CS471: Introduction to Artificial Intelligence
Assignment 1: Python

Andrew Ortiz

9/3/24

Link to my Assignment:
https://github.com/andrew-ortiz029/CS-471-AI/tree/main/Circle%20Clusters%20Assignment

For this problem, you are only allowed to use standard python libraries. You may not use third party libraries or call any shell/bash functions.

You are given a list of tuples of the form (<float> x, <float> y, <float> r) (Let's call these c-tuples). Each c-tuple represents a circle on a rectangular coordinate space, with x and y being the coordinates of the center, and r being the radius. Assume that each c-tuple has a unique radius.

Let a cluster of circles be a group of circles where each circle in the group overlaps with at least one other circle in that group. A path is formed between two circles when they overlap. Define a cluster as a group of n circles, where each circle is reachable from every other circle through the formed paths.

Write a python script that does the following: Return True if the given circles form a cluster and return false if they don't form a cluster.

Below are some test cases.

Test case 1:

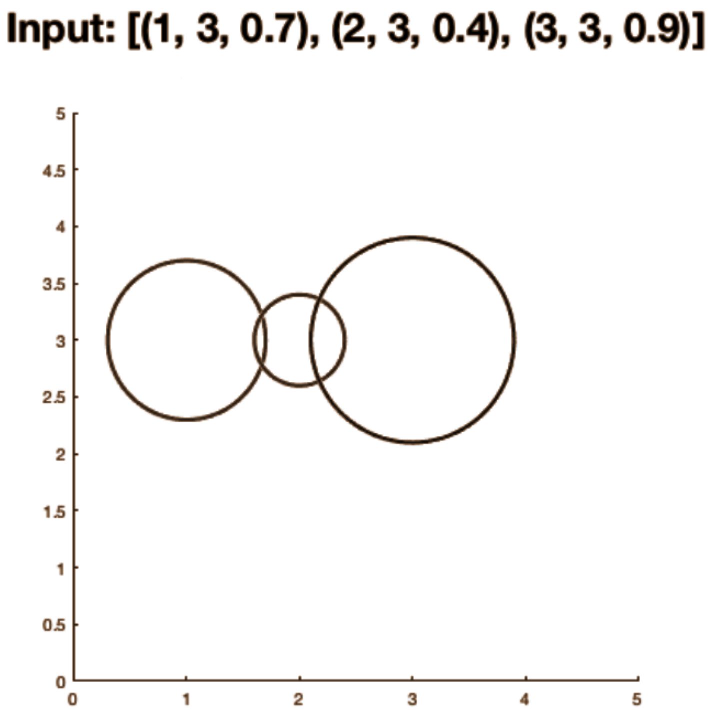
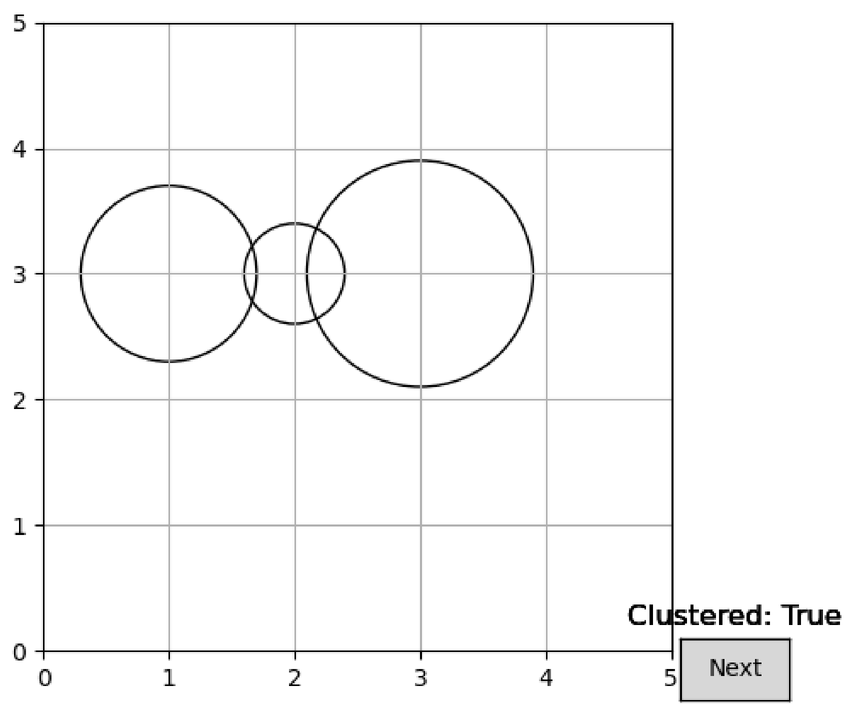


Figure 1: The three circles form the cluster. Output = True

My output:



Test case 2:

Input: [(1.5, 1.5, 1.3), (4, 4, 0.7)]

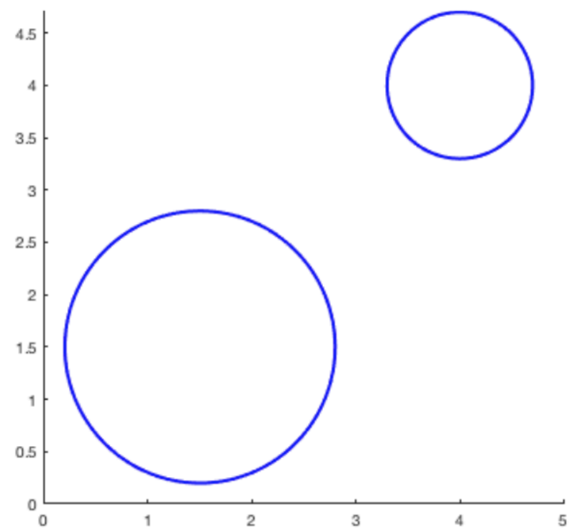
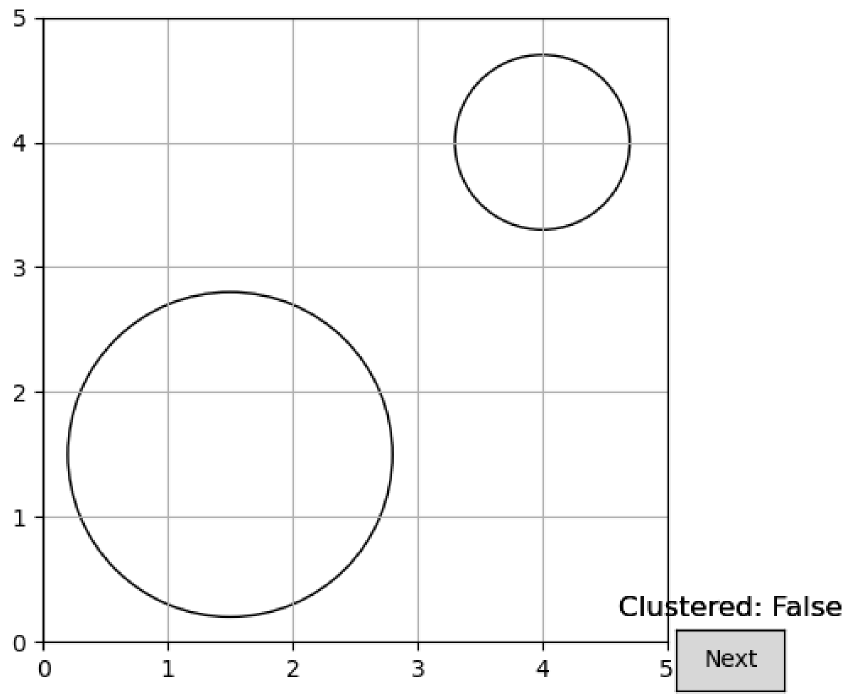


Figure 2: No clusters are found. Output = False

My Output:



Test case 3:

Input: [(0.5, 0.5, 0.5), (1.5, 1.5, 1.1), (0.7, 0.7, 0.4), (4, 4, 0.7)]

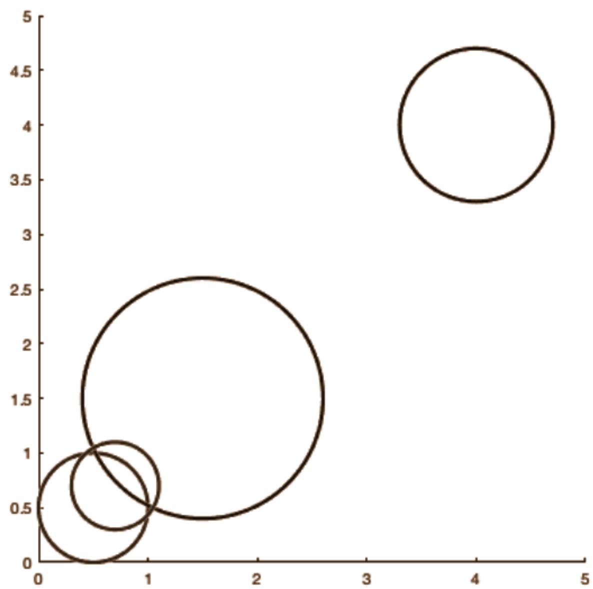
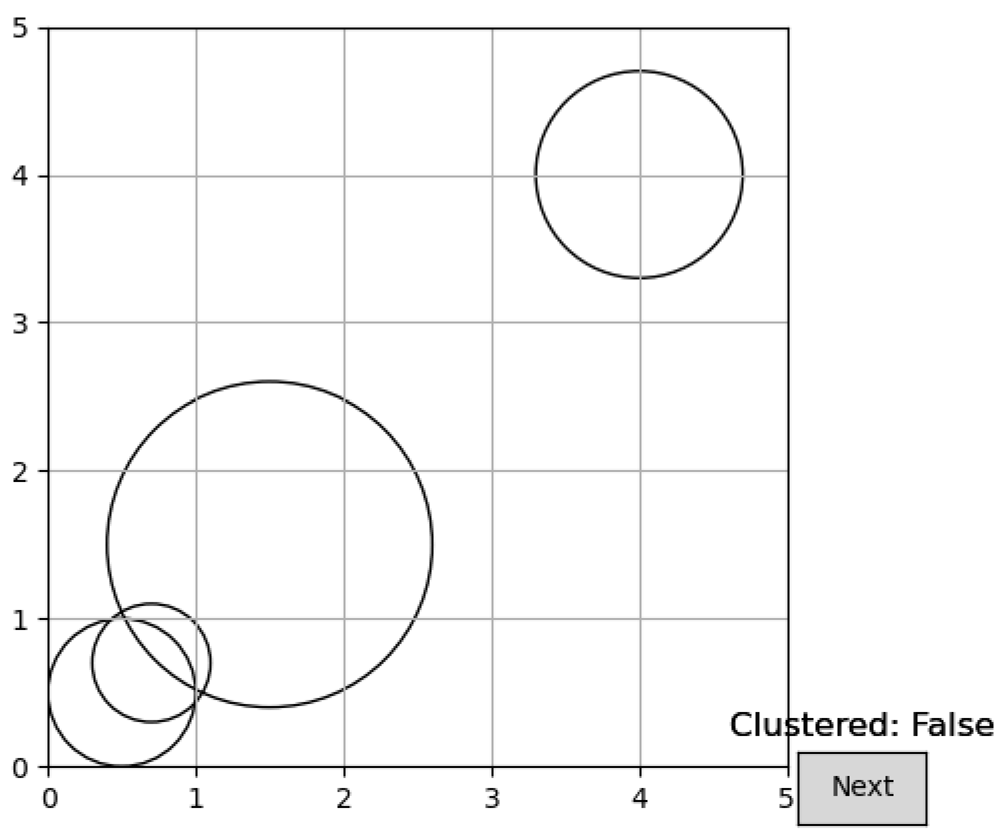


Figure 3: Given circles do not form a cluster. Output = False

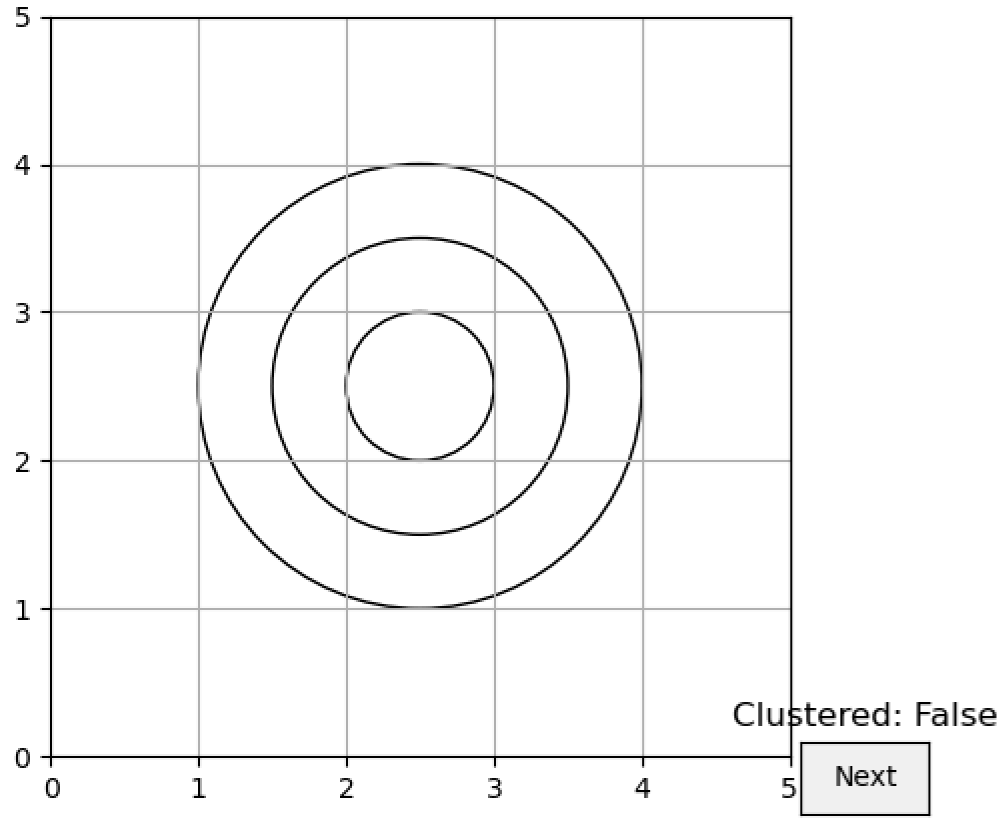
My Output:



Along with the above three test cases, design a fourth test case of your choice. Share in detail the approach you used to solve the problem.

Test Case 4:

My Output:



My Solution

So, before I started to solve the problem I began with structuring the test cases into a list. The final structure I went with was putting every test case into a list that contains more lists of tuples.

```
# Test cases list of lists containing tuples
test_cases = [[(1, 3, 0.7), (2, 3, 0.4), (3, 3, 0.9)],
              [(1.5, 1.5, 1.3), (4, 4, 0.7)],
              [(0.5, 0.5, 0.5), (1.5, 1.5, 1.1), (0.7, 0.7, 0.4), (4, 4, 0.7)],
              [(2.5, 2.5, 1.5), (2.5, 2.5, 1), (2.5, 2.5, .5)]]
```

This structure made it easy to iterate through the test cases for the graph_circles function to show the results of the circles. The function takes in a list and loops through its tuples to pull the values of each circle to graph, then pushes that same test case list to be processed by the cluster_check function to report whether or not it's a cluster.

```python
# function that takes in a list and adds the test cases to the graph
def graph_circles(circles):
    ax.clear()
    for x, y, radius in circles:

        # Create a circle for each circle in the test cases
        circle = patches.Circle((x, y), radius, edgecolor='black', facecolor='none')

        # Add the circle to the axis
        ax.add_patch(circle)

        # Set limits
        ax.set_xlim(0, 5)
        ax.set_ylim(0, 5)

        # Set equal scaling
        ax.set_aspect('equal')

        # Add grid for better visualization
        ax.grid(True)

        # Draw out the graph
        plt.draw()

    # Send to cluster_check for T/F output
    if cluster_check(circles) == True:
        plt.title("Clustered: True")
    else:
        plt.title("Clustered: False")
    #plt.title("Clustered: True")
```

The graph_circles function is called by an event handler function because of a button I added to the graph in order to iterate and observe all test cases.

```python
# Event handler for the 'Next' button
def next_graph(event):
    global index
    graph_circles(test_cases[index])
    if index == 3:
        index = 0
    else:
        index = index + 1 # next test case

# Create a button for next graph
next_button_ax = plt.axes([0.81, 0.05, 0.1, 0.075])
next_button = Button(next_button_ax, 'Next')
next_button.on_clicked(next_graph)
```

After I got those functions working together to graph the test cases, I moved on to checking the test cases for whether or not it was a cluster. The algorithm I chose to minimize comparisons between each circle was a modified DFT (depth first traversal). I would start with the first circle in the list and and traverse check each circle to see if it intersected with the first. If it did intersect with the first, I would mark it as visited in a list I made to indicate that it will be visited because of the intersection with the first. It would then be ‘pushed’ into a stack to then later be popped for traversal in the DFT driver loop. Because I’m not using DFT to plot out points and edges etc, the first instance that all circles have been marked as visited will stop the traversal, as this means that the graph is a cluster.

```
# List to keep track of visited circles
visited = [False] * len(circles)

# Stack to keep track of DFT
stack = []

# iterate through circle 0 connections and initialize the stack with that
x1 = circles[0][0]
y1 = circles[0][1]
r1 = circles[0][2]

visited[0] = True

i = len(circles) - 1
while i > 0:
    x2 = circles[i][0]
    y2 = circles[i][1]
    r2 = circles[i][2]

    # Equation for computing distance
    distance = math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
# If distance is less than the sum of the current radius then push that circl
# They get pushed as to check the next cirlces on the stack for insterction i
if distance < r1 + r2 and distance + min(r1, r2) > max(r1, r2):
    stack.append(i) # Push i into the stack for circles to traverse
    visited[i] = True # They are intersected so mark the circles are visited

i -= 1

# If all circles have been visited then return true as the graph is clustered
# If no circles have been visited then the first circle is not clusted and just return false
if visited == [True] * len(circles):
    return True
elif visited == [False] * len(circles):
    return False
```

```python
# Main DFT control will run as long as there's a connection in the stack or until returned
while stack: # run until stack is empty
    i = stack[0]
    stack.pop()

    x1 = circles[i][0]
    y1 = circles[i][1]
    r1 = circles[i][2]

    # Now check the non visited circles for intersection
    if visited == [True] * len(circles):
        return True
    else:
        first_non_visited = visited.index(False)

    x2 = circles[first_non_visited][0]
    y2 = circles[first_non_visited][1]
    r2 = circles[first_non_visited][2]

    # Equation for computing distance
    distance = math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))

    if distance < r1 + r2 and distance + min(r1, r2) > max(r1, r2):
        stack.append(first_non_visited) # Push i into the stack for circles to traverse
        visited[first_non_visited] = True # They are intersected so mark the circles are visited

# If all circles have been visited then return true as the graph is clustered else return false
if visited == [True] * len(circles):
    return True
else:
    return False

# If all circles have been visited then return true as the graph is clustered else return false
if visited == [True] * len(circles):
    return True
else:
    return False
```