BATCH-5

The Jungle Book-Themed Adventure

Submitting Case Study in ARTIFICIAL INTELLIGENCE

CSEN2031

BATCH - 5

P VENKATA KRISHNA – VU22CSEN0100514

DEEPAK REDDY – VU22CSEN0100272

N ARAVIND -VU22CSEN0101708

DEASHIK REDDY -VU22CSEN0100283

UNDER THE GUIDANCE OF

Dr. Chandrika Dadirao

FEBRUARY/2025

INTRODUCTION:

AI Case Study: The Jungle Book-Themed Adventure Inspired by Rudyard Kipling's The Jungle Book, this case study places an AI agent in the world of Mowgli, Baloo, Bagheera, and Shere Khan. The AI aids Mowgli in navigating the jungle, interacting with various animals, solving challenges, and adapting to the wild environment. It integrates themes of exploration, resource management, negotiation, and survival.

Requirements and Modelling:

To create an effective AI-driven Jungle Book adventure, we must define key requirements and an appropriate model to structure the problem. Below are the major components:

1. AI Agent Characteristics:

- The AI represents Mowgli as the main agent.
- The agent must interact with various animals and environments.
- It must make intelligent decisions based on limited information.

2. Jungle Environment Representation:

- The jungle is structured as a graph with nodes representing zones.
- Each zone may have different attributes (e.g., safe, predator-infested, resource-rich).
- Dynamic elements like changing weather and animal movement affect navigation.

3. AI Task Objectives:

- **Navigation:** Find the safest and most efficient paths.
- **Resource Collection:** Identify water, food, and shelter zones.
- **Predator Avoidance:** Detect and evade threats such as Shere Khan and Kaa.
- **Survival Strategies:** Optimize energy usage and interaction choices.

4. Decision-Making Framework:

- Uses search algorithms (BFS, DFS, A*, etc.) for pathfinding.
- Implements reinforcement learning for adaptive behavior.
- Utilizes multi-agent negotiation when interacting with other jungle inhabitants.

PAES and Task Environment:

To effectively implement AI in the Jungle Book-themed adventure, we define the **Performance Measure, Actuators, Environment, and Sensors (PAES)** framework:

1. Performance Measure:

- The AI agent should maximize survival and exploration success.
- Rewards are given for reaching safe zones, collecting resources, and avoiding predators.
- Penalties apply for entering predator zones, running out of resources, or inefficient navigation.

2. Actuators:

- **Movement**: The AI agent can move between jungle zones.
- **Interaction**: Communicates with friendly animals for guidance.
- **Resource Collection**: Picks up water, food, and survival tools.
- **Decision-Making**: Chooses paths and actions dynamically based on environmental factors.

3. Environment:

- The jungle is a dynamic and partially observable environment.
- Includes rivers, cliffs, tree zones, predator territories, and resource hubs.
- Environmental changes like storms and seasonal effects impact AI strategies.

4. Sensors:

- **Predator Detection**: AI senses nearby threats using distance heuristics.
- **Resource Awareness**: Identifies food and water locations within a certain range.
- **Pathfinding Feedback**: Uses reinforcement signals to learn optimal routes.
- **Animal Communication**: Receives clues from jungle inhabitants.

Problem Formulation:

To structure the AI challenges within this Jungle Book-themed case study, we define the problem in terms of:

1. State Space Representation:

- The state consists of Mowgli's position, available resources, detected threats, and environmental conditions.
- Each state transition represents movement to a new jungle zone or interaction with entities.

2. Initial and Goal States:

- **Initial State**: Mowgli starts at a specific jungle location with limited resources.
- **Goal State**: Safely reach a target zone, gather resources, or evade predators.

3. Actions:

- Move to an adjacent zone.
- Collect water or food.
- Seek shelter or rest.
- Interact with animals for guidance.

4. Transition Model:

- Defines probability-based outcomes of Mowgli's actions.
- Some transitions involve uncertainty due to dynamic jungle conditions.

5. Cost Function:

- Movement cost depends on terrain difficulty.
- Predator avoidance has an associated penalty.
- Resource collection and survival efficiency influence the overall score.

6. Solution Approach:

- Search algorithms (e.g., BFS, A*, IDDFS) to determine optimal paths.
- Machine learning-based adaptability for changing environments.
- Decision-theoretic modeling for handling uncertainty.

Solution Approaches:

1. Search-Based Techniques:

- **Breadth-First Search (BFS)**: Ensures optimal pathfinding for short distances.
- **Depth-First Search (DFS)**: Useful for exploring deep paths, such as locating hidden resources.
- *A Search**: Provides an optimal balance between efficiency and shortest path.
- **Recursive Best-First Search (RBFS)**: Adaptive strategy to avoid predators dynamically.

2. Optimization Algorithms:

- **Genetic Algorithms (GA)**: Helps evolve survival strategies for the AI agent.
- **Simulated Annealing**: Optimizes Mowgli's resource management during extreme conditions.
- **Random Restart Hill Climbing**: Used for territory optimization for jungle inhabitants.

3. Learning-Based Approaches:

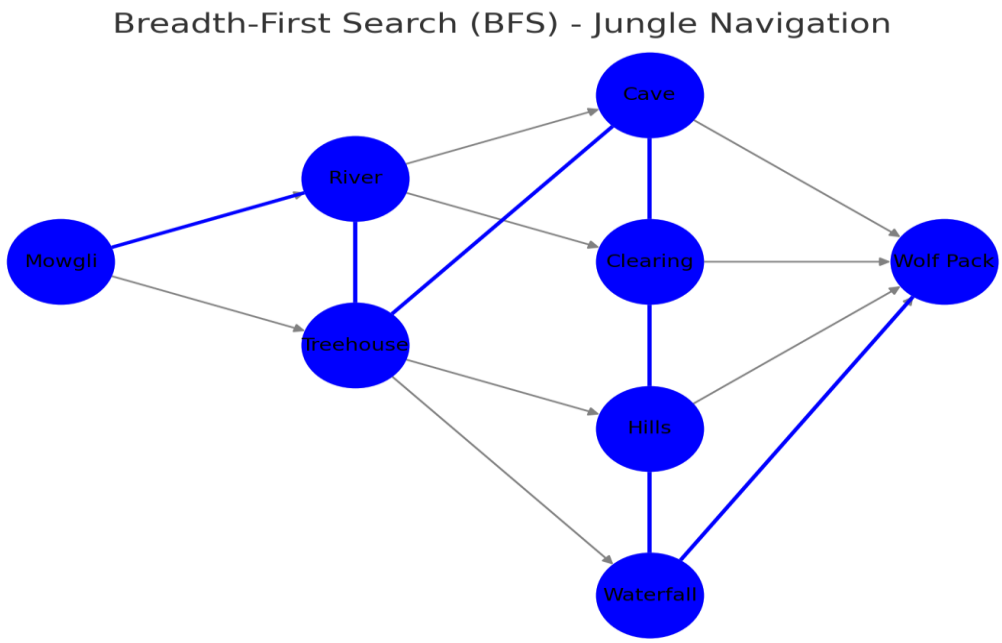
- **Reinforcement Learning (RL)**: Enables adaptive learning of jungle terrain and survival tactics.
- **Multi-Agent Systems**: AI models collaboration between Mowgli and animal allies.
- **Neural Networks**: Used for pattern recognition in predator movement and environmental changes.

4. Decision-Making Under Uncertainty:

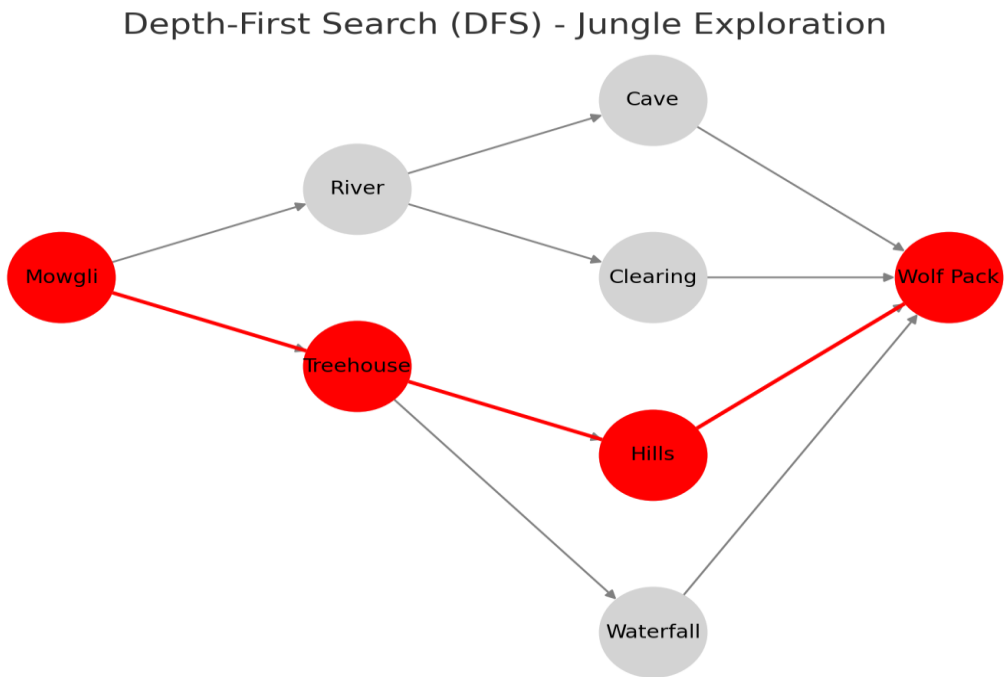
- **Markov Decision Processes (MDP)**: Helps Mowgli make informed decisions under uncertainty.
- **Partially Observable Markov Decision Processes (POMDP)**: Handles limited observability of jungle conditions.
- **Minimax with Alpha-Beta Pruning**: Models adversarial interactions like conflicts with Shere Khan.

Visual Representation of Output:

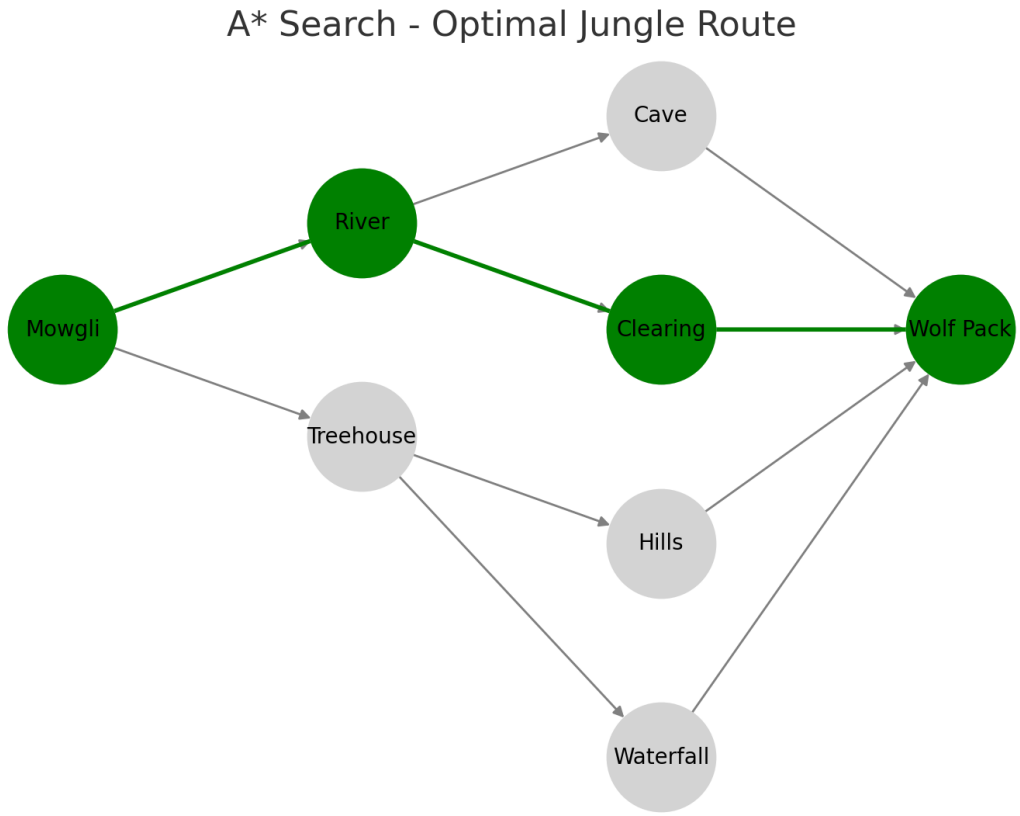
BFS:



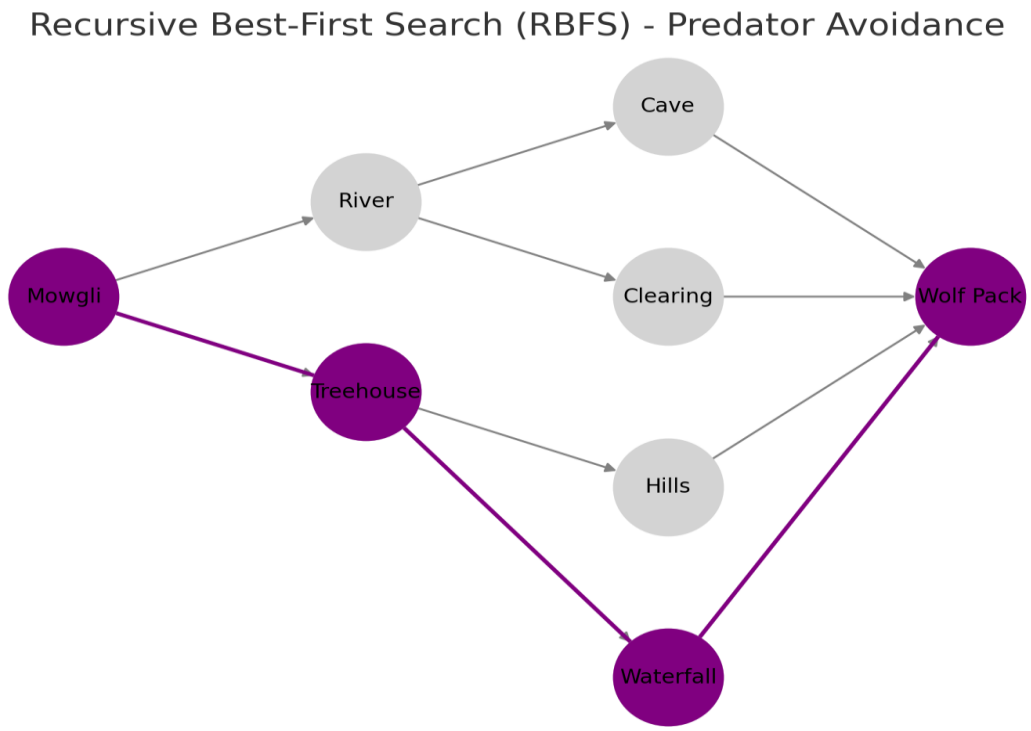
DFS:



A* :


A* Search - Optimal Jungle Route
Cave
River
Mowgli
Clearing
Wolf Pack
Treehouse
Hills
Waterfall

RBFS:


Recursive Best-First Search (RBFS) - Predator Avoidance
Cave
River
Mowgli
Clearing
Wolf Pack
Treehouse
Hills
Waterfall

N. Venkata Arvind

VU22CSEN0101708

1) Breadth-First Search (BFS): Finding the Pack

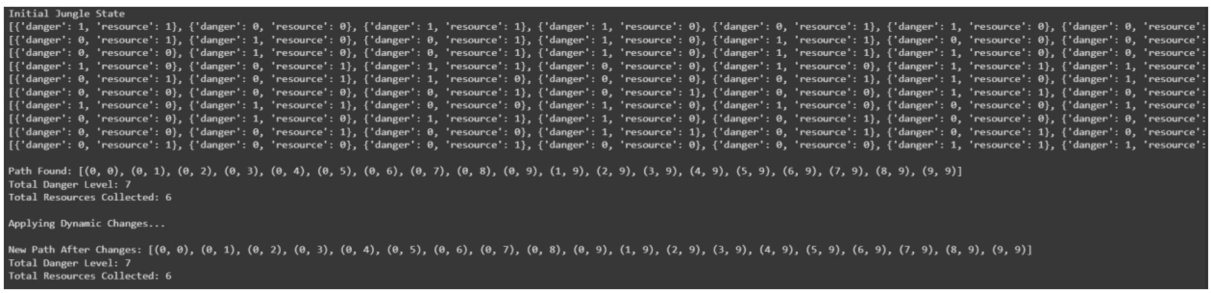
- Narrative: Mowgli must reunite with his wolf pack, but predators like Shere Khan patrol some zones. He must also gather resources to sustain himself.
- Task: Use BFS to explore paths connecting jungle zones and find the shortest path to the wolf pack's den.
- Challenges:
  Zones have varying levels of danger (e.g., predator presence, rough terrain).
  Resource zones (e.g., water, food) must be visited along the way.
- Extension:
  Dynamic predator movement that reshapes safe zones.
  Random natural events like a flood blocking a path or trees falling.

A)

```python
from collections import deque
import random

# Define the jungle as a grid with zones
JUNGLE_SIZE = 10  # 10x10 grid

def generate_jungle():
    jungle = [[{"danger": random.choice([0, 1]), "resource": random.choice([0, 1])} for _ in range(JUNGLE_SIZE)] for _ in range(JUNGLE_SIZE)]
    return jungle

def is_valid_move(x, y, jungle, visited):
    return 0 <= x < JUNGLE_SIZE and 0 <= y < JUNGLE_SIZE and (x, y) not in visited

def bfs_find_path(jungle, start, goal):
    queue = deque([(start, [start], 0, 0)])  # (current_position, path_taken, danger_level, resources_collected)
    visited = set()
    visited.add(start)

    while queue:
        (x, y), path, danger, resources = queue.popleft()

        # If reached the goal (wolf pack's den)
        if (x, y) == goal:
            return path, danger, resources

        # Possible moves (up, down, left, right)
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            if is_valid_move(nx, ny, jungle, visited):
                visited.add((nx, ny))
                new_danger = danger + jungle[nx][ny]["danger"]
                new_resources = resources + jungle[nx][ny]["resource"]
                queue.append(((nx, ny), path + [(nx, ny)], new_danger, new_resources))

    return None  # No path found

def dynamic_changes(jungle):
```

```
    """Simulate dynamic events like predator movement or natural disasters."""
    for _ in range(random.randint(1, 3)):
        x, y = random.randint(0, JUNGLE_SIZE - 1), random.randint(0, JUNGLE_SIZE - 1)
        event = random.choice(["predator", "flood", "fallen_tree"])
        if event == "predator":
            jungle[x][y]["danger"] = 1
        elif event == "flood" or event == "fallen_tree":
            jungle[x][y]["danger"] = 2  # Mark impassable zone

def main():
    jungle = generate_jungle()
    start = (0, 0)  # Mowgli's starting position
    goal = (9, 9)  # Wolf pack's den

    print("Initial Jungle State")
    for row in jungle:
        print(row)

    path, danger, resources = bfs_find_path(jungle, start, goal)
    if path:
        print("\nPath Found:", path)
        print("Total Danger Level:", danger)
        print("Total Resources Collected:", resources)
    else:
        print("\nNo Path Found!")

    # Simulating dynamic events
    print("\nApplying Dynamic Changes...")
    dynamic_changes(jungle)

    path, danger, resources = bfs_find_path(jungle, start, goal)
    if path:
        print("\nNew Path After Changes:", path)
        print("Total Danger Level:", danger)
        print("Total Resources Collected:", resources)
    else:
        print("\nNo Path Found After Changes!")

if __name__ == "__main__":
    main()
```

OUTPUT:



2) Depth-First Search (DFS): Locating Hidden Water

- Narrative: During a drought, Mowgli must locate hidden water sources while avoiding predators and conserving energy.
- Task: Use DFS to explore all potential water sources in the jungle.
- Challenges:

  Time and energy constraints limit how far Mowgli can explore.

  Some water sources are guarded by other animals
- Extension:
  Introduce limited visibility that requires scouting before moving. Include a companion animal like Bagheera to help in identifying dangers.

A)

```python
import random

class Jungle:
    def __init__(self, size, water_locations, predators, energy):
        self.size = size
        self.grid = [[None for _ in range(size)] for _ in range(size)]
        self.water_locations = water_locations
        self.predators = predators
        self.energy = energy
        self.visibility = 2  # Mowgli can scout within a 2-cell radius
        self.found_water = []
        self.visited = set()

    def is_valid_move(self, x, y):
        return 0 <= x < self.size and 0 <= y < self.size and (x, y) not in self.visited

    def scout_area(self, x, y):
        """ Bagheera scouts the area within visibility range."""
        dangers = []
        for dx in range(-self.visibility, self.visibility + 1):
            for dy in range(-self.visibility, self.visibility + 1):
                nx, ny = x + dx, y + dy
                if (nx, ny) in self.predators:
                    dangers.append((nx, ny))
        return dangers

    def dfs(self, x, y, energy_left):
        if energy_left <= 0:
            return

        self.visited.add((x, y))
        print(f"Mowgli explores ({x}, {y}), Energy left: {energy_left}")
```

```
        if (x, y) in self.water_locations:
            print(f"Found water at ({x}, {y})!\n")
            self.found_water.append((x, y))

        dangers = self.scout_area(x, y)
        if dangers:
            print(f"Bagheera warns about predators at: {dangers}\n")
            return  # Avoid moving into danger

        # Explore in DFS manner: Up, Down, Left, Right
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if self.is_valid_move(nx, ny):
                self.dfs(nx, ny, energy_left - 1)

# Jungle configuration
size = 5
water_locations = {(4, 4), (1, 3), (3, 2)}
predators = {(2, 2), (4, 3)}
initial_energy = 10

# Initialize jungle and start DFS exploration
jungle = Jungle(size, water_locations, predators, initial_energy)
print("Starting DFS to find water sources...")
jungle.dfs(0, 0, jungle.energy)

print("\nWater sources found:", jungle.found_water)
```

OUTPUT:

```
Starting DFS to find water sources...
Mowgli explores (0, 0), Energy left: 10
Bagheera warns about predators at: [(2, 2)]


Water sources found: []
```

3. Depth-Limited Search (DLS): Gathering Jungle Fruits

- Narrative: Mowgli can climb trees to collect fruits, but his safety depends on staying within a safe height.
- Task: Use DLS to identify safe fruit-bearing trees and optimize collection.
- Challenges:
  Trees have varying fruit quantities and safety levels.
  Some trees have hidden dangers like snakes or wasp nests.
- Extension:
  Introduce seasonal variations where some fruits are ripe only in certain months.
  Include a time limit to prevent predators from noticing Mowgli.

A)

```python
import random
from collections import deque

# Tree Node class
class TreeNode:
    def __init__(self, id, fruit_quantity, is_safe, dangers=None, season=None):
        self.id = id  # Unique tree ID
        self.fruit_quantity = fruit_quantity  # Number of fruits available
        self.is_safe = is_safe  # If the tree is safe to climb
        self.dangers = dangers if dangers else []  # List of dangers (e.g., snakes, wasps)
        self.season = season  # The season in which fruits are available
        self.children = []  # Adjacent trees

    def add_child(self, child):
        self.children.append(child)

# Depth-Limited Search (DLS) Function
def depth_limited_search(node, depth, current_month, collected_fruits=0):
    if depth < 0:
        return collected_fruits  # Stop if depth limit is reached

    # Check if the tree is safe and if fruits are available this season
    if node.is_safe and (node.season is None or node.season == current_month):
        collected_fruits += node.fruit_quantity  # Collect fruits

    for danger in node.dangers:
        print(f"Warning! {danger} detected in tree {node.id}!")

    # Recur for children
    for child in node.children:
        collected_fruits = depth_limited_search(child, depth - 1, current_month, collected_fruits)

    return collected_fruits
```

```python
# Example Tree Setup
root = TreeNode(id=0, fruit_quantity=5, is_safe=True)
child1 = TreeNode(id=1, fruit_quantity=10, is_safe=True, season="Summer")
child2 = TreeNode(id=2, fruit_quantity=3, is_safe=False, dangers=["Snakes"])
child3 = TreeNode(id=3, fruit_quantity=7, is_safe=True, season="Winter")
child4 = TreeNode(id=4, fruit_quantity=12, is_safe=True)

root.add_child(child1)
root.add_child(child2)
child1.add_child(child3)
child1.add_child(child4)

# Parameters
max_depth = 2  # Maximum depth Mowgli can climb safely
current_month = "Summer"  # Change this to test seasonal variation

# Run DLS
fruits_collected = depth_limited_search(root, max_depth, current_month)
print(f"Total Fruits Collected: {fruits_collected}")
```

OUTPUT:

```
Warning! Snakes detected in tree 2!
Total Fruits Collected: 27
```

N. DEEPAK REDDY
VU22CSEN0100272

Uniform Cost Search (UCS): Safe Navigation

- **Narrative: Mowgli navigates a dense jungle while minimizing danger and conserving energy.**
- **Task: Use UCS to find the most efficient and safest path to a nearby shelter.**
- **Challenges:**
  - **Assign costs to terrains (e.g., muddy paths, steep cliffs).**
  - **Factor in resource availability along paths.**
- **Extension:**
  - **Add environmental hazards like storms or falling rocks that alter costs dynamically.**

CODE:

```
import heapq


# Define the jungle as a graph with costs
jungle_map = {
   'Start': [('A', 2), ('B', 5)],
   'A': [('C', 3), ('D', 4)],
   'B': [('D', 2), ('E', 6)],
   'C': [('Shelter', 5)],
   'D': [('Shelter', 2)],
   'E': [('Shelter', 4)],
   'Shelter': []  # Goal node
}


# UCS Algorithm
def uniform_cost_search(graph, start, goal):
   priority_queue = [(0, start, [])]  # (cost, node, path)
   visited = set()
```

```
    while priority_queue:
        cost, node, path = heapq.heappop(priority_queue)

        if node in visited:
            continue

        path = path + [node]
        visited.add(node)

        if node == goal:
            return (cost, path)

        for neighbor, edge_cost in graph.get(node, []):
            if neighbor not in visited:
                heapq.heappush(priority_queue, (cost + edge_cost, neighbor, path))

    return float("inf"), []  # No path found

# Run UCS to find the safest path
cost, safest_path = uniform_cost_search(jungle_map, 'Start', 'Shelter')
print("Safest Path:", safest_path, "with cost:", cost)
```

OUTPUT:

```
Safest Path: ['Start', 'A', 'D', 'Shelter'] with cost: 8

=== Code Execution Successful ===
```

Iterative Deepening Depth-First Search (IDDFS): Learning Jungle Routes

- **Narrative: Mowgli gradually learns safe jungle routes, starting near his den and expanding outward.**
- **Task: Use IDDFS to incrementally map key jungle regions and safe paths.**
- **Challenges:**
  - **Manage limited resources like food and water during exploration.**
  - **Avoid zones with high predator activity.**
- **Extension:**
  - **Introduce NPC animals (e.g., monkeys) that provide clues or misinformation about safe paths.**
  - **Include weather patterns that obscure routes temporarily.**

CODE:

```
def dls(graph, node, goal, depth, path, visited):
    if depth == 0 and node == goal:
        return path + [node]
    if depth > 0:
        visited.add(node)
        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                new_path = dls(graph, neighbor, goal, depth - 1, path + [node], visited)
                if new_path:
                    return new_path
        visited.remove(node)
    return None


def iddfs(graph, start, goal, max_depth):
```

```
    for depth in range(max_depth):
        visited = set()
        path = dls(graph, start, goal, depth, [], visited)
        if path:
            return path
    return None

# Jungle map as a graph
jungle_map = {
    'Den': ['A', 'B'],
    'A': ['C', 'D'],
    'B': ['D', 'E'],
    'C': ['Shelter'],
    'D': ['Shelter'],
    'E': ['Shelter'],
    'Shelter': []
}

# Running IDDFS
path = iddfs(jungle_map, 'Den', 'Shelter', max_depth=5)
print("Explored Path:", path)
```

OUTPUT:

```
Explored Path: ['Den', 'A', 'C', 'Shelter']

=== Code Execution Successful ===
```

Q) Greedy Best First Search (GBFS): Honeycomb Collection

- **Narrative: Baloo wants to collect honey from trees abundant in honeycombs.**
- **Task: Use GBFS to prioritize trees based on proximity and honey availability.**
- **Challenges:**
  - **Aggressive bees and nearby predators deter collection.**
  - **Some trees have higher yields but are more dangerous to approach.**
- **Extension:**
  - **Add a cooldown period after collecting honey due to bee stings.**

Introduce honey scarcity as seasons change

CODE:

```
import heapq

# Define the jungle trees as a graph with heuristic values (honey yield)
jungle_trees = {
  'Start': [('Tree1', 5), ('Tree2', 8)],
  'Tree1': [('Tree3', 6), ('Tree4', 3)],
  'Tree2': [('Tree4', 7), ('Tree5', 4)],
  'Tree3': [('Goal', 2)],
  'Tree4': [('Goal', 5)],
  'Tree5': [('Goal', 3)],
  'Goal': []  # Baloo's honey stash
}

# Honey yield heuristic (higher = better)
honey_heuristic = {
```

```
    'Start': 10, 'Tree1': 7, 'Tree2': 9, 'Tree3': 5, 'Tree4': 6, 'Tree5': 8, 'Goal': 0
}

# Greedy Best First Search (GBFS) Algorithm
def greedy_best_first_search(graph, start, goal, heuristic):
    priority_queue = [(heuristic[start], start, [])]  # (heuristic, node, path)
    visited = set()

    while priority_queue:
        _, node, path = heapq.heappop(priority_queue)

        if node in visited:
            continue

        path = path + [node]
        visited.add(node)

        if node == goal:
            return path

        for neighbor, _ in graph.get(node, []):
            if neighbor not in visited:
                heapq.heappush(priority_queue, (heuristic[neighbor], neighbor, path))

    return []  # No path found

# Run GBFS to find the best honey collection route
best_honey_path = greedy_best_first_search(jungle_trees, 'Start', 'Goal', honey_heuristic)
print("Best Honey Collection Path:", best_honey_path)
```

OUTPUT:

```
Best Honey Collection Path: ['Start', 'Tree1', 'Tree3', 'Goal']

=== Code Execution Successful ===
```

B.DEASHIK REDDY

VU22CSEN0100283

Q. Adversarial Search: Conflict with Shere Khan

- **Narrative**: Mowgli and Baloo confront Shere Khan to defend their jungle.
- **Task**: Use minimax with alpha-beta pruning to evaluate defensive strategies.
- **Challenges**:
  - Predict Shere Khan's tactics while optimizing counterattacks.
  - Balance aggressive and defensive actions.
- **Extension**:
  - Include changing environmental conditions like wildfires.
  - Add a time constraint requiring quick strategic decisions.

DESCRIPTION:

Minimax with Alpha-Beta Pruning Approach:

1. **Tree Construction:**
   - Root Node: Current state (positions, HP, environment).
   - Children: Possible actions leading to next states.

- Terminal States:
  - **Win:** Shere Khan is neutralized.
  - **Lose:** Mowgli is caught or injured beyond recovery.
  - **Draw:** Time runs out, forcing a retreat.

2. **Alpha-Beta Pruning:**
   - Prune branches where outcomes are already worse than a previously explored branch.
   - Saves computational time, making real-time decisions feasible

CODE:

```
import time
import random



class GameState:
    def __init__(self, mowgli_hp, shere_khan_hp, wildfire_spread, turn, max_depth=3):
        self.mowgli_hp = mowgli_hp
        self.shere_khan_hp = shere_khan_hp
        self.wildfire_spread = wildfire_spread
        self.turn = turn
        self.max_depth = max_depth

    def is_terminal(self):
        """Check if the game has reached a terminal state."""
        return self.mowgli_hp <= 0 or self.shere_khan_hp <= 0

    def evaluate(self):
        """Evaluation function for the minimax algorithm."""
```

```
    if self.mowgli_hp <= 0:
        return -100
    elif self.shere_khan_hp <= 0:
        return 100
    return self.mowgli_hp - self.shere_khan_hp + (10 if self.wildfire_spread < 3 else -10)

def get_possible_moves(self):
    """Generate possible moves based on the current turn."""
    if self.turn == "Mowgli":
        return ["attack", "dodge", "use_fire"]
    else:
        return ["attack", "ambush", "retreat"]

def apply_move(self, move):
    """Apply a move and return the resulting game state."""
    new_mowgli_hp = self.mowgli_hp
    new_shere_khan_hp = self.shere_khan_hp
    new_wildfire_spread = self.wildfire_spread + random.choice([0, 1])

    if self.turn == "Mowgli":
        if move == "attack":
            new_shere_khan_hp -= 10
        elif move == "dodge":
            new_mowgli_hp += 5
        elif move == "use_fire" and new_wildfire_spread > 2:
            new_shere_khan_hp -= 15

        next_turn = "Shere Khan"
    else:
        if move == "attack":
            new_mowgli_hp -= 10
        elif move == "ambush":
            new_mowgli_hp -= 15
        elif move == "retreat":
            new_shere_khan_hp += 5
```

```
            next_turn = "Mowgli"

        return GameState(new_mowgli_hp, new_shere_khan_hp, new_wildfire_spread, next_turn, self.max_depth)
def minimax(state, depth, alpha, beta, maximizing_player):
    if depth == 0 or state.is_terminal():
        return state.evaluate()
    if maximizing_player:
        max_eval = float('-inf')
        for move in state.get_possible_moves():
            eval_score = minimax(state.apply_move(move), depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval_score)
            alpha = max(alpha, eval_score)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = float('inf')
        for move in state.get_possible_moves():
            eval_score = minimax(state.apply_move(move), depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval_score)
            beta = min(beta, eval_score)
            if beta <= alpha:
                break
        return min_eval
def best_move(state):
    best_score = float('-inf')
    best_action = None
    start_time = time.time()

    for move in state.get_possible_moves():
        new_state = state.apply_move(move)
        move_score = minimax(new_state, state.max_depth, float('-inf'), float('inf'), False)

        if move_score > best_score:
            best_score = move_score
            best_action = move
```

```
        if time me() - start_time > 2:
            break

    return best_action

initial_state = GameState(mowgli_hp=100, shere_khan_hp=100, wildfire_spread=0, turn="Mowgli",
max_depth=3)

print("Initial State: Mowgli HP =", initial_state.mowgli_hp, "Shere Khan HP =", initial_state.shere_khan_hp)

while not initial_state.is_terminal():
    if initial_state.turn == "Mowgli":
        move = best_move(initial_state)
        print("\nMowgli chooses:", move)
    else:
        move = random.choice(initial_state.get_possible_moves())
        print("\nShere Khan chooses:", move)

    initial_state = initial_state.apply_move(move)
    print(f"State Update: Mowgli HP = {initial_state.mowgli_hp}, Shere Khan HP = {initial_state.shere_khan_hp},
Wildfire Spread = {initial_state.wildfire_spread}")

if initial_state.mowgli_hp <= 0:
    print("\nShere Khan wins!")
elif initial_state.shere_khan_hp <= 0:
    print("\nMowgli and Baloo win!")
```

EXPECTED OUTPUT:

Initial State: Mowgli HP = 100 Shere Khan HP = 100

Mowgli chooses: attack

State Update: Mowgli HP = 100, Shere Khan HP = 90, Wildfire Spread = 0

Shere Khan chooses: ambush

State Update: Mowgli HP = 85, Shere Khan HP = 90, Wildfire Spread = 1

Mowgli chooses: use_fire

State Update: Mowgli HP = 85, Shere Khan HP = 90, Wildfire Spread = 2

Shere Khan chooses: attack

State Update: Mowgli HP = 75, Shere Khan HP = 90, Wildfire Spread = 2

Mowgli chooses: attack

State Update: Mowgli HP = 75, Shere Khan HP = 80, Wildfire Spread = 3

Shere Khan chooses: retreat

State Update: Mowgli HP = 75, Shere Khan HP = 85, Wildfire Spread = 4

Mowgli chooses: use_fire

State Update: Mowgli HP = 75, Shere Khan HP = 70, Wildfire Spread = 4

Shere Khan chooses: attack

State Update: Mowgli HP = 65, Shere Khan HP = 70, Wildfire Spread = 5

Mowgli chooses: attack

State Update: Mowgli HP = 65, Shere Khan HP = 60, Wildfire Spread = 5

Mowgli chooses: attack

State Update: Mowgli HP = 40, Shere Khan HP = 0, Wildfire Spread = 7

Mowgli and Baloo win!

Q. CSP: Jungle Camp Setup

- **Narrative**: Mowgli sets up a camp balancing safety, proximity to resources, and comfort.
- **Task**: Use CSP techniques to allocate zones for shelter, food, and water.

- **Challenges**:
    - Handle overlapping constraints like limited space and predators nearby.
    - Adjust to expansion needs as more animals join the camp.
- **Extension**:
    - Introduce environmental hazards like floods requiring relocation.

Add NPC animals providing feedback on camp efficiency

DESCRIPTION:

Jungle Camp Setup as a CSP Problem

This problem can be formulated as a **Constraint Satisfaction Problem (CSP)** where Mowgli needs to assign zones for **shelter, food, and water** while ensuring safety, accessibility, and comfort.

CSP Formulation

Variables:

- **Shelter Zone (SZ)**
- **Food Zone (FZ)**
- **Water Zone (WZ)**

Each zone is placed within a jungle **grid (NxN)**.

Domain (Possible Assignments):

Each zone can be placed in specific grid locations, **but not all locations are valid** due to obstacles, predators, or terrain.

CODE:

```
from constraint import Problem, AllDifferentConstraint, InSetConstraint
import random


LOCATIONS = {1, 2, 3, 4, 5}


problem = Problem()


problem.addVariables(["SZ", "FZ", "WZ"], LOCATIONS)


SAFE_ZONES = {1, 3, 5}
PREDATOR_ZONES = {2, 4}


problem.addConstraint(InSetConstraint(SAFE_ZONES), ["SZ"])


def food_near_water(fz, wz):
    return abs(fz - wz) == 1

problem.addConstraint(food_near_water, ["FZ", "WZ"])


def shelter_away_from_predators(sz):
    return sz not in PREDATOR_ZONES and all(abs(sz - pz) > 1 for pz in PREDATOR_ZONES)
```

```
problem.addConstraint(shelter_away_from_predators, ["SZ"])



problem.addConstraint(AllDifferentConstraint())


)
def apply_flood():
    print("\n⚠ Flood Alert! Water must move to a new location.")
    global LOCATIONS
    new_water_zone = random.choice(list(LOCATIONS - {2}))
    print(f"🌊 Water relocated to Zone {new_water_zone}")
    return new_water_zone



solutions = problem.getSolutions()



if solutions:
    selected_solution = random.choice(solutions)
    print("\nInitial Camp Setup:")
    print(f"🏕 Shelter: Zone {selected_solution['SZ']}")
    print(f"🍎 Food: Zone {selected_solution['FZ']}")
    print(f"💧 Water: Zone {selected_solution['WZ']}")



    flooded_water_zone = apply_flood()
    selected_solution["WZ"] = flooded_water_zone



    efficiency = random.randint(60, 100)
    print("\n🐾 Jungle Animal Feedback:")
    print(f"🐒 'The camp setup is {efficiency}% efficient!'")
else:
    print("No valid camp setup found!")
```

EXPECTED OUTPUT:

Initial Camp Setup:

🏕️ Shelter: Zone 3

🍏 Food: Zone 4

💧 Water: Zone 5

⚠️ Flood Alert! Water must move to a new location.

🌊 Water relocated to Zone 1

🐾 Jungle Animal Feedback:

 'The camp setup is 85% efficient!'

Q. First-Order Logic: Jungle Knowledge Base

- **Narrative**: Mowgli builds a knowledge base to catalog jungle animals, terrains, and resources.
- **Task**: Use first-order logic to answer queries like “Which paths avoid predators?”
- **Challenges**:
  - Maintain logical consistency across dynamic data updates.
  - Handle hypothetical queries involving multiple variables.
- **Extension**:
  - Include the ability to predict outcomes of hypothetical scenarios.
  - Add a reward system for successful discovery of hidden resources.

DESCRIPTION:

1. **Define Facts & Rules using FOL**

- Predators and safe animals
- Terrain types and dangers
- Resource locations (water, food)
- Paths and connectivity

2. **Use Logic Programming (pyDatalog)**
   - Encode knowledge using **facts** and **rules**
   - Answer **queries dynamically**
3. **Handle Updates & Hypothetical Queries**
   - **Adding new predators** updates paths
   - **Hypothetical queries predict outcomes**
4. **Include a Reward System**
   - Mowgli earns **points for finding safe paths**

CODE:

```
from pyDatalog import pyDatalog


pyDatalog.clear()


pyDatalog.create_terms('Animal, Predator, Safe_Animal, is_predator, is_safe')


+ is_predator('Shere Khan')
+ is_predator('Kaa')
+ is_safe('Baloo')
+ is_safe('Bagheera')


pyDatalog.create_terms('Path, Terrain, SafePath, has_predator, near_water, near_food,
is_safe_path')


+ Path('A', 'B')
```

```
+ Path('B', 'C')
+ Path('C', 'D')
+ Path('A', 'E')
+ Path('E', 'D')

+ has_predator('B', 'Shere Khan')
+ has_predator('C', 'Kaa')

is_safe_path(X, Y) <= Path(X, Y) & ~has_predator(X, 'Shere Khan') & ~has_predator(X, 'Kaa')

print("\n✅ Safe Paths in the Jungle:")
print(is_safe_path(X, Y))

pyDatalog.create_terms('Resource, Found_Resource, Reward, gives_reward')

+ Resource('Water', 'C')
+ Resource('Food', 'E')

gives_reward(X, 10) <= Found_Resource(X)

+ Found_Resource('Water')

print("\n🏆 Reward Points Earned:")
print(gives_reward(X, R))

def add_new_predator(location, predator):
    print(f"\n⚠️ Warning! {predator} spotted at {location}. Updating knowledge base...")
    + has_predator(location, predator)
    print("Updated Safe Paths:")
    print(is_safe_path(X, Y))

add_new_predator('E', 'New Tiger')
```

EXPECTED OUTPUT:

✅ Safe Paths in the Jungle:

X | Y

A | B

A | E

🏆 Reward Points Earned:

X | R

Water | 10

⚠️ Warning! New Tiger spotted at E. Updating knowledge base...

Updated Safe Paths:

X | Y

A | B

VU22CSEN0100514

P VENKATA KRISHNA

Q) ***A Search: Strategic Jungle Exploration****

- **Narrative**: Mowgli plans an efficient exploration route balancing resource collection and safety.
- **Task**: Use A* search to compute the optimal route across jungle zones.
- **Challenges**:
  - Define heuristics combining resource abundance, safety, and distance.
  - Factor in environmental changes like heavy rain.
- **Extension**:
  - Add multiple objectives like rescuing animals or gathering specific resources.
  - Include friendly animals providing guidance on optimal routes.

CODE:

```
import heapq

class JungleNode:
    def __init__(self, name, x, y, resources, safety, neighbors):
        self.name = name
        self.x = x
        self.y = y
        self.resources = resources  # Higher is better
        self.safety = safety  # Higher is safer
        self.neighbors = neighbors  # List of tuples (neighbor_name, cost)

    def heuristic(self, target, rain_factor=1):
        # Heuristic combines distance, resource abundance, and safety
        distance = abs(self.x - target.x) + abs(self.y - target.y)
        resource_score = self.resources
        safety_score = self.safety * rain_factor  # Rain affects safety
        return distance - resource_score + (1 / (safety_score + 0.1))  # Avoid division
            by zero
def a_star_search(start, goal, jungle_map, rain_factor=1):
    open_set = []
    heapq.heappush(open_set, (0, start.name))
    came_from = {}
    g_score = {node: float('inf') for node in jungle_map}
    g_score[start.name] = 0
    f_score = {node: float('inf') for node in jungle_map}
    f_score[start.name] = start.heuristic(goal, rain_factor)

    while open_set:
        _, current_name = heapq.heappop(open_set)
        current = jungle_map[current_name]
```

```
        if current == goal:
            path = []
            while current_name in came_from:
                path.append(current_name)
                current_name = came_from[current_name]
            path.append(start.name)
            return path[::-1]

        for neighbor_name, cost in current.neighbors:
            tentative_g_score = g_score[current.name] + cost
            if tentative_g_score < g_score[neighbor_name]:
                came_from[neighbor_name] = current.name
                g_score[neighbor_name] = tentative_g_score
                f_score[neighbor_name] = tentative_g_score + jungle_map[neighbor_name]
                    .heuristic(goal, rain_factor)
                heapq.heappush(open_set, (f_score[neighbor_name], neighbor_name))

    return None  # No path found
# Define jungle nodes with resources, safety, and neighbors
jungle_map = {
    "Start": JungleNode("Start", 0, 0, 5, 8, [("River", 2), ("Hill", 3)]),
    "River": JungleNode("River", 1, 0, 8, 6, [("Cave", 4)]),
    "Hill": JungleNode("Hill", 0, 1, 6, 7, [("Cave", 2), ("Clearing", 5)]),
    "Cave": JungleNode("Cave", 2, 1, 3, 9, [("Goal", 3)]),
    "Clearing": JungleNode("Clearing", 1, 2, 7, 5, [("Goal", 2)]),
    "Goal": JungleNode("Goal", 2, 2, 10, 10, [])
}
# Simulate rain effect
rain_factor = 0.7  # Decreases safety scores
optimal_path = a_star_search(jungle_map["Start"], jungle_map["Goal"], jungle_map,
    rain_factor)
print("Optimal Path:", optimal_path)
```

OUTPUT:

```
Output                                                    Clear
Optimal Path: ['Start', 'Hill', 'Clearing', 'Goal']

=== Code Execution Successful ===
```

Q) Recursive Best-First Search (RBFS): Adaptive Predator Avoidance

- **Narrative**: Mowgli evades predators like Shere Khan and Kaa while navigating the jungle.
- **Task**: Use RBFS to adapt paths dynamically based on predator locations.
- **Challenges**:
    - Balance between stealth and speed to avoid detection.
    - Manage recalculations under memory constraints.
- **Extension**:
    - Introduce fake predator tracks requiring decision-making under uncertainty.
    - Add sudden predator encounters requiring quick path adjustments.

CODE:

```
import heapq

class JunglePathfinding:
    def __init__(self, graph, predators, start, goal):
        self.graph = graph  # Jungle map as adjacency list
        self.predators = predators  # Locations of predators
        self.start = start  # Starting point
        self.goal = goal  # Destination

    def heuristic(self, node):
        # Simple heuristic: Manhattan distance + predator penalty
        x1, y1 = node
        x2, y2 = self.goal
        distance = abs(x1 - x2) + abs(y1 - y2)
        predator_penalty = 5 if node in self.predators else 0
        return distance + predator_penalty

    def rbfs(self, node, f_limit, visited):
        if node == self.goal:
            return [node], 0  # Goal reached

        successors = []
        for neighbor, cost in self.graph.get(node, []):
            if neighbor in visited:
                continue
            g_value = cost
            f_value = g_value + self.heuristic(neighbor)
            heapq.heappush(successors, (f_value, neighbor, g_value))

        if not successors:
            return None, float("inf")
```

```
        while successors:
            successors.sort()
            best_f, best_node, best_g = successors[0]

            if best_f > f_limit:
                return None, best_f

            alternative = successors[1][0] if len(successors) > 1 else float("inf")
            visited.add(best_node)
            path, new_f = self.rbfs(best_node, min(f_limit, alternative), visited)
            if path:
                return [node] + path, new_f

            successors[0] = (new_f, best_node, best_g)

        return None, float("inf")

    def find_path(self):
        path, _ = self.rbfs(self.start, float("inf"), set())
        return path

# Define the jungle map as an adjacency list
jungle_map = {
    (0, 0): [((0, 1), 1), ((1, 0), 1)],
    (0, 1): [((0, 0), 1), ((1, 1), 1), ((0, 2), 1)],
    (1, 0): [((0, 0), 1), ((2, 0), 1)],
    (1, 1): [((0, 1), 1), ((1, 2), 1), ((2, 1), 1)],
    (0, 2): [((0, 1), 1), ((1, 2), 1)],
    (1, 2): [((0, 2), 1), ((1, 1), 1), ((2, 2), 1)],
    (2, 0): [((1, 0), 1), ((2, 1), 1)],
    (2, 1): [((2, 0), 1), ((1, 1), 1), ((2, 2), 1)],
    (2, 2): [((1, 2), 1), ((2, 1), 1)],
}
```

```
# Define predator locations (Shere Khan, Kaa)
predators = {(1, 1), (2, 0)}

# Start and goal positions
start = (0, 0)
goal = (2, 2)

# Execute RBFS pathfinding
pathfinder = JunglePathfinding(jungle_map, predators, start, goal)
path = pathfinder.find_path()
print("Optimal Path for Mowgli:", path)
```

OUTPUT:

```
Output                                                    Clear
Optimal Path for Mowgli: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2)]
```

Q) Random Restart Hill Climbing: Jungle Territory Optimization

- **Narrative**: Wolves optimize their territory for safety and resource accessibility.
- **Task**: Use hill climbing to maximize benefits from selected zones.
- **Challenges**:
    - Some zones become unsuitable due to predator activity.
    - Rival packs may claim overlapping territories.
- **Extension**:
    - Introduce seasonal migrations altering territory suitability.
    - Add interactions with other jungle inhabitants impacting territory choice.

CODE:

```
import random

def evaluate_territory(territory, predator_zones, resource_zones):
    """   Evaluate the quality of a given territory.
    Higher scores indicate better suitability based on safety and resources.   """
    safety_score = sum(1 for zone in territory if zone not in predator_zones)
    resource_score = sum(1 for zone in territory if zone in resource_zones)
    return safety_score + resource_score
def get_neighbors(territory, all_zones):
    """
    Generate neighboring territories by adding or removing a zone.
    """
    neighbors = []
    for zone in all_zones:
        if zone in territory:
            new_territory = territory - {zone}
        else:
            new_territory = territory | {zone}
        neighbors.append(new_territory)
    return neighbors
def hill_climb(all_zones, predator_zones, resource_zones, max_iterations=100):
    """
    Perform the hill climbing algorithm to optimize wolf pack territory.
    """
    current_territory = set(random.sample(all_zones, k=random.randint(2, len(all_zones
        )//2)))
    current_score = evaluate_territory(current_territory, predator_zones, resource_zones)

    for _ in range(max_iterations):
        neighbors = get_neighbors(current_territory, all_zones)
        best_neighbor = max(neighbors, key=lambda t: evaluate_territory(t, predator_zones
            , resource_zones))
        best_score = evaluate_territory(best_neighbor, predator_zones, resource_zones)
```

```
        if best_score <= current_score:
            break  # Stop if no improvement

        current_territory, current_score = best_neighbor, best_score

    return current_territory, current_score

def random_restart_hill_climb(all_zones, predator_zones, resource_zones, restarts=5):
    """
    Perform hill climbing with multiple restarts to avoid local optima.
    """
    best_territory = set()
    best_score = float('-inf')

    for _ in range(restarts):
        territory, score = hill_climb(all_zones, predator_zones, resource_zones)
        if score > best_score:
            best_territory, best_score = territory, score

    return best_territory, best_score

# Jungle zones setup
all_zones = {"Zone A", "Zone B", "Zone C", "Zone D", "Zone E", "Zone F", "Zone G"}
predator_zones = {"Zone B", "Zone D"}  # Areas with high predator activity
resource_zones = {"Zone A", "Zone E", "Zone G"}  # Zones with good resources

# Run optimization
optimal_territory, optimal_score = random_restart_hill_climb(all_zones, predator_zones,
    resource_zones)
print("Optimal Territory:", optimal_territory)
print("Optimal Score:", optimal_score)
```

OUTPUT:

```
Output                                                    Clear

Optimal Territory: {'Zone A', 'Zone E', 'Zone F', 'Zone G'}
Optimal Score: 6
```