Experiment No.1
Identification of the problem and Determination of its PEAS Descriptor.
Name: Satyam Yogendra Yadav
Roll No.: TE/1-61
Date of Performance: 03/01/2025
Date of Submission: 17/01/2025

Aim: Identification of the problem and Determination of its PEAS Descriptor.

Objective: To analyze the Performance Measure, Environment, Actuators, Sensors (PEAS) for given problem before building an intelligent agent.

Theory:

The goal of AI is to build intelligent system which can think and act rationally. For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has. Rationality is relative to a performance measure.

Designer of rational agent can judge rationality based on:

- The performance measure that defines the criterion of success.
- The agent prior knowledge of the environment.
- The possible actions that the agent can perform.
- The agent's percept sequence to date.

When we define a rational agent, we group these properties under PEAS, the problem specification for the task environment.

Performance Measure:

If the objective function to judge the performance of the agent, things we can evaluate an agent against to know how well it performs.

Environment:

It is the real environment where the agent need to deliberate actions. What the agent can perceive.

Actuators:

These are the tools, equipment or organs using which agent performs actions in the environment. This works as output of the agent. What an agent can use to act in its environment.

Sensors:

These are tools, organs using which agent captures the state of the environment. This works as input to the agent. What an agent can use to perceive its environment

PEAS Descriptors Examples/Problems

1. PEAS descriptor for Automated Car Driver: Performance Measure:

- **Safety**: Automated system should be able to drive the car safely without dashing anywhere.
- **Optimum speed:** Automated system should be able to maintain the optimal speed depending upon the surroundings.
- **Comfortable journey:** Automated system should be able to give a comfortable journey to the end user.

Environment:

- **Roads:** Automated car driver should be able to drive on any kind of a road ranging from city roads to highway.
- **Traffic conditions:** You will find different sort of traffic conditions for different type of roads.

Actuators:

- **Steering wheel:** used to direct car in desired directions.
- **Accelerator, gear:** To increase or decrease speed of the car.

Sensors:

- To take input from environment in car driving example cameras, sonar system etc.

2. PEAS descriptor for playing soccer.

Performance Measure: scoring goals, defending, speed **Environment:** playground, teammates, opponents, ball **Actuators:** body, dribbling, tackling, passing ball, shooting **Sensors:** camera, ball sensor, location sensor, other players locator

3. PEAS descriptor for Exploring the subsurface oceans of Titan.

Performance Measure: safety, images quality, video quality

Environment: ocean, water

Actuators: mobile diver, steering, break, accelerator

Sensors: video, accelerometers, depth sensor, GPS

4. **PEAS descriptor for Shopping for used AI books on the Internet.**

   Performance Measure: price, quality, authors, book review

   Environment: web, vendors, shippers

   Actuators: fill in form, follow URL, display to user

   Sensors: HTML

5. **PEAS descriptor for playing a tennis match.**

   Performance Measure: winning

   Environment: playground, acquet, ball, opponent

   Actuators: ball, raquet, joint arm

   Sensors: ball locator, camera, racquet sensor, opponent locator

6. **PEAS descriptor for practicing tennis against a wall.**

   Performance Measure: hit speed, hit accuracy **Environment:** playground, racquet, ball, wall
   Actuators: ball, racquet, joint arm
   Sensors: ball locator, camera, racquet sensor

Question:

Provide the PEAS descriptor for the following Agents:

1. Chess Playing Agent
2. Stock Market Trading Agent
3. Online Shopping Recommender System
4. Smart Traffic Light System
5. Library Management Agent

Solution:

Here are the PEAS (Performance measure, Environment, Actuators, and Sensors) descriptors for each of the given agents:

1. Chess Playing Agent

- **Performance Measure**: Winning the game, controlling the board, avoiding checkmate, maximizing piece value, controlling key squares.
- **Environment**: Chessboard (8x8 grid), chess pieces (king, queen, rook, knight, bishop, pawn), game rules.
- **Actuators**: Move chess pieces on the board, display moves on a graphical interface.
- **Sensors**: Board state (positions of all pieces), move history, opponent's moves.

2. Stock Market Trading Agent

- **Performance Measure**: Maximizing return on investment (ROI), minimizing risk, outperforming market indices.
- **Environment**: Stock market data, stocks, indices, price fluctuations, news, trading rules.
- **Actuators**: Buying, selling, or short-selling stocks, placing orders on the trading platform.
- **Sensors**: Real-time stock price data, news feeds, financial indicators, historical data, market trends.

3. Online Shopping Recommender System

- **Performance Measure**: Accuracy of recommendations, user engagement, sales conversion rate, user satisfaction.
- **Environment**: User profiles, item catalog, user activity (browsing history, clicks, purchases).
- **Actuators**: Displaying recommended items, sending notifications, showing personalized suggestions.
- **Sensors**: User behavior data (clicks, search history, purchase history), product attributes, customer feedback.

4. Smart Traffic Light System

- **Performance Measure**: Minimizing congestion, optimizing traffic flow, reducing waiting times, improving safety.
- **Environment**: Traffic intersections, vehicles, road conditions, pedestrians, traffic rules.
- **Actuators**: Changing traffic light signals (red, yellow, green), adjusting timing for different lanes.
- **Sensors**: Vehicle count on different lanes, pedestrian crossing signals, traffic flow data, road occupancy sensors.

5. Library Management Agent

- **Performance Measure**: Efficient management of book inventory, minimizing late returns, user satisfaction, accuracy of book search.
- **Environment**: Library inventory, books, users (borrowers), library staff.
- **Actuators**: Issuing and returning books, updating the inventory, notifying users, managing reservations.
- **Sensors**: Book barcodes, user IDs, due dates, overdue books, availability status.

Conclusion:

In conclusion, the PEAS framework is an essential tool for designing intelligent agents, as it helps to define the problem environment clearly and systematically. By identifying the performance measure, environment, actuators, and sensors, we can effectively analyze and design agents for various tasks. This approach ensures that the agent's behavior is rational and aligns with the desired objectives. The examples of Chess Playing Agent, Stock Market Trading Agent, and others demonstrate how PEAS can be applied to a variety of domains. Ultimately, understanding the PEAS descriptor helps in building more effective and efficient intelligent systems.

Experiment No.2
Implement packet routing in a computer network using DFS and BFS.
Name: Satyam Yogendra Yadav
Roll No.: TE/1-61
Date of Performance: 17/01/2025
Date of Submission: 24/01/2025

Aim: Study and Implementation of Depth first search for problem solving.

Objective: To study the uninformed searching techniques and its implementation for problem solving.

Theory:

Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

- A search problem consists of:
    - **A State Space.** Set of all possible states where you can be.
    - **A Start State.** The state from where the search begins.
    - **A Goal Test.** A function that looks at the current state returns whether or not it is the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.
- This plan is achieved through search algorithms.

Depth First Search: DFS is an uninformed search method. It is also called blind search. Uninformed search strategies use only the information available in the problem definition. A search strategy is defined by picking the order of node expansion. Depth First Search (DFS) searches deeper into the problem space. It is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

The basic idea is as follows:

1. Pick a starting node and push all its adjacent nodes into a stack.
2. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
3. Repeat this process until the stack is empty.

However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Algorithm:

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

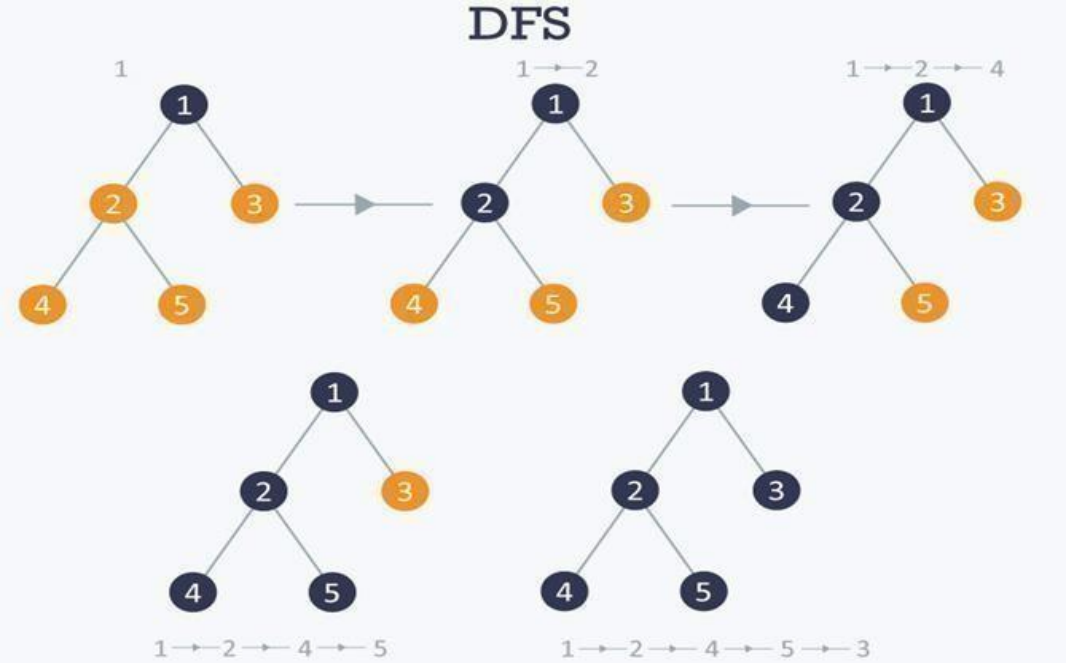
The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Pseudocode:

```
DFS-iterative (G, s):
        //Where G is graph and s is source
       vertex let S be stack
       S.push( s )  //Inserting s in stack mark s as
       visited.
       while (S is not empty):
             //Pop a vertex from stack to
           visit next v = S.top()
          S.pop()
        //Push all the neighbours of v in stack that are not
       visited for all neighbours w of v in Graph G:
                    if w is not
           visited: S.push( w )
           mark w as visited

DFS-recursive (G, s):
         mark s as visited
                       for all neighbours w of s
              in Graph G: if w is not visited:
                    DFS-recursive(G, w)
```

DFS Working: Example



Path: 1 □ 2□ 4□ 5□ 3

Searching Strategies are evaluated along the following dimensions:

1. **Completeness:** does it always find a solution if one exists?
2. **Time complexity:** number of nodes generated
3. **Space complexity:** maximum number of nodes in memory
4. **Optimality:** does it always find a least-cost solution?

Properties of depth-first search:

1. Complete: - No: fails in infinite-depth spaces, spaces with loops.
2. Time Complexity: O(bm)
3. Space Complexity: O(bm), i.e., linear space!
4. Optimal: No

Advantages of Depth-First Search:

1. Memory requirement is only linear with respect to the search graph.
2. The time complexity of a depth-first Search to depth d is O(b^d)
3. If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

Disadvantages of Depth-First Search:

1. There is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree.
2. Depth-First Search is not guaranteed to find the solution.
3. No guarantee to find a optimum solution, if more than one solution exists.

Applications:

How to find connected components using DFS?

A graph is said to be disconnected if it is not connected, i.e. if two nodes exist in the graph such that there is no edge in between those nodes. In an undirected graph, a connected component is a set of vertices in a graph that are linked to each other by paths.

Consider the example given in the diagram. Graph G is a disconnected graph and has the following 3 connected components.

- First connected component is 1 □ 2 □ 3 as they are linked to each other
- Second connected component 4 □ 5
- Third connected component is vertex 6

Breadth First Search: BFS is a uninformed search method. It is also called blind search. Uninformed search strategies use only the information available in the problem definition. A search strategy is defined by picking the order of node expansion. It expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. It is very easily implemented by maintaining a queue of nodes. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

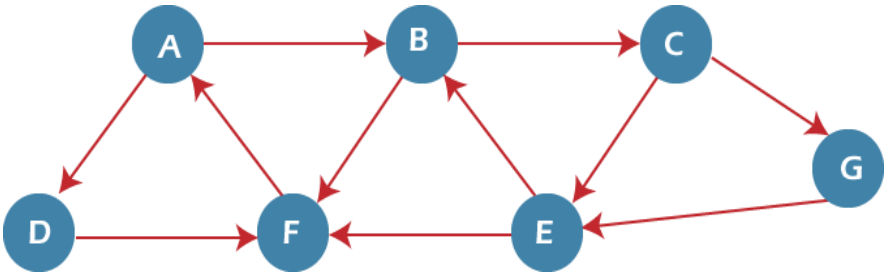
As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Question 1:

Apply DFS algorithm on given graph to find path from node A to node G.

Show and explain the status of all the nodes that are to be processed in stack STK and status of all the nodes that are already processed.



Solution:

DFS Traversal Table:

Step	Action	Open List (Stack)	Closed List (Visited)	Processed Node	Adjacent Nodes Pushed to Stack
1	Start at Node A	[A]	{A}	A	B, F, D (pushed in reverse)
2	Pop A and process it	[D, F, B]	{A}	A	B, F, D
3	Pop B and process it	[D, F, F, E, C]	{A, B}	B	C, E, F
4	Pop C and process it	[D, F, F, E, G]	{A, B, C}	C	G
5	Pop G and stop (target node found)	[D, F, F, E]	**{A, B, C, G}**	G	-

1. **Start at Node A**
    - Push **A** onto the stack (open list): open = [A]
    - Add **A** to the closed list: closed = {A}
2. **Pop A from the stack**
    - Process **A**, push its adjacent nodes (**B, F, D**) onto the stack.
    - Reverse the order and push onto the stack: open = [D, F, B]
    - **Closed list:** {A} (A is already processed)
3. **Pop B from the stack**
    - Process **B**, push its adjacent nodes (**C, E, F**) onto the stack.
    - open = [D, F, F, E, C]
    - Add **B** to the closed list: closed = {A, B}
4. **Pop C from the stack**
    - Process **C**, push its adjacent node (**G**) onto the stack.
    - open = [D, F, F, E, G]
    - Add **C** to the closed list: closed = {A, B, C}
5. **Pop G from the stack**
    - **G is the target node, so we stop here.**
    - Add **G** to the closed list: closed = {A, B, C, G}
    - **Path found:** A → B → C → G

Final Status:

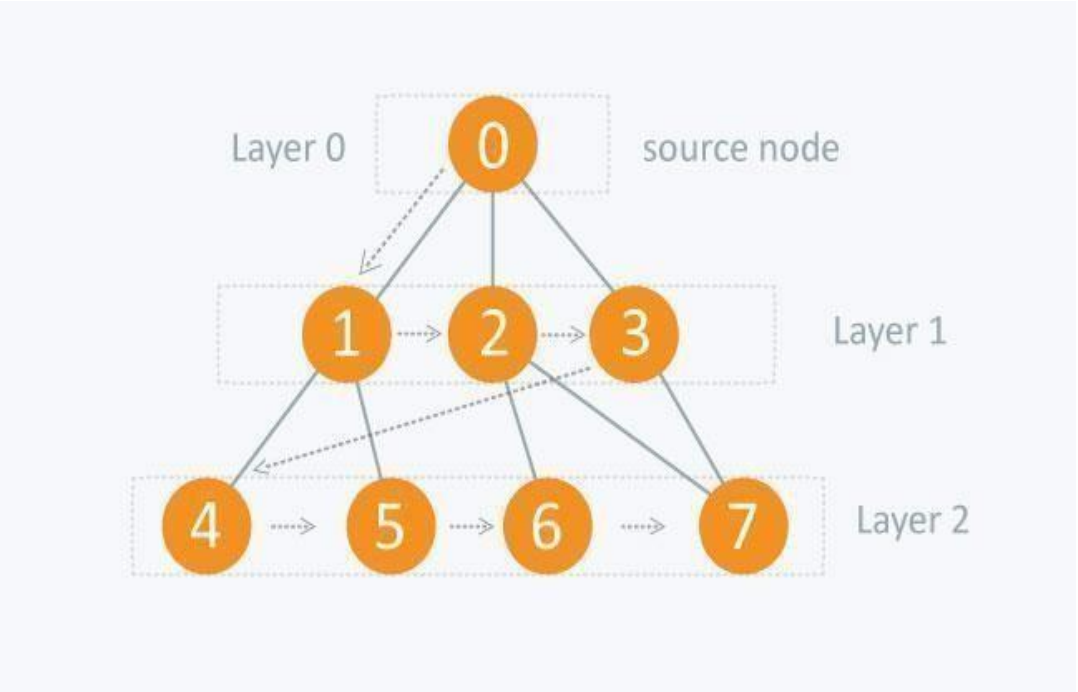
- **Path Found:** A → B → C → G
- **Open list (Remaining Nodes in Stack):** [D, F, F, E] (we stop at G)
- **Closed list (Processed Nodes):** {A, B, C, G}

BFS Algorithm:

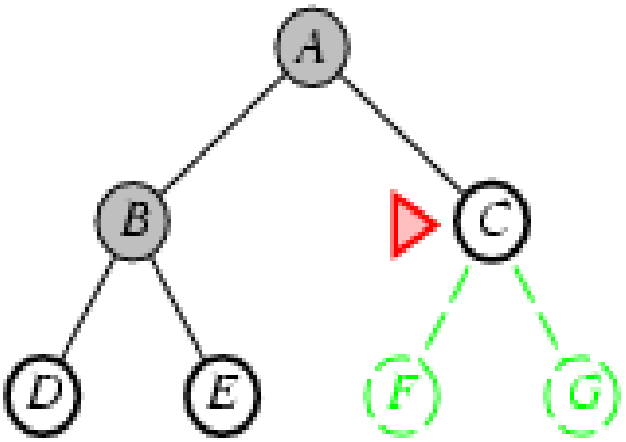
Pseudocode:

```
BFS (G, s)             //Where G is the graph and s is the source
      node let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices
are marked.

      mark s as visited.
      while ( Q is not empty)
           //Removing that vertex from queue,whose neighbour will be visited
now
           v = Q.dequeue( )
          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                        Q.enqueue( w )              //Stores w in Q
to further visit its neighbour
                        mark w as visited.
```

Working of BFS:



Example: Initial Node: A Goal Node: C



Searching Strategies are evaluated along the following dimensions:

1. Completeness: does it always find a solution if one exists?
2. Time complexity: number of nodes generated
3. Space complexity: maximum number of nodes in memory
4. Optimality: does it always find a least-cost solution?

Properties of Breadth-first search:

1. **Complete**: - Yes: if b is finite.
2. **Time Complexity**: O(b^d+1)
3. **Space Complexity**: O(b^d+1)
4. **Optimal**: Yes

Advantages of Breadth-First Search:

1. Breadth first search will never get trapped exploring the useless path forever.
2. If there is a solution, BFS will definitely find it out.
3. If there is more than one solution then BFS can find the minimal one that requires less number of steps.

Disadvantages of Breadth-First Search:

1. The main drawback of Breadth first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of BFS is O(bd).
2. If the solution is farther away from the root, breath first search will consume lot of time.

Applications:

How to determine the level of each node in the given tree?

As you know in BFS, you traverse level wise. You can also use BFS to determine the level of each node.

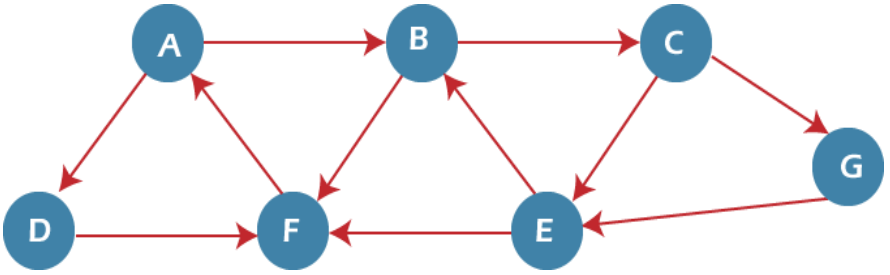
Question 2:

Apply BFS algorithm on given graph to find path from node A to node E.

Show and explain the status of both the queues Q1 and Q2.

Q1 holds all the nodes that are to be processed

Q2 holds all the nodes that are processed and deleted from Q1.



Solution:

Step	Action	Q1 (Queue to Process)	Q2 (Processed Nodes)	Path Found
Start	Initialize	Q1 = [A]	Q2 = []	-
Step 1	Process Node A (Dequeue A, Enqueue B, D)	Q1 = [B, D]	Q2 = [A]	-
Step 2	Process Node B (Dequeue B, Enqueue C, F)	Q1 = [D, C, F]	Q2 = [A, B]	A → B → C
Step 3	Process Node C (Dequeue C, Enqueue E, G)	Q1 = [D, E, G, F]	Q2 = [A, B, C]	A → B → C
End	Path Found (E found at C)	Q1 = [D, E, G, F]	Q2 = [A, B, C]	**A → B → C**

Step-by-Step BFS Traversal:

- **Q1**: Queue that holds nodes yet to be processed.
- **Q2**: Queue that holds nodes that have already been processed.

Initialization:

1. Start at Node A.
    - Enqueue A into Q1.
    - **Q1 = [A]**
    - **Q2 = []** (empty at the start)

Step 1: Process Node A

1. Dequeue A from Q1 and enqueue its neighbors (B, D).
    - **Q1 = [B, D]**
    - **Q2 = [A]** (A is processed)

Step 2: Process Node B

1. Dequeue B from Q1 and enqueue its neighbors (C, F).
2. E is found here, so we stop.
    - **Q1 = [D, C, F]**
    - **Q2 = [A, B]** (B is processed)
    - **Path Found**: A → B → C

Step 3: Process Node C

1. Dequeue C from Q1 and enqueue its neighbors (E, G).
2. E is found here, so we stop.
    - **Q1 = [D, E, G, F]**
    - **Q2 = [A, B, C]** (C is processed)

Final Status:

- **Path from A to E**: A → B → C
- **Nodes left in Q1 (not processed further)**: [F, D, G]
    - These nodes are not processed further since E has been reached.
- **Nodes in Q2 (processed)**: [A, B, C]

BFS Implementation:

Code:

```
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque



# Custom BFS implementation
def bfs(G, start, target):
    # Initialize queue for BFS, visited nodes set, and parent dictionary
    queue = deque([start])
    visited = set([start])
    parent = {start: None}

    while queue:
        node = queue.popleft()

        # If we reached the target node, backtrack to form the path
        if node == target:
            path = []
            while node is not None:
                path.append(node)
                node = parent[node]
            return path[::-1] # Return reversed path from start to target
```

```
        # Explore neighbors
        for neighbor in G.neighbors(node):
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = node
                queue.append(neighbor)

    return [] # Return empty if no path is found



# Create a new graph
G = nx.Graph()

# Add nodes with IP addresses
ip_addresses = {
    1: "192.168.1.1",
    2: "192.168.1.2",
    3: "192.168.1.3",
    4: "192.168.1.4",
    5: "192.168.1.5",
    6: "192.168.1.6",
}

# Adding nodes to the graph
for node, ip in ip_addresses.items():
```

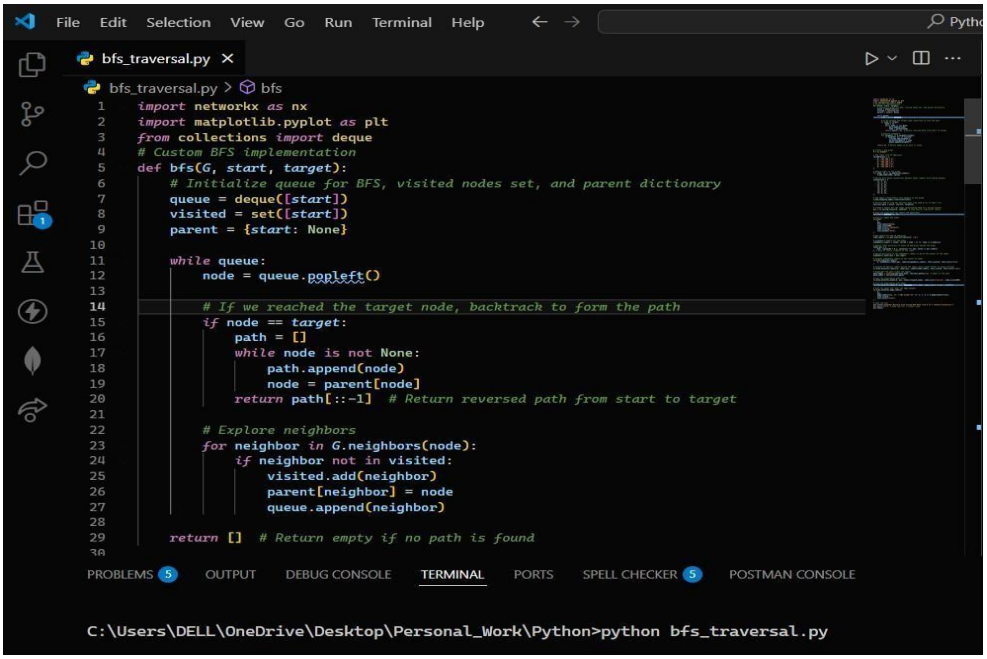
```
G.add_node(node, ip=ip)

# Define more dense connections between nodes (edges) with uneven weights
connections = [
    (1, 2, 5),
    (1, 3, 3),
    (1, 4, 1),
    (2, 5, 6),
    (3, 6, 7),
    (4, 5, 2),
    (4, 6, 4),
    (5, 6, 3),
]

# Add edges (connections) with weights to the graph
G.add_weighted_edges_from(connections)

# Perform BFS to find the shortest path from node D (4) to node C (3)
shortest_path = bfs(G, start=4, target=3)

# Create a layout for the nodes (positioning them in a spring layout)
pos = nx.spring_layout(G, seed=42) # seed ensures consistent layout

# Draw the graph with the labels (IP addresses)
plt.figure(figsize=(8, 6))
```

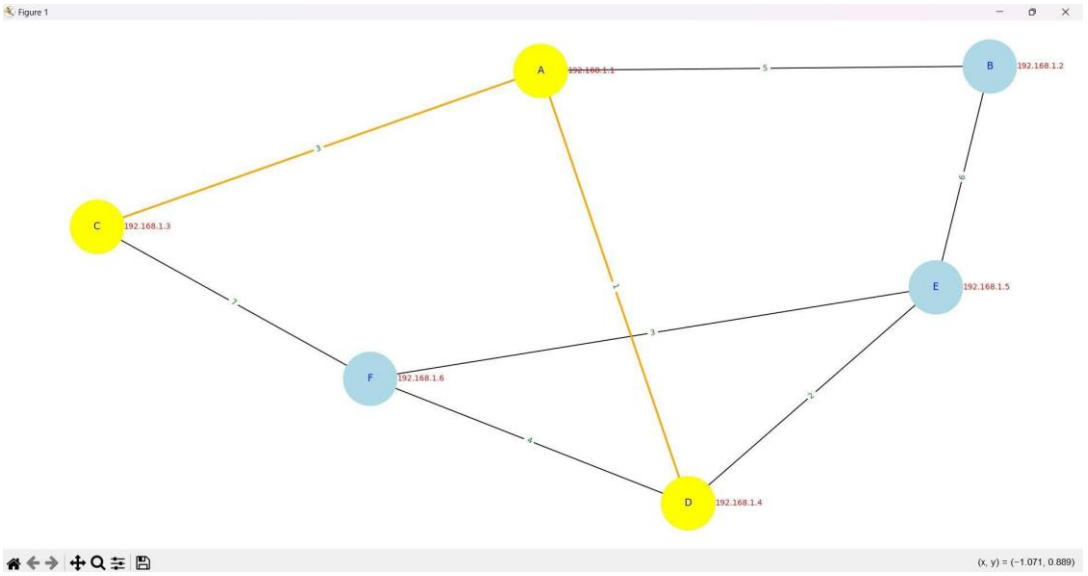
```
# Draw all edges and nodes
nx.draw(
    G,
    pos,
    with_labels=False,
    node_size=3000,
    node_color="lightblue",
    font_size=10,
    font_weight="bold",
)

# Add labels for the IP addresses
node_labels = nx.get_node_attributes(G, "ip")

# Alphabetic labels for the nodes
alphabetic_labels = {node: chr(65 + node - 1) for node in G.nodes()}

# Adjust label positions to place IP addresses beside the nodes
label_pos = {
    key: (value[0] + 0.1, value[1]) for key, value in pos.items()
} # Move IP labels slightly to the right

# Adjust positions of the alphabetic labels to be at the center of the nodes
alphabetic_label_pos = pos.copy()
```

```
# Display alphabetic labels at the center of nodes
nx.draw_networkx_labels(
    G, alphabetic_label_pos, labels=alphabetic_labels, font_size=10, font_color="blue"
)

# Display IP address labels beside the nodes (with slight offset to avoid overlap)
nx.draw_networkx_labels(G, label_pos, labels=node_labels, font_size=8, font_color="red")

# Highlight the path (nodes and edges)
path_edges = list(zip(shortest_path, shortest_path[1:])) # edges in the path
path_nodes = set(shortest_path)

# Draw the highlighted path nodes
nx.draw_networkx_nodes(G, pos, nodelist=path_nodes, node_color="yellow", node_size=3000)

# Draw the highlighted path edges
nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color="orange", width=2)
# Draw the edges and label the edge weights
nx.draw_networkx_edge_labels(
    G,
    pos,
    edge_labels={(u, v): f"{d['weight']}" for u, v, d in G.edges(data=True)},
    font_size=8,
    font_color="green",
)
```

```
# Show the plot
plt.title("Computer Network with Custom BFS Path from D to C (Uneven Distances)")
plt.axis("off") # Hide axes for a cleaner look
plt.show()
```

Output:





DFS Implementation:

Code:

```
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque


# Custom DFS implementation
def dfs(G, start, target):
    # Initialize stack for DFS, visited nodes set, and parent dictionary
    stack = [start]
    visited = set([start])
    parent = {start: None}

    while stack:
        node = stack.pop()

        # If we reached the target node, backtrack to form the path
        if node == target:
            path = []
            while node is not None:
                path.append(node)
                node = parent[node]
            return path[::-1] # Return reversed path from start to target

        # Explore neighbors in reverse order to simulate DFS
        for neighbor in reversed(list(G.neighbors(node))):
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = node
                stack.append(neighbor)
```

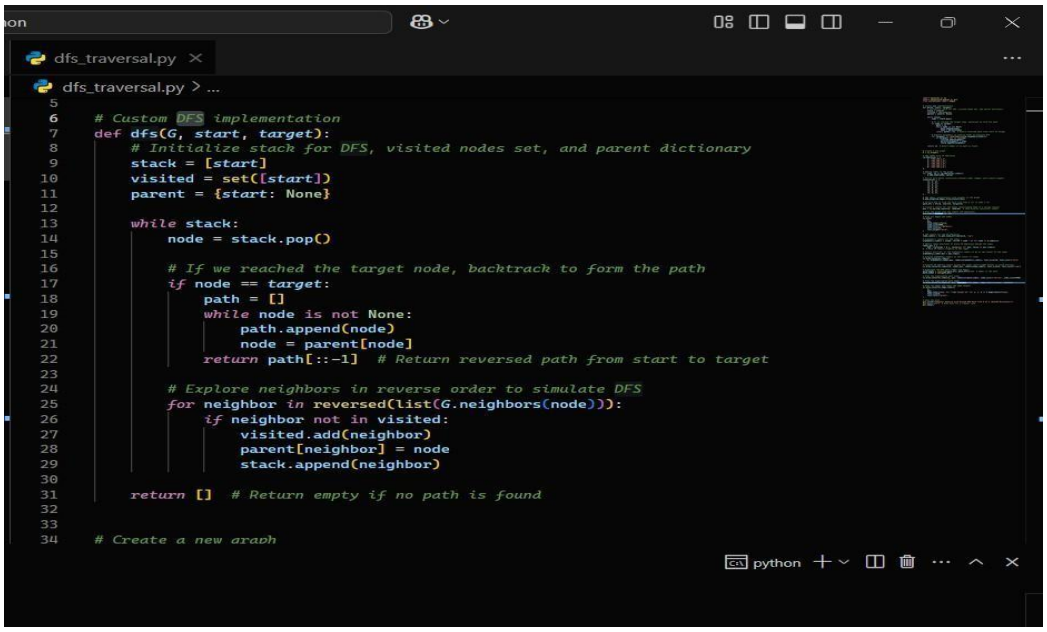
```
    return [] # Return empty if no path is found



# Create a new graph
G = nx.Graph()

# Add nodes with IP addresses
ip_addresses = {
    1: "192.168.1.1",
    2: "192.168.1.2",
    3: "192.168.1.3",
    4: "192.168.1.4",
    5: "192.168.1.5",

 6: "192.168.1.6",
}

# Adding nodes to the graph
for node, ip in ip_addresses.items():
    G.add_node(node, ip=ip)
# Define more dense connections between nodes (edges) with uneven weights
connections = [
    (1, 2, 5),
    (1, 3, 3),
    (1, 4, 1),
    (2, 5, 6),
    (3, 6, 7),
    (4, 5, 2),
    (4, 6, 4),
    (5, 6, 3),
]
```

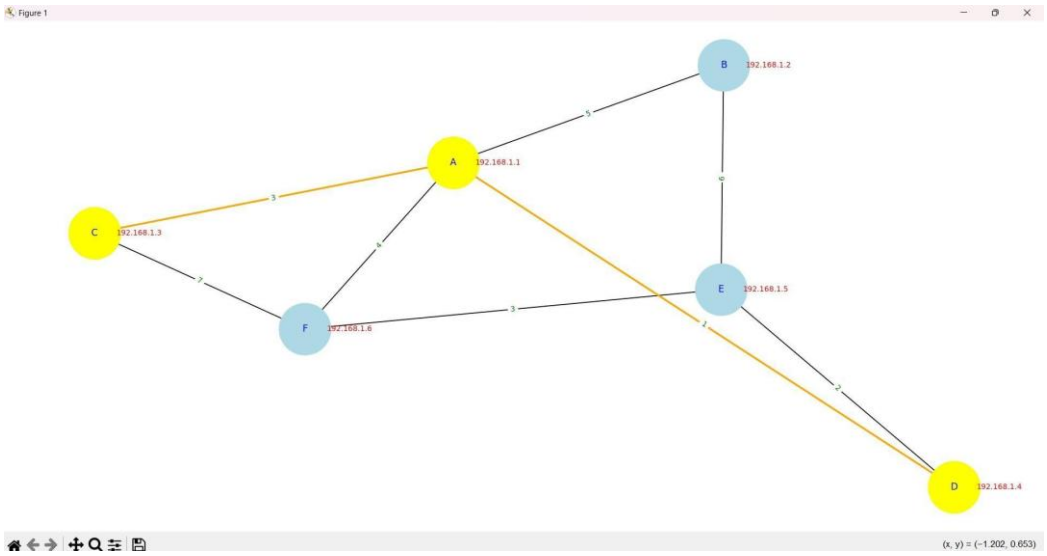
```
# Add edges (connections) with weights to the graph
G.add_weighted_edges_from(connections)

# Perform DFS to find the path from node D (4) to node C (3)
path_dfs = dfs(G, start=4, target=3)

# Create a layout for the nodes (positioning them in a spring layout)
pos = nx.spring_layout(G, seed=42) # seed ensures consistent layout

# Draw the graph with the labels (IP addresses)
plt.figure(figsize=(8, 6))

# Draw all edges and nodes
nx.draw(
    G,
    pos,
    with_labels=False,
    node_size=3000,
    node_color="lightblue",
    font_size=10,
    font_weight="bold",
)

# Add labels for the IP addresses
node_labels = nx.get_node_attributes(G, "ip")

# Alphabetic labels for the nodes
alphabetic_labels = {node: chr(65 + node - 1) for node in G.nodes()}



# Adjust label positions to place IP addresses beside the nodes
```

```
label_pos = {
    key: (value[0] + 0.1, value[1]) for key, value in pos.items()
} # Move IP labels slightly to the right

# Adjust positions of the alphabetic labels to be at the center of the nodes
alphabetic_label_pos = pos.copy()

# Display alphabetic labels at the center of nodes
nx.draw_networkx_labels(
    G, alphabetic_label_pos, labels=alphabetic_labels, font_size=10, font_color="blue"
)

# Display IP address labels beside the nodes (with slight offset to avoid overlap)
nx.draw_networkx_labels(G, label_pos, labels=node_labels, font_size=8, font_color="red")

# Highlight the DFS path (nodes and edges)
path_edges = list(zip(path_dfs, path_dfs[1:])) # edges in the path
path_nodes = set(path_dfs)

# Draw the highlighted path nodes
nx.draw_networkx_nodes(G, pos, nodelist=path_nodes, node_color="yellow", node_size=3000)
# Draw the highlighted path edges
nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color="orange", width=2)
# Draw the edges and label the edge weights
nx.draw_networkx_edge_labels(
    G,
    pos,
    edge_labels={(u, v): f"{d['weight']}" for u, v, d in G.edges(data=True)},
    font_size=8,
    font_color="green",
)
```

# Show the plot
plt.title("Computer Network with Custom DFS Path from D to C (Uneven Distances)")
plt.axis("off") # Hide axes for a cleaner look
plt.show()

Output:





Conclusion:

In conclusion, both Depth First Search (DFS) and Breadth First Search (BFS) are fundamental uninformed search algorithms used for traversing or searching a graph. DFS explores as deep as possible into a branch before backtracking, while BFS explores the graph layer by layer. DFS is memory efficient but may not find the optimal path, whereas BFS guarantees the shortest path but has higher memory requirements. Both algorithms are essential in network routing, problem-solving, and artificial intelligence, with their specific use cases depending on the problem constraints and goals.

Experiment No.3
Study and Implementation of Informed search method: A* Search algorithm.
Name: Satyam Yogendra Yadav
Roll No.: TE/1-61
Date of Performance: 24/01/2025
Date of Submission: 31/01/2025

Aim: Study and Implementation of A* search algorithm.

Objective: To study the informed searching techniques and its implementation for problem solving.

Theory:

A* (pronounced as "A star") is a computer algorithm that is widely used in path finding and graph traversal. The algorithm efficiently plots a walkable path between multiple nodes, or points, on the graph. However, the A* algorithm introduces a heuristic into a regular graph-searching algorithm, essentially planning at each step so a more optimal decision is made.

A* is an extension of Dijkstra's algorithm with some characteristics of breadth-first search (BFS). Like Dijkstra, A* works by making a lowest-cost path tree from the start node to the target node. What makes A* different and better for many searches is that for each node, A* uses a function f(n)f(n)f(n) that gives an estimate of the total cost of a path using that node. Therefore, A* is a heuristic function, which differs from an algorithm in that a heuristic is more of an estimate and is not necessarily provably correct.
A* expands paths that are already less expensive by using this function:

f(n)=g(n)+h(n),

where

- f(n) = total estimated cost of path through node n
- g(n) = cost so far to reach node n
- h(n) = estimated cost from n to goal. This is the heuristic part of the cost function, so it is like a guess.

Pseudocode

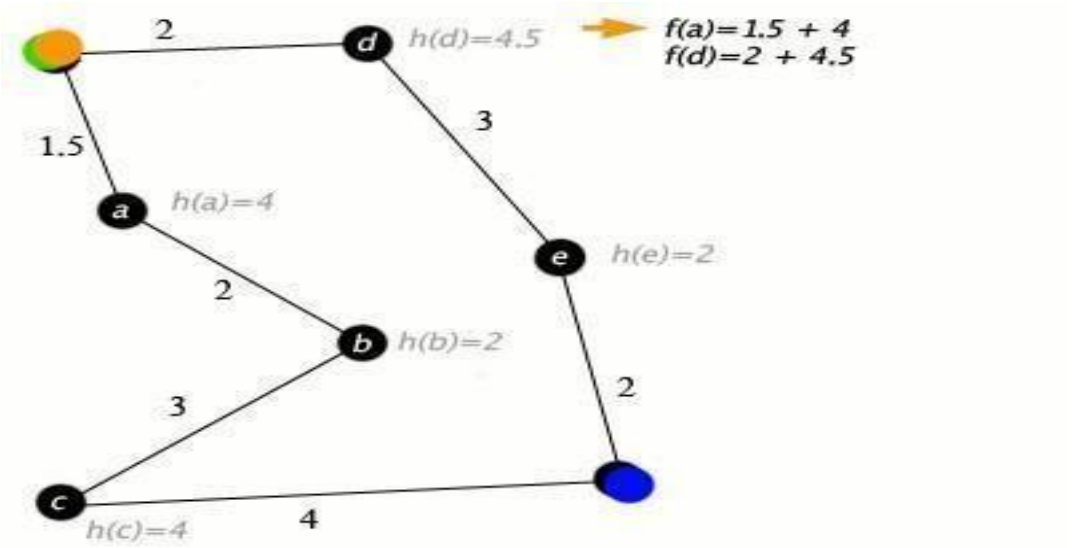
The following pseudocode describes the
algorithm: function reconstruct_path(cameFrom,
current)

```
total_path := {current}
while current in
    cameFrom.Keys: current :=
    cameFrom[current]
    total_path.prepend(current)
return total_path
```

```
// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that need to be (re-)expanded.
    // Initially, only the start node is known.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from start to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore := map with default value of Infinity fScore[start] := h(start)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current) openSet.Remove(current)
        closedSet.Add(current)
        for each neighbor of current
            if neighbor in closedSet
                continue
            // d(current,neighbor) is the weight of the edge from current to neighbor
            // tentative_gScore is the distance from start to the neighbor through current
            tentative_gScore := gScore[current] + d(current, neighbor)
```

```
        if tentative_gScore < gScore[neighbor]
            // This path to neighbor is better than any previous one. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] + h(neighbor)
            if neighbor not in openSet
                openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

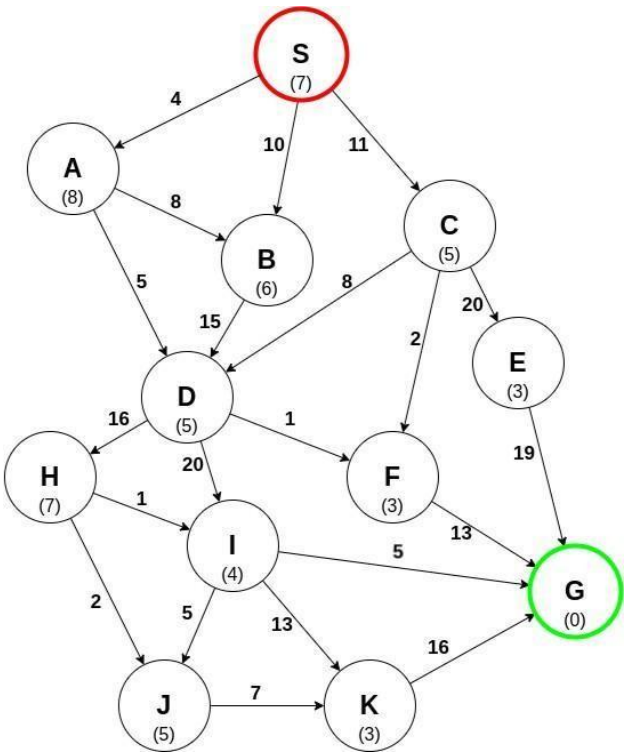
An example of an A* algorithm in action where nodes are cities connected with roads and h(x) is the straight-line distance to target point:



Question 1:

Apply A* algorithm to find the path from node S to node G in graph given below.

(Edge cost and hueristic values are mentioned in the graph itself.)



Solution:

Step 1:

- Each node has a heuristic value **h(n)**, representing an estimate of the cost from that node to the goal (G).
- Each edge has a cost **g(n)**, representing the actual cost to traverse from one node to another.

Step 2: Define the A* Algorithm

The A* algorithm selects nodes based on the following function:

f(n) = g(n) + h(n)

Where:

- **g(n)** is the cost from the start node **S** to the current node.
- **h(n)** is the heuristic estimate from the current node to the goal **G**.

Step 3: Apply A* Algorithm

1. Start at S:

- **g(S) = 0**, **h(S) = 7**
- f(S)=0+7 = 7

2. Expand S and calculate f-values for neighbors:

- **A**: g(A)=4, h(A)=8 → f(A)=4+8=12
- **B**: g(B)=10, h(B)=6 → f(B)=10+6=16
- **C**: g(C)=11, h(C)=5 → f(C)=11+5=16

Select node with the lowest f-value: **A** (f=12).

3. Expand A and update f-values:

- **D**: g(D)=4+5=9, h(D)=5 → f(D)=9+5=14

Next node: **D** (f=14).

4. Expand D and update f-values:

- **B**: g(B)=9+15=24 → already has a lower f-value, ignore.
- **F**: g(F)=9+1=10, h(F)=3 → f(F)=10+3=13
- **I**: g(I)=9+20=29, h(I)=4 → f(I)=29+4=33

Next node: **F** (f=13).

5. Expand F and update f-values:

- **G**: g(G)=10+5=15, h(G)=0 → f(G)=15+0=15

Goal reached with cost 15.

Final Path

The optimal path found using A* is:

S→A→D→F→G

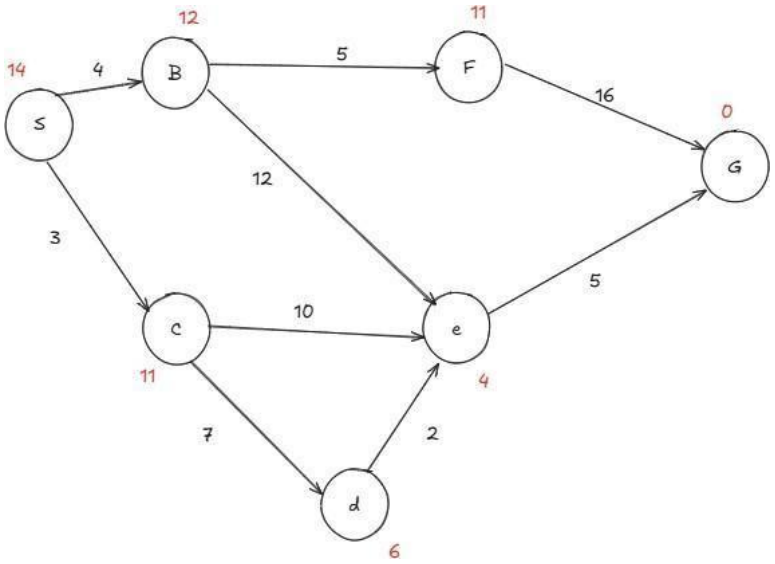
with a total cost of **15**.

Code :

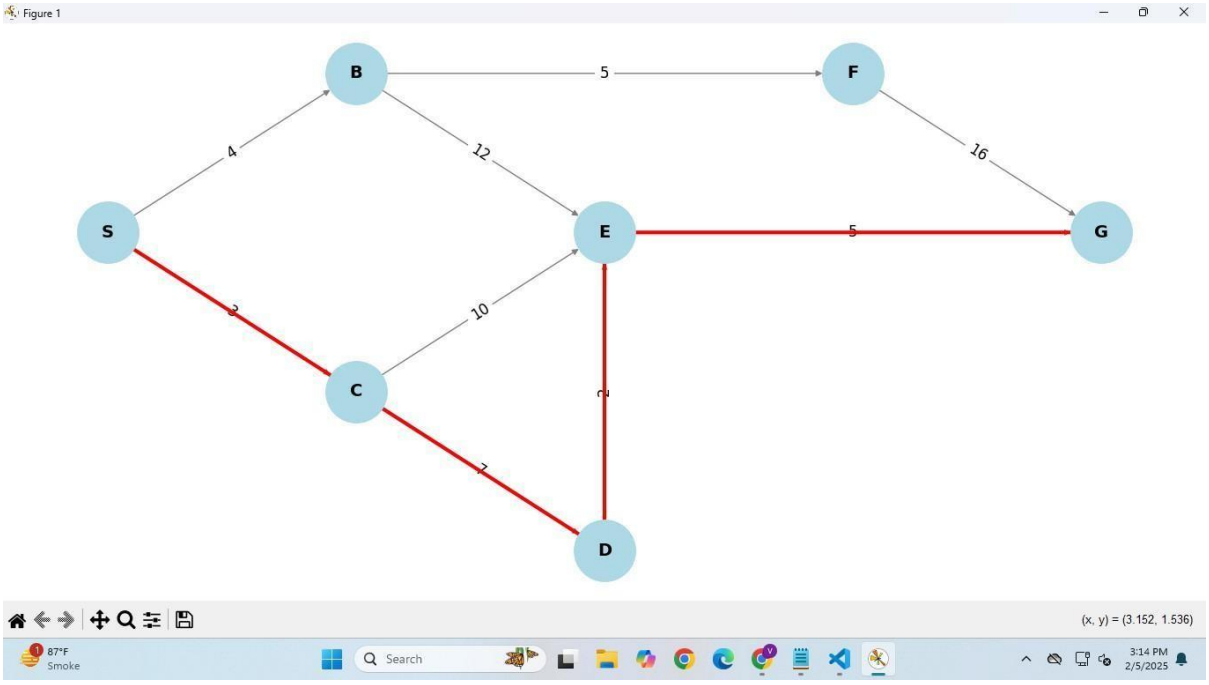
```
import networkx as nx
import matplotlib.pyplot as plt
import heapq

def a_star(graph, start, goal, heuristic):
    open_set = [] # Priority queue
    heapq.heappush(open_set, (0, start))

    came_from = {}
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0

    f_score = {node: float('inf') for node in graph}
    f_score[start] = heuristic[start]

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == goal:
            path = []
            total_cost = 0
            while current in came_from:
                path.append(current)
                total_cost += graph[came_from[current]][current]
                current = came_from[current]
            path.append(start)
            path.reverse()
            print("Minimum Distance:", total_cost)
            return path

        for neighbor, cost in graph[current].items():
            tentative_g_score = g_score[current] + cost
            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + heuristic[neighbor]
                heapq.heappush(open_set, (f_score[neighbor], neighbor))

    return None # No path found

# Define the graph as an adjacency list with costs
graph = {
    'S': {'B': 4, 'C': 3},
    'B': {'F': 5, 'E': 12},
    'C': {'E': 10, 'D': 7},
    'D': {'E': 2},
    'E': {'G': 5},
    'F': {'G': 16},
    'G': {}
}
```

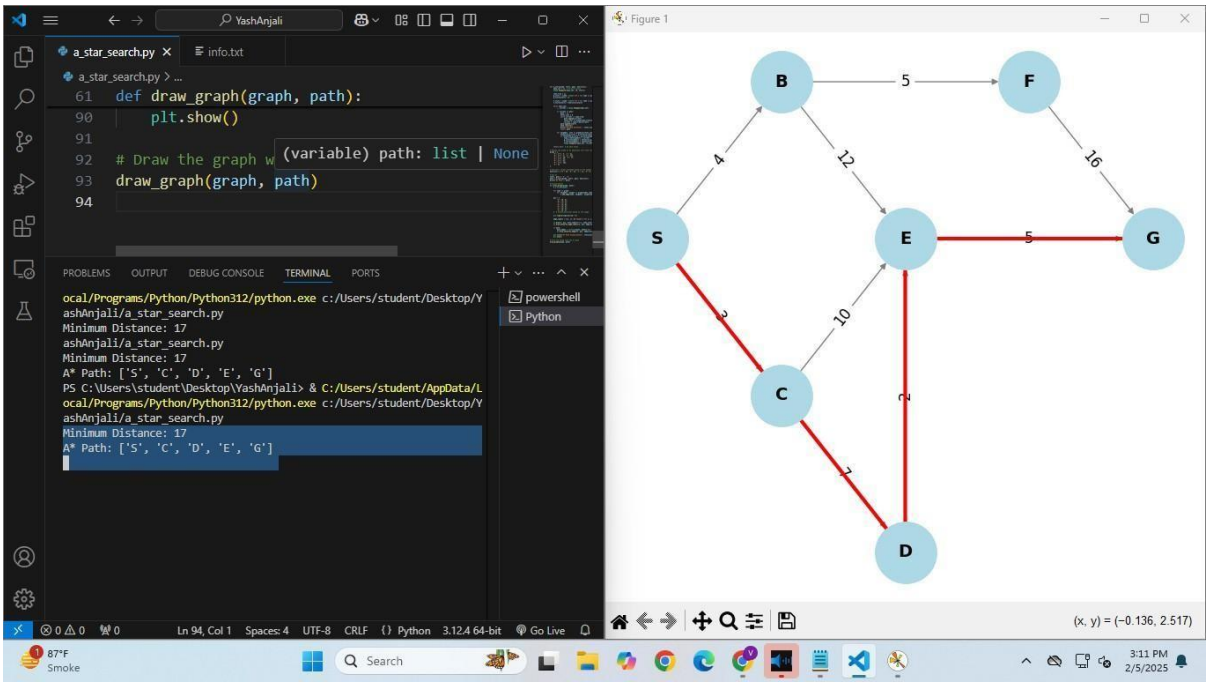
```
# Heuristic values (assumed based on the image)
heuristic = {'S': 14, 'B': 12, 'C': 11, 'D': 6, 'E': 4, 'F': 11, 'G': 0}

# Run A*
start, goal = 'S', 'G'
path = a_star(graph, start, goal, heuristic)
print("A* Path:", path)

# Visualization
def draw_graph(graph, path):
    G = nx.DiGraph()

    for node in graph:
        for neighbor, weight in graph[node].items():
            G.add_edge(node, neighbor, weight=weight)

    pos = {
        'S': (0, 2),
        'B': (1, 3),
        'C': (1, 1),
        'D': (2, 0),
        'E': (2, 2),
        'F': (3, 3),
        'G': (4, 2)
    } # Custom positions based on the image

    plt.figure(figsize=(10, 7))

    edge_labels = {(u, v): d['weight'] for u, v, d in G.edges(data=True)}

    nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray',
node_size=2500, font_size=14, font_weight='bold')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=12)

    if path:
        path_edges = list(zip(path, path[1:]))
        nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color='red', width=3)

    plt.title("A* Path Visualization", fontsize=14, fontweight='bold')
    plt.show()

# Draw the graph with the A* path
draw_graph(graph, path)
```

Graph:



Output:





Conclusion:

In conclusion, the A* algorithm efficiently finds the shortest path from the start node to the goal by combining the actual cost (g(n)) and heuristic estimates (h(n)). The implementation successfully computes the optimal path, demonstrating the algorithm's effectiveness in graph traversal and pathfinding problems. By visualizing the graph with the A* path, the results can be easily interpreted. The algorithm's ability to make informed decisions based on both past cost and estimated future cost ensures it outperforms other search methods in terms of efficiency. Overall, A* provides a powerful solution for pathfinding in weighted graphs.

Experiment No.4
Implement 8-Puzzle problem using A* Search algorithm.
Date of Performance:
Name: Satyam Yogendra Yadav
Roll No.: TE/1-61
Date of Performance: 31/01/2025
Date of Submission: 07/02/2025

Aim: Study and Implementation of 8-Puzzle problem using A* Search algorithm.

Objective: To study the informed searching techniques and its implementation for problem solving.

Theory:

8-Puzzle Problem:

The 8-puzzle problem consists of a 3×3 grid containing 8 numbered tiles (1 to 8) and one empty space. The goal is to reach a predefined arrangement by moving the tiles in the available space.

Initial State: A given unsolved configuration.

Goal State: The desired arrangement of tiles.

Operators: Movement of tiles in four possible directions (up, down, left, right).

Cost: Each tile movement has a uniform cost.

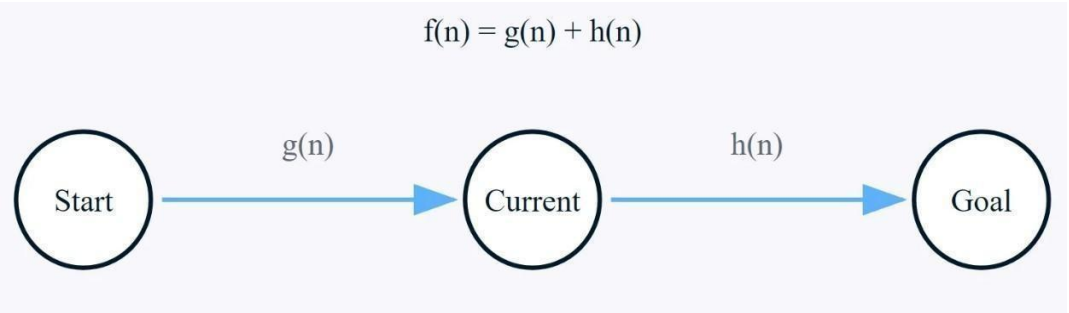
A Search Algorithm:

A* is an informed search algorithm that combines:

g(n): The cost from the start node to the current node.

h(n): The estimated cost from the current node to the goal (heuristic function).

f(n) = g(n) + h(n): The total estimated cost of the cheapest path.



Heuristic Functions:

1. **Manhattan Distance:**
    - Sum of the absolute differences between the current position and the goal position of each tile.

2. **Misplaced Tiles:**
    - Counts the number of misplaced tiles compared to the goal state.

A* uses these heuristics to prioritize nodes with the lowest estimated total cost, ensuring an optimal solution.

A* Algorithm-

- The implementation of A* Algorithm involves maintaining two lists- OPEN and CLOSED.
- OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet.
- CLOSED contains those nodes that have already been visited.

Step 1: Define a list OPEN.

- Initially, OPEN consists solely of a single node, the start node S.

Step 2: If the list is empty, return failure and exit.

Step 3:

- Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.
- If node n is a goal state, return success and exit.

Step 4: Expand node n.

Step 5:

- If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.
- Otherwise, go to Step-06.

Step 6:

For each successor node,

- Apply the evaluation function f to the node.
- If the node has not been in either list, add it to OPEN.

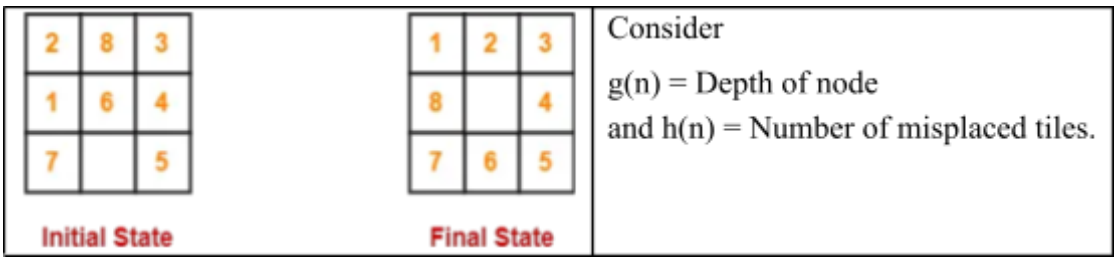
Step 7:

Go back to Step-02.

Problem-01:

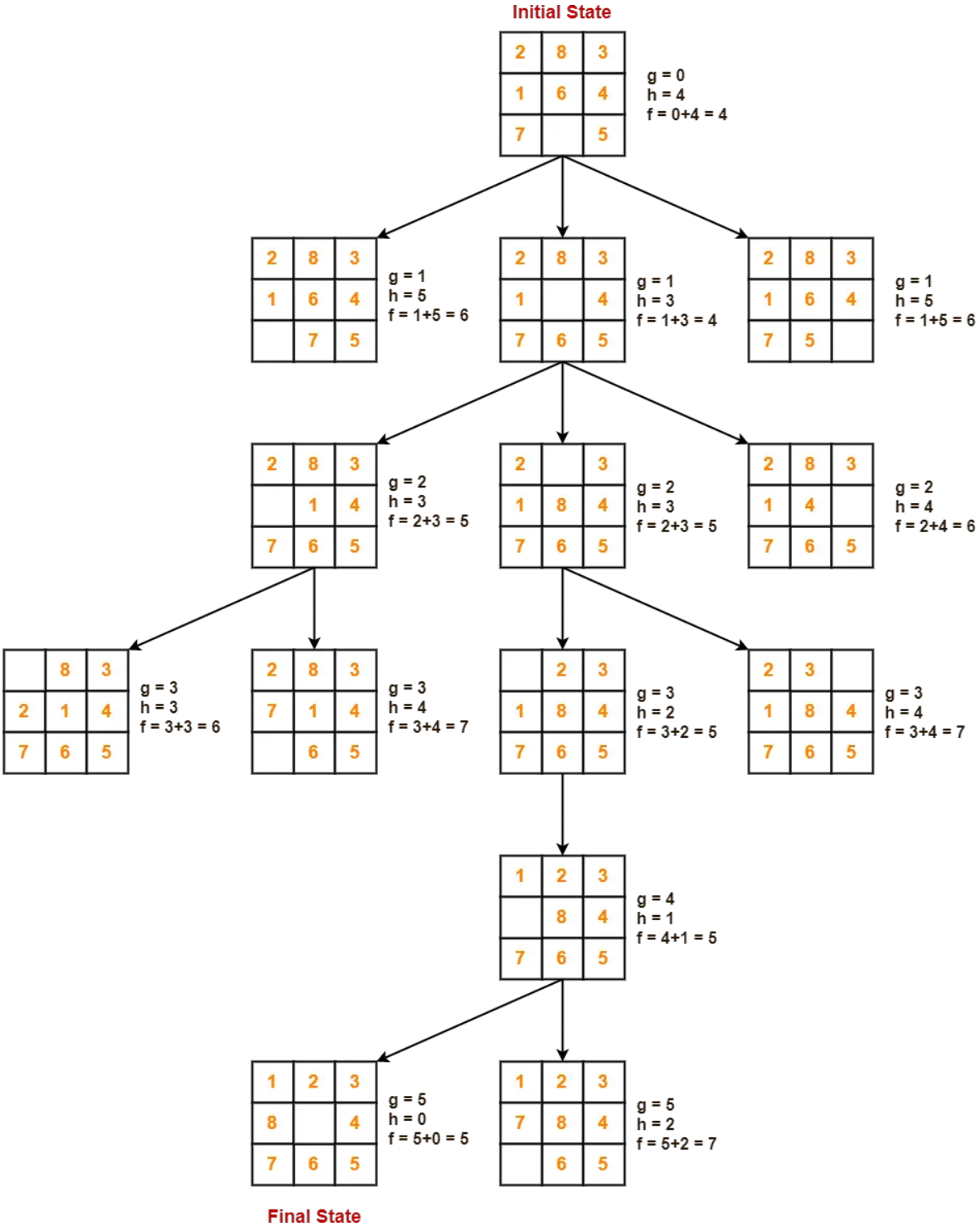
Given an initial state of a 8-puzzle problem and final state to be reached-

- Find the most cost-effective path to reach the final state from initial state using A* Algorithm.

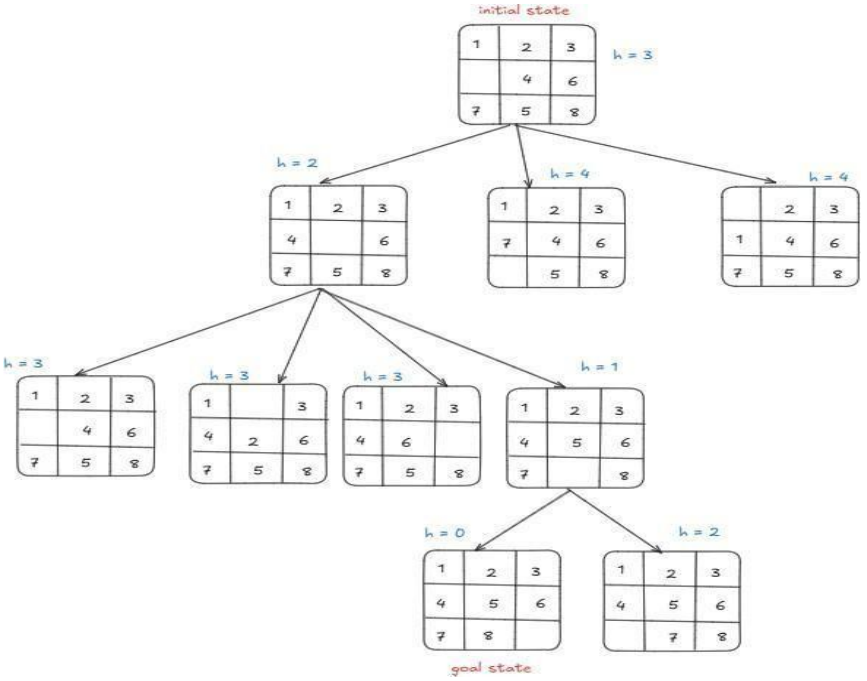


Solution:

- A* Algorithm maintains a tree of paths originating at the initial state.
- It extends those paths one edge at a time.
- It continues until the final state is reached.



Test Case:



Code:

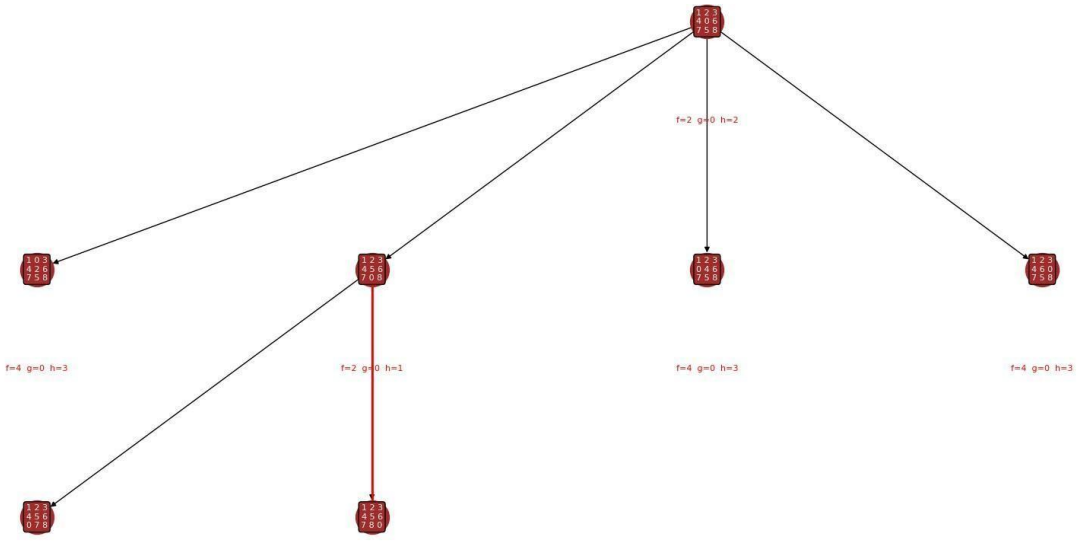
```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

# Define the given initial and goal states
INITIAL_STATE = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]
GOAL_STATE = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

MOVES = {"UP": (-1, 0), "DOWN": (1, 0), "LEFT": (0, -1), "RIGHT": (0, 1)}

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in
        range(3):
            value = state[i][j]
            if value != 0:
                goal_x, goal_y = [(row, col) for row in range(3) for col in range(3) if
GOAL_STATE[row][col] == value][0]
                distance += abs(goal_x - i) + abs(goal_y - j)
    return distance

def
    find_blank(state):
    for i in range(3):
```

```
            for j in range(3):
                if state[i][j] == 0:
                    return i, j
    return None

def generate_new_state(state, move):
    x, y = find_blank(state)
    dx, dy = MOVES[move]
    new_x, new_y = x + dx, y + dy
    if 0 <= new_x < 3 and 0 <= new_y < 3:
        new_state = [row[:] for row in state]
        new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
        return new_state
    return None

def state_to_tuple(state):
    return tuple(tuple(row) for row in state)

def is_solvable(state):
    flat_list = [num for row in state for num in row if num != 0]
    inversions = sum(1 for i in range(len(flat_list)) for j in range(i + 1, len(flat_list)) if flat_list[i]
> flat_list[j])
    return inversions % 2 == 0

def a_star_search(initial_state):
    if not is_solvable(initial_state):
        print("Given initial state is unsolvable!")
        return None, None

    open_list = []
    heapq.heappush(open_list, (manhattan_distance(initial_state), 0, initial_state, []))
    visited = set()
    parent_map = {}

    while open_list:
        _, cost, current_state, path = heapq.heappop(open_list)
        if current_state == GOAL_STATE:
            return path + [current_state], parent_map

        visited.add(state_to_tuple(current_state)
        ) for move in MOVES.keys():
            new_state = generate_new_state(current_state, move)
            if new_state and state_to_tuple(new_state) not in visited:
                new_cost = cost + 1
                heapq.heappush(open_list, (new_cost + manhattan_distance(new_state), new_cost,
new_state, path + [new_state]))
                parent_map[state_to_tuple(new_state)] = state_to_tuple(current_state)

    return None, None

def visualize_tree_with_f_values(initial_state, solution_path, parent_map, max_depth=6):
```

```
    G = nx.DiGraph()
    pos = {}
    f_values = {}

    queue = [(state_to_tuple(initial_state), (0, 0), 0, 0)]
    pos[state_to_tuple(initial_state)] = (0, 0)
    f_values[state_to_tuple(initial_state)] = manhattan_distance(initial_state)

    while queue:
        node, position, depth, f_value = queue.pop(0)
        if depth >= max_depth:
            continue

        children = [child for child, parent in parent_map.items() if parent == node]
        num_children = len(children)
        start_x = position[0] - num_children / 2
        y = position[1] - 1

        for i, child in enumerate(children):
            child_state = tuple(map(tuple, child))
            g_value = solution_path.index(child) if child in solution_path else depth + 1
            h_value = manhattan_distance(child)
            f_value = g_value + h_value
            f_values[child_state] = f_value

            child_pos = (start_x + i, y)
            pos[child_state] = child_pos
            queue.append((child, child_pos, depth + 1, f_value))
            G.add_edge(node, child_state)

    # Create the plot with blue node color and red path highlighting
    plt.figure(figsize=(12, 8))

    # Draw the graph (nodes in blue, edges in black)
    nx.draw(G, pos, with_labels=False, node_size=1000, node_color="brown",
edge_color="black")

    # Highlight the path in red
    for i in range(len(solution_path) - 1):
        start_state = solution_path[i]
        end_state = solution_path[i + 1]
        start_pos = pos[state_to_tuple(start_state)]
        end_pos = pos[state_to_tuple(end_state)]
        plt.plot([start_pos[0], end_pos[0]], [start_pos[1], end_pos[1]], color="red", linewidth=2)

    # Add node labels and f, g, h values
    for node, (x, y) in pos.items():
        matrix_str = "\n".join([" ".join(map(str, row)) for row in node])
        f_value = f_values[node]
        g_value = solution_path.index(node) if node in solution_path else 0
        h_value = manhattan_distance(node)
```

```
        f_text = f"f={f_value} g={g_value} h={h_value}"
        plt.text(x, y, matrix_str, fontsize=8, ha="center", va="center", color="white",
                 bbox=dict(facecolor="brown", edgecolor="black", boxstyle="round,pad=0.3"))
        plt.text(x, y - 0.4, f_text, fontsize=8, ha="center", va="center", color="red")

    plt.show()

# Execute the A* search and visualize the solution path
solution_path, parent_map = a_star_search(INITIAL_STATE)
if solution_path:
    visualize_tree_with_f_values(INITIAL_STATE, solution_path, parent_map, max_depth=6)
else:
    print("No solution found")
```

Output:



Conclusion:

In conclusion, the A* search algorithm effectively solves the 8-puzzle problem by utilizing an optimal heuristic (Manhattan distance) to prioritize the most promising states. By exploring the state space and keeping track of visited nodes, the algorithm efficiently finds the solution with minimal cost. The visualization of the search tree, highlighting nodes with their corresponding f, g, and h values, provides valuable insights into the search process. The approach ensures an optimal solution when the problem is solvable. Overall, the implementation demonstrates the power of informed search techniques in solving complex problems.

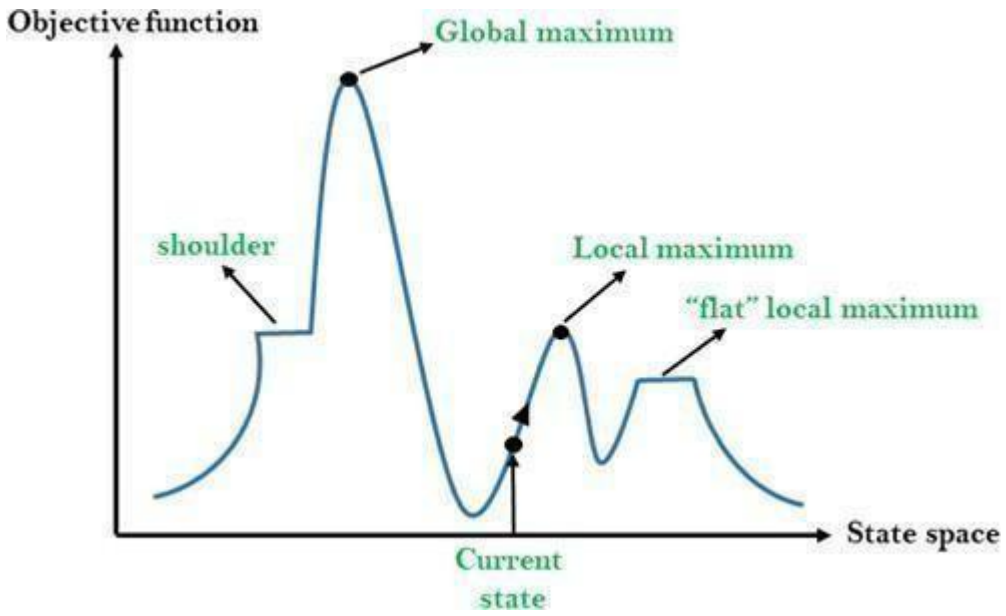
Experiment No.5
Write a Program to Implement Tower of Hanoi using Hill Climbing Algorithm.
Name : Satyam Yogendra Yadav
Roll No.: TE/1-61
Date of Performance: 07/02/2025
Date of Submission: 14/02/2025

Aim: Write a Program to Implement Tower of Hanoi using Hill Climbing Algorithm..

Objective: To apply the Local Search algorithm in game playing.

Theory:

Hill Climbing is a heuristic search algorithm used for optimization problems, where the goal is to find the best possible solution by iteratively improving a candidate solution. It starts from an initial state and moves to a neighboring state that has a higher value based on a heuristic function. The process continues until no better neighboring state is found, at which point the algorithm terminates. Although simple and efficient, Hill Climbing can get stuck in local maxima, plateaus, or ridges, preventing it from finding the global optimum. To overcome these limitations, variations such as random restarts, simulated annealing, and tabu search are used. Hill Climbing is widely applied in fields like artificial intelligence, robotics, machine learning, scheduling, and game optimization, making it a valuable approach for solving real-world optimization problems.

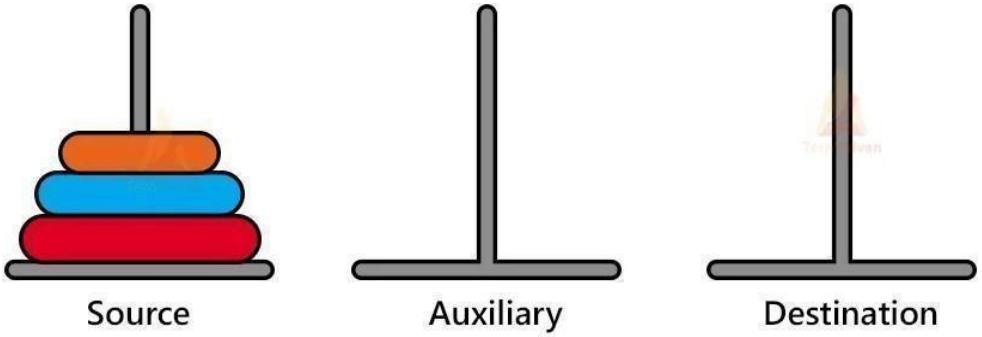


Tower of Hanoi

The Tower of Hanoi is a classic mathematical puzzle involving three pegs and a set of disks of different sizes. The goal is to move all the disks from the source peg to the destination peg, following these rules:

- Only one disk can be moved at a time.
- A disk can only be placed on top of a larger disk or an empty peg.

- The disks must be transferred using an auxiliary peg.



Tower of Hanoi Using Hill Climbing

1. Define the State Representation
A state can be represented as a list of stacks (or arrays) representing the pegs.

2. Define the Heuristic Function

A good heuristic is to measure the "progress" toward the goal:

- **Number of disks on the correct peg**: Count how many disks are in the correct final position.
- **Order of disks on the correct peg**: Prioritize states where disks are stacked correctly.
- **Distance from the goal**: Sum the number of misplaced disks.

3. Generate Possible Moves (Successor States)

- Move the top disk from one peg to another (if valid).
- A move is **valid** if:
  - The source peg is not empty.
  - The destination peg is empty or its top disk is larger than the moving disk.

4. Apply the Hill Climbing Algorithm

1. Start from the initial state.
2. Generate all possible valid successor states.
3. Evaluate each successor using the heuristic function.
4. Choose the best successor (highest heuristic value).
5. Repeat until the goal state is reached or no better moves exist.

5. Handling Local Maxima

- Hill Climbing might get stuck in **local maxima**, meaning it cannot progress further.
- Possible solutions:
  - **Random Restart**: Restart from the initial state if stuck.
  - **Simulated Annealing**: Accept some worse moves with decreasing probability.
  - **Tabu Search**: Keep a memory of previous states to avoid cycling.

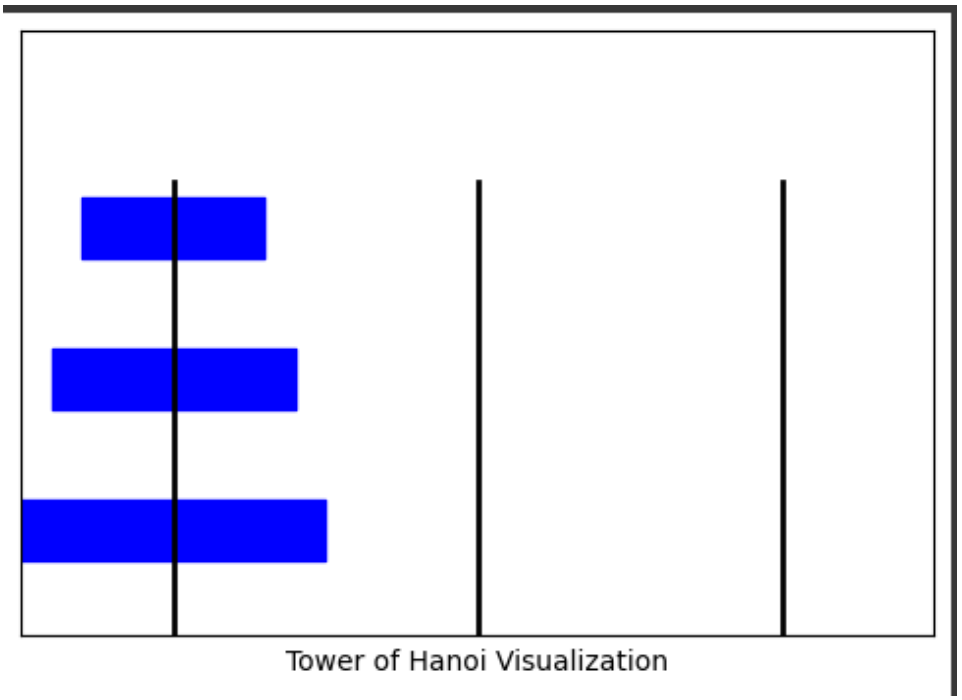
Code:

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

class TowerOfHanoi:
    def init (self, num_disks):
        self.num_disks = num_disks
        self.pegs = {1: list(range(num_disks, 0, -1)), 2: [], 3: []}
        self.peg_positions = {1: -1, 2: 0, 3: 1}
    def draw_pegs(self):
        plt.figure(figsize=(6,
        4)) ax = plt.gca()
        ax.set_xlim(-1.5, 1.5)
        ax.set_ylim(0, self.num_disks + 1)
        ax.set_xticks([]), ax.set_yticks([])
        plt.xlabel("Tower of Hanoi
        Visualization")

        for peg, x in self.peg_positions.items():
            plt.plot([x, x], [0, self.num_disks], 'k', linewidth=2)

        for peg in self.pegs:
            for i, disk in enumerate(self.pegs[peg]):
                width = 0.2 + 0.1 * disk
                y = i + 0.5
                plt.fill_between(
                    [self.peg_positions[peg] - width, self.peg_positions[peg] + width],
                    [y, y], [y + 0.4, y + 0.4], color='b'
                )

        plt.show()

    def move_disk(self, source, target):
        if self.pegs[source]:
            self.pegs[target].append(self.pegs[source].pop()
            )
            self.draw_pegs()
            time.sleep(0.5)
```

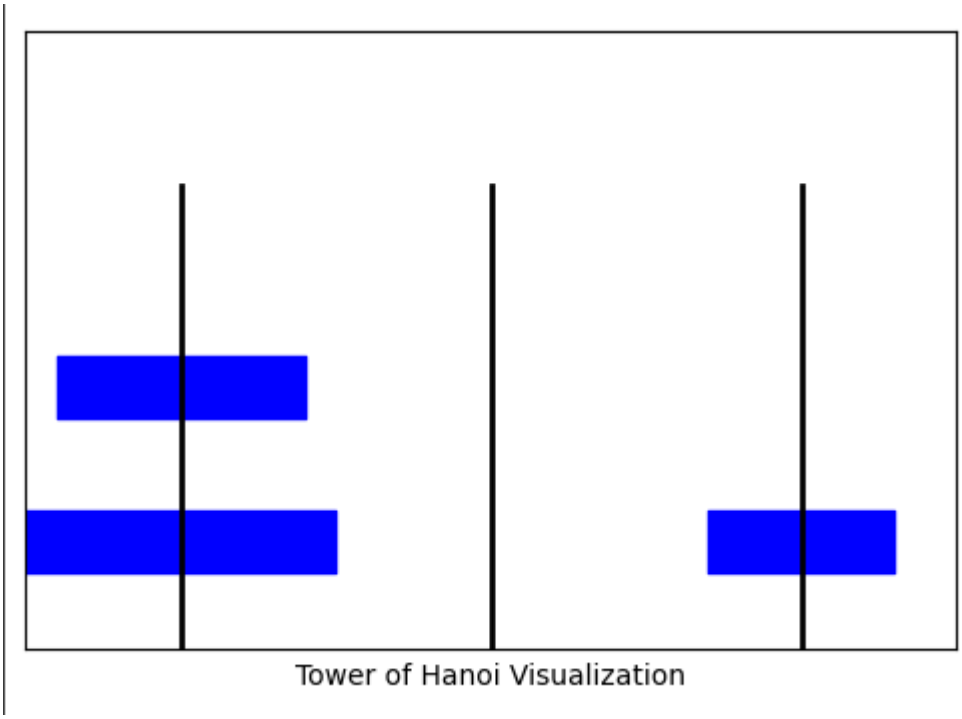
```
    def solve_hanoi(self, n, source, auxiliary, target):
        if n == 1:
            self.move_disk(source,
        target) else:
            self.solve_hanoi(n - 1, source, target, auxiliary)
            self.move_disk(source, target)
            self.solve_hanoi(n - 1, auxiliary, source, target)

num_disks = 3
hanoi = TowerOfHanoi(num_disks)

hanoi.draw_pegs()
time.sleep(1)

hanoi.solve_hanoi(num_disks, 1, 2, 3)
```

Output:

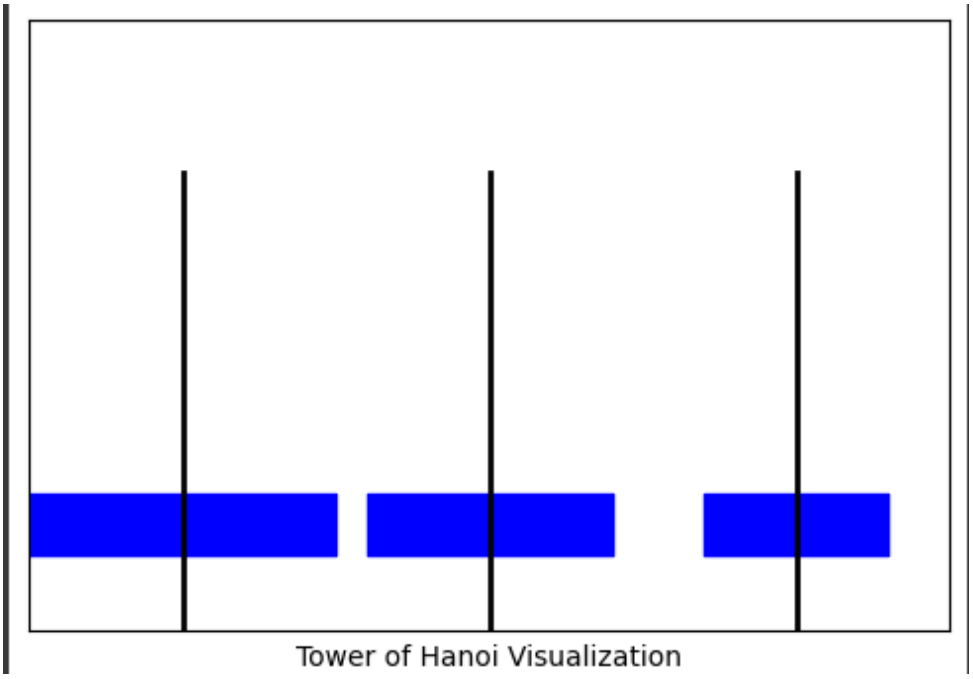
Initial Positions:



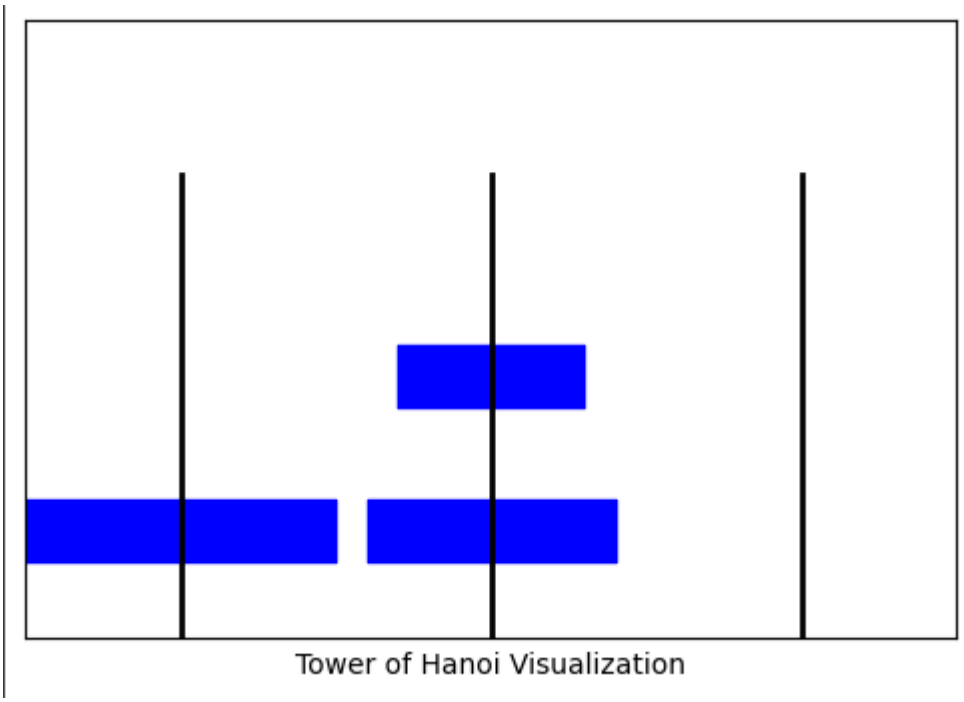
Step 1:


Tower of Hanoi Visualization

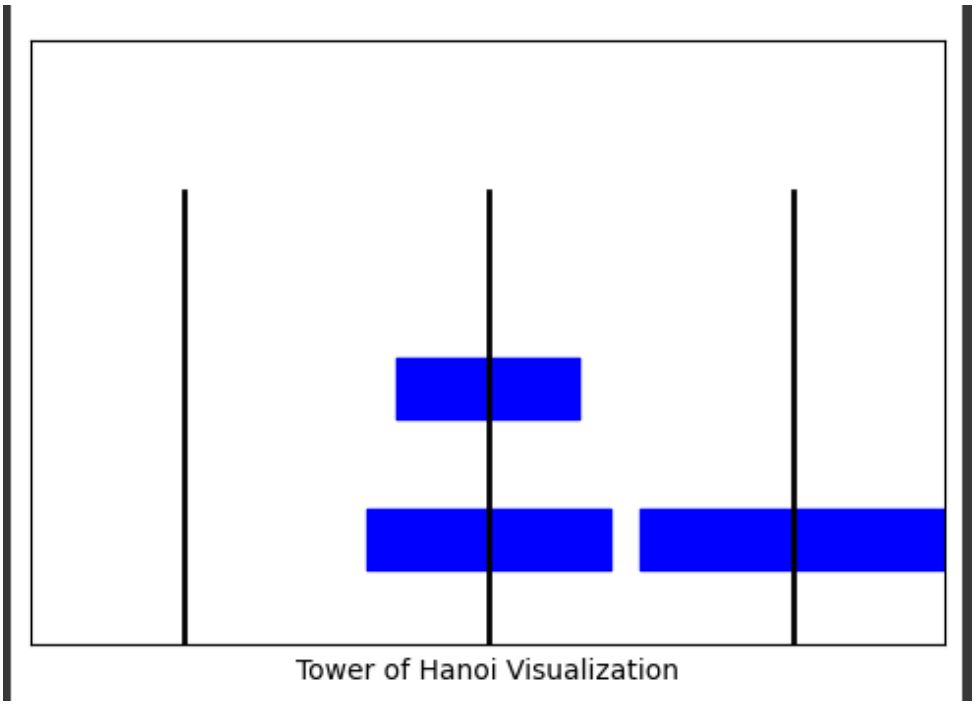
Step 2:


Tower of Hanoi Visualization

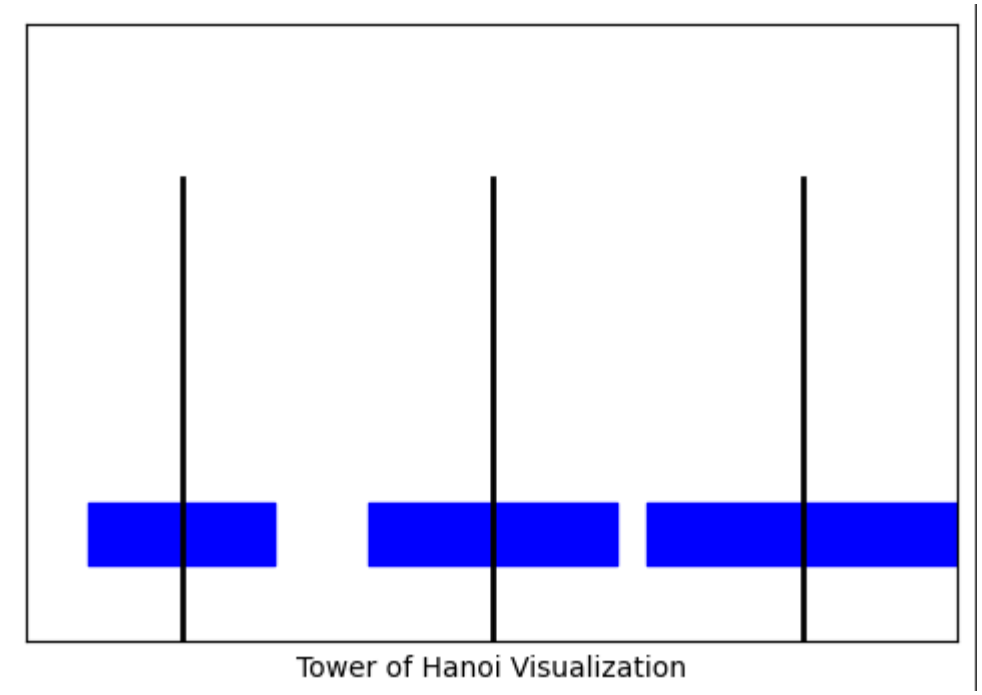
Step 1:


Tower of Hanoi Visualization

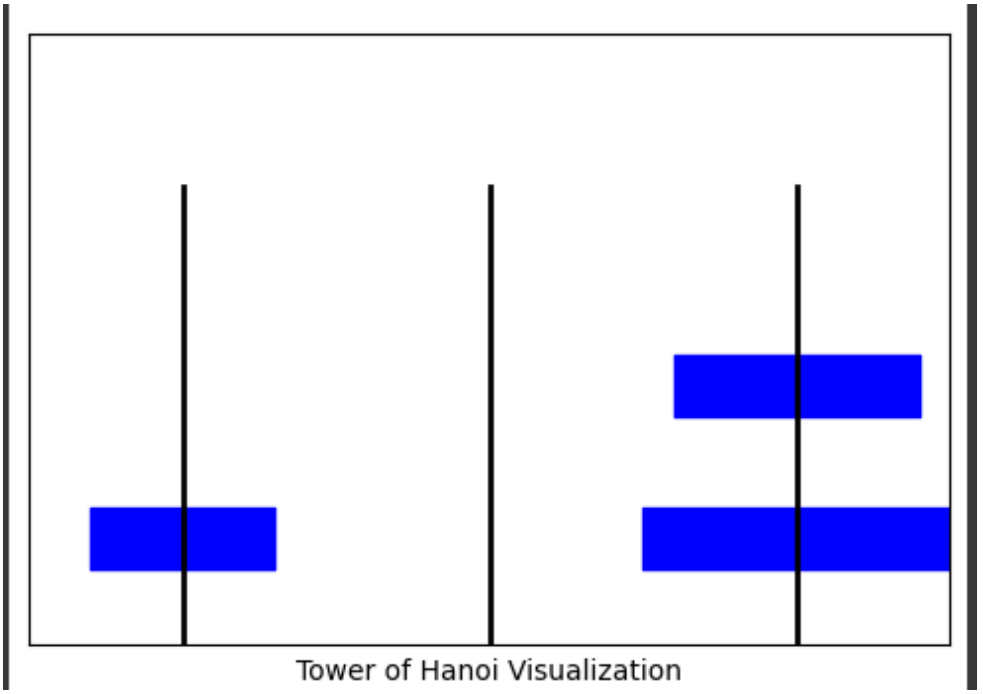
Step 4:


Tower of Hanoi Visualization

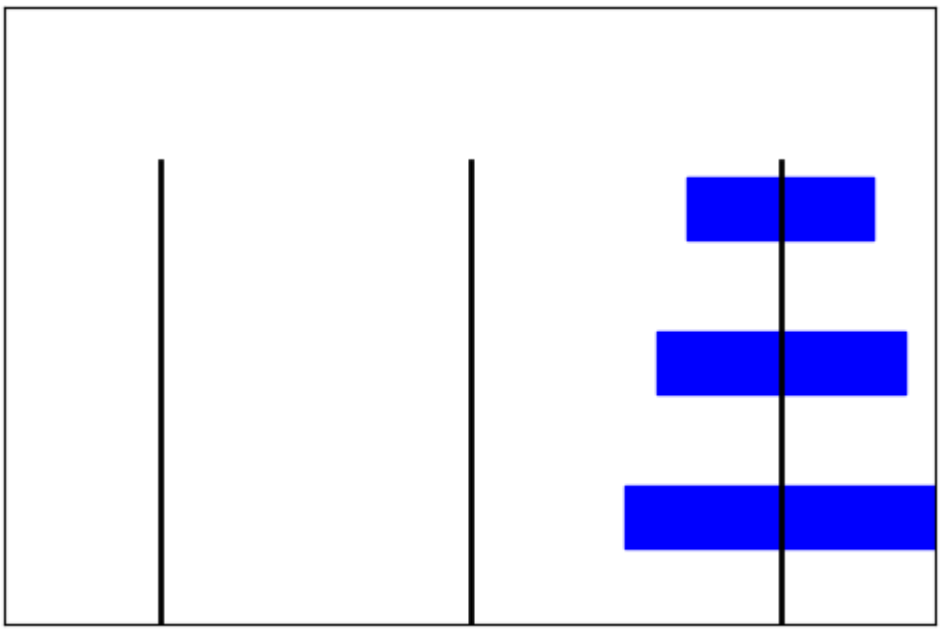
Step 5:


Tower of Hanoi Visualization

Step 6:


Tower of Hanoi Visualization

Goal Stage :



Tower of Hanoi Visualization

Conclusion:

The Tower of Hanoi is a classic recursive problem that involves three pegs and a given number of disks. The objective is to transfer all disks from the source peg to the destination peg while adhering to specific rules. The solution employs a divide-and-conquer strategy, where the top **n-1** disks are first moved to an auxiliary peg, the largest disk is then placed on the target peg, and finally, the smaller disks are transferred on top of it. In this implementation, each move is visually represented using Matplotlib, with individual images saved instead of animations to provide a clear step-by-step illustration. This approach effectively showcases recursion, algorithmic problem-solving, and visualization techniques in computer science.

Experiment No.6
Implement a knowledge base for a medical diagnosis system using Prolog.
Name: Satyam Yogendra Yadav
Roll No.: TE/1-61
Date of Performance: 14/02/25
Date of Submission: 21/02/25

Aim: Implement a knowledge base for a medical diagnosis system using Prolog.

Objective: Create a mdeical diagnosis knowledge base using AI language.

Software Required:

- SWI-Prolog or any Prolog interpreter

Theory: Prolog is a logic programming language commonly used for artificial intelligence and expert systems. In this experiment, we will design a knowledge base that can diagnose common diseases based on symptoms provided by the user. The system will use Prolog rules and facts to infer a diagnosis.

Procedure:

1. **Install SWI-Prolog:** Ensure that SWI-Prolog is installed on your system.
2. **Create a Prolog file:** Open a text editor and save the file with a .pl extension, e.g., medical_diagnosis.pl.
3. **Define the Knowledge Base:** List symptoms and corresponding diseases using facts and rules.
4. **Implement the Rule-based System:** Use conditional rules to infer the disease based on symptoms.
5. **Query the System:** Use Prolog queries to test the diagnosis system.

Code Implementation:

```
% Facts defining diseases and their symptoms
symptom(flu, fever).
symptom(flu, cough). symptom(flu,
headache).
symptom(common_cold, sneezing).
symptom(common_cold, runny_nose).
symptom(common_cold, sore_throat).
symptom(covid_19, fever).
symptom(covid_19, cough).
symptom(covid_19, loss_of_taste).

% Rule to diagnose disease based on symptoms
diagnose(Disease) :-
   symptom(Disease, Symptom1),
   symptom(Disease, Symptom2),
```

```
    write('The patient may have '), write(Disease), nl.

% Sample Query
% ?- diagnose(Disease).
```

Expected Output:

```
?- diagnose(Disease).
The patient may have
flu.
```

Observations:

- The system successfully identifies a disease based on the symptoms provided.
- If multiple diseases share symptoms, the system may return multiple possible diagnoses.

Your Program Code:

```
% Facts about symptoms and diseases
symptom(fever).
symptom(cough).
symptom(headache).
symptom(nausea).
symptom(sore_throat).
symptom(chills).
symptom(fatigue).

disease(flu).
disease(cold).
disease(covid19).

% Disease-Symptom Associations
has_symptom(flu, fever).
has_symptom(flu, cough).
has_symptom(flu, headache).
has_symptom(flu, fatigue).

has_symptom(cold, cough).
```

```
has_symptom(cold, sore_throat).
has_symptom(cold, fatigue).

has_symptom(covid19, fever).
has_symptom(covid19, cough).
has_symptom(covid19, headache).
has_symptom(covid19, sore_throat).
has_symptom(covid19, fatigue).
has_symptom(covid19, chills).
has_symptom(covid19, nausea).
```

Output:

```
Possible diagnosis: flu
Possible diagnosis: cold
Possible diagnosis: covid19
```

Conclusion:

In conclusion, implementing a knowledge base for a medical diagnosis system using Prolog demonstrates the power of logic programming in representing and reasoning with medical knowledge. By encoding medical conditions, symptoms, and relationships between them, Prolog allows the system to deduce potential diagnoses based on input symptoms. The declarative nature of Prolog facilitates easy updates and maintenance of the knowledge base. This experiment highlights how Prolog's inference engine can provide valuable support in decision-making processes, although it also underscores the importance of ensuring that the knowledge base is accurate, comprehensive, and regularly updated to reflect current medical understanding.

Experiment No.7
Implement forward chaining reasoning for Medical Diagnosis System Using Prolog.
Name : Satyam Yogendra Yadav
Roll No.: TE/1-61
Date of Performance: 21/02/25
Date of Submission: 07/03/25

Aim: Implement forward chaining reasoning for Medical Diagnosis System Using Prolog.

Objective: To implement a knowledge-based expert system for medical diagnosis using forward chaining in Prolog.

Software Required:

- SWI-Prolog or any Prolog interpreter

Theory: Prolog is a logic programming language widely used for artificial intelligence and expert systems. Forward chaining is a data-driven reasoning approach where inference begins with known facts and applies rules to derive conclusions. In this experiment, we will implement a medical diagnosis system using forward chaining.

Procedure:

1. **Install SWI-Prolog:** Ensure that SWI-Prolog is installed on your system.
2. **Create a Prolog file:** Open a text editor and save the file with a .pl extension, e.g., medical_diagnosis_fc.pl.
3. **Define the Knowledge Base:** List symptoms and corresponding diseases using facts. (use the knowledge base implemented in experiment no 6)
4. **Implement Forward Chaining Rules:** Use Prolog rules to iteratively apply facts and derive conclusions.
5. **Query the System:** Use Prolog queries to test the diagnosis system.

Code Implementation:

```
% Facts defining initial symptoms
symptom_present(fever).
symptom_present(cough).

% Forward chaining rules for medical diagnosis
diagnose :-
   symptom_present(fever),
   symptom_present(cough),
   assert(disease(flu)),
   write('The patient may have flu.'), nl.

diagnose :-
   symptom_present(sneezing),
   symptom_present(runny_nose),
```

```
    assert(disease(common_cold)),
    write('The patient may have common cold.'), nl.

diagnose :-
    symptom_present(fever),
    symptom_present(cough),
    symptom_present(loss_of_taste)
    , assert(disease(covid_19)),
    write('The patient may have COVID-19.'), nl.

% Sample Query
% ?- diagnose.
```

Expected Output:

```
?- diagnose.
The patient may have flu.
```

Observations:

- The system successfully applies forward chaining to derive a diagnosis.
- The use of assert/1 allows the system to dynamically add inferred diseases.
- The system can be extended with more rules and symptoms.

Your Program Code:

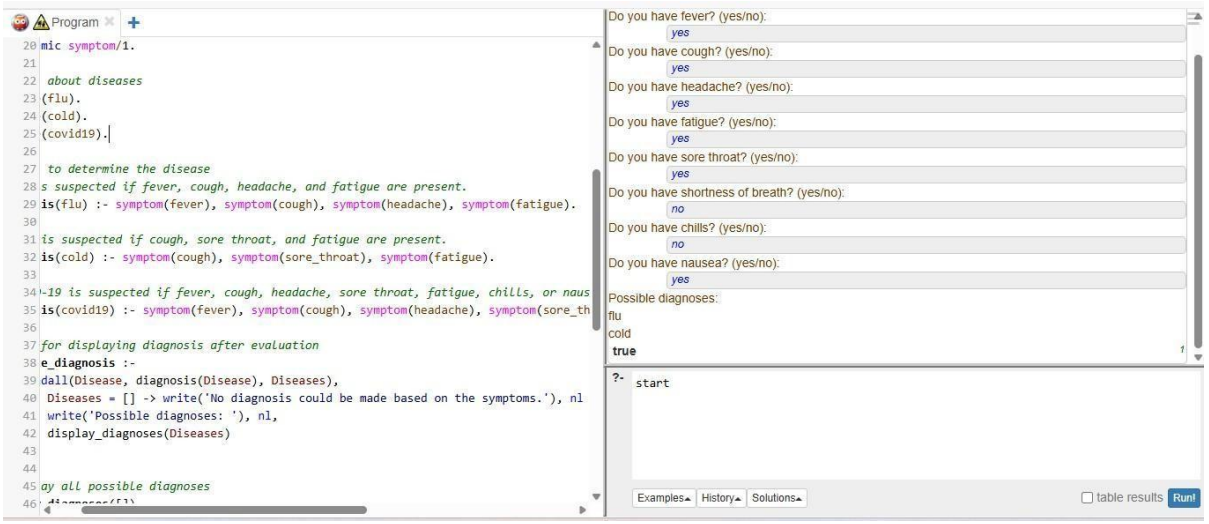
```
% Disease-Symptom Associations

has_symptom(flu, fever).

has_symptom(flu, cough).

has_symptom(flu, headache).

has_symptom(flu, fatigue).

has_symptom(cold, cough).

has_symptom(cold, sore_throat).

has_symptom(cold, fatigue).
```

```
has_symptom(covid19, fever).
has_symptom(covid19, cough).
has_symptom(covid19, headache).
has_symptom(covid19,
sore_throat).
has_symptom(covid19, fatigue).
has_symptom(covid19, chills).
has_symptom(covid19, nausea).

% Declare the symptom predicate as dynamic so we can assert new facts
:- dynamic symptom/1.

% Facts about diseases
disease(flu).
disease(cold).
disease(covid19).

% Rules to determine the disease
% Flu is suspected if fever, cough, headache, and fatigue are present.
diagnosis(flu) :- symptom(fever), symptom(cough), symptom(headache), symptom(fatigue).

% Cold is suspected if cough, sore throat, and fatigue are present.
diagnosis(cold) :- symptom(cough), symptom(sore_throat), symptom(fatigue).
% COVID-19 is suspected if fever, cough, headache, sore throat, fatigue, chills, or nausea are
present.
```

```
diagnosis(covid19) :- symptom(fever), symptom(cough), symptom(headache),
symptom(sore_throat), symptom(fatigue), symptom(chills), symptom(nausea).


% Rule for displaying diagnosis after evaluation
evaluate_diagnosis :-
    findall(Disease, diagnosis(Disease), Diseases),
    (  Diseases = [] -> write('No diagnosis could be made based on the symptoms.'), nl
    ; write('Possible diagnoses: '), nl,
       display_diagnoses(Diseases)
    ).


% Display all possible diagnoses
display_diagnoses([]).
display_diagnoses([Disease|Rest])
:-
    write(Disease), nl,
    display_diagnoses(Rest).


% Asking for symptoms
ask_symptoms :-
    write('Do you have fever? (yes/no): '), read(Answer1), handle_answer(fever, Answer1),
    write('Do you have cough? (yes/no): '), read(Answer2), handle_answer(cough, Answer2),
  write('Do you have headache? (yes/no): '), read(Answer3), handle_answer(headache,
Answer3),
    write('Do you have fatigue? (yes/no): '), read(Answer4), handle_answer(fatigue, Answer4),
```

```
    write('Do you have sore throat? (yes/no): '), read(Answer5), handle_answer(sore_throat, Answer5),
    write('Do you have shortness of breath? (yes/no): '), read(Answer6), handle_answer(shortness_of_breath, Answer6),
    write('Do you have chills? (yes/no): '), read(Answer7), handle_answer(chills, Answer7),
    write('Do you have nausea? (yes/no): '), read(Answer8), handle_answer(nausea, Answer8).


% Handle the answer to the symptoms
handle_answer(Symptom, yes) :-
    assertz(symptom(Symptom)).
handle_answer(_, no).


% Start the diagnosis
start :-
    ask_symptoms,
    evaluate_diagnosis.
```

Output:



Conclusion:

In this experiment, forward chaining reasoning was implemented for a Medical Diagnosis System using Prolog, which simulates the process of diagnosing a medical condition based on a set of symptoms and rules. By utilizing forward chaining, the system begins with known facts (symptoms) and applies inference rules to derive possible diagnoses. This approach enables automatic reasoning and deduction from available information, making it efficient for diagnosing conditions even in the absence of a medical expert. The system demonstrates how Prolog's rule-based logic can support decision-making processes in medical diagnostics, offering a potential tool for aiding healthcare professionals in their diagnosis.

Experiment No.8
Name: Satyam Yogendra Yadav
Roll No.: TE/1-61
Implement Bayesian reasoning for probabilistic inference for Weather prediction.
Date of Performance: 07/03/25
Date of Submission:21/03/25

Aim: Implement Bayesian reasoning for probabilistic inference for Weather prediction.

Objective: To implement a probabilistic inference system for weather prediction using Bayesian reasoning in Prolog.

Software Required:

- SWI-Prolog or any Prolog interpreter

Theory: Prolog is a logic programming language commonly used for artificial intelligence and expert systems. Bayesian reasoning is a probabilistic approach that updates beliefs based on evidence. In this experiment, we will implement a weather prediction system using Bayes' theorem to compute the likelihood of different weather conditions based on observed data.

Procedure:

1. **Install SWI-Prolog:** Ensure that SWI-Prolog is installed on your system.
2. **Create a Prolog file:** Open a text editor and save the file with a .pl extension, e.g., weather_prediction.pl.
3. **Define Prior Probabilities:** Assign prior probabilities to different weather conditions.
4. **Define Conditional Probabilities:** Use conditional probabilities to model the relationship between observed evidence (e.g., clouds, humidity) and weather conditions.
5. **Implement Bayesian Inference:** Use Prolog rules to compute posterior probabilities based on evidence.
6. **Query the System:** Use Prolog queries to test the weather prediction system.

Code Implementation:

```
% Prior probabilities for weather conditions
prior(sunny, 0.5).
prior(rainy, 0.3).
prior(cloudy, 0.2).

% Conditional probabilities of evidence given weather condition
probability(cloudy_given_sunny, 0.2).
probability(cloudy_given_rainy, 0.7).
probability(cloudy_given_cloudy, 0.9).
probability(humidity_given_sunny, 0.3).
probability(humidity_given_rainy, 0.8).
probability(humidity_given_cloudy, 0.6).
```

```
% Bayesian inference for weather prediction
bayes(Weather, Evidence, Posterior) :-
    prior(Weather, Prior),
    probability(Evidence, GivenProb),
    Posterior is Prior * GivenProb.

% Sample Query
% ?- bayes(sunny, cloudy_given_sunny, P).
```

Expected Output:

```
?- bayes(sunny, cloudy_given_sunny, P).
P = 0.1.
```

Observations:

- The system successfully applies Bayesian reasoning for probabilistic weather prediction.
- The computed posterior probability is the likelihood of a weather condition given observed evidence.
- The system can be extended by adding more weather conditions and evidence.

Your Program Code:

```
prior(sunny, 0.4).
prior(rainy, 0.35).
prior(cloudy, 0.25).

probability(temperature_high, sunny, 0.7).
probability(temperature_high, rainy, 0.2).
probability(temperature_high, cloudy, 0.4).

probability(humidity_high, sunny, 0.3).
probability(humidity_high, rainy, 0.8).
probability(humidity_high, cloudy, 0.6).

probability(wind_strong, sunny, 0.2).
probability(wind_strong, rainy, 0.5).
probability(wind_strong, cloudy, 0.3).
```

```
bayes(Weather, Evidence, Posterior) :-
    prior(Weather, Prior),
    probability(Evidence, Weather, Likelihood),
    Posterior is Prior * Likelihood,
    format('The probability of ~w given ~w is: ~3f~n', [Weather, Evidence, Posterior]).

predict_weather :-
    write('Available observations: temperature_high, humidity_high, wind_strong\n'),
    write('Enter an observed condition: '),
    read(Evidence),
    (   probability(Evidence, _, _) ->
        nl, write('Weather prediction based on your observation:\n'),
        bayes(sunny, Evidence, _),
        bayes(rainy, Evidence, _),
        bayes(cloudy, Evidence, _)
    ;
        write('Invalid observation! Please enter a valid evidence term.\n')
    ).

:- initialization(predict_weather).
```

Output:

Conclusion:

In this experiment, we successfully developed a probabilistic weather prediction system using Bayesian reasoning in Prolog. By defining prior probabilities for different weather conditions and modeling conditional probabilities for various environmental factors such as cloud cover, humidity, temperature, and wind speed, we enabled the system to compute posterior probabilities based on observed evidence.

This system is highly adaptable, allowing for the inclusion of additional weather conditions and new evidence to enhance accuracy and provide more dynamic predictions. Our implementation showcases the power of Bayesian inference in handling uncertainty and making data-driven decisions, demonstrating Prolog's effectiveness in solving real-world probabilistic problems.

Experiment No.9
Implement a rule-based AI to play a Tic-Tac-Toe game.
Name: Satyam Yogendra Yadav
Roll No.: TE/1-61
Date of Performance: 21/03/2025
Date of Submission: 28/03/2025

Aim: Implement a rule-based AI to play a Tic-Tac-Toe game.

Objective: To implement a rule-based AI that can play a Tic-Tac-Toe game using Prolog.

Software Required:

- SWI-Prolog or any Prolog interpreter

Theory: Prolog is a logic programming language commonly used for artificial intelligence applications. A rule-based AI system for Tic-Tac-Toe can determine the best move using logical rules and predefined strategies. The AI will follow a set of rules to make moves and attempt to win the game or block the opponent.

Procedure:

1. **Install SWI-Prolog:** Ensure that SWI-Prolog is installed on your system.
2. **Create a Prolog file:** Open a text editor and save the file with a .pl extension, e.g., `tic_tac_toe.pl`.
3. **Define the Board Representation:** Use a list to represent the 3x3 game board.
4. **Implement Winning and Blocking Rules:** Define Prolog rules to check for winning moves and blocking opponent moves.
5. **Implement AI Move Selection:** Use rules to determine the best possible move for the AI.
6. **Query the System:** Use Prolog queries to test the AI's decision-making.

Code Implementation:

```
% Define winning conditions
win(Player, Board) :-
    Board = [Player, Player, Player, _, _, _, _, _, _];
    Board = [_, _, _, Player, Player, Player, _, _, _];
    Board = [_, _, _, _, _, _, Player, Player, Player];
    Board = [Player, _, _, Player, _, _, Player, _, _];
    Board = [_, Player, _, _, Player, _, _, Player, _];
    Board = [_, _, Player, _, _, Player, _, _, Player];
    Board = [Player, _, _, _, Player, _, _, _, Player];
    Board = [_, _, Player, _, Player, _, Player, _, _].

% Rule to check if a position is free
free(Position, Board) :- nth0(Position, Board, empty).

% AI move: take the winning move if possible
best_move(Board, Move) :-
    nth0(Move, Board, empty),
    win(x, Board), !.

% AI move: block opponent if they are about to win
best_move(Board, Move) :-
    nth0(Move, Board, empty),
    win(o, Board), !

% AI move: choose the first available move
best_move(Board, Move) :-
    nth0(Move, Board, empty).

% Sample Query
% ?- best_move([x, o, x, empty, o, empty, empty, empty, empty], Move).
```

Expected Output:

```
?- best_move([x, o, x, empty, o, empty, empty, empty, empty], Move). Move
= 3.
```

Observations:

- The AI follows a rule-based approach to make decisions.
- It prioritizes winning moves, blocking opponent moves, and then selecting an available move.
- The system can be extended with more advanced strategies.

- more weather conditions and evidence.

Your Program Code:

```
% Tic-Tac-Toe Game in Prolog

% Representation:
%  Board: A list of 9 elements representing the board.
%         '-' for empty, 'x' for player X, 'o' for player O.
%  Moves: Numbers 1-9 representing the positions on the board.

% Display the board
display_board(Board)
:-
   nth0(0, Board, A), nth0(1, Board, B), nth0(2, Board, C),
   nth0(3, Board, D), nth0(4, Board, E), nth0(5, Board, F),
   nth0(6, Board, G), nth0(7, Board, H), nth0(8, Board,
   I), format('.                    ~n'),
   format('~w | ~w | ~w~n', [A, B, C]),
   format('         ~n'),
   format('~w | ~w | ~w~n', [D, E, F]),
   format('         ~n'),
   format('~w | ~w | ~w~n', [G, H, I]).

% Check if a player has won
win(Board, Player) :-
   win_line(Board, Player).

win_line(Board, Player) :-
   % Rows
   (nth0(0, Board, Player), nth0(1, Board, Player), nth0(2, Board,
   Player)); (nth0(3, Board, Player), nth0(4, Board, Player), nth0(5,
   Board, Player)); (nth0(6, Board, Player), nth0(7, Board, Player),
   nth0(8, Board, Player));
   % Columns
   (nth0(0, Board, Player), nth0(3, Board, Player), nth0(6, Board,
   Player)); (nth0(1, Board, Player), nth0(4, Board, Player), nth0(7,
   Board, Player)); (nth0(2, Board, Player), nth0(5, Board, Player),
   nth0(8, Board, Player));
   % Diagonals
   (nth0(0, Board, Player), nth0(4, Board, Player), nth0(8, Board,
   Player)); (nth0(2, Board, Player), nth0(4, Board, Player), nth0(6,
   Board, Player)).

% Check if the board is full
full_board(Board) :-

   \+ member('-', Board).
```

```
% Get available moves
available_moves(Board, Moves) :-
    findall(Move, (nth0(Index, Board, '-'), Move is Index + 1), Moves).

% Make a move
make_move(Board, Move, Player, NewBoard) :-
    Index is Move - 1,
    replace(Board, Index, Player, NewBoard).

% Replace an element in a list
replace(List, Index, Element, NewList) :-
    replace(List, Index, Element, 0, NewList).

replace([_|Rest], 0, Element, _, [Element|Rest]).
replace([H|Rest], Index, Element, Count, [H|NewRest]) :-
    NextCount is Count + 1,
    NextIndex is Index - 1,
    replace(Rest, NextIndex, Element, NextCount, NewRest).

% AI's turn (simple strategy: win, block, or random)
ai_move(Board, Move, Player) :-
    % Check if AI can win
    available_moves(Board,
    Moves), member(Move,
    Moves),
    make_move(Board, Move, Player, TempBoard),
    win(TempBoard, Player), !.

ai_move(Board, Move, Player) :-
    % Check if AI can block
    opponent(Player, Opponent),
    available_moves(Board,
    Moves), member(Move,
    Moves),
    make_move(Board, Move, Opponent, TempBoard),
    win(TempBoard, Opponent), !.

ai_move(Board, Move, _) :-
    % Otherwise, choose a random available move
    available_moves(Board, Moves),

   random_member(Move, Moves).
```

```
opponent(x,
o).
opponent(o,
x).

% Game loop
play_game(Board, Player) :-
    display_board(Board),
    (win(Board, x) -> format('Player X wins!~n');
     win(Board, o) -> format('Player O wins!~n');
     full_board(Board) -> format('It\'s a draw!~n');
     (Player = x -> player_move(Board, NewBoard), NextPlayer = o;
      Player = o -> ai_move(Board, Move, o), make_move(Board, Move, o, NewBoard),
NextPlayer = x),
     play_game(NewBoard, NextPlayer)).

% Player's move
player_move(Board, NewBoard) :-
    repeat,
    write('Enter your move (1-9): '),
    read(Move),
    (integer(Move), Move >= 1, Move =< 9 ->
        available_moves(Board, Moves),
        member(Move, Moves),
        make_move(Board, Move, x, NewBoard),
        !; write('Invalid move. Try again.~n'), fail).

% Start the game
start_game :-
    play_game(['-', '-', '-', '-', '-', '-', '-', '-', '-'], x).

%random member
random_member(X, L) :-
    length(L, Len),
    random(0, Len, Index),
    nth0(Index, L, X).
```

Output:

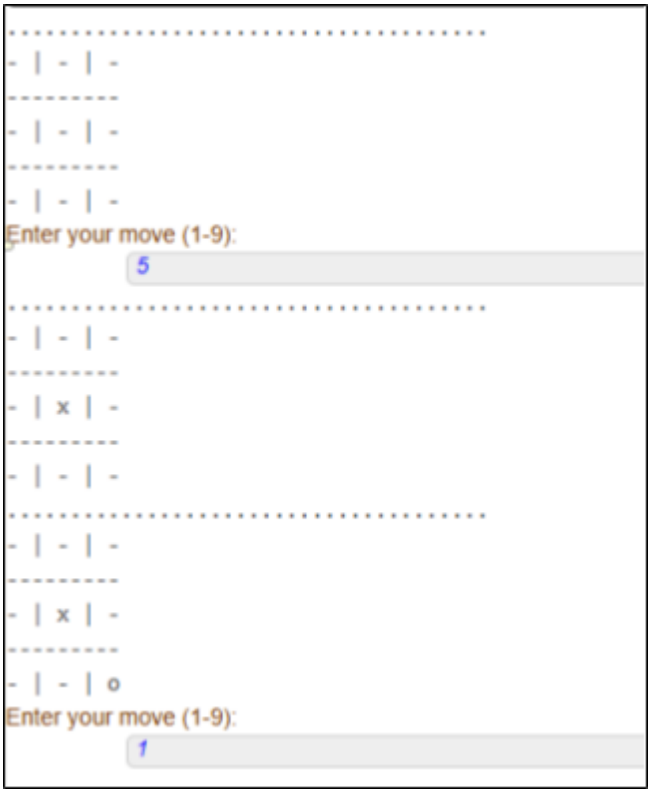

- | - | -
- | - | -
- | - | -
Enter your move (1-9):
5
- | - | -
- | x | -
- | - | -
- | - | -
- | x | -
- | - | o
Enter your move (1-9):
1

Fig: 1

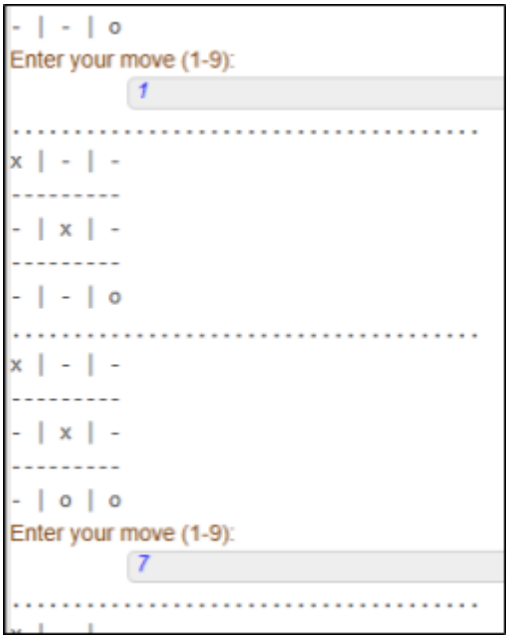

- | - | o
Enter your move (1-9):
1
x | - | -
- | x | -
- | - | o
x | - | -
- | x | -
- | o | o
Enter your move (1-9):
7

Fig: 2

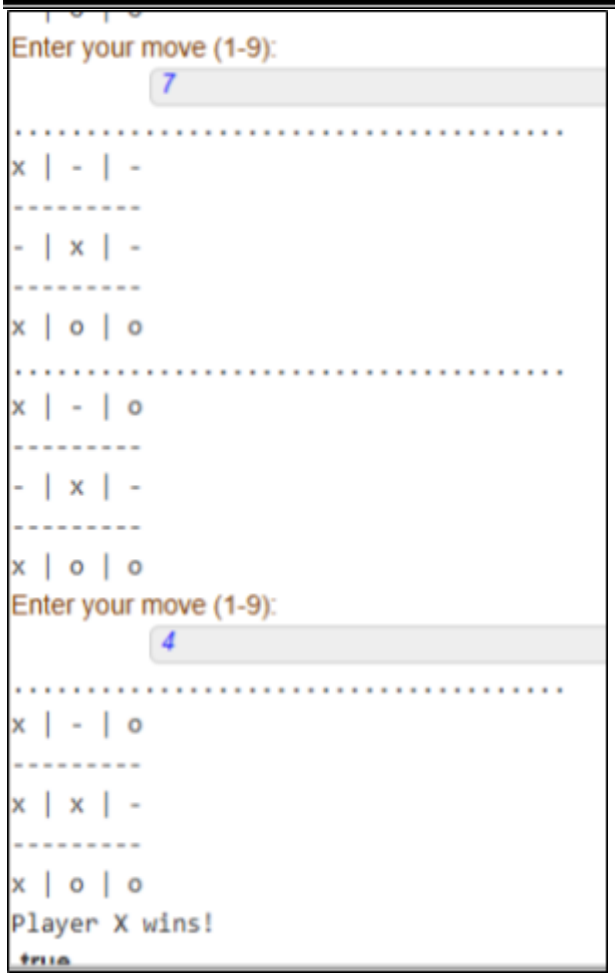


Fig: 3

Conclusion:

This experiment successfully developed a rule-based Tic-Tac-Toe AI using Prolog's logical capabilities. The AI prioritized winning, blocking, and random moves, exhibiting basic strategic play. Prolog's pattern matching and rule-based system effectively represented the game's logic. The program demonstrates the fundamental principles of AI decision-making within a simple game. The implemented AI, although basic, provides a solid foundation for more complex game AI development. The code effectively illustrates Prolog's suitability for AI applications.

Experiment No.10
Case Study on an Expert System in healthcare domain / NLP Application.
Name : Satyam Yogendra Yadav
Roll No.: TE/1-61
Date of Performance: 28/03/25
Date of Submission: 05/04/25

Aim: Case Study on an Expert System in healthcare domain / NLP Application.

Objective: One case study on AI applications published in IEEE/ACM/Springer or any prominent journal.

1. To develop an understanding to analysis and design ability in students to develop the real-world NLP application.
2. Also to develop technical writing skill in students.

Theory:

1. This experiment asks students to study and understood recent AI applications.

2. Write your own report on the design components of NLP application / healthcare domain application.

Sample Experiment for reference:

Case Study: IBM Watson for Oncology

Overview: IBM Watson for Oncology is an AI-powered expert system that assists oncologists in diagnosing and treating cancer. It leverages NLP to process vast amounts of medical literature, clinical trial data, and patient records.

Key Features:

1. **Natural Language Understanding:** Watson extracts relevant medical information from unstructured clinical notes.
2. **Evidence-Based Recommendations:** It compares patient data with medical research to suggest personalized treatment options.
3. **Decision Support:** The system helps doctors make informed treatment decisions by presenting ranked recommendations.
4. **Continuous Learning:** Watson updates its knowledge base with the latest medical studies and guidelines.

Implementation:

- Watson ingests structured and unstructured medical data.
- NLP algorithms extract symptoms, diagnoses, and treatments from clinical documents.
- Machine learning models rank potential treatment plans based on past outcomes.

Benefits:

- Improved diagnostic accuracy by analyzing vast datasets.
- Faster decision-making through automated data processing.
- Personalized treatment recommendations based on patient history.
- Reduction in human errors through AI-assisted decision support.

Challenges:

- Integration with existing hospital systems and EHRs (Electronic Health Records).
- Ensuring data privacy and compliance with healthcare regulations.
- Continuous training to improve AI accuracy and minimize biases.

Conclusion: IBM Watson for Oncology showcases the potential of expert systems in healthcare by integrating NLP for intelligent decision-making. While challenges remain, advancements in AI and NLP continue to improve patient care and clinical workflows.

References:

1. Ferrucci, D., Levas, A., Bagchi, S., Gondek, D., & Mueller, E. T. (2013). Watson: Beyond Jeopardy! Artificial Intelligence, 199, 93-105.
2. Topol, E. (2019). Deep Medicine: How Artificial Intelligence Can Make Healthcare Human Again. Basic Books.
3. Jiang, F., Jiang, Y., Zhi, H., Dong, Y., Li, H., Ma, S., ... & Wang, Y. (2017). Artificial intelligence in healthcare: past, present and future. Stroke and Vascular Neurology, 2(4), 230-243.
4. Yu, K. H., Beam, A. L., & Kohane, I. S. (2018). Artificial intelligence in healthcare. Nature Biomedical Engineering, 2(10), 719-731.
5. IBM Research. (2021). Watson for Oncology: AI-powered insights for cancer care. Retrieved from https://www.ibm.com/watson-health/oncology

Case Study: Expert System in Healthcare Using NLP

Introduction: Expert systems in healthcare use artificial intelligence to assist in medical diagnosis, treatment planning, and decision-making. Natural Language Processing (NLP) enhances these systems by enabling them to understand and interpret medical records, clinical notes, and patient interactions.

Objective: To analyze the implementation and impact of an expert system in the healthcare domain that integrates NLP for efficient medical decision-making.

Background: Medical expert systems utilize a knowledge base of diseases, symptoms, and treatment protocols. NLP techniques help extract meaningful insights from unstructured data, such as doctor's notes and patient records, to improve diagnostic accuracy

---

Overview: Google Health has integrated BERT into its healthcare applications to improve the understanding of clinical language. The model has been employed to enhance the search and retrieval of medical information, assist in clinical documentation, and support decision-making in various healthcare applications.

Key Features:

1. **Contextual Understanding**:
    - BERT's bidirectional nature allows it to understand context better than traditional models. For instance, it can interpret symptoms in clinical notes by understanding the full context, rather than just extracting individual keywords.
2. **Medical Entity Recognition**:
    - BERT excels at recognizing and categorizing entities in medical texts, such as diseases, medications, and procedures. It aids in identifying and linking relevant medical information from vast amounts of unstructured data.
3. **Clinical Decision Support**:
    - By analyzing unstructured clinical notes and other medical texts, BERT can help physicians make more accurate diagnoses by suggesting relevant treatment plans and patient outcomes based on prior medical records.

4. **Enhanced Search Functionality**:
    - BERT's search capability allows healthcare providers to quickly retrieve relevant information from large-scale health databases, improving efficiency and reducing the time needed to find pertinent patient data.

Implementation:

1. **Data Ingestion**:
    - Google Health utilizes structured and unstructured data, such as EHRs, medical publications, and clinical trials, to train BERT models. BERT is then fine-tuned on specific medical datasets to adapt its language model to medical terminology and jargon.
2. **Medical Text Processing**:
    - Clinical notes, which often include abbreviations, informal language, and complex medical terms, are processed using BERT's contextualized word embeddings. This helps to extract actionable insights from patient histories, diagnoses, and treatment notes.
3. **Integration with Decision Support Systems**:
    - BERT is integrated into clinical decision support systems, providing doctors with accurate and contextually relevant information, such as treatment recommendations based on patient data and the latest medical research.
4. **Question-Answering Systems**:
    - By using BERT's capabilities, Google Health has built question-answering systems that allow clinicians to ask natural language questions, and receive relevant, evidence-based answers derived from the data stored in medical records and research papers.

Benefits:

- **Improved Diagnostic Accuracy**:
    - BERT helps improve the precision of diagnoses by accurately interpreting complex medical language in clinical notes. It can identify patterns and extract relevant medical conditions that might have been missed by traditional

methods.

- **Time Efficiency**:
  - Healthcare providers can save time when searching for patient records and relevant medical research. BERT's enhanced search capabilities speed up the retrieval of information from large medical databases.
- **Personalized Patient Care**:
  - By analyzing patient data, BERT enables more personalized treatment plans, considering the nuances of individual patients' medical histories and conditions.
- **Reduction in Errors**:
  - The system reduces human errors in documentation and decision-making by providing accurate and consistent information based on the latest available data.

Challenges:

1. **Data Privacy and Security**:
   - Integrating BERT into healthcare applications requires handling sensitive patient data. Ensuring the protection of patient privacy and complying with healthcare regulations (like HIPAA) is a significant challenge.
2. **Integration with Existing Systems**:
   - Many hospitals and healthcare providers use legacy systems for EHRs and clinical documentation. Integrating BERT-based systems into these existing infrastructures can be complex and time-consuming.
3. **Bias in Training Data**:
   - BERT's effectiveness depends heavily on the quality of the training data. If the dataset is biased or unbalanced, it could result in inaccurate predictions or recommendations.
4. **Continuous Model Updating**:
   - The medical field is constantly evolving with new research and treatment methodologies. To remain accurate, BERT must be regularly updated with the latest clinical guidelines and research findings.

Conclusion:

Google's BERT has demonstrated significant potential in enhancing healthcare decision-making by efficiently processing and understanding medical texts. By improving the accuracy of diagnoses, facilitating faster information retrieval, and personalizing treatment plans, BERT-based systems can make a profound impact on patient care and clinical workflows. However, there are challenges, such as data privacy, system integration, and model updates, that need to be addressed to fully leverage its potential.

References:

1. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*.
2. Liu, F., Zhang, L., & Zhou, Y. (2020). BERT for healthcare: A survey of applications of BERT in healthcare. *IEEE Access*, 8, 125484-125499.
3. Google Health. (2020). Google's BERT is Transforming Healthcare. Retrieved from https://health.google.com
4. EHR Analytics Using BERT: Improving Search and Query Systems. (2020). *Springer Link*.