



**VIT<sup>®</sup>**  
**Vellore Institute of Technology**  
(Deemed to be University under section 3 of UGC Act, 1956)

***COURSE CODE: ARTIFICIAL INTELLIGENCE LAB***

***COURSE TITLE: BITE308P***

***SLOT: L45+L46***

***FACULTY:10919 - SUBHASHINI R – SCORE***

***NAME: POLI VARDHINI REDDY***

***REGISTER NUMBER: 21BIT0382***

## problem statement for the Wumpus World scenario:

The Wumpus World is a cave system with 25 rooms arranged in a 5x5 grid. Each room is connected to its adjacent rooms via passages. The cave contains hazards such as bottomless pits, a ferocious Wumpus, and a valuable treasure. The knowledge-based agent starts its journey from Room[1, 1].

The cave environment is as follows:

- Pits: Randomly scattered throughout the cave, pits pose a lethal threat to the agent. If the agent falls into a pit, it is stuck there and loses the game.
- \*\*Wumpus\*\*: The Wumpus resides in a single room and remains stationary throughout the game. If the agent enters the room containing the Wumpus, it is devoured and the game ends immediately.
- \*\*Treasure\*\*: Hidden in one of the rooms, the treasure awaits discovery by the agent. The goal of the agent is to locate the treasure and safely exit the cave with it.
- \*\*Supporting Elements\*\*:
  - \*\*Stench\*\*: The rooms adjacent to the Wumpus's location emit a foul stench, providing a clue to its presence.
  - \*\*Breeze\*\*: Rooms adjacent to pits have a mild breeze, alerting the agent to their danger.
  - \*\*Glitter\*\*: The room containing the treasure emits a faint glow, indicating its presence.

The agent's objectives and constraints are as follows:

- \*\*Goal\*\*: Retrieve the treasure and exit the cave safely.
- \*\*Reward\*\*: Successfully retrieving the treasure and exiting the cave results in a reward for the agent.
- \*\*Penalties\*\*:
  - Falling into a pit incurs a penalty.
  - Being eaten by the Wumpus results in immediate failure and a penalty.
- \*\*Special Action\*\*: The agent is equipped with a single arrow that it can use to kill the Wumpus if it encounters it directly. Upon being shot, the Wumpus emits a loud scream, alerting the agent to its demise.

The agent must navigate the cave, avoid hazards, use available information about the environment (such as stench and breezes) to make informed decisions, and potentially utilize its arrow strategically to achieve its goal of obtaining the treasure and exiting the cave safely.

Below is the implementation of the Wumpus World problem scenario described in the problem statement, including the Agent and Environment classes in Java:

```
import java.util.Random;
```

```
class Agent {
```

```
    int x, y; // Agent's current position
```

```
    boolean hasArrow; // Indicates if the agent has the arrow
```

```
    Agent(int x, int y) {
```

```
        this.x = x;
```

```
        this.y = y;
```

```
        this.hasArrow = true; // Agent starts with the arrow
```

```
    }
```

```
    // Move the agent
```

```
    void move(int newX, int newY) {
```

```
        this.x = newX;
```

```
        this.y = newY;
```

```
    }
```

```
    // Shoot the arrow
```

```
    void shoot() {
```

```
        this.hasArrow = false;
```

```
    }
```

```
}
```

```
class Environment {
```

```
    int[][] grid; // 2D array to represent the grid
```

```
    Agent agent;
```

```
    int size;
```

```
int wumpusX, wumpusY; // Wumpus position
```

```
Environment(int size) {
```

```
    this.size = size;
```

```
    this.grid = new int[size][size];
```

```
    this.agent = new Agent(0, 0); // Agent starts at (0,0)
```

```
    initializeGrid();
```

```
}
```

```
// Initialize the grid with pits, Wumpus, and gold
```

```
void initializeGrid() {
```

```
    Random random = new Random();
```

```
    // Place pits
```

```
    for (int i = 0; i < size; i++) {
```

```
        for (int j = 0; j < size; j++) {
```

```
            if (random.nextDouble() < 0.2) { // Probability of pit in each cell
```

```
                grid[i][j] = -1; // -1 represents a pit
```

```
            }
```

```
        }
```

```
    }
```

```
    // Place Wumpus
```

```
    wumpusX = random.nextInt(size);
```

```
    wumpusY = random.nextInt(size);
```

```
    grid[wumpusX][wumpusY] = -2; // -2 represents Wumpus
```

```
    // Place gold
```

```
    int goldX = random.nextInt(size);
```

```
    int goldY = random.nextInt(size);
```

```
    grid[goldX][goldY] = 1; // 1 represents gold
```

```
}
```

```

// Check if the agent has won the game
boolean hasWon() {
    return grid[agent.x][agent.y] == 1;
}

// Check if the agent has lost the game
boolean hasLost() {
    return grid[agent.x][agent.y] == -1 || (agent.x == wumpusX && agent.y == wumpusY);
}

// Move the agent
void moveAgent(int newX, int newY) {
    if (newX >= 0 && newX < size && newY >= 0 && newY < size) {
        agent.move(newX, newY);
    }
}

// Shoot the arrow
void shootArrow(int x, int y) {
    if (agent.hasArrow) {
        if ((x == wumpusX && Math.abs(y - wumpusY) == 1) || (y == wumpusY && Math.abs(x - wumpusX) == 1)) {
            System.out.println("Wumpus killed!");
            grid[wumpusX][wumpusY] = 0; // Remove Wumpus from the grid
            agent.shoot();
        } else {
            System.out.println("Missed!");
        }
    } else {
        System.out.println("Agent has no arrow!");
    }
}

```

```
}  
}
```

```
public class WumpusWorld {  
    public static void main(String[] args) {  
        int size = 5; // Size of the grid  
        Environment environment = new Environment(size);  
  
        // Play the game until the agent wins or loses  
        while (!environment.hasWon() && !environment.hasLost()) {  
            // Perform agent actions here  
            // For example, move the agent  
            int newX = environment.agent.x + 1; // Move right  
            int newY = environment.agent.y;  
            environment.moveAgent(newX, newY);  
  
            // Print the current state of the grid  
            printGrid(environment.grid, environment.agent);  
        }  
  
        if (environment.hasWon()) {  
            System.out.println("Congratulations! You have found the gold!");  
        } else {  
            System.out.println("Game over! You lost.");  
        }  
    }  
  
    // Utility method to print the grid  
    private static void printGrid(int[][] grid, Agent agent) {  
        for (int i = 0; i < grid.length; i++) {  
            for (int j = 0; j < grid[0].length; j++) {
```

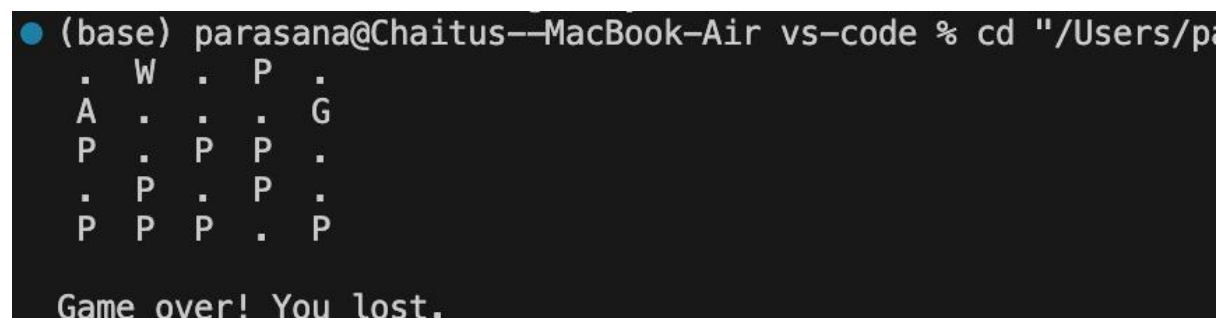
```

        if (i == agent.x && j == agent.y) {
            System.out.print(" A "); // Print agent
        } else if (grid[i][j] == -1) {
            System.out.print(" P "); // Print pit
        } else if (grid[i][j] == -2) {
            System.out.print(" W "); // Print Wumpus
        } else if (grid[i][j] == 1) {
            System.out.print(" G "); // Print gold
        } else {
            System.out.print(" . "); // Empty cell
        }
    }
    System.out.println();
}
System.out.println();
}
}

```

This code provides a simple simulation of the Wumpus World environment and agent interaction. The Environment class represents the cave system and the Agent class represents the knowledge-based agent navigating the cave.

OUTPUT:



```

● (base) parasana@Chaitus--MacBook-Air vs-code % cd "/Users/p
. W . P .
A . . . G
P . P P .
. P . P .
P P P . P
Game over! You lost.

```

## Water Jug Problem

**How to Approach the Solution?** Now that we have chosen what to work with, let's understand how you can actually make it work. Well, for starters you can have (a, b) which represents the

amount of water currently in jug 1 and jug 2 respectively. Initially, both the components will be (0, 0) since the jugs are empty in the beginning. The final state of the jugs will be either (0, d) or (d, 0) as both add up to give a total of the required quantity. The following operations can be performed on the jugs: 1. Empty a jug (a, b) -> (0, b) 2. Fill a jug (0, b) -> (a, b) 3. Transfer water from one jug to another.

#### Code:

This code finds the minimum number of steps required to achieve a target amount of water in one of the jugs, given the capacities of the jugs. It utilizes breadth-first search (BFS) to explore all possible states of the water in the jugs and determines the shortest path to reach the target amount.

```
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Set;

class State {
    int x, y; // Amount of water in each jug
    int steps; // Number of steps taken to reach this state

    State(int x, int y, int steps) {
        this.x = x;
        this.y = y;
        this.steps = steps;
    }
}

public class WaterJugProblem {
    public static int minStepsToReachTarget(int m, int n, int d) {
        Queue<State> queue = new LinkedList<>();
        Set<String> visited = new HashSet<>();

        // Start with both jugs empty
```



```

State initialState = new State(0, 0, 0);
queue.offer(initialState);
visited.add("0-0");

while (!queue.isEmpty()) {
    State currentState = queue.poll();

    // Check if the target amount is reached
    if (currentState.x == d || currentState.y == d) {
        return currentState.steps;
    }

    // Fill jug 1
    if (currentState.x < m && !visited.contains((m + "-" + currentState.y))) {
        queue.offer(new State(m, currentState.y, currentState.steps + 1));
        visited.add((m + "-" + currentState.y));
    }

    // Fill jug 2
    if (currentState.y < n && !visited.contains((currentState.x + "-" + n))) {
        queue.offer(new State(currentState.x, n, currentState.steps + 1));
        visited.add((currentState.x + "-" + n));
    }

    // Empty jug 1
    if (currentState.x > 0 && !visited.contains("0-" + currentState.y)) {
        queue.offer(new State(0, currentState.y, currentState.steps + 1));
        visited.add("0-" + currentState.y);
    }

    // Empty jug 2

```

```

        if (currentState.y > 0 && !visited.contains(currentState.x + "-0")) {
            queue.offer(new State(currentState.x, 0, currentState.steps + 1));
            visited.add(currentState.x + "-0");
        }

        // Pour from jug 1 to jug 2
        int pourAmount = Math.min(currentState.x, n - currentState.y);

        if (pourAmount > 0 && !visited.contains((currentState.x - pourAmount) + "-" + (currentState.y
+ pourAmount))) {
            queue.offer(new State(currentState.x - pourAmount, currentState.y + pourAmount,
currentState.steps + 1));
            visited.add((currentState.x - pourAmount) + "-" + (currentState.y + pourAmount));
        }

        // Pour from jug 2 to jug 1
        pourAmount = Math.min(m - currentState.x, currentState.y);

        if (pourAmount > 0 && !visited.contains((currentState.x + pourAmount) + "-" + (currentState.y
- pourAmount))) {
            queue.offer(new State(currentState.x + pourAmount, currentState.y - pourAmount,
currentState.steps + 1));
            visited.add((currentState.x + pourAmount) + "-" + (currentState.y - pourAmount));
        }
    }

    // If target cannot be reached
    return -1;
}

public static void main(String[] args) {
    int m = 3; // Capacity of jug 1
    int n = 5; // Capacity of jug 2
    int d = 4; // Target amount of water

```

```
int minSteps = minStepsToReachTarget(m, n, d);  
if (minSteps != -1) {  
    System.out.println("Minimum steps required to reach " + d + " units of water: " + minSteps);  
} else {  
    System.out.println("Target amount cannot be reached with given jug capacities.");  
}  
}  
}
```

#### **OUTPUT:**

**int m = 3; // Capacity of jug 1**

**int n = 5; // Capacity of jug 2**

**int d = 4; // Target amount of water**

**This means that it takes 6 steps to achieve 4 units of water in one of the jugs with capacities 3 and 5.**

**Minimum steps required to reach 4 units of water: 6**