a_star_manas_ronen_vaibhav.py

```python
# Github link: https://github.com/manasdesai/Project3_Phase1_Planning
#!/usr/bin/env python3

"""
## Team Members
1. Manas Desai (120998973) (mdesai01)
2. Ronen Aniti (112095116) (raniti)
3. Vaibhav Yuwaraj Shende (121206817) (svaibhav)

"""

# Import Libraries
import time
import matplotlib.colors as mcolors
import numpy as np
import matplotlib.pyplot as plt
from enum import Enum
from queue import PriorityQueue
from matplotlib.animation import FuncAnimation
from matplotlib.colors import ListedColormap, BoundaryNorm
from matplotlib.patches import Patch
from matplotlib.lines import Line2D
from typing import Tuple

################################################################################
################## CONSTANTS ##########################
################################################################################
LOGGING = False


################################################################################
################## DEFINE THE ACTIONS ##########################
################################################################################
class Action(Enum):
    """
    Enum to represent the actions for the A* search algorithm.

    """

    def __init__(self, r: int, theta: int) -> None:
        """Represents unit vector version of each action"""
        self.r = r
        self.theta = theta

    LEFT60 = (1, -60)
    LEFT30 = (1, -30)
    STRAIGHT = (1, 0)
    RIGHT30 = (1, 30)
```

```python
    RIGHT60 = (1, 60)


################################################################################
################## DEFINE THE COORDINATE TRANSFORMATIONS ##########################
################################################################################
def coordinate_transformation(
    standard_frame: Tuple, spatial_resolution, angular_resolution
) -> Tuple:
    """
    Convert a standard frame (mm, mm, deg) to a grid frame (grid_x, grid_y, grid_theta).

    Args:
        standard_frame (Tuple): A tuple representing the standard frame (x, y, theta) in mm and degrees.
        spatial_resolution (): _spatial resolution in mm per unit_
        angular_resolution (_type_): _angular resolution in degrees per unit_

    Returns:
        Tuple: A tuple representing the grid frame (grid_x, grid_y, grid_theta) in grid units.
    """
    return (
        int(standard_frame[0] * spatial_resolution),
        int(standard_frame[1] * spatial_resolution),
        int(standard_frame[2] / angular_resolution),
    )


def coordinate_transformation_2dof(standard_frame_2d, spatial_resolution) -> Tuple:
    """
    Convert a 2D standard frame (mm, mm) to a grid frame (grid_x, grid_y).

    Args:
        standard_frame_2d (Tuple): A tuple representing the 2D standard frame (x, y) in mm.
        spatial_resolution (): _spatial resolution in mm per unit_

    Returns:
        Tuple: A tuple representing the grid frame (grid_x, grid_y) in grid units.
    """
    return (
        int(standard_frame_2d[0] * spatial_resolution),
        int(standard_frame_2d[1] * spatial_resolution),
    )


def coordinate_transformation_inverse(
    grid_frame, spatial_resolution, angular_resolution
) -> Tuple:
```

```python
    return (
        int(grid_frame[0] / spatial_resolution),
        int(grid_frame[1] / spatial_resolution),
        int(grid_frame[2] * angular_resolution),
    )


def coordinate_transformation_inverse_2dof(grid_frame, spatial_resolution) -> Tuple:
    return (
        int(grid_frame[0] / spatial_resolution),
        int(grid_frame[1] / spatial_resolution),
    )


def angle_to_index(angle_deg, angular_resolution) -> int:
    return int(angle_deg / angular_resolution)


def index_to_angle(angle_index, angular_resolution) -> int:
    return int(angle_index * angular_resolution)


###############################################################################
################## DEFINE COLLISION DETECTION ##########################
###############################################################################
def collision(x: int, y: int, scale: int, safety: int) -> bool:
    """
    Check if a point (x, y) collides with the obstacles in the workspace.

    Args:
        x (int): x coordinate in mm
        y (int): y coordinate in mm
        scale (int): scale factor for the workspace
        safety (int): safety margin around obstacles in mm

    Returns:
        bool: True if the point collides with an obstacle.
    """
    regions = []

    # Wall buffer region
    regions.append(
        (x - safety <= 0)
        or (x - 180 * scale + safety >= 0)
        or (y - safety <= 0)
        or (y - 50 * scale + safety >= 0)
    )

    # E
    ##########################
```

```
    # E, vertical rectangle primitive
    regions.append(
        (x - 20 * scale + safety >= 0)
        and (x - 25 * scale - safety <= 0)
        and (y - 10 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )

    # E, bottom rectangle primitive
    regions.append(
        (x - 20 * scale + safety >= 0)
        and (x - 33 * scale - safety <= 0)
        and (y - 10 * scale + safety >= 0)
        and (y - 15 * scale - safety <= 0)
    )

    # E, middle rectangle primitive
    regions.append(
        (x - 20 * scale + safety >= 0)
        and (x - 33 * scale - safety <= 0)
        and (y - 20 * scale + safety >= 0)
        and (y - 25 * scale - safety <= 0)
    )

    # E, top rectangle primitive
    regions.append(
        (x - 20 * scale + safety >= 0)
        and (x - 33 * scale - safety <= 0)
        and (y - 30 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )

    # N
    ##########################
    # N, left vertical rectangle primitive
    regions.append(
        (x - 43 * scale + safety >= 0)
        and (x - 48 * scale - safety <= 0)
        and (y - 10 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )

    # N, middle in between two segment region
    regions.append(
        (y + 3 * x - 179 * scale - safety * np.sqrt(10) <= 0)
        and (y + 3 * x - 169 * scale + safety * np.sqrt(10) >= 0)
        and (x - 48 * scale + safety >= 0)
        and (x - 53 * scale - safety <= 0)
        and (y - 10 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
```

```
    )

    # N, right vertical rectangle primitive
    regions.append(
        (x - 53 * scale + safety >= 0)
        and (x - 58 * scale - safety <= 0)
        and (y - 10 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )

    # P
    ###############
    # P, left vertical bar
    regions.append(
        (x - 68 * scale + safety >= 0)
        and (x - 73 * scale - safety <= 0)
        and (y - 10 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )

    # P, semi-circular region
    regions.append(
        (
            (x - 73 * scale) ** 2
            + (y - 28.75 * scale) ** 2
            - (6.25 * scale + safety) ** 2
            <= 0
        )
        and (x - 73 * scale - safety >= 0)
    )

    # M
    ##################
    # M, first vertical bar (left vertical)
    regions.append(
        (x - 85 * scale + safety >= 0)
        and (x - 90 * scale - safety <= 0)
        and (y - 10 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )
    # M, left diagonal region
    regions.append(
        (5 * x + y - 485 * scale - safety * np.sqrt(26) <= 0)
        and (5 * x + y - 483 * scale + safety * np.sqrt(26) >= 0)
        and (x - 90 * scale + safety >= 0)
        and (x - 95 * scale - safety <= 0)
        and (y - 10 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )
    # M, right diagonal region
```

```
    regions.append(
        (5 * x - y - 465 * scale - safety * np.sqrt(26) <= 0)
        and (5 * x - y - 463 * scale + safety * np.sqrt(26) >= 0)
        and (x - 95 * scale + safety >= 0)
        and (x - 100 * scale - safety <= 0)
        and (y - 10 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )
    # M, second vertical bar (right vertical)
    regions.append(
        (x - 100 * scale + safety >= 0)
        and (x - 105 * scale - safety <= 0)
        and (y - 10 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )

    # 6
    ##################
    # 6, circle primitive
    regions.append(
        (
            ((x - 120 * scale) ** 2 + (y - 17.5 * scale) ** 2)
            <= (7.5 * scale + safety) ** 2
        )
    )
    # 6, vertical rectangle on top of the circle
    regions.append(
        (x - 112.5 * scale + safety >= 0)
        and (x - 117.5 * scale - safety <= 0)
        and (y - 17.5 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )
    # 6, horizontal rectangle attached to the right of the vertical rectangle
    regions.append(
        (x - 117.5 * scale + safety >= 0)
        and (x - 123 * scale - safety <= 0)
        and (y - 33 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )

    # 6 (second 6)
    ##################
    # 6 (second 6), circle primitive
    regions.append(
        (
            ((x - 139.5 * scale) ** 2 + (y - 17.5 * scale) ** 2)
            <= (7.5 * scale + safety) ** 2
        )
    )
    # 6 (second 6), vertical rectangle
```

```python
    regions.append(
        (x - 132 * scale + safety >= 0)
        and (x - 137 * scale - safety <= 0)
        and (y - 17.5 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )
    # 6 (second 6), horizontal rectangle
    regions.append(
        (x - 137 * scale + safety >= 0)
        and (x - 143 * scale - safety <= 0)
        and (y - 33 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )

    # 1
    ###############
    # 1, rectangle primitive
    regions.append(
        (x - 155 * scale + safety >= 0)
        and (x - 160 * scale - safety <= 0)
        and (y - 10 * scale + safety >= 0)
        and (y - 35 * scale - safety <= 0)
    )

    return any(regions)


################################################################################
## DEFINE FUNCTIONALITY TO CREATE DIFFERENT TYPES OF OCCUPANCY GRIDS  ########
################################################################################
def generate_occupancy_grid(workspace_dimension, spatial_resolution, scale, safety):
    """
    Generate an occupancy grid for the workspace based on the specified dimensions, spatial resolution, scale, and safety margin.

    Args:
        workspace_dimension (): The dimensions of the workspace (width, height) in mm.
        spatial_resolution (): The spatial resolution in mm per unit.
        scale (): The scale factor for the workspace, used to determine the size of obstacles.
        safety (): The safety margin around obstacles in mm.

    Returns:
        np.ndarray: A 2D boolean occupancy grid where True indicates an obstacle and False indicates free space.
    """
    occupancy_grid_shape = (
        int(workspace_dimension[0] * spatial_resolution) + 1,
        int(workspace_dimension[1] * spatial_resolution) + 1,
    )
```

```python

    occupancy_grid = np.zeros(occupancy_grid_shape, dtype=bool)

    for x in range(occupancy_grid_shape[0]):
        for y in range(occupancy_grid_shape[1]):
            grid_coord = (x, y)
            frame_coord = coordinate_transformation_inverse_2dof(
                grid_coord, spatial_resolution
            )
            if collision(frame_coord[0], frame_coord[1], scale, safety):
                occupancy_grid[x, y] = True

    return occupancy_grid


def generate_occupancy_grid_for_plotting(
    workspace_dimension, spatial_resolution, scale, safety
):
    occupancy_grid_shape = (
        int(workspace_dimension[0] * spatial_resolution) + 1,
        int(workspace_dimension[1] * spatial_resolution) + 1,
    )

    occupancy_grid_for_plotting = np.zeros(occupancy_grid_shape, dtype=int)

    for x in range(occupancy_grid_shape[0]):
        for y in range(occupancy_grid_shape[1]):
            grid_coord = (x, y)
            frame_coord = coordinate_transformation_inverse_2dof(
                grid_coord, spatial_resolution
            )
            if collision(frame_coord[0], frame_coord[1], scale, 0.0):
                occupancy_grid_for_plotting[x, y] = (
                    2  # Obstacle region but not in padding
                )
            elif collision(frame_coord[0], frame_coord[1], scale, safety):
                occupancy_grid_for_plotting[x, y] = (
                    1  # Padded region but not in obstacle region
                )

    return occupancy_grid_for_plotting


##############################################################################
##### DEFINE A HEURISTIC FUNCTION FOR A* SEARCH ##########################
##############################################################################
def euclidean_ground_distance(node1, node2):
    """
    Get the Euclidean distance between 2 points

```

```
    Args:
        node1 (tuple): coordinates of first point
        node2 (tuple): coordinates of second point

    Returns:
        float: euclidean distance
    """
    return round(np.sqrt((node1[0] - node2[0]) ** 2 + (node1[1] - node2[1]) ** 2), 2)


##############################################################################
# DEFINE A NEIGHBOR EXPANSION FUNCTION FOR A* SEARCH ##########
##############################################################################
def find_valid_neighbors(
    occupancy_grid, current, step_size, spatial_resolution, angular_resolution
):
    # The output of this must be two lists in grid units only

    # Define a list of valid actions
    actions_list = [
        Action.LEFT60,
        Action.LEFT30,
        Action.STRAIGHT,
        Action.RIGHT30,
        Action.RIGHT60,
    ]

    # Define lists for valid neighbors and distances
    valid_neighbors_list = []  # grid frame
    distances_list = []  # grid distances

    # Convert to standard frame (mm, mm, deg) for collision checking
    current_standard_frame = coordinate_transformation_inverse(
        current, spatial_resolution, angular_resolution
    )

    for action in actions_list:
        # Get the dimensionless radius of expansion
        r = action.r

        # The action angular displacement in degrees:
        delta_theta = action.theta

        # The new x coordinate in mm
        new_node_standard_x = current_standard_frame[0] + r * step_size * np.cos(
            np.deg2rad(current_standard_frame[2] + delta_theta)
        )

        # The new y coordinate in mm
        new_node_standard_y = current_standard_frame[1] + r * step_size * np.sin(
```

```
            np.deg2rad(current_standard_frame[2] + delta_theta)
        )

        # The new theta coordinate in degrees
        new_node_standard_theta = current_standard_frame[2] + delta_theta
        new_node_standard_theta = (
            new_node_standard_theta % 360
        )  # Wrap around 360 degrees

        # The new node in (mm, mm, deg) format
        new_node_standard = (
            new_node_standard_x,
            new_node_standard_y,
            new_node_standard_theta,
        )

        # The new node in grid units
        new_node_grid = coordinate_transformation(
            new_node_standard, spatial_resolution, angular_resolution
        )

        # If the grid coordinate of the new node is out of bounds, skip
        if new_node_grid[0] < 0 or new_node_grid[0] >= occupancy_grid.shape[0]:
            continue

        if new_node_grid[1] < 0 or new_node_grid[1] >= occupancy_grid.shape[1]:
            continue

        # Mark the new node as valid if its in bounds and not an obstacle
        if not occupancy_grid[new_node_grid[0], new_node_grid[1]]:
            valid_neighbors_list.append(new_node_grid)
            distances_list.append(
                r * step_size * spatial_resolution
            )  # The cost is measured in grid units

    return valid_neighbors_list, distances_list


###############################################################################
##### DEFINE A BACKTRACKING FUNCTION FOR A* SEARCH ###########################
###############################################################################
def backtrack(predecessors, start, goal):
    """
    Simple backtracking routine to extract the path from the branching dictionary
    """
    path = [goal]
    current = goal
    while predecessors[current] != None:
        parent = predecessors[current]
        path.append(parent)
```

```
        current = parent
    return path[::-1]


################################################################################
##### IMPLEMENTATION OF A* PATHFINDING ##########################
################################################################################
def astar(
    occupancy_grid,
    color_occupancy_grid,
    start,
    goal,
    scale_factor,
    safety_margin,
    step_size,
    spatial_resolution,
    angular_resolution,
    h=euclidean_ground_distance,
    logging=False,
):
    # *Assume start and goal are in grid units*

    # Convert the start node to standard coordinate from for validity check
    start_standard_frame = coordinate_transformation_inverse(
        start, spatial_resolution, angular_resolution
    )

    # Convert the goal node to standard coordinate frame for validity check
    goal_standard_frame = coordinate_transformation_inverse(
        goal, spatial_resolution, angular_resolution
    )

    # Check the validity of the start node
    if collision(
        start_standard_frame[0], start_standard_frame[1], scale_factor, safety_margin
    ):
        print("The start node is invalid")
        return None, None

    # Check the validity of the goal node
    if collision(
        goal_standard_frame[0], goal_standard_frame[1], scale_factor, safety_margin
    ):
        print("The goal node is invalid")
        return None, None

    # Construct lists to store open set and closed set history for animation
    if logging:
        search_array_history = []

```

```
    # Initialize A* data structures
    parents = dict()

    # Construct an array data structure to combine open set and closed set information.
    # Unvisited and unopened = 0
    # Visited = 1
    # Open = 2
    search_array_shape = (
        occupancy_grid.shape[0],
        occupancy_grid.shape[1],
        int(360 / angular_resolution),
    )
    search_array = np.zeros(search_array_shape, dtype=np.int8)

    g_scores = dict()
    queue = PriorityQueue()
    goal_is_found = False

    # Handle the start node
    search_array[start[0], start[1], start[2]] = 2  # 2 -> open
    parents[start] = None
    g_scores[start] = 0.0
    f_score_start = g_scores[start] + h(start, goal)
    queue.put((f_score_start, start))

    # Logging
    iteration = 1

    # Begin the A* main loop
    while not queue.empty():
        # Logging
        iteration += 1
        if logging and (iteration % 5000 == 0):
            print(f"Logging the {iteration}th iteration of A*")
            search_array_history.append(search_array.copy())

        # POP the most promising node
        f_current, current = queue.get()

        # ASSUME that some queued nodes may be visited nodes. Skip them.
        if search_array[current[0], current[1], current[2]] == 1:  # 1 -> visited
            continue

        # ONLY proceed to visit and process unvisited nodes:

        # Mark the current node as visited
        search_array[current[0], current[1], current[2]] = 1  # 1-> visited

        # Stop search if the goal is found
```

```python
        if current[:2] == goal[:2]:
            print("A* has found the goal")
            found_goal = current  # May have different orientation, but that's OK
            goal_is_found = True

            if logging:
                print(f"Logging the {iteration}th iteration of A*")
                search_array_history.append(search_array.copy())
            break

        # If this current node is NOT the goal node:

        # Expand the neighbors of this current node:
        valid_neighbors_list, distances_list = find_valid_neighbors(
            occupancy_grid, current, step_size, spatial_resolution, angular_resolution
        )
        for i, neighbor in enumerate(valid_neighbors_list):
            # ASSUME that some NEIGHBORS may be VISITED already. Skip them.
            if search_array[neighbor[0], neighbor[1], neighbor[2]] == 1:  # 1 -> visited
                continue

            # ASSUME that some NEIGHBORS may already be in the OPEN set. Process them, but IF AND ONLY IF a better partial plan would result.
            if search_array[neighbor[0], neighbor[1], neighbor[2]] == 2:  # 2 -> open
                g_current = g_scores[current]  # g-score of current node
                g_tentative = g_current + distances_list[i]
                if g_tentative < g_scores[neighbor]:
                    g_scores[neighbor] = g_tentative
                    parents[neighbor] = current
                    f_score_neighbor = g_tentative + h(neighbor, goal)
                    queue.put((f_score_neighbor, neighbor))

            # ASSUME that some NEIGHBORS may be NOT in the OPEN SET and NOT in the CLOSED SET.
            if (
                search_array[neighbor[0], neighbor[1], neighbor[2]] == 0
            ):  # 0 -> unvisited and unseen
                search_array[neighbor[0], neighbor[1], neighbor[2]] = 2  # 2 -> open
                g_tentative = g_scores[current] + distances_list[i]
                parents[neighbor] = current
                g_scores[neighbor] = g_tentative
                f_score_neighbor = g_tentative + h(neighbor, goal)
                queue.put((f_score_neighbor, neighbor))

    if goal_is_found:
        cost = g_scores[found_goal]  # cost in grid units

        path = backtrack(parents, start, found_goal)  # path is in grid units

```

```python
        # Logging
        if logging:
            animate_search(color_occupancy_grid, search_array_history, path)

        # Return the path and cost
        return path, cost

    path("No path found")
    return None, None


##########################################################################
##### GENERATE AN ANIMATION FUNCTION FOR VISUALIZATION OF A* RESULTS ##
##########################################################################
def animate_search(color_occ_grid, search_array_history, path):
    # Append 10 copies of the final frame to ensure the final frame is shown on screen for a sufficient amount of time.
    final_frame = search_array_history[-1]
    for _ in range(10):
        search_array_history.append(final_frame)

    # Create the figure and axis.
    fig, ax = plt.subplots(figsize=(8, 6))

    # Plot the colored occupancy grid as the background.
    ax.imshow(
        color_occ_grid.T,
        origin="lower",
        cmap="inferno",
        alpha=1.0,
        extent=[0, color_occ_grid.shape[0], 0, color_occ_grid.shape[1]],
    )

    # Create a discrete colormap to show the search evolution
    # 0: unvisited (transparent), 1: Closed (blue), 2: Open (yellow)
    search_cmap = ListedColormap([(0, 0, 0, 0), "blue", "yellow"])
    search_norm = BoundaryNorm([0, 0.5, 1.5, 2.5], search_cmap.N)

    # Code that ensures that each x, y location displays on the plot as  VISITED (blue) if ANY orientation at that location has been VISITED
    first_frame = search_array_history[0]
    H, W, O = first_frame.shape
    first_frame_list_of_lists = [[0 for _ in range(W)] for _ in range(H)]
    for i in range(H):
        for j in range(W):
            location_2d = first_frame[i, j, :]
            if 1 in location_2d:
                first_frame_list_of_lists[i][j] = 1
            elif 2 in location_2d:
                first_frame_list_of_lists[i][j] = 2
```

```
            else:
                first_frame_list_of_lists[i][j] = 0

    # Display the initial search space
    search_space = ax.imshow(
        np.array(first_frame_list_of_lists).T,
        origin="lower",
        cmap=search_cmap,
        norm=search_norm,
        alpha=0.6,
        extent=[0, color_occ_grid.shape[0], 0, color_occ_grid.shape[1]],
    )

    # Extract start and goal coordinates from the path.
    start_coords = path[0]
    goal_coords = path[-1]

    # Plot start and goal as static objects
    ax.scatter(start_coords[0], start_coords[1], s=50, color="chartreuse")
    ax.scatter(goal_coords[0], goal_coords[1], s=50, color="magenta")

    # Initialize a line object to eventually contain the path line
    (path_line,) = ax.plot([], [], "r-", linewidth=2)

    # Set title and axis labels.
    ax.set_title("A* Search Evolution")
    ax.set_xlabel("X-Coordinate Grid Frame")
    ax.set_ylabel("Y-Coordinate Grid Frame")

    # Create the legend markers
    # CLOSED SET patch (blue)
    closed_patch = Patch(facecolor="blue", edgecolor="blue", label="Closed")
    # OPEN SET patch (yellow)
    open_patch = Patch(facecolor="yellow", edgecolor="yellow", label="Open")
    # START marker
    start_marker = Line2D(
        [0],
        [0],
        marker="o",
        color="w",
        markerfacecolor="chartreuse",
        markersize=8,
        label="Start",
    )
    # GOAL marker
    goal_marker = Line2D(
        [0],
        [0],
        marker="o",
        color="w",
```

```python
        markerfacecolor="magenta",
        markersize=8,
        label="Goal",
    )
    # PATH marker
    path_marker = Line2D([0], [0], color="red", lw=2, label="Final Path")
    # PLACE the legend in the top right corner
    ax.legend(
        handles=[closed_patch, open_patch, start_marker, goal_marker, path_marker],
        loc="upper right",
    )

    # Define an update function for the animation of the A* search evolution
    def update_frame(frame_index):
        frame = search_array_history[frame_index]
        H, W, O = frame.shape
        # Collapse the 3D frame into 2D explicitly.
        frame_as_list_of_lists = [[0 for _ in range(W)] for _ in range(H)]
        for i in range(H):
            for j in range(W):
                location_2d = frame[i, j, :]
                if 1 in location_2d:
                    frame_as_list_of_lists[i][j] = 1
                elif 2 in location_2d:
                    frame_as_list_of_lists[i][j] = 2
                else:
                    frame_as_list_of_lists[i][j] = 0

        # Add the frame data to the visualization
        search_space.set_data(np.array(frame_as_list_of_lists).T)

        # Show the final path for the final 10 iterations of the animation
        if frame_index >= len(search_array_history) - 10:
            xs = [p[0] for p in path]
            ys = [p[1] for p in path]
            path_line.set_data(xs, ys)

        return search_space, path_line

    # Create the animation.
    anim = FuncAnimation(
        fig,
        update_frame,
        frames=len(search_array_history),
        interval=200,
        blit=False,
        repeat=True,
    )

    print("Saving the animation (gif)...")
```

```python
    # Save the animation as a GIF.
    anim.save("astar_animation.gif", writer="pillow", fps=5)
    print("Animation saved as astar_animation.gif")

    # Save the animation as an MP4.
    print("Saving the animation (mp4)...")
    anim.save("astar_animation.mp4", writer="ffmpeg", fps=5)
    print("Animation saved as astar_animation.mp4")

    plt.show()


################################################################################
##### THE MAIN FUNCTION ##########################
################################################################################
if __name__ == "__main__":
    # This project is an extension of Project 2.
    # The obstacles from Project 2 will be used.
    # Recall the exact x, y dimensions of the workspace from Project 2
    # However they will be dilated by a fixed factor.
    ORIGINAL_WORKSPACE_DIMENSION = (180, 50)
    SCALE_FACTOR = 3.0

    # The resulting workspace dimension is this:
    NEW_WORKSPACE_DIMENSION = (
        ORIGINAL_WORKSPACE_DIMENSION[0] * SCALE_FACTOR,
        ORIGINAL_WORKSPACE_DIMENSION[1] * SCALE_FACTOR,
    )

    # The Project 2 specifications also call for a spatial resolution of 0.5mm/ unit (2 units/mm) and for an angular resolution of 30 degrees per unit. So I am setting that here.
    # The coordinates of the workspace are assumed to be
    # (mm, mm, degrees) for (x, y, theta).
    SPATIAL_RESOLUTION = 2.0
    ANGULAR_RESOLUTION = 30.0

    # Define the step size for the branch expansion process for the A* component of the pathfinding operation. This will be in mm.
    step_size = float(input("Enter the step size (mm): "))

    # Define a safety margin around obstacles
    safety_margin = float(input("Enter a safety margin around obstacles (mm): "))

    robot_radius = float(input("Enter the robot radius (mm): "))

    # Use the larger of the input `robot_radius` and the input `safety_margin` to serve as the actual `safety_margin` used in pathfinding.
    safety_margin = max(safety_margin, robot_radius)

    # Define the start pose (mm, mm, deg)
```

```python
    x_start, y_start, theta_start = map(
        float,
        input(
            "Enter the (mm, mm, deg) coordinates for the start pose, separated by only spaces:"
        ).split(),
    )
    start = (x_start, y_start, theta_start)

    # Define the goal pose (mm, mm, deg)
    x_goal, y_goal, theta_goal = map(
        float,
        input(
            "Enter the (mm, mm, deg) coordinates for the goal pose, separated by only spaces:"
        ).split(),
    )
    goal = (x_goal, y_goal, theta_goal)

    # Convert the start and goal poses to the grid frame
    start = coordinate_transformation(start, SPATIAL_RESOLUTION, ANGULAR_RESOLUTION)

    goal = coordinate_transformation(goal, SPATIAL_RESOLUTION, ANGULAR_RESOLUTION)

    # Define an occupancy grid for Project 3 having the structure:
    # grid[i][j] -> i -> units of mm * spatial_resolution
    #              j -> units of mm * spatial_resolution
    # The grid will be a Numpy bool array and will be constructed using
    # a `collision` function having the hard-coded obstacle information
    # specified in Project 2
    print("Constructing the occupancy grid...")
    occupancy_grid = generate_occupancy_grid(
        NEW_WORKSPACE_DIMENSION, SPATIAL_RESOLUTION, SCALE_FACTOR, safety_margin
    )
    print("The occupancy grid construction is complete.")

    # This is a step non-necessary for pathfinding but necessary for static visualization and for animation. We will proceed with static visualization to aid the grader in assessing the result. But we will skip animation generation, instead including an animation from a prior run with our submission.
    print("Generating a color occupancy grid for plotting purposes")
    color_occupancy_grid = generate_occupancy_grid_for_plotting(
        NEW_WORKSPACE_DIMENSION, SPATIAL_RESOLUTION, SCALE_FACTOR, safety_margin
    )

    ############################################################################
    ########CALL TO A* #####
    ############################################################################
    # Executes A* search, return path and cost
    print("Executing A* search from start to goal.")
```

```
    # ONLY TURN LOGGING TO `True` IF YOU WANT TO GENERATE ANIMATION FILES AND HAVE MATPLOTLIB ALSO SHOW THE ANIMATION.
    # RECOMMENDED: KEEP FALSE. THIS PROCESS IS TIME CONSUMING.
    START_TIME = time.perf_counter()
    path, cost = astar(
        occupancy_grid,
        color_occupancy_grid,
        start,
        goal,
        SCALE_FACTOR,
        safety_margin,
        step_size,
        SPATIAL_RESOLUTION,
        ANGULAR_RESOLUTION,
        logging=LOGGING,
    )
    END_TIME = time.perf_counter()
    RUN_TIME = END_TIME - START_TIME

    print(f"Total time for execution: {RUN_TIME} seconds.")

    # Proceed with static visualization to aid the grader in assessing the results. This shows the color occupancy grid with the final path.
    if path:
        # Print the path in grid units
        print(f"The path, in grid units, is as follows: {path}")

        # Print the cost in grid units
        print(f"The total cost (length) of the path, in grid units, is {cost}")

        # Print the path in units of (mm, mm deg)
        print(
            f"The final path in units of (mm, mm, deg) is {[coordinate_transformation_inverse(p, SPATIAL_RESOLUTION, ANGULAR_RESOLUTION) for p in path]}"
        )

        # Print the cost in standard units (mm)
        print(f"The path length in mm is {cost / SPATIAL_RESOLUTION}")

        # Construct a static plot of the output to assist the grader in assessing performance
        print(
            "Constructing an occupancy grid for plotting to highlight the padded region..."
        )

        occupancy_grid_for_plotting = generate_occupancy_grid_for_plotting(
            NEW_WORKSPACE_DIMENSION, SPATIAL_RESOLUTION, SCALE_FACTOR, safety_margin
        )

        print("The occupancy grid for plotting is complete.")
```

```

        # Show the color occupancy grid
        plt.imshow(
            occupancy_grid_for_plotting.T,
            origin="lower",
            cmap="inferno",
            extent=[0, occupancy_grid.shape[0], 0, occupancy_grid.shape[1]],
        )

        # Make a chart title
        plt.title("Workspace with Path")

        # Overlay the path on the occupancy grid
        xs_path = [node[0] for node in path]
        ys_path = [node[1] for node in path]
        us_path = [np.cos(np.deg2rad(node[2] * ANGULAR_RESOLUTION)) for node in path]
        vs_path = [np.sin(np.deg2rad(node[2] * ANGULAR_RESOLUTION)) for node in path]

        # Include orientation in the static visualiation
        plt.quiver(
            xs_path,
            ys_path,
            us_path,
            vs_path,
            color="r",
            width=0.005,
            scale_units="xy",
            angles="xy",
            scale=0.5,
        )
        plt.show()

```