Homework 2
UCLA-CS180-S18

Quentin Truong

April 21, 2018

1 Question 1

Piazza says this question only requires proof of main lemma and algorithm

Proof of Main Lemma:
All points within points_sy_strip fall within min_dist from the center (due to definition). The strip is 2 * min_dist wide. If we vertically divide this strip into four, then horizontally divide this strip every 1/2 min_dist, we will have 4 squares per row. Each square will have length 1/2 min_dist. The maximum distance between two points within one square is 1/2 * min_dist + 1/2 * min_dist due to l1_distance. Knowing that this maximum distance is equal to min_dist along with the fact that both points within a square are only ever on one side or the other (so their distance cannot be less than min_dist, due to definition), we can claim that there is only one point per square. Further, we know that closer points could only ever be separated by two full rows, because each row is 1/2 * min_dist and we want points closer than this. If a point lay on the top of a row 1, and the bottom of row 4, they would be separated by min_dist. There are 4 boxes per row and 4 total rows, therefore there are 16 boxes. The pair of points cannot be in the same box, otherwise they would be on the same side, and we would have a different min_dist. Therefore, there are 15 possible boxes. Therefore, we only need to check the next 15 points (sorted along the y coordinate). The time complexity of this is O(nlogn) due to master theorem (a = 2, b = 2, f(n) = n).

Algorithm:

```
function l1_distance (first, second):
    return abs(first.x − second.x) + abs(first.y − second.y)

function find_closest_pair_brute (points):
    min_dist = inf
    for (int i = 0; i < points.size(); i++)
        for (int j = i + 1; j < points.size(); j++)
            min_dist = min(min_dist, l1_distance(points[i], points[j]))
            min_pair = (points[i], points[j])
    return min_pair, min_dist

function find_closest_pair_strip (points_sy):
    min_dist = inf
    for (int i = 0; i < points_sy.size(); i++)
        for (int j = 1; j <= 15 AND i + j < points_sy.size(); j++)
            min_dist = min(min_dist, l1_distance(points_sy[i], points_sy[i + j]))
            min_pair = (points_sy[i], points_sy[i + j])
    return min_pair, min_dist

function find_closest_pair_l1_helper (points_sx, points_sy):
    if (points_sx.size() <= 3)
        return find_closest_pair_brute(points_sx)
    // O(n) by just copying stuff over in a good way
    left_points_sx, left_points_sy = left_half(points_sx, points_sy)
    right_points_sx, right_points_sy = right_half(points_sx, points_sy)

    left_pair, left_dist = find_closest_pair_l1_helper(left_points_sx, left_points_sy)
    min_pair, min_dist = find_closest_pair_l1_helper(right_points_sx, right_points_sy)

    if left_dist < min_dist
        min_dist = left_dist
        min_pair = left_pair

    // O(n) to find points within min_dist of center
    points_sy_strip = middle_strip(points_sx, min_dist)
    strip_pair, strip_dist = find_closest_pair_strip(points_sy_strip, min_dist)

    if strip_pair < min_dist
        min_dist = strip_dist
        min_pair = strip_pair

    return min_pair, min_dist

function find_closest_pair_l1 (points):
    points_sx = sort_x(set_to_list(points))
    points_sy = sort_y(set_to_list(points))

    return find_closest_pair_l1_helper(points_sx, points_sy)
```

2 Question 2

We split the problem into two subproblems (a = 2), each of size n / 2 (b = 2), and merging their solutions requires at most a single traversal (f(n) = O(n)). From master theorem, the time complexity is O(nlogn).

```
function find_majority_element(elements):
    if (elements.size() <= 2)
        if elements[0] == elements[elements.size() - 1]
            return elements.size(), elements[0]
        return 0, null

    left_elements, right_elements = split_in_half(elements)

    count_left, majority_element_left = find_majority_element(left_elements)
    count_right, majority_element_right = find_majority_element(right_elements)

    if majority_element_left == majority_element_right
        return count_left + count_right, majority_element_left
    else if majority_element_left != null
        for element in right_elements
            if element == majority_element_left
                count_left++
        if count_left > elements.size() / 2
            return count_left, majority_element_left
        else
            return 0, null
    else if majority_element_right != null
        for element in left_elements
            if element == majority_element_right
                count_right++
        if count_right > elements.size() / 2
            return count_right, majority_element_right
        else
            return 0, null
```

3 Question 3

4 Question 4

We traverse all vertices and check all their edges. Therefore, the time complexity is O(V * E).

```
function has_triangle(graph, prev_layer, discovered)
    if (prev_layer.size() == 0)
        return false

    children = array of bool size n, init to false

    for curr in prev_layer
        for node in graph[curr]
            if discovered[node] == false
                next_layer.append(node)
                discovered[node] = true
                children[node] = true

    for curr in next_layer
        for node in graph[curr]
            if children[node] == true
                return true

    return has_triangle(graph, next_layer, discovered)

function has_triangle(graph)
    if (graph.size() > 0)
        discovered = array of bool of size n, init to false
        discovered[0] = true
        return has_triangle(graph, [0], discovered)
    return false
```