Foundation of Algorithms prog Assignment

Austin Jin

October 10, 2022

Conditions

(a) Consider a set, P, of n points, (x1,y1),...,(xn,yn), in a two dimensional plane.

(b) A metric for the distance between two points (xi,yi) and (xj,yj) in this plane is the Euclidean distance $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

2. Closest Pairs [90 points]

(a) [40 points] Construct an algorithm for finding the $m \leq \binom{n}{2}$ closest pairs of points in P. Your algorithm inputs are P and m. Return the distances between the m closest pairs of points, including their x and y coordinates.

i. [25 points] Define your algorithm using pseudocode.

In this algorithm, my idea would be using divide-and-conquer method. But I didn't store the local min-distance points-pair for each round. I actually designed a m-sized linked list, every time we insert the pairnode into the list to find if the distance is smaller than any node in m-list, and then insert at right position and delete the tail node if the length of the list is over m. This method would be space-saving and time-saving since it only need m extra space and sort m-list.

For divide-and-conquer part, I try to separate points based on x value recursively and find m minimum distance pair-points. Then we use the largest distance in the m minimum distances as threshold to limit input size and decrease unnecessary comparisons.

Algorithm 1. Find closest-pairs from P

Input: a set of points and number of closest pairs
Output: a closest pairs
Data Structure Node: point node, storing x and y.
Data Structure Pairnode: pair of points node, storing x and y of two points and its distance
Data Structure m-autosort linked list: sorted linked list of pairnode, the size of the list is m

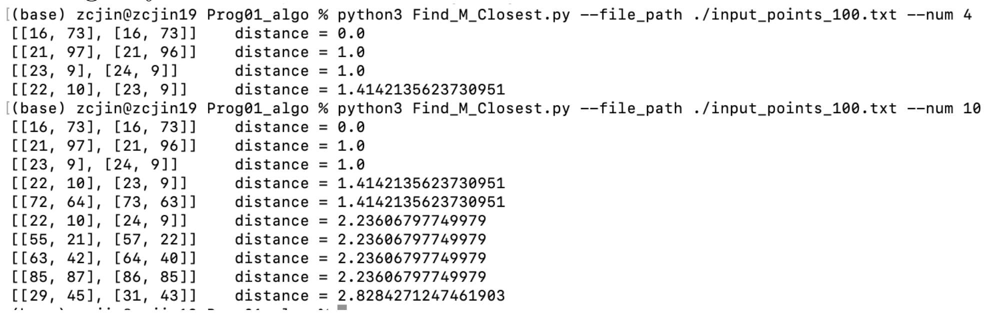
```
1: function INSERT(pairnode)
2:     if The linked list is empty then
3:         self.head = pairnode
4:     else
5:         while currentnode.next != Null do
6:             Insert the pairnode when its distance is smaller.
7:     if self.length > size limit then
8:         Delete self.tail
```

```
function MERGESORT(arr)
    $n = len(arr)$
    $mid = len(arr)/2$
    $l = arr[: mid]$
    $r = arr[mid + 1 :]$
    Mergesort($l$)
    Mergesort($r$)
    Merge($l + r$)
function CLOSEST-PAIR((PX, m))
    $n = PX.length$
    $m = m_linkedlist$
    if $n == 2$ then
        Store xy of nodes into pairnode.
        $pairnode.dist =$ Distance($PX[1], PX[2]$)
        $m$.insert($pairnode$) return $m$
    if $n == 3$ then
        store xy of nodes in to pairnode
        $m.insert$(pairnode($PX[1], PX[2]$), pairnode($PX[1], PX[3]$), pairnode($PX[2], PX[3]$))
return $m$
    $mid = n/2$
    $left = Closest - pair$(PX[1 to mid], m)
    $right = Closest - pair$(PX[mid+1 to n], m)
    $min - Node = m.tail.distance$
    $M =$ Points in X whose x values is among the range [mid.x-d to mid.x+d]
    for $i \leftarrow 1$ to $M.length$ do
        for $j \leftarrow 1$ to $M.length - i$ do
            $m.insert(pairnode(M[i], M[i + j])$
    return m
function M-CLOSET-PAIR(P, m)
    PX = Sort P based on x value
    m = autosort linked list with size m
    Closest-Pair(PX, m)
    for item in m linked list do
        print (x,y) pair-point's coordinates and its distance
```

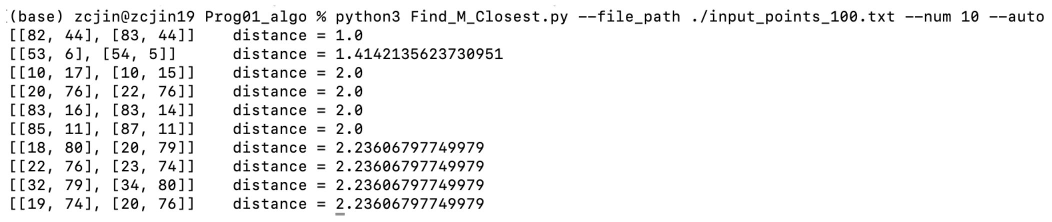
ii. [15 points] Determine the worst-case running time (page 25) of your algorithm (call this the algorithm's worst-case running time).

To calculate the running time, we divided it into three part: Merge sort, closest-pair and link-list insertion.
Merge sort:
the worst-case running time of merge sort would be $\Theta(nlogn)$.
link-list insertion:
For the link-list insertion, we used the linear comparing, so the average cost of time would be m/2, and the worst case would be m. since we defined the size of the linked-list, and the size would be a constant. So the time cost of insertion would be constant when size is small enough.
Closest-pair:
For Closest-pair part, I think the most confused part would be choosing points-pair in the middle range. Even though there are two for loop, but the M.length did not associated with n but distribution of points, the M.length would more like a small variable. So I would recognize it as constant. so $T(n) = 2 * T(n/2) + O(n)$ The worst time complexity of the closest-pair would be $O(mnlgn)$
(b) [20 points] Implement your algorithm. Your code must have a reasonable, consistent, style and docu- mentation. It must have appropriate data structures, modularity, and error checking. (Your submission must include a readme text file which gives clear directions on how to run your program. Please refer to Programming Assignment Guidelines and Pseudocode Restrictions for more details.)

(c) [10 points] Perform and submit trace runs demonstrating the proper functioning of your code.

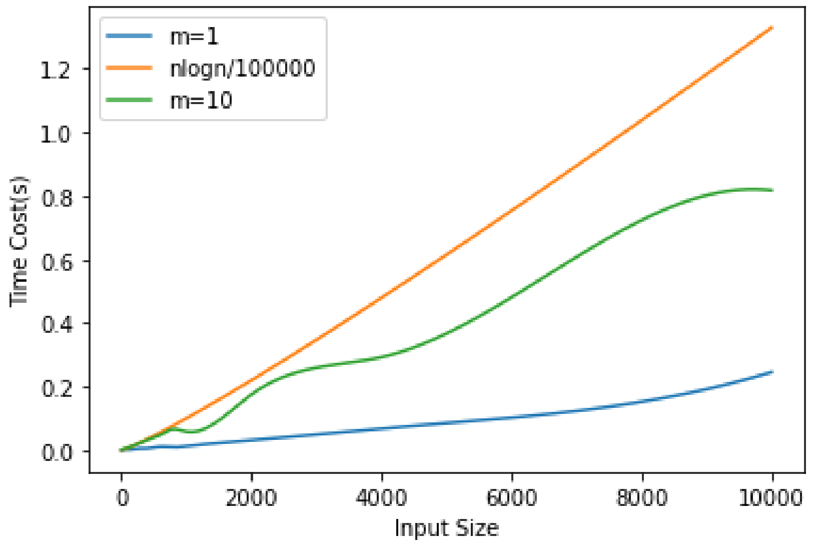


```
(base) zcjin@zcjin19 Prog01_algo % python3 Find_M_Closest.py --file_path ./input_points_100.txt --num 4
[[16, 73], [16, 73]]    distance = 0.0
[[21, 97], [21, 96]]    distance = 1.0
[[23, 9], [24, 9]]      distance = 1.0
[[22, 10], [23, 9]]     distance = 1.4142135623730951
(base) zcjin@zcjin19 Prog01_algo % python3 Find_M_Closest.py --file_path ./input_points_100.txt --num 10
[[16, 73], [16, 73]]    distance = 0.0
[[21, 97], [21, 96]]    distance = 1.0
[[23, 9], [24, 9]]      distance = 1.0
[[22, 10], [23, 9]]     distance = 1.4142135623730951
[[72, 64], [73, 63]]    distance = 1.4142135623730951
[[22, 10], [24, 9]]     distance = 2.23606797749979
[[55, 21], [57, 22]]    distance = 2.23606797749979
[[63, 42], [64, 40]]    distance = 2.23606797749979
[[85, 87], [86, 85]]    distance = 2.23606797749979
[[29, 45], [31, 43]]    distance = 2.8284271247461903
```



```
(base) zcjin@zcjin19 Prog01_algo % python3 Find_M_Closest.py --file_path ./input_points_100.txt --num 10 --auto
[[82, 44], [83, 44]]    distance = 1.0
[[53, 6], [54, 5]]      distance = 1.4142135623730951
[[10, 17], [10, 15]]    distance = 2.0
[[20, 76], [22, 76]]    distance = 2.0
[[83, 16], [83, 14]]    distance = 2.0
[[85, 11], [87, 11]]    distance = 2.0
[[18, 80], [20, 79]]    distance = 2.23606797749979
[[22, 76], [23, 74]]    distance = 2.23606797749979
[[32, 79], [34, 80]]    distance = 2.23606797749979
[[19, 74], [20, 76]]    distance = 2.23606797749979
```

(d) [10 points] Perform tests to measure the asymptotic behavior of your program (call this the code's worst- case running time). This should include a table or figure which clearly shows the asymptotic behavior of your program.



The worst case running time testing code is in the `script_prog01.ipynb` jupyter notebook file.

(e) [10 points] Analysis comparing your algorithm's worst-case running time to your code's worst-case run- ning time.

According to the Time cost analysis in the question d, we found that the time complexity of own designed algorithm is lower than $O(\frac{1}{100000}nlgn$, however if we keep increasing the m which means collecting m paired points with smallest distance, we would found the time complexity of our algorithm is increasing. if m is large enough which is close to n, the complexity would be $O(n^2lgn)$ instead. Like the line m=10 in the plot shows, at the beginning part, the cost time is equal or larger than nlgn. Secondly, we found the time complexity of our algorithm is also dependent on distribution of the input points. It would run faster when the points are randomly distributed or m is

smaller enough. That's because what I mentioned above counting the points in the middle region.
So, I think there are two factors that can influence the time complexity of our algorithm:

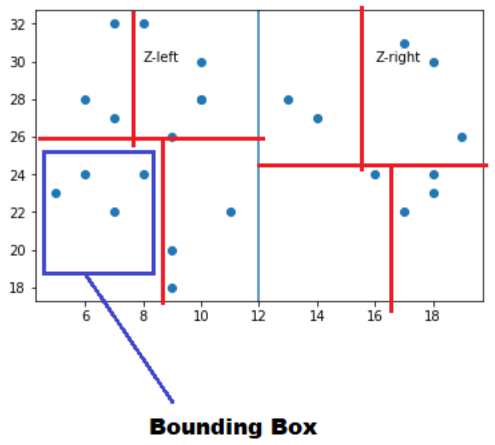
- Input size m
- Distribution of the points

3. Retrospection [10 points]
(a) [10 points] Now that you have designed, implemented, and tested your algorithm, what aspects of your algorithm and/or code could change and reduce the worst-case running time of your algorithm? Be specific in your response to this question.

For this question, I think the improvement is required when m is large or points are clustering in a certain region.

1. m is large enough When m is large enough, what we can do to improve the time efficiency is that change the comparing algorithm used by m-linked list(which is linear searching initially). Using binary search instead of linear search!

2. Distribution of the points If most of points are cluster at certain region, there are two methods we can use. The first one would be stop checking every point-pairs in the middle region or narrow down the middle region. The second method is based on the k-D tree. We can divide the points plot into multiple dimensions, like the picture below, and we would calculate the minimum distance in the region with clustered points.



Reference

Kanoki, Finding nearest neighbor using KD tree, retrieved from: Link
Algorithm Tutor, Finding the Closest Pair of Points, retrieved from: Link