University of Washington March 7, 2021
Department of Computer Science and Engineering
CSEP 521, Winter 2021

Homework 8 Solution

Problem 3 (10 points):

This problem is to work out the details of a bucketing approach to the nearest neighbors problem in 1-D. You *do not* need to implement your algorithm.

Let S be a set of n points from $[0, 1)$ with minimum separation $\delta \geq 2^{-k}$. Think of the line segment as being divided into overlapping buckets $B = \{B_j^i \mid 0 \leq j \leq k \text{ and } 0 \leq i < 2^j\}$ where B_j^i corresponds to the interval $[i2^{-j}, (i+1)2^{-j})$. Describe a nearest neighbors algorithm that relies on looking for the query point in appropriate buckets, and uses hashing to avoid storing unnecessary buckets. You should describe the run time of your algorithm in terms of the ~~expected~~ number of buckets accesses per query point. (You can assume that hashing is an $O(1)$ time operation.)

Solution 3:

The basic idea is to build the binary version of a quad tree (a BQ-tree) for the points in S, and then look up the query points in BQ-tree. We use hashing to store the nodes of the BQ-tree so we don't need to explicitly navigate with pointers. The importance of hashing is that we can look up nodes that are not present in the BQ-tree, and be notified that they are missing.

The interval B_j^i corresponds to the interval $[i2^{-j}, (i+1)2^{-j})$. The hash key for B_j^i is a hash of (i, j). We can map (i, j) to $h = 2^j + i - 1$ and then use a standard integer hash of h. We can compute the interval at the j-th level for the point $x \in [0, 1)$ by the formula $i = \lfloor 2^j x \rfloor$.

When we create the BQ-tree, we record the largest and smallest element from S placed in each interval B_j^i.

Let y be a query point, so we want to find the closest point in S to y. The first thing we do is we locate the leaf l that would contain y if y were inserted into the tree. The parent p contains at least one point of S. These points are all located in the sibling s of l at the same level. We can find this node l either bottom up, starting at a node on level k and moving up the tree, or top down, starting at level zero and going down the tree. We can now find y's nearest neighbor in s by either looking at the largest value in s (if s is a left child of p or looking at the smallest point in s if s the right child of p.

To find the nearest neighbor of y, we need to find the closest points above and below y, so from looking y up in the tree, we have found one of the candidate points. Assume that l is the right child of p, so we have found the closest point, z below y in S. Now we need to find the point above y. We could traverse the tree, but there is a simpler way. If the tree node containing y is the interval B_j^i, then we need to look into the interval $B_{j-1}^{i'}$ that is the parent of B_j^{i+1} for a potential closest point above y. If $B_{j-1}^{i'}$ is in the tree, we take the smallest point t in $B_{j-1}^{i'}$, and compare $y - z$ and $t - y$ to determine y's nearest neighbor. If $B_{j-1}^{i'}$ is not in the tree, then y's nearest neighbor is z.

Note that we need to look in the “uncle” node instead of the “cousin” node. The case where y is the left child is similar.

This takes k node visits in the worst case to locate where y fits in the tree. We then need to do two more node lookups to find the closest neighbor.

Can we do better than k? Remarkably, we can do much better, and do the point location in $\log k$ operations. We need to find the parent p of y in the BQ-Tree. The initial proposal was a linear search - either top down or bottom up for k steps. However, we can use binary search to locate the parent. We just perform a binary search on the levels (there are k). We can determine which node would be y's parent on level j - if this node exists we need to move down the tree, if this node does not exist, we move up the tree. Thus, we can locate the parent in $\log k$ steps.