1924. Erect the Fence II

Problem Explanation

We are provided with a 2D integer array `trees` that represents the multiple locations of the trees in a garden. Each location is represented by a pair [x,y] where x & y are coordinates of a tree in the garden. We need to find the smallest circle that would enclose all the trees in the garden.

The garden is considered well-fenced only if all the trees are enclosed and the rope forms a perfect circle. A tree is considered enclosed if it resides within or is on the border of the circle. So we need to return the coordinates of the center of the circle [x,y] and its radius 'r' as a length 3 array [x,y,r]. Answers within 10-5 of the actual answer will be accepted.

For example consider a garden that has two trees with coordinates [1,2] and [2,2]. In this case, the smallest enclosing circle would have center coordinates as [1.5,2] with a radius of 0.5. So, the output would be [1.5,2,0.5]

Solution Approach

We can solve this problem using Welzl's algorithm, famously known for solving the minimal enclosing circle problem.

The idea of the algorithm is simple. It recursively constructs the smallest enclosing circle from the given points. Based on the principle of Welzl's algorithm, the smallest disk is either the smallest disk with 0 points on the boundary (which we return Disk(Point(0, 0), 0) for), 1 point on the boundary (which we return the Disk with center at this point and radius 0 for), 2 points on the boundary (which we calculate a Disk that just covers these 2 points with getDisk() function for), or 3 points on the boundary (which we calculate a Disk that just covers these 3 points with getDisk() function for). If we use the trivial() function to return the Disk for these 4 cases.

The Welzl's algorithm is based on the following main tasks:

1. Finding the smallest disk that encloses all points: we can solve this problem recursively by iterating over all points and if a point lies outside of current minimal disk, then we solve the problem again with this point added to boundary points.
2. Checking if a point is within a disk: we can use the Euclidean distance from the point to the center of the disk which should be less than or equal to the radius of the disk.
3. Finding the smallest disk with 2 points on the border: The center of the disk is the midpoint of the 2 points and the radius is half the distance between the 2 points.
4. Finding the smallest disk with 3 points on the border: The center of the disk is the intersection of the perpendicular bisectors of the line segments made by the 3 points. The radius is the distance from the center to any of the 3 points.

This algorithm is slightly complex but efficient and provides the most optimal solution for the problem.

Detailed Example

Let's consider a simple example.

Input: [[1,2], [2,2], [1,1]]

```
2  x x
1  x
0
  0 1 2 3
```

The enclosing circle for the points have a center at [1.5,1.5] and radius 0.707107. Hence [[1.5,1.5,0.707107]] is the output.

In the first iteration, the algorithm calls itself with the first point on the boundary. Because that is one point on the boundary the smallest disc enclosing rest of the points as well as those boundary points is centered at [1,2] with radius 0. In the next recursion the second point [2,2] is also added to boundary points. Now as there are two boundary points the smallest disc is calculated centered at midpoint of [1,2] and [2,2] i.e., at [1.5, 2] and with radius 0.5. The algorithm continues until all points are considered.

Python Solution

```
python
import math

class Point:
    def   init  (self, x, y):
        self.x, self.y = x, y

class Disk:
    def   init  (self, center, radius):
        self.center, self.radius = center, radius

class Solution:
    def outerTrees(self, trees):
        points = [Point(x, y) for x, y in trees]
        disk = self.welzl(points, [], len(points))
        return [disk.center.x, disk.center.y, disk.radius]
```

Java Solution

```
java
import java.util.ArrayList;
import java.util.List;

class Solution {
    // define the structure of Point and Disk as in Python solution

    public double[] outerTrees(int[][] trees) {
        List<Point> points = new ArrayList<>();
        for (int[] tree : trees) {
            points.add(new Point(tree[0], tree[1]));
        }
        Disk disk = welzl(points, new ArrayList<>(), points.size());
        return new double[]{disk.center.x, disk.center.y, disk.radius};
    }

    // insert the rest functions here

}
```

JavaScript Solution

```
javascript
class Solution {
    // define the constructor, same as in Python solution

    outerTrees(trees) {
        let points = trees.map(tree => new Point(tree[0], tree[1]));
        let disk = this.welzl(points, [], points.length);
        return [disk.center.x, disk.center.y, disk.radius];
    }

    // insert the rest functions here

}
```

C++ Solution

```
cpp
class Solution {
    // define the structure of Point and Disk, similar to Python solution

public:
    vector<double> outerTrees(vector<vector<int>>& trees) {
        vector<Point> points;
        for (auto& tree : trees) {
            points.push_back(Point(tree[0], tree[1]));
        }
        Disk disk = welzl(points, {}, points.size());
        return {disk.center.x, disk.center.y, disk.radius};
    }

    // insert the rest functions here

};
```

C# Solution

```
csharp
public class Solution {
    // define the structure of Point and Disk, similar to Python solution

    public double[] outerTrees(int[][] trees) {
        List<Point> points = new List<Point>();
        foreach (var tree in trees) {
            points.Add(new Point(tree[0], tree[1]));
        }
        Disk disk = welzl(points, new List<Point>(), points.Count);
        return new double[]{disk.center.x, disk.center.y, disk.radius};
    }

    // insert the rest functions here

}
```

Implementation Details

After defining Point class with coordinates x and y and Disk class with point and radius, we can start implementing the Welzl's algorithm in detail as mentioned below.

`dist(a, b)`: This method takes two parameters as two points and calculates the Euclidean distance between them. We calculate the difference between each coordinate, square it, add them together and finally return the square root of that sum.

`getDisk(a, b)`: This method takes two points a and b as parameters and returns a Disk that just encloses both points. The center of the disk is simply the midpoint of a and b, and the radius is half of the distance between a and b.

`getDisk(a, b, c)`: This method accepts three points a, b and c as input and returns a Disk that encloses all three points. The center of the disk can be calculated using the formula for the circumcenter of a triangle and the radius is the distance from the center to any of the three points.

`trivial(points)`: This method takes 0, 1, 2 or 3 points and returns a Disk that encloses all these points. This is a trivial computation for zero points or when all points are on the border of the disk and hence the method trivial().

`welzl(points, boundaryPoints, n)`: This is the main algorithm method that goes over all points and checks if it lies within the current minimum enclosing disk. If it doesn't then we solve the problem again recursively with this point added to the boundary points.

Please note, Disk(center, radius) creates a disk with center and radius:

- For Python, a class named Disk is created where `Disk(Point(0, 0), 0)` creates a disk of zero radius with center at origin
- For Java, use new Disk(new Point(0, 0), 0)
- In JavaScript, use new Disk(new Point(0, 0), 0)
- In C++ and C#, use Disk(Point(0, 0), 0)

"outerTrees()" is the main calling function that transforms the input list of tree co-ordinates to a list of points which is then supplied as an argument to welzl recursive function.

Conclusion

Before wrapping up, it's important to note that the Welzl's algorithm gives the most efficient solution to this problem with an O(n) complexity. While the recursive approach might seem complex initially, understanding the logic behind all the helper functions as explained above makes it easy to comprehend overall. It has wide applications in computational geometry and image processing.

However, it's not the only solution to this problem. The naive approach of checking all possible subsets and picking the smallest enclosing circle is of high complexity but can help understand the problem if you are not familiar with Welzl's algorithm. Other complex methods involve Computational Geometry algorithms such as the Quickhull or Gift wrapping algorithm.