963. Minimum Area Rectangle II

Medium · Geometry · Array · Math — Leetcode Link

Problem Description

In this problem, we are given a set of points in the X-Y plane, with each point represented as a pair of coordinates `[x_i, y_i]`. The task is to find the minimum area of any rectangle that can be formed using these points. Importantly, the sides of these rectangles do not have to be parallel to the X and Y axes, which means we could be dealing with rectangles at any orientation in the plane.

The output should be the area of the smallest such rectangle. If no rectangle can be formed from the given points, we should return `0`. In the answer, a small tolerance is allowed - the returned result must be within `10^-5` (0.00001) of the actual minimum area.

The challenge is to find a way to efficiently check all combinations of points to determine which ones can form the corners of a valid rectangle and calculate the minimum area among all possible rectangles.

Intuition

The solution makes use of a brute-force approach, iterating through each combination of three points `(x1, y1), (x2, y2), (x3, y3)` to determine whether a fourth point exists that would complete a rectangle. For any given three points, the idea is to calculate the fourth point `(x4, y4)` which would form two vectors representing two adjacent sides of the rectangle.

To ensure that these sides are perpendicular (which is a requirement for a rectangle), the dot product of the vectors is calculated. If the dot product is zero, we have a right angle, and thus a candidate for rectangle corners.

Here's the breakdown of the solution:

1. Store all given points in a set for constant time look-up.
2. Iterate over each pair of points `(x1, y1)` and `(x2, y2)`, treating them as adjacent corners of a prospective rectangle.
3. For each pair, explore another point `(x3, y3)` to act as the third corner.
4. Calculate the potential fourth corner's coordinates `(x4, y4)` using vector addition.
5. Check if the calculated fourth point actually exists in the provided points set.
6. Confirm that the angle between the vectors representing potential rectangle sides is a right angle by checking if their dot product is zero.
7. If we have a valid rectangle, calculate its area using the lengths of the vectors, which are the rectangle's sides.
8. Keep track of the minimum area found so far.
9. After all points have been considered, return the minimum area or `0` if no rectangle is found.

By iterating over all possible combinations and considering only those sets of points that pass the perpendicularity test, the solution finds the rectangle with the minimum area. This approach guarantees finding the correct answer, but it may be slow for a large set of points because of its cubic time complexity.

Solution Approach

The Reference Solution Approach relies on algorithmic geometry and set data structures to solve the problem efficiently. Here's a step-by-step breakdown of the solution implementation:

1. **Using a Set for Lookup**: A set `s` is created to store all the points. A set is used here because it allows for O(1) complexity for look-up operations, which is crucial since we want to check the existence of the potential fourth point quickly in our algorithm.

2. **Iterating Over Combinations of Points**: The solution uses three nested loops to iterate over all possible combinations of three different points`(x1, y1)`, `(x2, y2)`, and `(x3, y3)`. The outermost loop picks the first point, the second loop picks the second point, and the third loop picks the third point.

3. **Calculating the Fourth Point**: With `(x1, y1)`, `(x2, y2)`, and `(x3, y3)` determined, the coordinates for the potential fourth point `(x4, y4)` are calculated. This is done by vector addition:

   ```
   x4 = x2 - x1 + x3
   y4 = y2 - y1 + y3
   ```

   This represents the diagonal translation from the first point to the second and then from the first to the third, arriving at the fourth point's location.

4. **Verifying Perpendicular Sides**: To ensure the points can form a rectangle, the algorithm verifies that the vectors representing the sides are perpendicular. This is done by computing the dot product of the two vectors:

   ```
   v21 = (x2 - x1, y2 - y1)  # Vector from point 1 to point 2
   v31 = (x3 - x1, y3 - y1)  # Vector from point 1 to point 3
   ```

   The dot product of `v21` and `v31` should be zero for the sides to be perpendicular:

   ```
   v21[0] * v31[0] + v21[1] * v31[1] == 0
   ```

5. **Calculating Area of Rectangle**: If a valid rectangle is found, its area is calculated using the lengths of the sides:

   ```
   w = sqrt(v21[0] ** 2 + v21[1] ** 2)  # Width of the rectangle
   h = sqrt(v31[0] ** 2 + v31[1] ** 2)  # Height of the rectangle
   ```

   The area `w * h` is computed and compared with the running minimum.

6. **Tracking the Minimum Area**: A variable `ans` is initialized to `inf` (infinity) to keep track of the minimum area seen so far. Whenever a new rectangle area is computed, we update `ans` with the lesser of the current minimum or the new area.

7. **Returning the Result**: After all possible point combinations have been considered, the solution checks if `ans` is still `inf`. If it is, it means no rectangle was found, so we return `0`. If a minimum area was found, we return that minimum area.

This approach systematically explores all combinations of points that could possibly form rectangles, ensuring no potential rectangle is missed. Although the solution's complexity is high (O(n^3)), it is acceptable given that the problem size (the number of points) is not excessively large.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Suppose we are given the following set of points: `[(1, 0), (0, 1), (2, 1), (1, 2)]`, and we need to find the minimum area of a rectangle that can be formed with these points.

Following the solution approach:

1. **Using a Set for Lookup**: The first step is to store all points in a set for quick lookup. Let's define our set `s`:

   ```
   s = {(1, 0), (0, 1), (2, 1), (1, 2)}
   ```

2. **Iterating Over Combinations of Points**: We start iterating over combinations of three different points. Let's take `(x1, y1) = (1, 0)`, `(x2, y2) = (0, 1)`, and `(x3, y3) = (2, 1)`.

3. **Calculating the Fourth Point**: We calculate the potential fourth point `(x4, y4)` using vector addition:

   ```
   x4 = x2 - x1 + x3 = 0 - 1 + 2 = 1
   y4 = y2 - y1 + y3 = 1 - 0 + 1 = 2
   ```

   So the fourth point's coordinates are `(x4, y4) = (1, 2)`.

4. **Verifying Perpendicular Sides**: We have to check if the sides are perpendicular, which is a requirement for a rectangle. We calculate the vectors from point 1 to point 2 `v21`, and from point 1 to point 3 `v31`:

   ```
   v21 = (x2 - x1, y2 - y1) = (0 - 1, 1 - 0) = (-1, 1)
   v31 = (x3 - x1, y3 - y1) = (2 - 1, 1 - 0) = (1, 1)
   ```

   The dot product of `v21` and `v31` is `(-1*1 + 1*1) = 0`. Since the dot product is zero, `v21` and `v31` are perpendicular.

5. **Calculating Area of Rectangle**: Now that we confirmed we have a rectangle, we can calculate its area:

   ```
   w = sqrt(v21[0] ** 2 + v21[1] ** 2) = sqrt((-1) ** 2 + 1 ** 2) = sqrt(2)
   h = sqrt(v31[0] ** 2 + v31[1] ** 2) = sqrt(1 ** 2 + 1 ** 2) = sqrt(2)
   ```

   The area of the rectangle is `w * h = sqrt(2) * sqrt(2) = 2`.

6. **Tracking the Minimum Area**: Since this is our first rectangle, we set the minimum area `ans = 2`.

7. **Returning the Result**: After completing the iterations through all possible point combinations (since our example is small, we only had a single combination to check), we find that the minimum area `ans` is `2`. Therefore, the minimum area of a rectangle that can be formed with our given set of points is `2`.

This simple example demonstrates the steps outlined in the solution approach, and how it can be used to find the minimum area rectangle given a set of points in the X-Y plane.

Python Solution

```python
from typing import List
from math import sqrt, inf

class Solution:
    def minAreaFreeRect(self, points: List[List[int]]) -> float:
        # Create a set of point tuples for O(1) look-up times
        point_set = {(x, y) for x, y in points}

        # Get the number of points
        num_points = len(points)

        # Initialize answer as infinity to track minimum
        min_area = inf

        # Iterate over all points to check for rectangles
        for i in range(num_points):
            x1, y1 = points[i]
            for j in range(num_points):
                if j != i:
                    x2, y2 = points[j]
                    for k in range(j + 1, num_points):
                        if k != i:
                            x3, y3 = points[k]
                            # Calculate coordinates for the potential fourth point
                            x4 = x2 - x1 + x3
                            y4 = y2 - y1 + y3
                            # Check if the fourth point exists in the set
                            if (x4, y4) in point_set:
                                # Check for perpendicular vectors (dot product == 0)
                                vector_21 = (x2 - x1, y2 - y1)
                                vector_31 = (x3 - x1, y3 - y1)
                                if vector_21[0] * vector_31[0] + vector_21[1] * vector_31[1] == 0:
                                    # Calculate widths and heights of the rectangle
                                    width = sqrt(vector_21[0] ** 2 + vector_21[1] ** 2)
                                    height = sqrt(vector_31[0] ** 2 + vector_31[1] ** 2)
                                    # Update minimum area if a smaller one is found
                                    min_area = min(min_area, width * height)
        # Return 0 if no rectangle was found, else return the minimum area
        return 0 if min_area == inf else min_area
```

Java Solution

```java
class Solution {
    public double minAreaFreeRect(int[][] points) {
        int n = points.length;
        // Using a HashSet to store unique representations of points
        Set<Integer> pointSet = new HashSet<>();
        for (int[] point : points) {
            pointSet.add(encode(point[0], point[1]));
        }
        // Initialize the minimum area to the maximum possible value
        double minArea = Double.MAX_VALUE;
        // Iterate through all combinations of three points to find the fourth point
        for (int i = 0; i < n; ++i) {
            int x1 = points[i][0], y1 = points[i][1];
            for (int j = 0; j < n; ++j) {
                if (j != i) {
                    int x2 = points[j][0], y2 = points[j][1];
                    for (int k = j + 1; k < n; ++k) {
                        if (k != i) {
                            int x3 = points[k][0], y3 = points[k][1];
                            // Calculate potential fourth point's coordinates
                            int x4 = x2 - x1 + x3, y4 = y2 - y1 + y3;
                            // Check if the fourth point exists in the set
                            if (pointSet.contains(encode(x4, y4))) {
                                // Check if the three points form a right angle
                                if ((x2 - x1) * (x3 - x1) + (y2 - y1) * (y3 - y1) == 0) {
                                    // Calculate the square of the rectangle's width and height
                                    int widthSquared = (x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1);
                                    int heightSquared = (x3 - x1) * (x3 - x1) + (y3 - y1) * (y3 - y1);
                                    // Update the minimum area, if smaller than the current minimum area
                                    minArea = Math.min(minArea, Math.sqrt((long) widthSquared * heightSquared));
                                }
                            }
                        }
                    }
                }
            }
        }
        // Return 0 if no rectangle is found, otherwise return the minimum area found
        return minArea == Double.MAX_VALUE ? 0 : minArea;
    }

    // Helper function to encode a 2D point into a unique integer
    private int encode(int x, int y) {
        // We use 40001 as a base for encoding since the problem statement might give a max coordinate value of 40000
        return x * 40001 + y;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_set>
#include <cmath>
#include <algorithm>

class Solution {
public:
    double minAreaFreeRect(std::vector<std::vector<int>>& points) {
        // A lambda function to hash the coordinates uniquely by assigning
        // a unique number to each coordinate pair.
        auto hash_pair = [](int x, int y) {
            return x * 40001 + y;
        };

        int num_points = points.size(); // Number of points in the input.

        // Populating the hash set with all the given points.
        std::unordered_set<int> point_set;
        for (const auto& point : points) {
            point_set.insert(hash_pair(point[0], point[1]));
        }

        double min_area = std::numeric_limits<double>::max();

        // Triple nested loop to check every combination of three points.
        for (int i = 0; i < num_points; ++i) {
            int x1 = points[i][0], y1 = points[i][1];
            for (int j = 0; j < num_points; ++j) {
                if (j != i) {
                    int x2 = points[j][0], y2 = points[j][1];
                    for (int k = 0; k < num_points; ++k) {
                        if (k != i && k != j) {
                            int x3 = points[k][0], y3 = points[k][1];

                            // Calculating coordinates of the fourth point, assuming rectangularity.
                            int x4 = x2 - x1 + x3, y4 = y2 - y1 + y3;

                            // Check if the calculated point lies within the bounds and is present in the set.
                            if (point_set.count(hash_pair(x4, y4))) {
                                // Check for orthogonality between vectors (x2 - x1, y2 - y1) and (x3 - x1, y3 - y1).
                                if ((x2 - x1) * (x3 - x1) + (y2 - y1) * (y3 - y1) == 0) {
                                    // Compute sides squared of the rectangle.
                                    long long width_squared = (x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1);
                                    long long height_squared = (x3 - x1) * (x3 - x1) + (y3 - y1) * (y3 - y1);

                                    // Update minimum area if a smaller one is found.
                                    min_area = std::min(min_area, std::sqrt(width_squared * height_squared));
                                }
                            }
                        }
                    }
                }
            }
        }

        // Return 0 if no rectangle is found, otherwise return the min_area.
        return min_area == std::numeric_limits<double>::max() ? 0.0 : min_area;
    }
};
```

Typescript Solution

```typescript
function minAreaFreeRect(points: number[][]): number {
    const numOfPoints = points.length; // Total number of points
    const pointKey = (x: number, y: number): number => x * 40001 + y; // Unique key for a point
    const pointSet: Set<number> = new Set(); // Set to hold unique keys for points

    // Add all points to the set with their unique keys
    points.forEach(([x, y]) => pointSet.add(pointKey(x, y)));

    let minArea = Number.MAX_VALUE; // Initialize minArea with the maximum possible value

    // Generate all combinations of three points to find rectangles
    for (let i = 0; i < numOfPoints; ++i) {
        const [x1, y1] = points[i];
        for (let j = 0; j < numOfPoints; ++j) {
            if (j !== i) { // Ensure different points are chosen
                const [x2, y2] = points[j];
                for (let k = 0; k < numOfPoints; ++k) {
                    if (k !== i && k !== j) { // Ensure all three points are distinct
                        const [x3, y3] = points[k];
                        // Calculate the fourth point assuming the points form a rectangle
                        const x4 = x2 - x1 + x3;
                        const y4 = y2 - y1 + y3;
                        // Check if the calculated fourth point exists in our set
                        if (pointSet.has(pointKey(x4, y4))) {
                            // Check if the vectors are orthogonal
                            if ((x2 - x1) * (x3 - x1) + (y2 - y1) * (y3 - y1) === 0) {
                                // Calculate squares of the lengths of the rectangle sides
                                const widthSquared = (x2 - x1) ** 2 + (y2 - y1) ** 2;
                                const heightSquared = (x3 - x1) ** 2 + (y3 - y1) ** 2;
                                // Minimize minArea with the area of the current rectangle
                                minArea = Math.min(minArea, Math.sqrt(widthSquared * heightSquared));
                            }
                        }
                    }
                }
            }
        }
    }

    // Return 0 if no rectangle found, otherwise return the minimum area
    return minArea === Number.MAX_VALUE ? 0 : minArea;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be determined by analyzing the nested loops and the operations inside them:

- The first loop runs over all the points, which gives us `O(n)`.
- Nested within the first loop, the second loop runs over all other points except the one chosen by the first loop, contributing `O(n-1)` complexity.
- Inside the second loop, there's another loop that starts from the current index of the second loop and goes over the remaining points, which, in the worst case, contributes `O(n/2)` (this is an approximation since it depends on the current index of the second loop).

Putting these together, the complexity of the loops is approximately `O(n) * O(n-1) * O(n/2)`, which simplifies to `O(n^3)` for the worst case.

Additionally, within the innermost loop, there are constant-time operations such as checking if a point exists in the set, and basic arithmetic operations. The dot product and the calculation of rectangle area also take constant time, which does not affect the overall `O(n^3)` time complexity.

Space Complexity

The space complexity can be considered by looking at the data structures used:

- A set `s` that holds all the points. In the worst case, every point is unique, so the space complexity for the set is `O(n)`.
- Constant amount of extra space is used for variables such as `x1`, `y1`, `x2`, `y2`, `x3`, `y3`, `x4`, `y4`, `v21`, `v31`, `w`, `h`, and `ans`.

Therefore, the overall space complexity of the algorithm is `O(n)` for storing the set of points.