Tutorial of OOM and Pathfinding Algorithm

This tutorial will cover two parts: OOM and Pathfinding algorithm. In each part, basic concepts will be first introduced, and code practice will be followed. All code demos can be found from GitHub (**https://github.com/Enigma-li/ILP23-24_tutorial**).

Part One: Object-oriented Modeling/Programming (OOM/OOP)

1. Explain the concept of OOP (refer to the lecture slides)

2. Explain the four principles, especially Inheritance and Abstraction, since students are not quite clear how to implement inheritance and abstraction, e.g., the method overriding or implementing, the abstract class and interface class.

3. Code practice

Scenario: Building a Zoo Management System

In this scenario, you are tasked with developing a Java application for managing a zoo. The zoo is home to various types of animals, each with unique attributes and behaviors. Your application should demonstrate the core principles of object-oriented programming.

Classes and Concepts:

1. Animal (Abstraction): Create an abstract class **'Animal'** that serves as the base class for all animals in the zoo. This class should have attributes like **'name', 'age',** and **'gender'**. Define an abstract method **'makeSound()'** that will be implemented by each specific animal type.

2. Encapsulation: Ensure that the **'name', 'age',** and **'gender'** attributes in the **'Animal'** class are encapsulated using private access modifiers and provide public getter and setter methods for each attribute.

3. Inheritance: Create several concrete animal classes that are inherited from the **'Animal'** class, such as **'Lion', 'Elephant'**, and **'Giraffe'**. Each of these classes should implement the **'makeSound()'** method to produce the corresponding sound of the animal.

4. Polymorphism: Implement a method in the main class to display information about each animal in the zoo. This method should take an array or list of **'Animal'** objects as input and use polymorphism to call the **'makeSound()'** method for each animal.

Sample Code Structure: download from GitHub

- ***Task1***: Please complete the sample code and have a test.

Scenario Extension: Adding More Animal Types and Behaviors

In this extension, you'll add additional animal types and behaviors to your zoo management system, allowing students to practice inheritance and polymorphism more extensively.

Classes and Concepts:

1. Animal Types (Inheritance): Introduce new animal types such as **'Tiger', 'Penguin'**, and **'Kangaroo'** as subclasses of the **'Animal'** class. Each animal type should have its unique attributes and behaviors. For example, a **'Tiger'** can have a **'roar()'** method, while a **'Penguin'** can have a **'swim()'** method.

2. Polymorphism: Implement a method in the main class that simulates various interactions in the zoo. This method should take an array or list of **'Animal'** objects, and it should demonstrate different actions that animals can perform, such as making sounds, eating, and special behaviors unique to their type (e.g., swimming, flying, climbing).

Sample Code Structure (Extension): download from GitHub.

- ***Task2***: Please complete the sample code and have a test.

Updated Scenario: Using Interfaces for Abstraction

In this updated scenario, you'll define interfaces for specific behaviors that animals can exhibit. This will further emphasize abstraction and polymorphism.

Interfaces and Concepts:

1. **Animal Interface (Abstraction)**: Create an interface called **'Animal'** that defines a method for making a sound (e.g., **'makeSound()'**). All animal classes should implement this interface.

2. **SpecialBehaviors Interface (Abstraction)**: Create a new interface called **'SpecialBehaviors'** that defines methods for special actions such as swimming, flying, or jumping. Not all animals will implement this interface, but those that do will provide specific implementations for their unique behaviors.

3. **Implement Interfaces**: Modify the animal classes to implement the **'Animal'** interface. Additionally, make specific animal classes implement the **'SpecialBehaviors'** interface if they exhibit unique behaviors.

Sample Code Structure (Updated): download from GitHub.

- *Task3*: Please complete the sample code and have a test.

Updated Scenario2: Using an Abstract Class and Interface Class for Abstraction

In this updated scenario, we'll define an abstract class for the Animal class, which will include common attributes and behavior. Individual animal types will extend this abstract class and provide specific implementations.

Abstract Class and Concepts:

1. **Animal Abstract Class (Abstraction)**: Create an abstract class **'Animal'** that defines attributes like **'name', 'age',** and **'gender'**, and includes an abstract method **'makeSound()'**. This abstract class serves as a common base for all animal types.

2. **SpecialBehaviors Interface (Abstraction)**: Continue using the **'SpecialBehaviors'** interface to define methods for special actions, as some animals may implement unique behaviors.

Sample Code Structure (Updated): download from GitHub.

- *Task4*: Please complete the sample code and have a test.

Part Two: A* algorithm

Task One - Solving the Shortest Path in a Maze using A* Algorithm

Problem Description:

Imagine you have a 2D maze represented as a grid, where some cells are walls (impassable), and others are open paths. You need to find the shortest path from a start cell to a goal cell while avoiding the walls. You'll guide your students through solving this problem using the A* algorithm.

Step-by-Step Tutorial:

1. Maze Representation:

Start by representing the maze as a 2D grid, where each cell can be either:

- Open path (denoted as '.')
- Wall (denoted as '#')
- Start point (denoted as 'S')
- Goal point (denoted as 'G')

2. A* Algorithm Implementation:

Teach your students how to implement the A* algorithm. This algorithm requires the following components:

- Priority Queue (Min Heap) for open nodes
- Closed list to track explored nodes
- Evaluation function (f-cost) for each cell

3. Pseudocode:

Provide a pseudocode for the A* algorithm. Here's a simple example:

```
function AStar(maze, start, goal):
  openSet = {start}
  cameFrom = {}
  gScore = {cell: infinity for cell in maze}
  gScore[start] = 0
  fScore = {cell: infinity for cell in maze}
  fScore[start] = heuristic(start, goal)

  while openSet is not empty:
    current = cell in openSet with lowest fScore
    if current == goal:
      return reconstructPath(cameFrom, current)
    openSet.remove(current)
    for neighbor in getNeighbors(current):
      tentativeGScore = gScore[current] + distance(current, neighbor)
      if tentativeGScore < gScore[neighbor]:
        cameFrom[neighbor] = current
        gScore[neighbor] = tentativeGScore
        fScore[neighbor] = gScore[neighbor] + heuristic(neighbor, goal)
        if neighbor not in openSet:
          openSet.add(neighbor)
  return "No path found"
```

4. Heuristic Function:

Explain what a heuristic function is and provide an example. Common heuristics include Manhattan distance, Euclidean distance, and diagonal distance.

5. Visualization:

Use a graphical tool or code to visualize the A* algorithm as it progresses through the maze. This will help students understand how the algorithm explores different paths. **For Jave, just print the results maze**.

6. Implementation:

Ask students to implement the A* algorithm to find the shortest path from 'S' to 'G' in a given maze. They can use the pseudocode as a guideline.

7. Testing:

Provide different maze scenarios for students to test their A* algorithm implementation. Ensure that they consider edge cases, such as mazes with no solution or mazes with multiple paths.

8. Performance Analysis:

Discuss the time and space complexity of the A* algorithm, as well as its advantages and limitations. Help students understand when to use A* and when to consider alternative algorithms.

Example Input:

Map:
```
############################################################
#..........................................................#
#.............................#............................#
#.............................#............................#
#.............................#............................#
#.......S.....................#............................#
#.............................#............................#
#.............................#............................#
#.............................#............................#
#.............................#............................#
#.............................#............................#
#.............................#............................#
#.............................#............................#
#######.#######################################............#
#....#........#............................................#
#....#........#............................................#
#....##########............................................#
#..........................................................#
#..........................................................#
#..........................................................#
#..........................................................#
#..........................................................#
#...............................##############.............#
#...............................#........G...#.............#
#...............................#............#.............#
#...............................#............#.............#
#...............................#............#.............#
#...............................###########..#.............#
#..........................................................#
#..........................................................#
############################################################
```

Sample Code: download from GitHub.

- *Task*: implement the find shortest path function

Extension - Solving the 15-puzzle problem.

The 15-puzzle, also known as the "Game of Fifteen," is a sliding puzzle that consists of a 4x4 grid with 15 numbered tiles and one empty space. The objective of the game is to rearrange the tiles by sliding them into the empty space to achieve a specific goal configuration, typically with the tiles arranged in ascending order, starting from the top-left corner.

The rules for the 15-puzzle are as follows:

- You can only slide a tile into the empty space, and you can't move tiles diagonally.
- The puzzle is typically solved when the tiles are arranged in ascending order from left to right, top to bottom, with the empty space in the bottom-right corner.
- The tiles can be shuffled into various configurations, making it a challenging puzzle.

The 15-puzzle is a classic example of a sliding puzzle and is often used as a recreational puzzle game and a simple problem for teaching algorithms like search and heuristic methods, such as the A* algorithm, to find an optimal solution. Solving the 15-puzzle can be done through a sequence of moves that minimize a specific cost function or heuristic.

Example:

Start Status:

5	1	2	4
9	6	3	8
13	15	10	11
14	0	7	12

Goal Status:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

General Tips:

You can treat the status after one movement as a grid position, from the start status, you can move tiles to achieve the goal status.

g function: calculate the moving cost from the start status to the current status h function: calculate the number of misplaced tiles

Sample Code: download from GitHub.

- *Task*: implement the find shortest path function.