```python
import numpy as np
import heapq as hq
import cv2
import time

from typing import Dict, Tuple, List, Callable, Union
from numpy.typing import NDArray



# Define function for visualizing environment

def visualize_environment(obstacles, clearances, start, goal, path,
explored_nodes):

    # Create a blank 600x250 white frame
    frame = np.ones((250, 600, 3), dtype=np.uint8) * 255

    # Generate meshgrid of all (x, y) coordinates
    x_grid, y_grid = np.meshgrid(np.arange(600), np.arange(250))

    # Compute clearance area and display as gray
    for conditions in clearances.values():
        mask = np.ones_like(x_grid, dtype=bool)
        for cond in conditions:
            mask &= cond(x_grid, y_grid)
        frame[mask] = (150, 150, 150)

    # Compute obstacle area and display as black
    obstacle_mask = np.zeros_like(x_grid, dtype=bool)
    for conditions in obstacles.values():
        temp_mask = np.ones_like(x_grid, dtype=bool)
        for cond in conditions:
            temp_mask &= cond(x_grid, y_grid)
        obstacle_mask |= temp_mask
    frame[np.where(obstacle_mask)] = (0, 0, 0)

    # Draw the start and goal points
    cv2.circle(frame, (int(start[0]), int(start[1])), 2, (0, 0, 255), -1)
    cv2.circle(frame, (int(goal[0]), int(goal[1])), 2, (0, 255, 0), -1)

    # Flip to match coordinate system
    frame = cv2.flip(frame, 0)

    # Draw the explored nodes
```

```
    for i in range(len(explored_nodes) - 1):

        # Gather arrow end points
        x1, y1, theta1 = explored_nodes[i]
        x2, y2, theta2 = explored_nodes[i + 1]

        dx = int(3 * np.cos(np.radians(theta2 * 30)))
        dy = int(3 * np.sin(np.radians(theta2 * 30)))

        # Flip to match coordinate system
        y1_flipped = 250 - y1
        dy_flipped = -dy

        # Draw arrow of explored node
        cv2.arrowedLine(frame, (int(x1), int(y1_flipped)), (int(x1 + dx),
int(y1_flipped + dy_flipped)),
                        (0, 200, 200), 1, tipLength=1)

        # Display animation
        if i%5 == 0:
            cv2.imshow("A* Path Visualization", frame)
            cv2.waitKey(1)


    # Draw path nodes
    for i in range(len(path) - 1):

        # Gather arrow end points
        x1, y1, theta1 = path[i]
        x2, y2, theta2 = path[i + 1]

        dx = int(5 * np.cos(np.radians(theta2 * 30)))  # Scale for better
visibility
        dy = int(5 * np.sin(np.radians(theta2 * 30)))

        # Flip to match coordinate system
        y1_flipped = 250 - y1
        y2_flipped = 250 - y2
        dy_flipped = -dy

        # Draw path node
        cv2.arrowedLine(frame, (int(x1), int(y1_flipped)), (int(x2),
int(y2_flipped)),
                         (255, 0, 0), 1, tipLength=1)  # Blue arrows
```

```
    # Draw start and goal points
    cv2.circle(frame, (int(start[0]), int(250 - start[1])), 2, (0, 0, 255), -
1)  # Red (start)
    cv2.circle(frame, (int(goal[0]), int(250 - goal[1])), 2, (0, 255, 0), -1)  #
Green (goal)

    # Final visualization
    cv2.imshow("A* Path Visualization", frame)
    cv2.waitKey(0)
    cv2.destroyAllWindows()



# Define function for determining whether a location is valid

def is_valid(x: float | int, y: float | int, clearances: Dict) -> bool:

    # If location is within obstacle constraints
    if any(all(constraint(x, y) for constraint in constraints) for constraints in
clearances.values()):

        # Return invalid
        return False

    # If location is not within obstacle constraints
    else:

        # Return valid
        return True



# Define function for gathering a pose

def get_pose(location: str, clearances: Dict) -> Tuple:

    # Loop until pose is valid
    while True:

        # Gather user input
        user_input = input(f"{location} pose separated by commas in the format
of: x, y, θ\n- x: 1 - 600\n- y: 1 - 250\n- θ: Intervals of 30\nEnter: ").strip()

        # Return default start pose if input is empty
        if user_input is None:
```

```
            return ("Please enter a pose.")

        # Break user input into pose coordinates
        parts = user_input.split(",")

        # Ensure all three pose coordinates are present
        if len(parts) == 3:
            try:

                # Assign input coordinates
                x = float(parts[0].strip())
                y = float(parts[1].strip())
                theta = int(parts[2].strip())

                # If coordinates are within bounds
                if 1 <= x <= 600 and 1 <= y <=250 and theta in [0, 30, 60, 90,
120, 150, 180, 210, 240, 270, 300, 330, 360]:

                    # Convert positional coordinates to 1 - n scale
                    x = x - 1
                    y = y - 1

                    # If location is not an obstacle, return pose
                    if is_valid(x + 1, y + 1, clearances):
                        return (x, y, (theta / 30) % 12)

                    # Inform user of invalid location
                    else:
                        print("Sorry, this point is within the obstacle space.
Try again.")

                # Inform user of invalid location
                else:
                    print("Invalid input. Please ensure both x and y are within
the bounds of the space and theta is in [-60,-30,0,30,60].")

            # Inform user of invalid input format
            except ValueError:
                print("Invalid input. Please enter integers for x, y, and
theta.")

        # Inform user of invalid input dimension
            else:
                print("Invalid input. Please enter exactly three integers
separated by a comma.")
```

```python
def a_star(start: Tuple[float, float, int], goal: Tuple[float, float, int],
clearances: Dict, actions: List, map_size: Tuple[int, int] = (600, 250)) ->
Union[List, None]:

    # Mark start time
    start_time = time.time()

    # Define function for computing heuristic
    def heuristic(node: Tuple[float, float, int], goal: Tuple[float, float, int])
-> float:
        return np.sqrt((node[0] - goal[0]) ** 2 + (node[1] - goal[1]) ** 2) + 0.5
* (node[2] - goal[2])

    # Define function for backtracking
    def backtrack(goal: Tuple[float, float, int], parent_map: Dict) -> List:
        path = []
        while goal in parent_map:
            path.append(goal)
            goal = parent_map[goal]
        path.reverse()
        return path

    # Define function for getting node neighbors
    def get_neighbors(node: Tuple[float, int, int], visited: np.ndarray,
clearances: Dict, actions: List, map_size: Tuple[int, int] = (600, 250)) -> List:
        x, y, theta = node
        neighbors = []
        for move, delta_angle in actions:
            new_theta = (theta + (delta_angle / 30)) % 12
            new_x = x + move * np.cos(np.deg2rad(new_theta * 30))
            new_y = y + move * np.sin(np.deg2rad(new_theta * 30))
            int_x, int_y, int_theta = int(round(new_x)), int(round(new_y)),
int(new_theta)
            if 0 <= int_x < map_size[0] and 0 <= int_y < map_size[1]:
                if is_valid(new_x, new_y, clearances) and visited[int_y, int_x,
int_theta] == 0:
                    visited[int_y, int_x, int_theta] = 1
                    neighbors.append((new_x, new_y, new_theta))
        return neighbors

    # Create configuration map for visited nodes
    visited = np.zeros((map_size[1], map_size[0], 12), dtype=np.uint8)
```

```
    # Initialize open list
    open_list = []
    hq.heappush(open_list, (0, start))

    # Initialize dictionary for storing parent information
    parent_map = {}

    # Initialize dictionary for storing cost information
    cost_map = {start: 0}

    # Initialize list for storing closed nodes and explored nodes
    closed_nodes = []
    explored_nodes = []

    # Loop until queue is empty
    while open_list:

        current_node_info = hq.heappop(open_list)
        current_node: Tuple[float, float, int] = current_node_info[1]

        # Add node to closed list
        closed_nodes.append(current_node)

        # Record explored node for visualization
        explored_nodes.append(current_node)

        # Determine if solution is found
        if np.sqrt((current_node[0] - goal[0]) ** 2 + (current_node[1] - goal[1])
** 2) <= 1.5 and current_node[2] == goal[2]:

            # Mark end time
            end_time = time.time()

            print(f"Time to search: {end_time - start_time:.4f} seconds")

            # Backtrack to find path from goal
            return backtrack(current_node, parent_map), explored_nodes

        # Loop through neighbors
        for neighbor in get_neighbors(current_node, visited, clearances,
actions):
            new_cost = cost_map[current_node] + 1
            if neighbor not in cost_map or new_cost < cost_map[neighbor]:
                cost_map[neighbor] = new_cost
```

```python
                total_cost = new_cost + heuristic(neighbor, goal)
                hq.heappush(open_list, (total_cost, neighbor))
                parent_map[neighbor] = current_node

    return None, explored_nodes  # Return None if no path is found



# Define function for main execution

def main():


    # Define obstacles
    obstacles = {

            "Obstacle 1": [
                lambda x, y: x >= 26.25,
                lambda x, y: x <= 51.25,
                lambda x, y: y >= 50,
                lambda x, y: y <= 175
            ],

            "Obstacle 2": [
                lambda x, y: x >= 51.25,
                lambda x, y: x <= 81.25,
                lambda x, y: y >= 50,
                lambda x, y: y <= 75
            ],

            "Obstacle 3": [
                lambda x, y: x >= 51.25,
                lambda x, y: x <= 81.25,
                lambda x, y: y >= 100,
                lambda x, y: y <= 125
            ],

            "Obstacle 4": [
                lambda x, y: x >= 51.25,
                lambda x, y: x <= 81.25,
                lambda x, y: y >= 150,
                lambda x, y: y <= 175
            ],

            "Obstacle 5": [
```

```
            lambda x, y: x >= 96.25,
            lambda x, y: x <= 121.25,
            lambda x, y: y >= 50,
            lambda x, y: y <= 175
        ],

        "Obstacle 6": [
            lambda x, y: x >= 121.25,
            lambda x, y: x <= 136.25,
            lambda x, y: y >= -(3 + (1/3)) * x + (504 + (1/6)),
            lambda x, y: y <= -(3 + (1/3)) * x + (579 + (1/6))
        ],

        "Obstacle 7": [
            lambda x, y: x >= 136.25,
            lambda x, y: x <= 161.25,
            lambda x, y: y >= 50,
            lambda x, y: y <= 175,
        ],

        "Obstacle 8": [
            lambda x, y: x >= 176.25,
            lambda x, y: x <= 201.25,
            lambda x, y: y >= 50,
            lambda x, y: y <= 175
        ],

        "Obstacle 9": [
            lambda x, y: x >= 201.25,
            lambda x, y: (x - 201.25)**2 + (y - 150) ** 2 <= 625
        ],

        "Obstacle 10": [
            lambda x, y: x >= 241.25,
            lambda x, y: x <= 266.25,
            lambda x, y: y >= 50,
            lambda x, y: y <= 175
        ],

        "Obstacle 11": [
            lambda x, y: x >= 266.25,
            lambda x, y: x <= 297.5,
            lambda x, y: y >= 50,
            lambda x, y: y <= 175,
            lambda x, y: y <= -3.2 * x + 1027,
```

```
        lambda x, y: y >= -3.2 * x + 952
    ],

    "Obstacle 12": [
        lambda x, y: x >= 297.5,
        lambda x, y: x <= 328.75,
        lambda x, y: y >= 50,
        lambda x, y: y <= 175,
        lambda x, y: y <= 3.2 * x - 877,
        lambda x, y: y >= 3.2 * x - 952
    ],

    "Obstacle 13": [
        lambda x, y: x >= 328.75,
        lambda x, y: x <= 353.75,
        lambda x, y: y >= 50,
        lambda x, y: y <= 175,
    ],

    "Obstacle 14": [
        lambda x, y: (x - 406.25)**2 + (y - 87.5) ** 2 <= 1406.25,
    ],

    "Obstacle 15": [
        lambda x, y: (x - 476.5)**2 + (y - 87.5) ** 2 <= 11556.25,
        lambda x, y: (x - 406.25)**2 + (y - 87.5) ** 2 >= 1406.25,
        lambda x, y: (x - 476.5)**2 + (y - 87.5) ** 2 >= 6806.25,
        lambda x, y: y >= 87.5,
        lambda x, y: x <= 426.25,
    ],

    "Obstacle 16": [
        lambda x, y: (x - 496.25)**2 + (y - 87.5) ** 2 <= 1406.25,
    ],

    "Obstacle 17": [
        lambda x, y: (x - 566.25)**2 + (y - 87.5) ** 2 <= 11556.25,
        lambda x, y: (x - 496.25)**2 + (y - 87.5) ** 2 >= 1406.25,
        lambda x, y: (x - 566.25)**2 + (y - 87.5) ** 2 >= 6806.25,
        lambda x, y: y >= 87.5,
        lambda x, y: x <= 516.25,
    ],

    "Obstacle 18": [
        lambda x, y: x >= 548.75,
```

```
            lambda x, y: x <= 573.75,
            lambda x, y: y >= 50,
            lambda x, y: y <= 183
        ],

}

# Define clearances
clearances = {

        "Clearance 1": [
            lambda x, y: x >= 21.25,
            lambda x, y: x <= 56.25,
            lambda x, y: y >= 45,
            lambda x, y: y <= 180
        ],

        "Clearance 2": [
            lambda x, y: x >= 56.25,
            lambda x, y: x <= 86.25,
            lambda x, y: y >= 45,
            lambda x, y: y <= 80
        ],

        "Clearance 3": [
            lambda x, y: x >= 56.25,
            lambda x, y: x <= 86.25,
            lambda x, y: y >= 95,
            lambda x, y: y <= 130
        ],

        "Clearance 4": [
            lambda x, y: x >= 56.25,
            lambda x, y: x <= 86.25,
            lambda x, y: y >= 145,
            lambda x, y: y <= 180
        ],

        "Clearance 5": [
            lambda x, y: x >= 91.25,
            lambda x, y: x <= 126.25,
            lambda x, y: y >= 45,
            lambda x, y: y <= 180,
            lambda x, y: y <= -89.12565661 * x + 11318.04697,
        ],
```

```python
    "Clearance 6": [
        lambda x, y: x >= 124.9701533,
        lambda x, y: x <= 132.52984675,
        lambda x, y: y <= -(3 + (1/3)) * x + 596.5671771,
        lambda x, y: y >= -(3 + (1/3)) * x + 486.7661641
    ],

    "Clearance 7": [
        lambda x, y: x >= 131.25,
        lambda x, y: x <= 166.25,
        lambda x, y: y >= 45,
        lambda x, y: y <= 180,
        lambda x, y: y >= -89.12565315 * x + 11856.80915,
    ],

    "Clearance 8": [
        lambda x, y: x >= 171.25,
        lambda x, y: x <= 206.25,
        lambda x, y: y >= 45,
        lambda x, y: y <= 180
    ],

    "Clearance 9": [
        lambda x, y: x >= 201.25,
        lambda x, y: (x - 201.25)**2 + (y - 150) ** 2 <= 900
    ],

    "Clearance 10": [
        lambda x, y: x >= 236.25,
        lambda x, y: x <= 271.25,
        lambda x, y: y >= 45,
        lambda x, y: y <= 180,
        lambda x, y: y <= -85.165548 * x + 23168.391984
    ],

    "Clearance 11": [
        lambda x, y: x >= 269.92595457,
        lambda x, y: x <= 297.5,
        lambda x, y: y >= 45,
        lambda x, y: y <= 180,
        lambda x, y: y >= -3.2 * x + 935.2369454,
        lambda x, y: y <= -3.2 * x + 1043.763055
    ],
```

```
        "Clearance 12": [
            lambda x, y: x >= 297.5,
            lambda x, y: x <= 325.07404543,
            lambda x, y: y >= 45,
            lambda x, y: y <= 180,
            lambda x, y: y <= 3.2 * x - 860.2369454,
            lambda x, y: y >= 3.2 * x - 968.7630546
        ],

        "Clearance 13": [
            lambda x, y: x >= 323.75,
            lambda x, y: x <= 358.75,
            lambda x, y: y >= 45,
            lambda x, y: y <= 180,
            lambda x, y: y <= 85.165548 * x - 27505.10922
        ],

        "Clearance 14": [
            lambda x, y: (x - 406.25)**2 + (y - 87.5) ** 2 <= 1806.25,
        ],

        "Clearance 15": [
            lambda x, y: (x - 476.5)**2 + (y - 87.5) ** 2 <= 12656.25,
            lambda x, y: (x - 406.25)**2 + (y - 87.5) ** 2 >= 1806.25,
            lambda x, y: (x - 476.5)**2 + (y - 87.5) ** 2 >= 6006.25,
            lambda x, y: y >= 87.5,
            lambda x, y: x <= 431.25,
        ],

        "Clearance 16": [
            lambda x, y: (x - 496.25)**2 + (y - 87.5) ** 2 <= 1806.25,
        ],

        "Clearance 17": [
            lambda x, y: (x - 566.25)**2 + (y - 87.5) ** 2 <= 12556.25,
            lambda x, y: (x - 496.25)**2 + (y - 87.5) ** 2 >= 1806.25,
            lambda x, y: (x - 566.25)**2 + (y - 87.5) ** 2 >= 6006.25,
            lambda x, y: y >= 87.5,
            lambda x, y: x <= 521.25,
        ],

        "Clearance 18": [
            lambda x, y: x >= 543.75,
            lambda x, y: x <= 578.75,
            lambda x, y: y >= 45,
```

```
                lambda x, y: y <= 188
            ],

        }

    # Define action set
    actions = [
        (1.25, 60),
        (1.25, 30),
        (1.25, 0),
        (1.25, -30),
        (1.25, -60)
    ]

    # Gather start pose
    start = get_pose("Start", clearances)

    # Gather goal pose
    goal = get_pose("Goal", clearances)

    # Run search algorithm
    path, explored_nodes = a_star(start, goal, clearances, actions)

    # Visualize the environment
    visualize_environment(obstacles, clearances, start, goal, path,
explored_nodes)


# Execute script

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