```
# %%
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from collections import deque
import cv2
import matplotlib.patches as patches
import math
import heapq
from matplotlib import animation
import os
import time


def in_circle(x, y, cx, cy, r):
    # Check if (x, y) is in the circle centered at (cx, cy) with radius r
    return (x - cx)**2 + (y - cy)**2 <= r**2


def in_partial_ring(x, y,
                    cx, cy,
                    outer_r, inner_r,
                    start_deg, end_deg):
    """
    Check if (x, y) is in the partial ring defined by
        - circle centered at (cx, cy) with
          - outer_radius  = outer_r
          - inner_r       = inner_r
```

```
        - angles between start_deg and end_deg
    Coords wrt bottom-left origin
    """

    dx, dy   = x - cx, y - cy        # Vector from center of both outer/inner circles to point (x, y)
    angle_rad = math.atan2(dy, dx)     # Angle in radians from circle's center to point (x, y)
    angle_deg = math.degrees(angle_rad) # Convert angle to degrees

    pt_in_outer_circle = in_circle(x, y, cx, cy, outer_r) # Check if point is in outer circle
    pt_in_inner_circle = in_circle(x, y, cx, cy, inner_r) # Check if point is in inner circle

    if angle_deg < 0: # Convert negative angles to positive
        angle_deg += 360

    pt_is_in_ring     = pt_in_outer_circle and not pt_in_inner_circle  # Check if point is in ring
    pt_is_in_deg_range = start_deg <= angle_deg <= end_deg          # Check if point is in
degree range

    # If points is in ring and in degree range, return True, else return False
    if pt_is_in_deg_range:
        if pt_is_in_ring:
            return True
    return False

def in_right_half_circle(x, y, cx, cy, radius):
    # Check if (x, y) is in the right half of the circle centered at (cx, cy) with radius r
```

```
    # Coords wrt bottom-left origin
    inside_circle = (x - cx)**2 + (y - cy)**2 <= radius**2
    inside_right_half   = x >= cx
    return inside_circle and inside_right_half


def to_the_right(ex1, ey1, ex2, ey2, x, y):
    # Helper function to in_parallellogram, checks if point (x, y) is to the left of the line
defined by points (px1, py1) and (px2, py2)
    # (ex1, ey1) and (ex2, ey2) define the input edge of parallelogram
    # Vector1: (edge_x, edge_y) = (ex2 - ex1, ey2 - ey1)   points in direction of input edge from
start to end
    # Vector2: (pt_x, pt_y) = (x - ex1, y - ey1)      points is direction from edge start to point (x, y)
    # Cross Product of 2-D vector defined by
    #           (edge_x, edge_y, 0) x (pt_x, pt_y, 0) = (0, 0, edge_x * pt_y - edge_y * pt_x)
    # Coords wrt bottom-left origin

    edge_x, edge_y = ex2 - ex1, ey2 - ey1
    pt_x, pt_y    = x - ex1, y - ey1
    cross        = (edge_x * pt_y) - (edge_y * pt_x)

    # If cross product is positive, point is to the right of the edge
    # If cross product is negative, point is to the left of the edge
    return cross <= 0


def in_parallelogram(x, y, A, B, C, D):
    # Check if (x, y) is inside the parallelogram defined by the four points A, B, C, D
    # A, B, C, D are defined in clockwise order, so we check if (x, y) is to the right of each edge
```

```
    # Coords wrt bottom-left origin

    return (to_the_right(A[0], A[1], B[0], B[1], x, y) and
        to_the_right(B[0], B[1], C[0], C[1], x, y) and
        to_the_right(C[0], C[1], D[0], D[1], x, y) and
        to_the_right(D[0], D[1], A[0], A[1], x, y))

def in_rectangle(x, y, xmin, xmax, ymin, ymax):
    # Returns True if (x, y) is inside rectangle defined by (xmin, ymin), (xmax, ymax) corners
    # Coords wrt bottom-left origin
    return (x >= xmin) and (x <= xmax) and (y >= ymin) and (y <= ymax)

def in_E(x, y, start_x, start_y):
    # Check if x, y is inside the letter 'E'
    # Coords wrt bottom-left origin
    R_v   = in_rectangle(x, y, start_x, start_x+5,  start_y,    start_y+25)    # vertical bar
    R_top = in_rectangle(x, y, start_x, start_x+13, start_y+20, start_y+25) # top horizontal
    R_mid = in_rectangle(x, y, start_x, start_x+13, start_y+10, start_y+15)  # middle horizontal
    R_bot = in_rectangle(x, y, start_x, start_x+13, start_y,    start_y+ 5)   # bottom horizontal

    return R_v or R_top or R_mid or R_bot

def in_1(x, y, start_x, start_y):
    # Check if x, y is inside our coordinate for the number 1 in map_data
    # Coords wrt bottom-left origin
    R   = in_rectangle(x, y, start_x, start_x+5,  start_y,    start_y+28)    # vertical bar
```

```
    return R


def in_N(x, y, start_x, start_y):
    # Check whether (x, y) is inside the letter 'N'
    # Coords wrt bottom-left origin
    # Define N's Diagonal as parallelograms of points defined in clockwise order
    A = (start_x,    start_y+25)
    B = (start_x+5,  start_y+25)
    C = (start_x+20, start_y)
    D = (start_x+15, start_y)

    # Check if (x, y) is in our geometric definition of N
    R_left   = in_rectangle(x, y,    start_x, start_x+5,  start_y,   start_y+25)    # Left  Vertical Bar of
N
    R_right  = in_rectangle(x, y, start_x+15, start_x+20,  start_y,  start_y+25)    # Right Vertical
Bar of N
    diagonal = in_parallelogram(x, y, A, B, C, D)                          # Diagonal of N
    return R_left or R_right or diagonal


def in_P(x, y, start_x, start_y):
    # Check whether (x, y) is inside the letter 'P'
    # Coords wrt bottom-left origin

    radius = 6                 # Radius of our P's Half-Circle
    cx     = start_x + 5         # Half-Circle Center X Coordinate (On top of our P's Vertical Bar)
```

```
    cy    = start_y + 25-radius   # Half-Circle Center Y Coordinate (On top of our P's Vertical Bar)

    #Check if points is in our geometrical definition of P
    bar = in_rectangle(x, y,      # Vertical bar of our P
                xmin=start_x,
                xmax=start_x+5,
                ymin=start_y,
                ymax=start_y+25)

    top_half_circle = in_right_half_circle(x, y, cx, cy, radius) # Half-Circle of P

    return bar or top_half_circle

def in_M(x, y, start_x, start_y):
    # Check whether (x, y) is inside the letter 'M'
    # Coords wrt bottom-left origin
    # Diagonals of M Defined as parallelograms of points defined clockwise

    # Values below were defined in project prompt or were manually checked to get shape close to project prompt
    m_gap       = 5         # Horizontal Gap between the two vertical bars of M and the box connecting the diagonals in the middle
    bottom_w     = 7         # Width of the bottom rectangle in the middle of the M connecting the two diagonals
    bottom_offset = 5 + m_gap   # Offset from the start_x to the start of the bottom rectangle in the middle
```

```
    bottom_w     = 7         # Width of the bottom rectangle in the middle of the M connecting
the two diagonals

    second_box_offset = bottom_offset + bottom_w + m_gap # Leftmost X coord of second
vertical bar of M

    # First Vertical Box to Middle Rectangle Box Diagonal:
    A = (start_x,              start_y+25) # Diagonal 1, Top Left
    B = (start_x+5,            start_y+25) # Diagonal 1, Top Right
    C = (start_x+bottom_offset+1, start_y+5) # Diagonal 1, Bottom Right
    D = (start_x+bottom_offset,   start_y+0) # Diagonal 1, Bottom Left

    # Middle Rectangle Box Diagonal to Second Vertical Box:
    A1 = (start_x+second_box_offset,      start_y+25) # Diagonal 2, Top Left
    B1 = (start_x+second_box_offset+5,    start_y+25) # Diagonal 2, Top Right
    C1 = (start_x+bottom_offset+bottom_w,    start_y) # Diagonal 2, Bottom Left
    D1 = (start_x+bottom_offset+bottom_w-1, start_y+5) # Diagonal 2, Bottom Right

    # Check if (x, y) is in our geometric definition of M:
    R_left   = in_rectangle(x, y,
              start_x, start_x+5,  start_y,   start_y+25)    # First Vertical Bar

    R_right  = in_rectangle(x, y,                            # Second Vertical Bar
                  start_x+second_box_offset,
                  start_x+second_box_offset+5,
                  start_y,
                  start_y+25)
```

```
  R_bottom = in_rectangle(x, y,                    # Middle Retangle between two vertical
bars
                start_x+bottom_offset,
                start_x+bottom_offset+bottom_w,
                start_y,
                start_y+5)

  diagonal1 = in_parallelogram(x, y, A,  B,  C,  D)   # From First Vertical Bar to Middle
Retangle between the two vertical bars
  diagonal2 = in_parallelogram(x, y, A1, B1, C1, D1)  # Middle Rectangle Box Diagonal to
Second Vertical Box
  return R_left or R_right or R_bottom or diagonal1 or diagonal2



def in_6(x, y, start_x, start_y):
  # Check whether (x, y) is inside the number '6'
  # Coords wrt bottom-left origin

  cx = start_x + 20   # Adjusted Manually to get shape close to project prompt
  cy = start_y + 10   # Adjusted Manually to get shape close to project prompt

  outer_r = 21.5     # From Project Prompt
  inner_r = 16.5     # From Project Prompt
  start_deg, end_deg = 120, 180    # Degrees to sweep for curly top part of 6 curves,
Adjusted Manually to get shape close to project prompt
  bottom_cx  = start_x + 7       # Adjusted Manually to get shape close to project prompt
```

```
    bottom_cy  = start_y + 7        # Adjusted Manually to get shape close to project prompt
    bottom_r   = 9                  # From Project Prompt

    # Define Circle centered at outer tip of 6 and inner tip of 6
    tip_radius = 2.5    # From Project Prompt
    outer_tip  = cx + outer_r * np.cos(np.deg2rad(start_deg)), cy + outer_r *
np.sin(np.deg2rad(start_deg))
    inner_tip  = cx + inner_r * np.cos(np.deg2rad(start_deg)), cy + inner_r *
np.sin(np.deg2rad(start_deg))
    center_tip = (outer_tip[0] + inner_tip[0]) / 2, (outer_tip[1] + inner_tip[1]) / 2

    # Checks if (x, y) is in curled top part of 6
    curly_top     =  in_partial_ring(x, y, cx, cy, outer_r, inner_r,
                                     start_deg, end_deg)

    # Checks if (x, y) is in circular part of the bottom of the 6
    bottom_circle = in_circle(x, y, bottom_cx, bottom_cy, bottom_r)

    # Checks if (x, y) is in circular part at end of upper curled part of 6
    tip_circle    = in_circle(x, y, center_tip[0], center_tip[1], tip_radius)

    return curly_top, bottom_circle, tip_circle

def in_wall(x, y, w, h):
    # Check if (x, y) is inside the wall
    # Coords wrt bottom-left origin\
    clearance = 5
```

```
  return (
      x < clearance             # left edge
      or x >= w - clearance      # right edge
      or y < clearance          # bottom edge
      or y >= h - clearance      # top edge
    )

def draw(map_img, start_x, start_y, letter='E'):
  h, w = map_img.shape
  for py in range(h):
    for px in range(w):
      x_bl = px
      y_bl = py

      if letter == 'E': # Draw E
        if in_E(x_bl, y_bl, start_x, start_y):
          map_img[py, px] = 0

      elif letter == 'N': # Draw N
        if in_N(x_bl, y_bl, start_x, start_y):
          map_img[py, px] = 0

      elif letter == 'P': # Draw P
        if in_P(x_bl, y_bl, start_x, start_y):
          map_img[py, px] = 0
```

```
        elif letter == 'M': # Draw M
            if in_M(x_bl, y_bl, start_x, start_y):
                map_img[py, px] = 0

        elif letter == '6': # Draw 6
            curly_top, bottom_circle, tip_circle = in_6(x_bl, y_bl, start_x, start_y)
            if curly_top or bottom_circle or tip_circle:
                map_img[py, px] = 0

        elif letter == '1': # Draw 1
            if in_1(x_bl, y_bl, start_x, start_y):
                map_img[py, px] = 0

        elif letter == 'Wall': # Draw Wall
            if in_wall(x_bl, y_bl, w, h):
                map_img[py, px] = 0

    return map_img

def add_buffer(map_img, buffer_size=5):
    # Add 2 pixels to our map_data by dilating obstacles with a circular kernel with
radius=buffer_size
    map_img_copy = map_img.copy()

    # Create Circular Dilation Kernel, for morphology operations, we need a single center
pixel, and a 2x2 circle has no center pixel, so we use a 3x3 circle
```

```
    # The center pixel is 1 pixel, and the 8 surrounding pixels extend 1 pixel, so total radius is
2
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (buffer_size*2+1,
buffer_size*2+1))

    # In OpenCV, white(255) is treated as the foreground, Black(0) is Obstacle Space, so we
need to invert the colors
    map_with_clearance = cv2.dilate(255 - map_img_copy, kernel)

    # Invert back (to original representation: obstacles=0, free=255)
    map_img_copy     = 255 - map_with_clearance
    obstacles        = np.where(map_img_copy == 0)
    obstacles        = set(zip(obstacles[1], obstacles[0]))

    # new_obstacles = (map_img == 255) & (map_with_clearance == 0)

    return map_img_copy, obstacles

def create_cost_matrix(map_img):
    # Create cost matrix with obstacles as -1 and free space as infinity
    # We use [y, x] indexing to match openCV's (row, col) convention
    h, w      = map_img.shape
    cost_matrix = np.ones((h, w)) * np.inf

    for py in range(h):
        for px in range(w):
            if map_img[py, px] == 0:
```

```
            cost_matrix[py, px] = -1
    # Double resolution while keeping the same cost values
    upscaled_cost_matrix = np.repeat(np.repeat(cost_matrix, 2, axis=0), 2, axis=1)

    return upscaled_cost_matrix

def get_map_cost_matrix():
    # Create Map with Obstacles, Map with Obstacles + 2mm clearance and Cost Matrix
    start              = time.time()
    map_width, map_height = 180, 50 # mm
    start_x, start_y     = 12,  12 # mm
    resize_x, resize_y    = 600, 250 # mm
    robot_clearance      = 5      # mm

    map_img   = np.ones((map_height, map_width), dtype=np.uint8) * 255

    # Start x / y coordinates for each letter determined through trial/error and inspection
    map_img = draw(map_img, start_x, start_y, letter='E')

    start_x  = start_x + 21
    map_img = draw(map_img, start_x, start_y, letter='N')

    start_x   = start_x + 28
    map_img = draw(map_img, start_x, start_y, letter='P')

    start_x  = start_x + 18
```

```
    map_img = draw(map_img, start_x, start_y, letter='M')

    start_x = start_x  + 37
    map_img = draw(map_img, start_x, start_y, letter='6')

    start_x   = start_x + 26
    map_img = draw(map_img, start_x, start_y, letter='6')

    start_x   = start_x + 25
    map_img = draw(map_img, start_x, start_y, letter='1')

    upscaled_map      = cv2.resize(map_img, (resize_x, resize_y),
interpolation=cv2.INTER_NEAREST)
    map_with_clearance, obstacles = add_buffer(upscaled_map,
buffer_size=robot_clearance)

    map_with_clearance = draw(map_with_clearance, 0, 0, letter='Wall')

    cost_matrix       = create_cost_matrix(map_with_clearance)

    plt.figure(figsize=(10, 10))
    plt.imshow(upscaled_map, cmap='gray', origin='lower')
    plt.title('Map with Obstacles')
    plt.show()

    plt.figure(figsize=(10, 10))
    plt.imshow(map_with_clearance, cmap='gray', origin='lower')
```

```
    plt.title('Map with Obstacles and Clearance')
    plt.show()

    end = time.time()
    print("Time to Create Map: ", round((end-start), 2), " seconds")

    return upscaled_map, map_with_clearance, cost_matrix, obstacles

def get_theta_index(theta):
    theta = theta % 360
    look_up_dict = {
        0:   0,
        30:  1,
        60:  2,
        90:  3,
        120: 4,
        150: 5,
        180: 6,
        210: 7,
        240: 8,
        270: 9,
        300: 10,
        330: 11
    }
    return look_up_dict[theta]
```

```
def round_and_get_v_index(node):
    """
    Round x, y coordinates to nearest half to ensure we
    """
    x          = round(node[0] * 2) / 2
    y          = round(node[1] * 2) / 2
    theta      = node[2]
    x_v_idx    = int(x * 2)
    y_v_idx    = int(y * 2)
    theta_v_idx = get_theta_index(theta)

    return (x, y, theta), x_v_idx, y_v_idx, theta_v_idx

def check_if_visited(V, curr_node_v, stepsize):
    h, w = V.shape[:2]
    y, x, theta = curr_node_v

    step_size_i = max(int(stepsize // 2), 1)

    x1 = max(x - step_size_i, 0)
    x2 = min(x + step_size_i, w-1)
    y1 = max(y - step_size_i, 0)
    y2 = min(y + step_size_i, h-1)
    # print(x1, x2, y1, y2)

    sum_over_region = np.sum(V[y1:y2, x1:x2, :])
```

```
    if sum_over_region > 0:
        return True
    else:
        return False

def get_xy(node, move_theta, r=1):
    theta = node[2] + move_theta
    x     = node[0] + r * np.cos(np.deg2rad(theta))
    y     = node[1] + r * np.sin(np.deg2rad(theta))
    return (x, y, theta)

def move_theta_0(node, r=1):
    theta =  0
    return get_xy(node, theta, r), r
def move_diag_up_30(node, r=1):
    theta = 30
    return get_xy(node, theta, r), r
def move_diag_up60(node, r=1):
    theta = 60
    return get_xy(node, theta, r), r
def move_diag_down_30(node, r=1):
    theta = -30
    return get_xy(node, theta, r), r
def move_diag_down60(node, r=1):
    theta = -60
```

```
    return get_xy(node, theta, r), r


def is_valid_move(node, map):
    h, w = map.shape
    x, y, theta = node

    x, y = int(round(x)), int(round(y))

    if x < 0 or x >= w-1 or y < 0 or y >= h-1:
        return False
    if map[y, x] == 0:
        return False
    return True


def euclidean_distance(node, goal_state):
    """
    Calculate Euclidean Distance between current node and goal state
    Euclidean Distance is the straight line distance between two points
    distance metric used in A* Search
    """
    return math.sqrt((node[0] - goal_state[0])**2 + (node[1] - goal_state[1])**2)




def check_user_theta(node):
    """
```

```
    Check if theta is a multiple of 30 degrees, if not prompt user to input new theta
    """
    theta = node[2]
    if theta % 30 != 0:
        print("Theta is not a multiple of 30 degrees")
        theta = tuple(map(int, input(f"Enter new theta (0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330) as one number: ").split()))
    return theta


def check_validity_with_user(map_data, start_state, goal_state, max_attempts=10):
    """
    Check if initial and goal states are valid, if not prompt user to input new states

    start_state: Initial state of point robot
    goal_state:  Goal state of point robot
    map_data:       Map with obstacles

    Returns:    Tuple of valid start and goal states

    """
    i = 0

    while True:
        i += 1
        try:
            if not is_valid_move(start_state, map_data): # Check if initial state is valid, if not prompt user to input new state
```

```
            print("Initial state is invalid")
            start_state = tuple(map(int, input(f"{str(start_state)} invalid, Enter new start state (x y theta) as three numbers seperated by space: ").split()))
            start_theta = check_user_theta(start_state)             # Check if theta is a multiple of 30 degrees, if not prompt user to input new theta
            start_state = (start_state[0], start_state[1], start_theta) # Set new theta to start state

        if not is_valid_move(goal_state, map_data): # Check if goal state is valid, if not prompt user to input new state
            print("Goal state is invalid")
            goal_state = tuple(map(int, input(f"{str(goal_state)} invalid, Enter new goal state (x y theta) as three numbers seperated by space: ").split()))
            goal_theta = check_user_theta(goal_state)             # Check if theta is a multiple of 30 degrees, if not prompt user to input new theta
            goal_state = (goal_state[0], goal_state[1], goal_theta) # Set new theta to start state

        if is_valid_move(start_state, map_data) and is_valid_move(goal_state, map_data): # If both states are valid, plot map_data with start and goal states
            return start_state, goal_state

        if i > max_attempts: # if User has tried more than 50 times, exit
            print("Too many attempts, exiting")
            return None, None
    except:
        print("Invalid input")
        continue
```

```
def generate_path(parent, goal_state):
    """
    Generate the path from start state to goal state leveraging parent-child dictionary as
input,
    mapping child nodes to parent nodes. We start at goal state and work back to start state,
appending
    each state to a list. We then reverse our list to get correct order of nodes
    from start to goal.

    parent:     Dictionary mapping child states to parent states
    goal_state: Goal state of the puzzle

    Returns:    List of states from the start state to the goal state
    """
    path    = []
    current = goal_state

    while current is not None:
        try: # Try to append current state to path and set current state to parent of current state
            path.append(current)
            current = parent[current]
        except: # If error print the current state and break the loop (This is for debugging, but
should not be reached to run DFS or BFS)
            print(current)
            break
    path.reverse()
    return path
```

```
def plot_cost_matrix(cost_matrix, start_state, goal_state,  title="Cost Matrix Heatmap"):
    plt.figure(figsize=(8, 6))
    # Plot the cost matrix as a heatmap
    plt.imshow(cost_matrix, cmap='jet', origin='lower')
    plt.plot(start_state[0]*2, start_state[1]*2, 'ro', label='Start State')
    plt.plot(goal_state[0]*2, goal_state[1]*2, 'go', label='Goal State')
    plt.colorbar(label='Cost Value') # Add colorbar to show range of cost values
    plt.title(title)
    plt.xlabel("X (columns)")
    plt.ylabel("Y (rows)")
    plt.legend(
    loc='upper center',
    bbox_to_anchor=(0.5, -0.4),
    ncol=2
    )
    plt.show()

def solution_path_video(map_data, solution_path, save_folder_path, algo="Dijkstra"):
    fps      = 30
    h, w     = map_data.shape
    color_map = map_data.copy()
    color_map = cv2.cvtColor(map_data, cv2.COLOR_GRAY2RGB)
    my_path   = os.path.expanduser("~")

    for folder in save_folder_path:
```

```
    my_path = os.path.join(my_path, folder)

  video_path  = os.path.join(my_path, "chris_collins_solution_proj3_" + algo + ".mp4")

  if os.path.exists(video_path):
    os.remove(video_path)

  # Define the codec and create VideoWriter object
  fourcc = cv2.VideoWriter_fourcc(*'mp4v')
  writer = cv2.VideoWriter(video_path, fourcc, fps, (w, h))

  for i in range(len(solution_path)-1):
    # Draw solution path on map_data
    solution_path_xy  = (int(solution_path[i][0]), int(solution_path[i][1]) )
    solution_path_xy1  = (int(solution_path[i+1][0]), int(solution_path[i+1][1]) )

    cv2.line(color_map, solution_path_xy, solution_path_xy1, (0, 0, 255), 1) # Draw line
between current and next node
    frame_inverted = color_map.copy()      # Copy to ensure we don't draw on same frame
from previous iteration
    frame_inverted = cv2.flip(color_map, 0) # Flip to ensure bottom-left origin
    writer.write(frame_inverted)          # Write frame to video
  writer.release()

def explored_path_video(map_data, explored_path, save_folder_path, algo="A_Star",
solution_path=None, goal_reached=None):
  """
```

```
Create a video of the explored path and solution path

For large search cases, we skip frames to reduce run time

by only plotting every len(explored_path) // 1000 frames


explored_path:      List of states explored by algorithm

save_folder_path:   Path to save  video

algo:               Algorithm used to find path (A_Star, Dijkstra, etc.)

solution_path:      List of states in the solution path (optional)

goal_reached:       Goal state reached by the algorithm (optional)


Returns: None


"""


n           = len(explored_path)

video_length = 10 # Seconds

fps         = 100

num_frames   = video_length * fps # FPS

my_path      = os.path.expanduser("~")

if n < num_frames:

   skip = 1

else:

   skip        = n // num_frames # Skip frames to ensure we get all explored nodes in video


h, w      = map_data.shape

color_map = map_data.copy()
```

```
    color_map = cv2.cvtColor(map_data, cv2.COLOR_GRAY2RGB)
    color_map_inverted = cv2.flip(color_map, 0)
    start_state       = explored_path[0]

    for folder in save_folder_path:
        my_path = os.path.join(my_path, folder)

    video_path = os.path.join(my_path, "chris_collins_explored_proj3_" + algo + ".mp4")
    if os.path.exists(video_path):
        os.remove(video_path)

    # Define the codec and create VideoWriter object
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    writer = cv2.VideoWriter(video_path, fourcc, fps, (w, h))

    # Draw start and goal states on map_data
    cv2.circle(color_map_inverted, (int(start_state[0]), int((h-1) - start_state[1])), 3, (255, 0, 0),
5)
    cv2.circle(color_map_inverted, (int(goal_reached[0]),  int((h-1) - goal_reached[1])),  3,
(255, 0, 0), 5)
    writer.write(color_map_inverted)

    for i, node in enumerate(explored_path):
            # Only draw/write every skip expansions
            if i % skip == 0:

                x, y = int(node[0]), int(node[1])
```

```
            y_cv = (h - 1) - y  # Flip y coordinate to match OpenCV's (row, col) convention
            cv2.circle(color_map_inverted, (x, y_cv), 1, (0, 255, 0), -1)
            writer.write(color_map_inverted)

    for i in range(len(solution_path)-1):
        # Draw solution path on map_data
        solution_path_xy  = (int(solution_path[i][0]),    (h-1) - int(solution_path[i][1]) )
        solution_path_xy1  = (int(solution_path[i+1][0]), (h-1) - int(solution_path[i+1][1]) )

        cv2.line(color_map_inverted, solution_path_xy, solution_path_xy1, (0, 0, 255), 3)
        writer.write(color_map_inverted)

    writer.release()


def generate_random_state(map_img, obstacles):
    """
    Generate a random state within the map_data that is not an obstacle

    map_img:   Map with obstacles
    obstacles: Set of obstacle coordinates

    Returns:   Random state within map_data that is not an obstacle
    """
    h, w = map_img.shape
    while True:
```

```
        x    = np.random.randint(0, w-2)
        y    = np.random.randint(0, h-2)
        theta = np.random.randint(0, 12, 1)[0] * 30 # Random theta in [0, 360) degrees

        if (x, y) not in obstacles:
            return (x, y, theta)




def main(generate_random=True, start_in=(5, 48, 30), goal_in=(175, 2, 30),
save_folder_path=None, algo='Dijkstra', r=1):
    '''
    Main function to run A_Star Search to find lowest cost / shortest path from start to goal
state
    and create videos of solution path and explored path

    generate_random:  Boolean to generate random start/ goal state (if True) or use user
provided start/ goal state (if False)
    start_in:        User provided start state
    goal_in:         User provided goal state
    save_folder_path: List of folder names from root to save videos
    '''

    found_valid = False
    start      = time.time()
```

```
    # Step 1: Create Map with Obstacles, Map with Obstacles + 2mm clearance and Cost
Matrix
    (map_data, map_with_clearance, cost_matrix, obstacles
    )= get_map_cost_matrix()
    map_data_wit_clearance   = map_with_clearance.copy()   # Copy map_data with
obstacles and clearance

    end = time.time()

    # Step 2: Get Start/ Goal State, either from user or generate random valid start/ goal state
    if generate_random: # Generate Random Start/ Goal State
        while not found_valid:
            start_state = generate_random_state(map_data_wit_clearance, obstacles)
            goal_state  = generate_random_state(map_data_wit_clearance, obstacles)
            if is_valid_move(start_state, map_data_wit_clearance) and is_valid_move(goal_state,
map_data_wit_clearance):
                found_valid = True
                print("Start State: ", start_state)
                print("Goal State: ", goal_state)
                break

            if not is_valid_move(start_state, map_data_wit_clearance):
                start_state = generate_random_state(map_data_wit_clearance, obstacles)

            if not is_valid_move(goal_state, map_data_wit_clearance):
                goal_state  = generate_random_state(map_data_wit_clearance, obstacles)
```

```
    else: # Use User Provided Start/ Goal State
        start_state, goal_state = check_validity_with_user(map_data_wit_clearance, start_in,
goal_in) # Use User Provided Start/ Goal State

    # Step 3: Run Search Algorithm
    start = time.time()

    # Run A* Algorithm
    (solution_path, cost_to_come, parent, cost_matrix, explored_path, V, goal_state_reached
    ) = a_star(start_state, goal_state, map_data_wit_clearance, cost_matrix, obstacles, r=r)

    # Plot Heat Map of Cost Matrix
    plot_cost_matrix(cost_matrix, start_state, goal_state_reached, title=f"Cost Matrix
Heatmap {algo}" )

    # Plot "Heat Map" of V Matrix
    V_2d = np.sum(V, axis=2)
    plot_cost_matrix(V_2d, start_state, goal_state_reached, title=f"V Matrix Heatmap {algo}"
)



    if solution_path:
        # Create Videos of Solution Path and Explored Path
        solution_path_video(map_data_wit_clearance, solution_path, save_folder_path,
algo=algo)
```

```
        explored_path_video(map_data_wit_clearance, explored_path, save_folder_path,
algo=algo, solution_path=solution_path, goal_reached=goal_state_reached)

    else:
        print("No solution found, aborting video generation")
    end = time.time()
    print("Time to Find Path, Plot Cost Matrix, and create videos: ", round((end-start), 2), "
seconds")

    return start_state, goal_state, map_data, map_with_clearance, cost_matrix, obstacles,
solution_path, cost_to_come, parent, explored_path, V

def a_star(start_state, goal_state, map_data_wit_clearance, cost_matrix, obstacles, r=1):
    """
    Perform A* Search to find shortest path from start state to goal state based on provided
map
    and an 8-connected grid.

    Data Structure and Algorithm are same as Dijkstra's Algorithm, but we use Euclideon
distance to goal
    from current state as our heuristic function + cost to come to current state.

    Parameters:
        start_state:      Initial state of point robot as tuple of (x, y) coordinates
        goal_state:       Goal state of point robot as tuple of (x, y) coordinates
```

map_data: Map with obstacles

cost_matrix: Cost matrix with obstacles as -1 and free space as infinity

obstacles: Set of obstacle coordinates

Returns:

solution_path: List of states from the start state to goal state

cost_to_come: Dictionary of cost to reach each node

parent: Dictionary mapping child states to parent states

cost_matrix: Cost matrix with updated costs to reach each node

explored_path: List of all nodes expanded by the algorithm in search

V: Visited nodes matrix with 1 for visited nodes and 0 for unvisited nodes

goal_state_reached: Goal state reached by the algorithm

Steps:

1. Initialize Open List (priority queue) and Closed List (cost_to_come dictionary)

2. Add start state to Open List with cost to come + euclideon distance to reach goal

3. While Open List is not empty:

    1. Pop node with lowest cost_to_come + cost_to_go from Open List

    2. Check if node is within 1.5 mm of goal state, if it is, generate path and break loop

    3. Check if node has higher cost than previously found cost, if it does, skip and continue

    4. Add node to Closed List

    5. Generate possible moves from current node

    6. For each possible move:

        1. Check if move is valid and not an obstacle

        2. Calculate cost to reach next node

3. Check if next node has not been visited or if new cost is lower than previous cost to reach node

4. If so, update cost_to_come, parent, and cost_matrix and add node back to Open List

5. If not, skip and continue

7. If no solution found, return None

```
    """
    solution_path = None
    pq          = []                              # Open List
    cost_to_come  = {}                              # Closed List
    explored_path = []                              # List of all nodes expanded in search
    parent       = {start_state: None}                    # Dictionary to map child->parent to
backtrack path to goal state
    f_start      = euclidean_distance(start_state, goal_state) # Heuristic function for start state
    thresh       = 0.5
    V          = np.zeros(                          # Visited Nodes
                (int(map_data_wit_clearance.shape[0] / thresh),
                 int(map_data_wit_clearance.shape[1] / thresh),
                   12)
              )

    start_state, x_v_idx, y_v_idx, theta_v_idx    = round_and_get_v_index(start_state)
    print("Starting A_Star Search for:")
    print("Start State: ", start_state)
    print("Goal State: ", goal_state)
```

```
    cost_to_come[(y_v_idx, x_v_idx, theta_v_idx)] = 0.0      # cost_to_come is our Closed List
    cost_matrix[y_v_idx, x_v_idx]    = f_start             # we'll store cost to reach node + heuristic
cost to reach goal
    V[y_v_idx, x_v_idx, theta_v_idx] = 1
    goal_reached = goal_state

    heapq.heappush(pq, (f_start, start_state))   # pq is our Open List

    while pq:
        curr_f, curr_node = heapq.heappop(pq) # Pop node with lowest cost from priority
queue

        curr_node_round, curr_x_v_idx, curr_y_v_idx, curr_theta_v_idx =
round_and_get_v_index(curr_node) # Round to nearest half
        curr_cost_node = (curr_y_v_idx, curr_x_v_idx, curr_theta_v_idx) # Get cost node for
current node
        V[curr_y_v_idx, curr_x_v_idx, curr_theta_v_idx] = 1

        if euclidean_distance(curr_node_round, goal_state) <= 1.5:           # If goal state
reached, generate path from start to gaol and break the loop
            solution_path = generate_path(parent, curr_node_round)
            print("Found Solution to Goal:")
            print(goal_state)
            print("Cost: ", cost_to_come[curr_cost_node])
            goal_reached = curr_node_round
            break
```

```
    if curr_f > cost_to_come[curr_cost_node] + euclidean_distance(curr_node,
goal_state):   # If we've found lower cost for this node,
        continue                        # skip and don't expand this node
    # else:                             # Only add node to explored path if it is visited and expanded
    #     explored_path.append(curr_node)       # If we've found a lower cost for the node,
then we have already explored it

    possible_moves = [  move_theta_0(      curr_node, r),
                move_diag_up_30(   curr_node, r),
                move_diag_up60(    curr_node, r),
                move_diag_down_30( curr_node, r),
                move_diag_down60(  curr_node, r),
                ]

    for next_node, next_cost in possible_moves:   # For each move, check if it is valid and
not an obstacle
        next_node_round, next_x_v_idx, next_y_v_idx, next_theta_v_idx =
round_and_get_v_index(next_node)
        next_cost_node = (next_y_v_idx, next_x_v_idx, next_theta_v_idx)
        next_v_node    = (next_y_v_idx, next_x_v_idx, next_theta_v_idx)

        valid_move   = is_valid_move(next_node_round, map_data_wit_clearance)
        not_obstacle = (next_node_round[0], next_node_round[1]) not in obstacles

        if valid_move and not_obstacle:     # Check if next node is valid and not on top of an
obstacle
```

```
            # We don't use our heuristic function here, we just use the cost to come to the current node + cost to reach next node
            # This is the parameter we want to minimize, but we use the heuristic function to prioritize our queue
            new_cost = cost_to_come[curr_cost_node] + next_cost

            # Check if next has not been visited or if new cost is lower than previous cost to reach node
            # For cases where we've found a lower cost to reach a node, we update the cost_to_come, parent, and cost_matrix
            # and add the node back-in to the priority queue without removing the old node, if the old node is reached again
            # we skip it with the continue statement above

            visited = (V[next_y_v_idx, next_x_v_idx, next_theta_v_idx] == 1)

            if (not visited) or (new_cost < cost_to_come.get(next_v_node, float('inf')) ):

              explored_path.append(next_node_round)
              cost_to_come[next_cost_node] = new_cost

              parent[next_node_round]        = curr_node_round
              # Add Heurstic cost to reach goal to cost to come to current node for prioritization
              f_next                = new_cost + euclidean_distance(next_node_round, goal_state)
              heapq.heappush(pq, (f_next, next_node_round))
              cost_matrix[next_y_v_idx, next_x_v_idx] = new_cost
```

```
                V[next_y_v_idx, next_x_v_idx, next_theta_v_idx] = 1

    if solution_path is None:
        print("No Solution Found")

    print("A_star Expanded States: ", len(explored_path))

    return solution_path, cost_to_come, parent, cost_matrix, explored_path, V, goal_reached



# %%  Main Block to Run Test Case

found_valid = True
start      = time.time()
algo       = "A_star"
save_folder_path = ["Dropbox", "UMD", "ENPM_661 - Path Planning for Robots",
"ENPM661_Project03_Phase01"]
generate_random = False
start_in = (10, 48, 30)
goal_in  = (500, 220, 30)
r      = 1

if __name__ == "__main__":
    save_folder_path = ["Dropbox", "UMD", "ENPM_661 - Path Planning for Robots",
"ENPM661_Project03_Phase01"]
    algo          = "A_star"
```

```
(start_state, goal_state, map_data, map_with_clearance, cost_matrix, obstacles,
solution_path, cost_to_come, parent, explored_path, V
) = main(
    generate_random=generate_random, start_in=start_in, goal_in=goal_in,
save_folder_path=save_folder_path, algo=algo, r=r)
```