### Topic 7 Walkthrough

#### 1. Image Processing

**Overview:** The image\_processing function performs basic image processing tasks on an input image. It converts the image to grayscale, applies a binary threshold, and detects edges using the Canny edge detection algorithm.

**Detailed Pseudo Code:**

FUNCTION image\_processing(image\_path)  
 READ image from 'image\_path' using cv2.imread  
 CONVERT image to grayscale using cv2.cvtColor with cv2.COLOR\_BGR2GRAY  
 APPLY binary threshold on grayscale image using cv2.threshold with a threshold value of 127  
 DETECT edges in the grayscale image using cv2.Canny with thresholds 100 and 200  
 STORE grayscale, thresholded, and edges images in a dictionary 'processed\_images'  
 RETURN 'processed\_images'  
END FUNCTION

**Implementation Guide:** - Start by importing cv2 for OpenCV functions and numpy for array manipulations. - Use cv2.imread() to load the image from the given path. - Convert the loaded image to grayscale and apply a binary threshold. - Use the Canny edge detection algorithm to detect edges in the grayscale image. - Return a dictionary containing the grayscale, thresholded, and edges images for further analysis or visualization.

#### 2. Sensory Processing

**Overview:** The sensory\_processing function analyzes sensor data to identify obstacles based on a distance threshold. It’s a simplified model of how a robotic system might process sensory input to navigate an environment.

**Detailed Pseudo Code:**

FUNCTION sensory\_processing(sensor\_data)  
 INITIALIZE 'obstacle\_threshold' with a value (e.g., 1.5)  
 IDENTIFY 'obstacles' by enumerating over 'sensor\_data' and selecting indices where distance < 'obstacle\_threshold'  
 RETURN 'obstacles'  
END FUNCTION

**Implementation Guide:** - Define a threshold value that represents the maximum distance at which an object is considered an obstacle. - Iterate through the sensor data, which contains distances to objects detected by the sensors. - Collect the indices of data points where the distance is less than the threshold, indicating an obstacle. - Return the list of indices corresponding to obstacles.

#### 3. Pathfinding

**Overview:** pathfinding implements the A\* search algorithm to find the shortest path from a start point to a goal in a grid. This algorithm is widely used in robotics and game development for navigation and pathfinding.

**Detailed Pseudo Code:**

FUNCTION pathfinding(grid, start, goal)  
 DEFINE heuristic function to calculate the Manhattan distance between two points  
 INITIALIZE a priority queue 'frontier' with the start point  
 INITIALIZE 'came\_from' and 'cost\_so\_far' dictionaries with the start point  
 WHILE the 'frontier' is not empty  
 GET the current point from 'frontier'  
 IF current point is the goal, exit the loop  
 FOR each possible move from the current point  
 CALCULATE the new cost and determine if this path is better than any previously explored  
 IF better path is found, update 'came\_from' and 'cost\_so\_far', and add the next point to 'frontier'  
 RECONSTRUCT the path from goal to start using 'came\_from'  
 RETURN the path  
END FUNCTION

**Implementation Guide:** - Use the PriorityQueue class from the queue module to manage the frontier. - Define a heuristic function (Manhattan distance is used here) for estimating the cost from a point to the goal. - Explore the grid from the start point, expanding paths to neighboring points and using the priority queue to prioritize exploration based on the estimated total cost (path cost plus heuristic). - Keep track of the path cost and the points from which each point was reached. - Once the goal is reached, or the frontier is empty, reconstruct the path from the goal back to the start. - Return the reconstructed path or an empty list if no path is found.