CS471: Introduction to Artificial Intelligence
Assignment #1 - Python

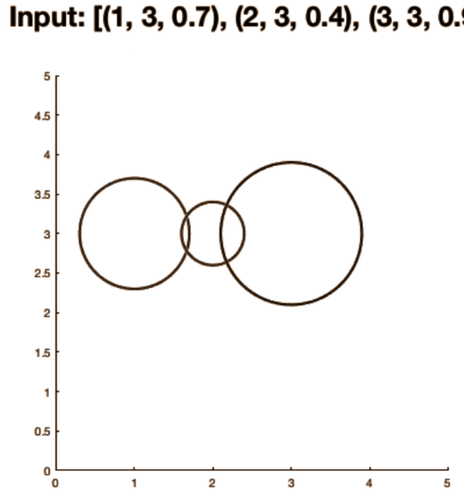
Malcolm Zartman
09/05/2025

Question 1: Python
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.

Test case 1:



Input: [(1, 3, 0.7), (2, 3, 0.4), (3, 3, 0.9)]

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

<u>Test case 2:</u>

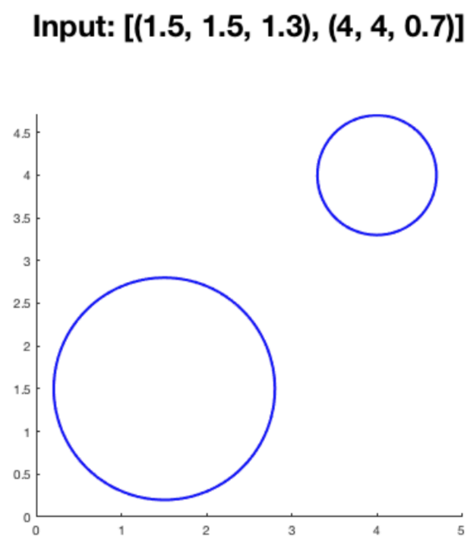


Figure 2: No clusters are found. Output = False

<u>Test case 3:</u>

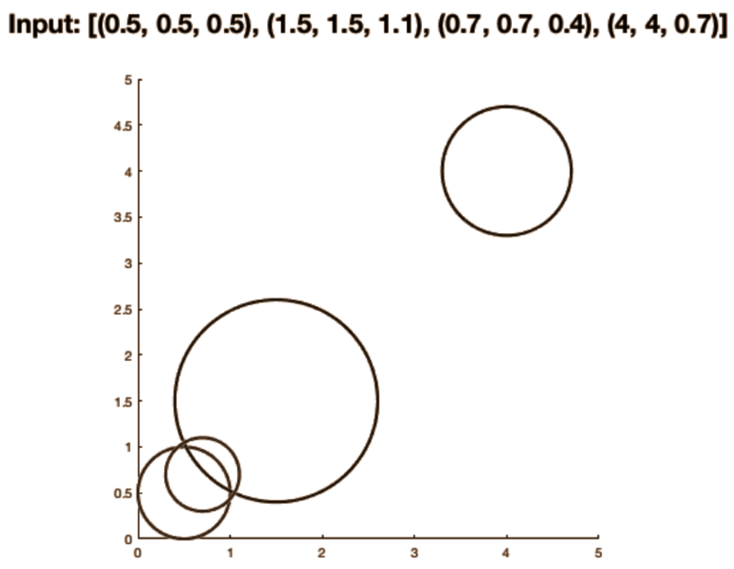


Figure 3: Given circles do not form a cluster. Output = False

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.

Colab link:
https://colab.research.google.com/drive/1SoTNKPLdaOpH69g-ddCE-0e6DymXnnXb?usp=sharing

Before I determine whether or not the circles form a cluster, I will create a helper function to figure out whether or not a pair of circles overlap or not.

Initialization of the ‘doesOverlap(t1,t2)’ function that uses the distance formula to see if two circles overlap or not:

```
import math

# helper function to see if two circles overlap with each other
# the parameters consist of two tuples that contain (x,y) coordinates of
the...
# center of the circles, as well as the radius of the circles: (x,y,r)
def doesOverlap(c_tuple1, c_tuple2):
  # getting the x,y coordinates & the radius from inputted tuples
  x1, y1, r1 = c_tuple1[0], c_tuple1[1], c_tuple1[2]
  x2, y2, r2 = c_tuple2[0], c_tuple2[1], c_tuple2[2]
  # finding the distance between the centers of the circles
  distance = math.sqrt((x2-x1)**2 + (y2-y1)**2)
   # if the sum of the radii is greater than or equal to the distance, then
they overlap
  if distance <= (r1+r2):
    return True
  else:
    return False
```

Below is the first half of the ‘isCluster(tList)’ that takes a list of tuples and determines if each set of circles overlaps with each other and then makes a resulting graph from that:

```
# function to check if the inputted tuples are a cluster
def isCluster(list_of_cTuple):
  n = len(list_of_cTuple)
  # if there are 1 or less circles that automatically counts as a cluster
  if n <= 1:
    return True

  # creating an empty adjacency list the size of the tuple list
  adj_list = [[] for _ in range(n)]
  # looping through each set of circles to see if they overlap
  for i in range(n):
    for j in range(i + 1, n):
        #  if  they  overlap  add  them  to  the  adjacency  list  otherwise  do
nothing
      if doesOverlap(list_of_cTuple[i], list_of_cTuple[j]):
        # this creates a list of lists (a graph) where for every cirlce it
has a list of circles it overlaps with
        adj_list[i].append(j)
        adj_list[j].append(i)
```

After looping through the entire tuple list, I will use the adjacency list or essentially all of the circles that overlap with another, along with the Depth-First Search algorithm to determine whether or not every circle overlaps to form a cluster.

Here are the steps for my ‘dfs(node)’ algorithm:

- Add the current node (a circle's index) to a visited set to not “recount” it again.
- For each neighbor, if it hasn't been visited yet, recursively call the dfs function on that neighbor.
- The function starts with a call on the first circle, dfs(0), to begin the traversal.

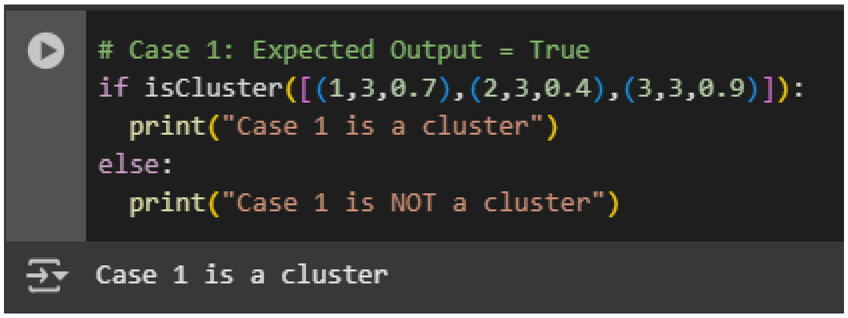
Once I finish processing each node, I will now have a count of how many circles overlap with another, and if that matches the total amount of circles in the input list, then we have a cluster, otherwise not.

```
   #  initializing  a  visited  set  (efficent  for  checking)  to  keep  track  of
circles already visited
  visited = set()
  # using Depth-First Search to efficiently traverse through the graph
  def dfs(node):
    # once visited, add to list in order to not process it again
    visited.add(node)
     # for each node in the adjacency list, we call the function again for
the ones we haven't visited yet
```

```
    for neighbor in adj_list[node]:
      if neighbor not in visited:
        dfs(neighbor)
    # once we have visited all nodes in the adj_list, the function ends.

  # start the recursive function
  dfs(0)
   # if the number of nodes we visited matches the length of the tuple
list, we have a cluster
  return len(visited) == n
```

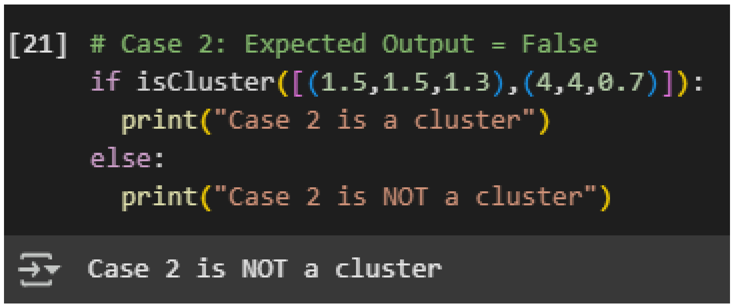
Test case: 1



```
# Case 1: Expected Output = True
if isCluster([(1,3,0.7),(2,3,0.4),(3,3,0.9)]):
  print("Case 1 is a cluster")
else:
  print("Case 1 is NOT a cluster")
```

```
Case 1 is a cluster
```

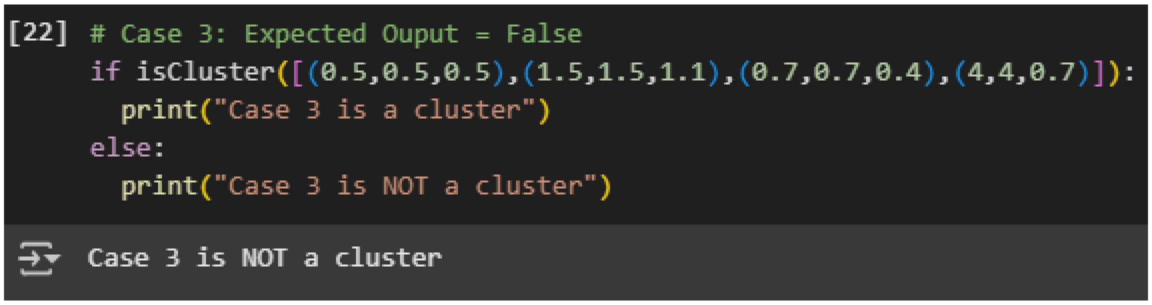
Test case: 2



```
[21] # Case 2: Expected Output = False
     if isCluster([(1.5,1.5,1.3),(4,4,0.7)]):
       print("Case 2 is a cluster")
     else:
       print("Case 2 is NOT a cluster")
```

```
Case 2 is NOT a cluster
```

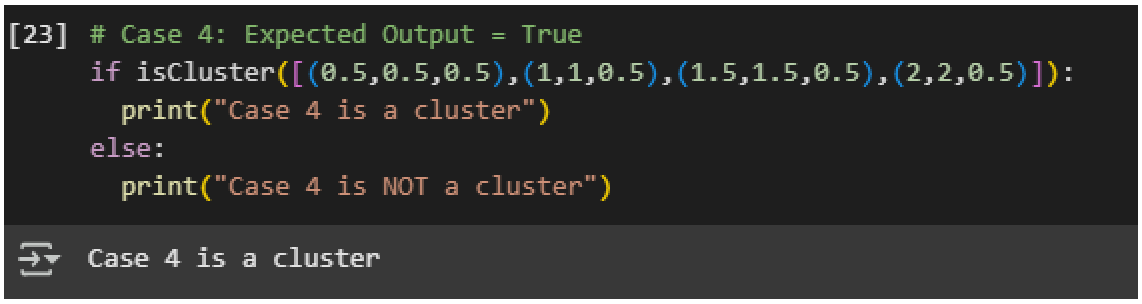
Test case 3:



```
[22] # Case 3: Expected Ouput = False
     if isCluster([(0.5,0.5,0.5),(1.5,1.5,1.1),(0.7,0.7,0.4),(4,4,0.7)]):
       print("Case 3 is a cluster")
     else:
       print("Case 3 is NOT a cluster")
```

```
Case 3 is NOT a cluster
```

Test case 4:



```
[23] # Case 4: Expected Output = True
     if isCluster([(0.5,0.5,0.5),(1,1,0.5),(1.5,1.5,0.5),(2,2,0.5)]):
       print("Case 4 is a cluster")
     else:
       print("Case 4 is NOT a cluster")
```

```
Case 4 is a cluster
```

Input: [(0.5,0.5,0.5),(1,1,0.5),(1.5,1.5,0.5),(2,2,0.5)]

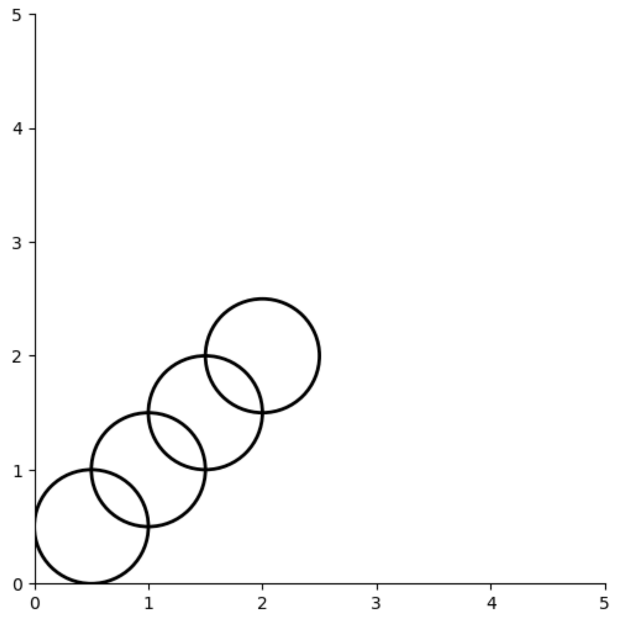


Figure 4: The four circles form a cluster. Output = True

Code for visualization:

```python
import matplotlib.pyplot as plt
import matplotlib.patches as patches

circles = [(0.5, 0.5, 0.5), (1, 1, 0.5), (1.5, 1.5, 0.5), (2, 2, 0.5)]
fig, ax = plt.subplots(figsize=(6, 6))

ax.set_xlim(0, 5)
ax.set_ylim(0, 5)

for x, y, r in circles:
    circle = patches.Circle((x, y), r, fill=False, edgecolor='black',
linewidth=2)
    ax.add_patch(circle)

ax.set_aspect('equal', adjustable='box')
# to match the example test cases
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

plt.grid(False)
plt.show()
```