self, node):
        """Update heuristic values based on computed AND-OR costs"""
        children = self.expand_node(node)
        if children is not None:
            self.heuristics[node] = self.compute_and_or_cost(node, childre
n)

    def ao_star_search(self, node):
        """Main AO* recursive search algorithm to find optimal solution pat
h"""
        self.update_heuristics(node)

        # If leaf node (no children), return its heuristic as cost
        if node not in self.graph or not self.graph[node]:
            return self.heuristics[node]

        # Explore each child node and recursively apply AO* search
        for child in self.expand_node(node):
            self.ao_star_search(child)

        # Update solution graph for optimal path tracking
        self.solution_graph[node] = self.graph[node]
        return self.heuristics[node]
```

In [3]:
```python
# Initialize Graph with Threats and Response Actions
graph = {
    'Start': ['Identify Threat', 'Evaluate Impact'],
    'Identify Threat': ['Analyze Traffic', 'Scan Logs'],
    'Evaluate Impact': ['Low Impact', 'High Impact'],
    'Analyze Traffic': None,  # Leaf node, mitigation ends here
    'Scan Logs': None,  # Leaf node, mitigation ends here
    'Low Impact': None,  # Leaf node, minimal action needed
    'High Impact': ['Mitigate Attack', 'Isolate System'],
    'Mitigate Attack': None,
    'Isolate System': None,
}
```

In [4]:
```python
# Define Heuristic Values for Each Node (Assume some initial values)
heuristics = {
    'Start': 5,
    'Identify Threat': 3,
    'Evaluate Impact': 3,
    'Analyze Traffic': 2,
    'Scan Logs': 2,
    'Low Impact': 1,
    'High Impact': 5,
    'Mitigate Attack': 2,
    'Isolate System': 2,
}
```

In [5]:
```python
# Initialize AO* Algorithm and Run Search
ao_star = AOStar(graph=graph, start_node='Start', heuristics=heuristics)
optimal_cost = ao_star.ao_star_search('Start')

# Print Optimal Solution Path and Heuristic Values
print("Optimal Solution Path:")
print(ao_star.solution_graph)
print("Heuristic Values after AO* Search:")
print(ao_star.heuristics)
print("Optimal Cost from Start:", optimal_cost)
```

```
Optimal Solution Path:
{'Identify Threat': ['Analyze Traffic', 'Scan Logs'], 'High Impact': ['Mit
igate Attack', 'Isolate System'], 'Evaluate Impact': ['Low Impact', 'High
Impact'], 'Start': ['Identify Threat', 'Evaluate Impact']}
Heuristic Values after AO* Search:
{'Start': 3, 'Identify Threat': 2, 'Evaluate Impact': 1, 'Analyze Traffi
c': 2, 'Scan Logs': 2, 'Low Impact': 1, 'High Impact': 2, 'Mitigate Attac
k': 2, 'Isolate System': 2}
Optimal Cost from Start: 3
```

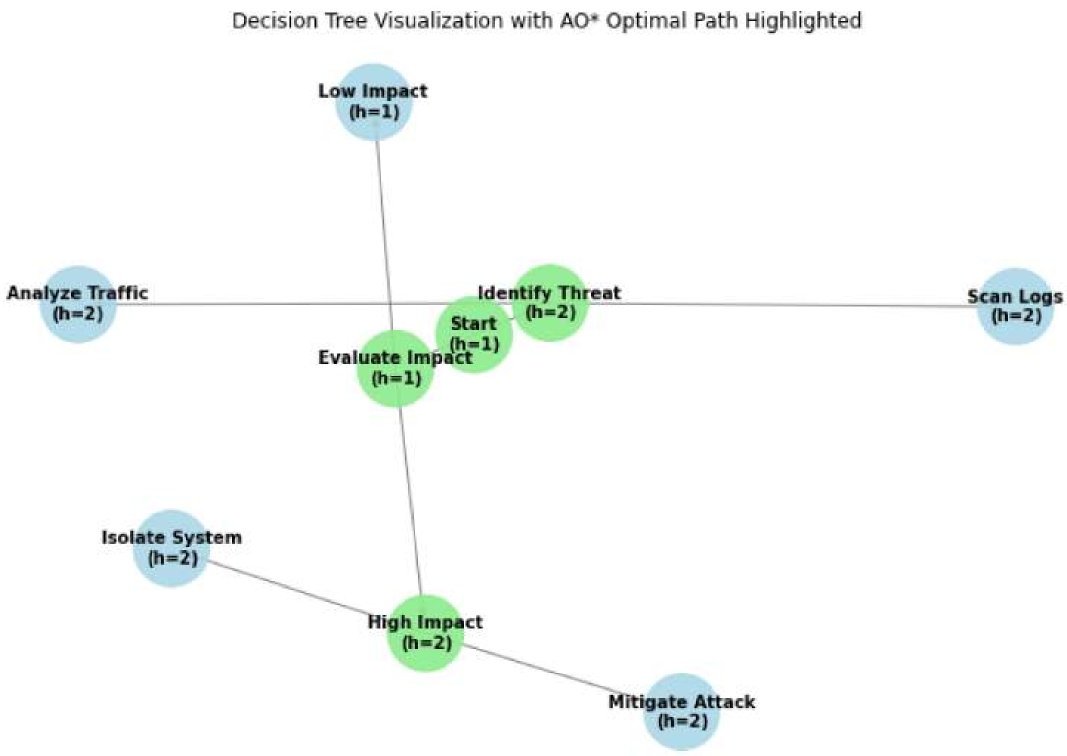
In [6]:
```python
# Initialize AO* Algorithm and Run Search
ao_star = AOStar(graph=graph, start_node='Start', heuristics=heuristics)
optimal_cost = ao_star.ao_star_search('Start')

# Print Optimal Solution Path and Heuristic Values
print("Optimal Solution Path:")
print(ao_star.solution_graph)
print("Heuristic Values after AO* Search:")
print(ao_star.heuristics)
print("Optimal Cost from Start:", optimal_cost)
```

```
Optimal Solution Path:
{'Identify Threat': ['Analyze Traffic', 'Scan Logs'], 'High Impact': ['Mit
igate Attack', 'Isolate System'], 'Evaluate Impact': ['Low Impact', 'High
Impact'], 'Start': ['Identify Threat', 'Evaluate Impact']}
Heuristic Values after AO* Search:
{'Start': 1, 'Identify Threat': 2, 'Evaluate Impact': 1, 'Analyze Traffi
c': 2, 'Scan Logs': 2, 'Low Impact': 1, 'High Impact': 2, 'Mitigate Attac
k': 2, 'Isolate System': 2}
Optimal Cost from Start: 1
```

In [7]:

```python
# Enhanced Visualization of the Decision Tree with Optimal Path Highlighted
def visualize_graph(graph, solution_graph, heuristics):
    # Create a directed graph for visualization
    G = nx.DiGraph()

    # Add edges to the graph based on the structure of the decision tree
    for node, children in graph.items():
        if children:
            for child in children:
                G.add_edge(node, child)

    # Define node colors based on whether they're in the optimal path
    node_colors = ['lightgreen' if node in solution_graph else 'lightblue' for node in G.nodes()]

    # Node labels with heuristic values for clearer understanding
    node_labels = {node: f"{node}\n(h={heuristics[node]})" for node in G.nodes()}

    # Define layout for visualization
    pos = nx.spring_layout(G, seed=42)  # Set seed for consistent layout

    # Draw the graph
    plt.figure(figsize=(12, 8))
    nx.draw_networkx_nodes(G, pos, node_color=node_colors, node_size=2000, alpha=0.9)
    nx.draw_networkx_edges(G, pos, edgelist=G.edges(), edge_color="gray", arrows=True)
    nx.draw_networkx_labels(G, pos, labels=node_labels, font_size=10, font_weight="bold")

    # Draw edge labels
    edge_labels = {(node, child): "" for node in solution_graph for child in graph.get(node, [])}
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=8)

    plt.title("Decision Tree Visualization with AO* Optimal Path Highlighted")
    plt.axis("off")  # Hide axis for a cleaner look
    plt.show()
```

In [8]:
```
# Run visualization function
visualize_graph(graph, ao_star.solution_graph, ao_star.heuristics)
```



In [ ]: