Algorithms Project 1

Greyson Brothers

March 12, 2023

Statement of Integrity: I, Greyson Brothers, attempted to answer each question honestly and to the best of my abilities. I cited any and all help that I received in completing this project.

1 Problem Statement

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.

1.1 Define your algorithm using pseudocode.

This algorithm will make use of three data structures - a node, a min heap, and a max heap. The node is a simple object holding a pair of (x, y) points and the distance between those points. It also holds the index of its current position in the min heap. Comparing two node objects involves comparing their distances.

The min heap is implemented as standard binary tree implemented via an array, with the nodes of the tree being node objects. The tree is maintained such that each node in the tree is less than its children, which results in the minimum always being at the root of the tree. To insert a node into the tree, it is added as a leaf in the bottom layer of the tree. If the bottom layer is full, it is added as a leaf in a new layer. This way the tree is always built in a complete fashion such that it is always $log_2 n$ layers deep. After the point is inserted, a function $heapify\_up$ is called on the new node to preserve the min structure of the heap. Heapify recursively compares the new node to its parent and swaps the nodes if the parent is greater than it. This repeats until the new node finds a parent that is less than it or until it reaches the node, which involves at most $log_2 n$ comparisons.

To extract the min from the tree, the root node is returned. Then the last node in the tree is swapped with the old root and the old root is dropped from the tree. In order to preserve the heap structure, a function called $heapify\_down$ is called on the new root. In this function, the new node is recursively compared

with the smallest of its two children. If that child is less than the new node, they are swapped. This continues until either both children are larger than the new node or the new node reaches the bottom of the heap, taking at most $2log_2 n$ comparisons. When nodes are inserted or swapped, the index field in the node object is updated with its new position in the min heap array. The two heapify functions will be important for estimating the run time complexity of our algorithm defined below.

The max heap is implemented identically to the min heap, but with the opposite comparison operator. This maintains the maximum value at the root and keeps all parents greater than or equal to their children. This implementation of the max heap does not update or alter the index fields of the nodes it contains.

Let p be our set of (x, y) points, $|p| = n$, $M = \binom{n}{2}$ and $2 \leq m \leq M$ be the parameters described in the problem statement. These definitions will be used throughout the rest of this analysis.

In order to determine the m closest pair of points in p, we need to check the distance between every unique pair of points and then sort the unique pairs by distances. Then we can return the m points at the beginning of our sorted list. Note that for any n points, there are M unique pairs, hence why M is an upper bound for m.

We can loop through all unique pairs by careful indexing with nested loops, as outlined in the psuedocode below. On each step of the inner loop, we compute this distance between the points i and j, which is a constant time operation. We then construct a node object from the two points and the computed distance (constant time) and insert it into both the min heap and max heap. Both heaps cannot hold the same node object simultaneously, so they instead hold pointers to the underlying node object. The node is assigned an index upon insertion to the min heap. Since we expect to insert M nodes into our tree, the worst case insert will take $log_2 M$ node comparisons.

This can actually be improved upon. We only care about returning the closest $m \leq M$ pairs of points, so we only ever need to store m nodes in our min heap. In order to do this, once the heap has m nodes inserted, any new nodes being inserted must replace the largest node in the heap. This is where the max heap comes in. By looking at the root of the max heap, we can see what the largest node in the min heap is as well as its index. If a new node is smaller than the current largest node in the min heap, we can insert the new node in its place and heapify. We also extract the max from the max heap and insert the new node in the standard fashion ensure that both heaps point to the same set of nodes. This prevents both heaps from getting bigger than size m and limits their depths to $log_2 m \leq log_2 M$. If the new node is bigger than the root of the max heap, we simply ignore it and skip to the next iteration.

In simple terms, rather than sorting all pairs of points by their distances, we are only sorting the m closest pairs. The result is a large improvement in efficiency when $m << M$. The idea for using heaps in this manner came from the Module 7 lecture video titled "Heap-Based Order Statistics".

Once we have looked at all pairs of points, we can start building our list of sorted closest pairs for output. To do this we simply loop m times, with each

iteration extracting the min node from the heap and appending it to our output list (constant time). We return a list of node objects, but this can easily be converted to a list of distances or a list of point pairs in linear time depending on the use case.

Algorithm 1 m Closest Pairs

```
1: p ←                                                          ▷ len(n)
2: m ←
3: n ← len(p)
4: min_heap ← MinHeap
5: max_heap ← MaxHeap
6: for i = 0; i < n; i++ do
7:     for j = i+1; j < n; i++ do                                ▷ M
8:         d = dist(p[i], p[j])                                  ▷ → 1
9:         new_node = Node(d, p[i], p[j])                        ▷ → 1
10:        insert_idx ← null                                     ▷ → 1
11:        if min_heap.size >= m then                            ▷ 0 → 1
12:            min_heap.size >= m                                ▷ 0 → 1
13:            if max_heap.get_max() < new_node then             ▷ 0 → 1
14:                > throw away new_node and continue            ▷ 0 → 1
15:            end if
16:            max_node = max_heap.extract_max()                 ▷ 0 → log2m
17:            insert_idx = max_node.idx                         ▷ 0 → 1
18:        end if
19:        max_heap.insert(new_node)                             ▷ → log2m
20:        min_heap.insert(new_node, insert_idx)                 ▷ → log2m
21:    end for
22: end for
23: output ← emptylist
24: for i = 0; i >= m; i++ do                                    ▷ m
25:    min_node = min_heap.get_min()                             ▷ → log2m
26:    output.append(min_node)                                   ▷ → 1
27: end for
```

Note the conditional logic in the if statements can execute any number of times from 0 to n depending on the input array. For analysis we take the worst case n for all statements. Also, in practice, the worst case in one loop (i.e. an array with all R's) is the best case for the other loop.

1.2 Determine the worst-case run time of your algorithm.

In the absolute worst case, the input data is distributed such that all if conditions are met on each iteration and each heapify call performs $log_2 m$ comparisons. In that case we get the following:

$$
\begin{aligned}
T(n) &= M(1+1+1+1+1+1+1+log_2m+1+log_2m+log_2m)+m(log_2m+1) \\
&= 8M+3Mlog_2m+m+mlog_2m \\
&= O(Mlog_2m+mlog_2m) \\
&= O(Mlog_2m)
\end{aligned}
\tag{1}
$$

1.3 Trace Runs of the Code

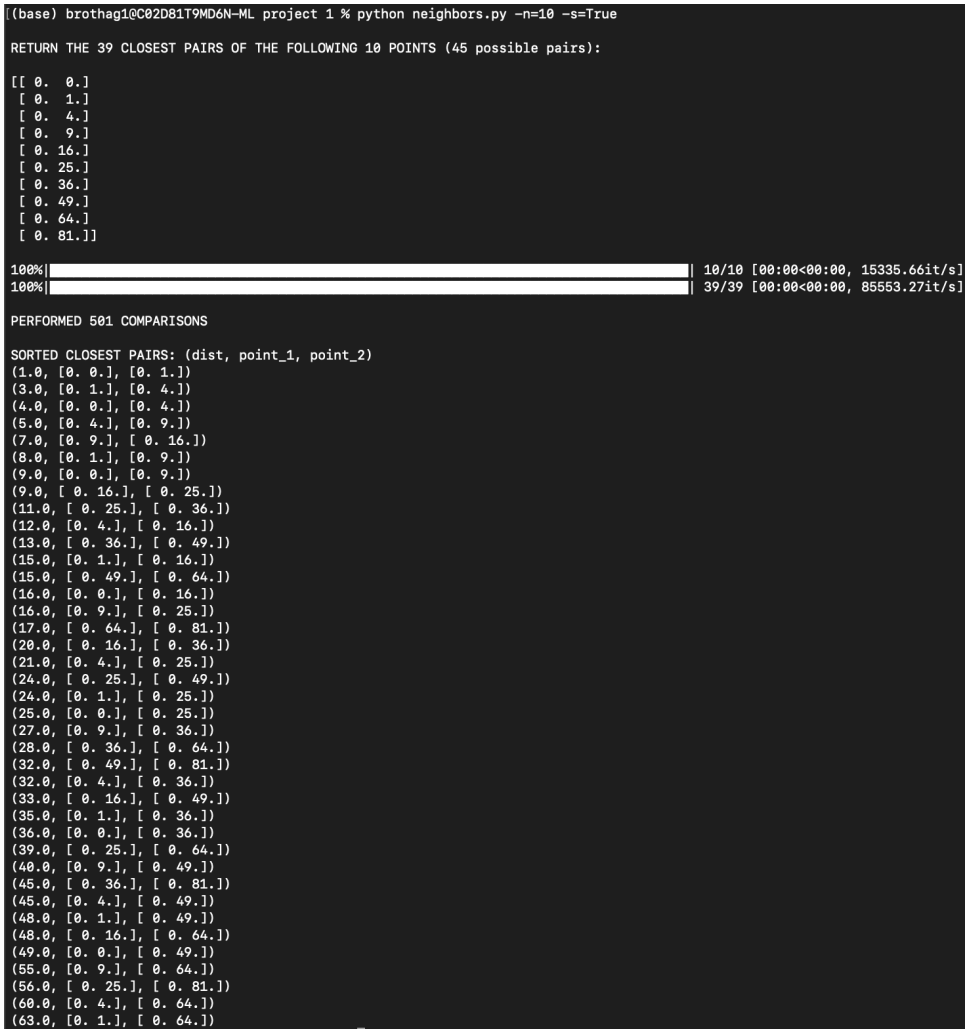

Figure 1: Example program simple execution and output.

The program has an option to run in "simple" mode as seen above. In this mode the points are generated on the x axis in the following manner: $p = [(0,1),(0,4),...,(0,n^2)]$. This solution to this set of points is very easy to inspect visually and allows the user to quickly sanity check the validity of the program.

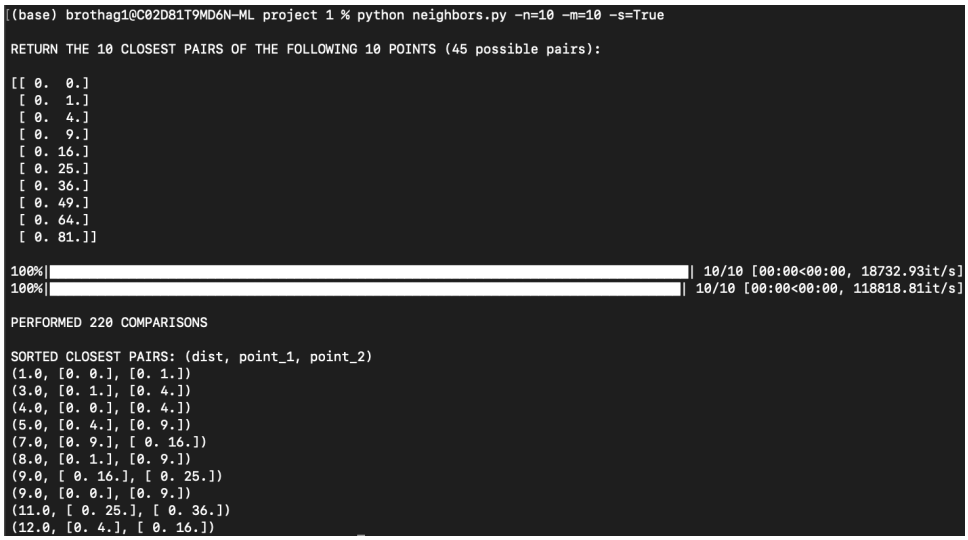

Figure 2: Program simple execution with m=10.

The user can also specify m. By not specifying -m in the command line, the program takes $m = M$ by default. The code performs checks to assert that m is a valid value. We see that the output here is consistent with the output in Figure 1.

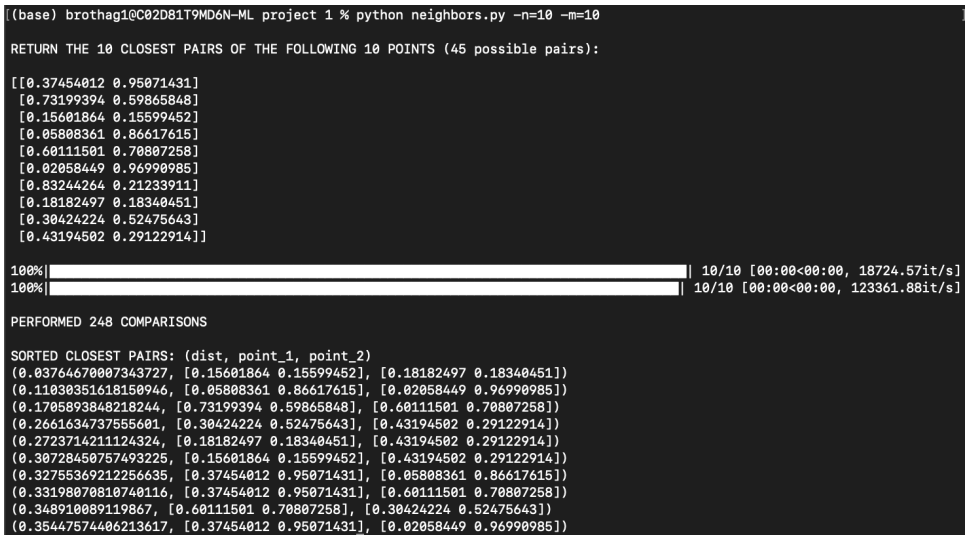

Figure 3: Program execution on random uniform points with m=10.

By excluding the -s argument in the command line, the program will use numpy to generate random uniform points between (-1, -1) and (1, 1). Beyond examining the output above for validity, there are a set of assert statements and print statements that can be enabled by setting "DEBUG=True" in neighbors.py. These ensure that the min heaps return the actual min, max heaps return the actual max, actual numbers of comparisons don't exceed their theoretical bounds, etc. There is also a plotting file used to generate the graph below which be done by simply running plotting.py.

1.4 Perform tests to measure the asymptotic behavior of your program (call this the code's worst- case running time)

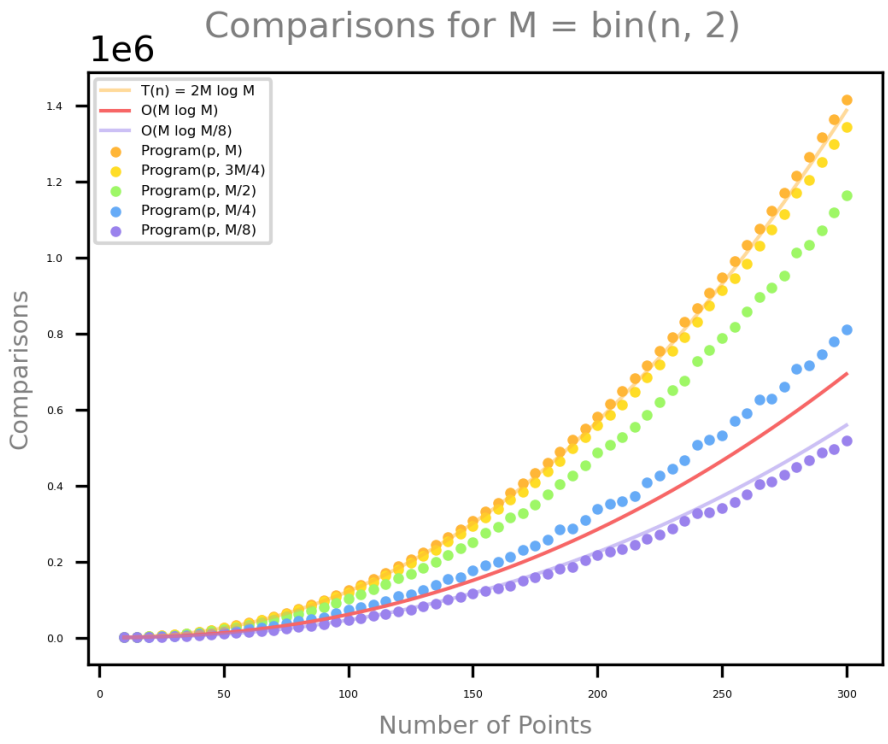

Figure 4: Analysis of program execution time vs theoretical run time complexity.

In the figure above we have the theoretical number of comparisons by the algorithm and samples of the number of comparisons by the code implementation with respect to various input sizes and values of m. For these results, p was sampled using random uniform points as described in the code trace section, hence the noise in the scatter plots. For a given n, the same set of points were used to evaluate each different value of m.

The number of comparisons is incremented each time the distance between two points is computed and each time the distances of two pairs of points are compared in a heap. Other constant time operations are not counted as they do not scale with the respect to the input.

The predicted worst case run time $T(n) = 2M\ln M$ is plotted in solid orange, which almost perfectly aligns with orange samples of the program worst case run time for returning M points. The big O worst case run time is $O(M\ln M)$, which is plotted in red.

The program was then sampled for different values of $m < M$, which demonstrates the improvement of this algorithm over the naïve approach. In fact, we are able to show that the program marginally outperforms $M \ln m$ for $m = \frac{M}{8}$. After a lot of debugging and improvement on the code implementation side, the theoretical worst case times and the actual worst case times converge.

2 Retrospection

I spent a lot of time thinking about the algorithm, tweaking the code, and making any changes I could think of to improve efficiency. One thing that made a big difference at the end was the preventing the insertion of new nodes that were greater than all nodes currently in the max heap when the heap already held m nodes. Before, I was inserting the new node into the max heap and then extracting the max to see which index the new node should replace in the min heap. Fixing this improved the performance considerably, and there may be other parts of the code that I overlooked where similar changes are possible. Barring that, this algorithm using heaps is as efficient as I can currently come up with.

There are certain limits to how much more efficient this algorithm can get. In order to guarantee a correct solution, the algorithm must compute the distance between each pair of points, requiring no less than M computations. In order to return the m closest points, the algorithm must sort the pairs by distance. With randomly ordered points, the best you can do here is $M \log m$, where you are only sorting the closest m pairs to be returned. This is what the above algorithm does.

Perhaps there might be a method where one is able to compute the relative distances between the M pairs of points in less than M computations, through some form of dimensionality reduction or randomization. Reducing the factor of M would cause the greatest improvement in the performance of this algorithm. I spent a bit of time thinking about those approaches but was not able to come up with anything that would produce a correct solution in all cases.

I would love to hear about any ideas I missed or methods that might improve upon my work here. Thank you!