Closest Pair Analysis

Kenley Arai
CS 141
SID: 861128466

April 25, 2015

Definition 1. Closest Pair
Given a set of points on a Euclidean plane, find the pair of points whose Euclidean distance is the smallest value.

1 Brute force method

1.1 Closest Pair with Brute Force

The first way to approach this problem would be to implement a naive algorithm in Python.

```
def closest_pair_bf(points):
    number_of_points = len(points)
    if number_of_points <= 1:
        return -1
    smallest = float("inf")
    for point_indx in range(number_of_points):
        for compare_point_indx in range(point_indx+1, number_of_points):
            smallest = min(smallest, euclidean_dist(points[point_indx], points[compare_p
    return smallest
```

1.2 Analysis

The assignments for $PointCount$ and smallest both take constant time.
Next the nested for loop contains a comparison and a an assignment giving $O(n^2)$ performance.
Because this nested loop will always occur we also get $\Omega(n^2)$ performance.
Since $c_1\Omega(n^2) = c_2O(n^2)$ we can say the run time will be $\theta(n^2)$

2 Divide and conquer method

2.1 Closest Pair with Divide and Conquer

The next way to approach this is by implementing it with a divide and conquer method in Python:

```
def closest_pair_d_and_c(points): #Assuming the points are already sorted
    if len(points) < 4:
        return closest_pair_bf(points)

    mid_point = int(len(points)/2)
    mid_x = points[mid_point][0]

    smallest_left = closest_pair_d_and_c(points[mid_point:])
    smallest_right = closest_pair_d_and_c(points[:mid_point])

    smallest = min(smallest_left, smallest_right)

    between_2_d = sorted([item for item in points if \
                    mid_x - smallest < item[0] < mid_x + smallest], key=lambda x: x[

    y_smallest = smallest
    for i in range(len(between_2_d) - 1):
        if(between_2_d[i+1][1] - between_2_d[i][1] < y_smallest):
            y_smallest = euclidean_dist(between_2_d[i], between_2_d[i+1])

    return min(y_smallest, smallest)
```

2.2 Analysis

Assuming the points are already sorted we start with a run time of at least $\theta(n \log n)$. The next impact are the assignments for *smallest_left* and *smallest_right* each contributing $2T(\frac{n}{2})$ operations.Next we sort on y values which will give $n \log n$ operations.By comparing y values we can evaluate if any values that could be smaller than our current smallest value contributing n amount of operations.

This gives the recurrence relation:

$$T(n) = 2T(\frac{n}{2}) + n \tag{1}$$

Using Masters Theorem we get $\theta(n \log n)$.

3 Run time Analysis

3.1 Testing

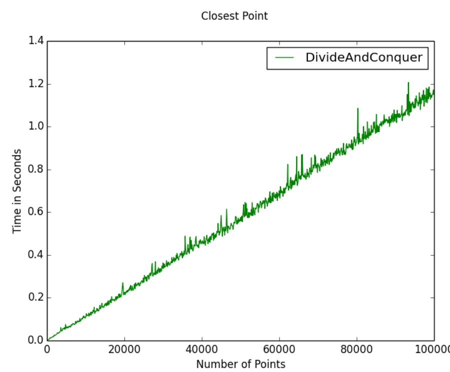
Running both algorithms we get the following table of nodes against, time.

3.2 Times using dataset provided

	10 nodes	100 nodes	10e5 nodes	10e6 nodes
Brute Force time	0.03s	0.05s	140.30s	107m
Divide and conquer time	0.03s	0.04s	0.15s	1.42s

3.3 Graphs using random points

(a) Brute Force



(b) Divide and Conquer

4 Conclusion

From the testing and from the graphical representation we can confirm that the run time for the brute-force algorithm is $\theta(n^2)$ and divide and conquer is $\theta(nlogn)$.