# %%

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)