CS 266 Homework 5

Zachary DeStefano, PhD Student, 15247592

Due Date: May 15

Problem 10.1

In Section 10.1 we solved the problem of finding all horizontal line segments in a set that intersect a vertical segment. For this we used an interval tree with priority search trees as associated structures. There is also an alternative approach. We can use a 1-dimensional range tree on the y-coordinate of the segments to determine those segments whose y-coordinate lies in the y-range of the query segment. The resulting segments cannot lie above or below the query segment, but they may lie completely to the left or to the right of it. We get those segments in O(logn) canonical subsets. For each of these subsets we use as an associated structure an interval tree on the x-coordinates to find the segments that actually intersect the query segment. q a. Give the algorithm in pseudocode. b. Prove that the data structure correctly solves the queries. c. What are the bounds for preprocessing time, storage, and query time of this structure? Prove your answers.

Part A

Here is the algorithm:
Traverse binary search tree for y-coordinate until you hit a split vertex
Traverse right subtree:
- Every time you make a right turn, call the left node n and do Recurse(n)
Traverse left subtree:
- Every time you make a left turn, call the right node n and do Recurse(n)

Recurse(n):
- Report intervals in the binary search tree for n that lie in our point.

Part B

We have a binary search tree that will correctly get us the intervals with the correct y-coordinate. We then have an interval tree on our x-coordinates that will get us the matching x's. Thus we will get segments that correctly fit our requirements.

Part C

Preprocessing:
The binary search tree can be constructed in $O(n \cdot logn)$
Each segment tree can be constructed in $O(n \cdot logn)$ time.
Total thus is $O(n^2 log^2(n))$ time.

Query:
Traversing the first one will take O(log n) operations.
Traversing the query time will be done up to log(n) times.
For each time, we have a segment tree, so the count will take $logn$ times to obtain.
We thus have a query time of $O(log^2(n))$

Storage:
A binary search tree on the y-coordinate can be stored in $O(n)$ space.
The segment trees are each $O(n \cdot log(n))$ space.
Thus the total space is $O(n^2 \cdot log(n))$

Problem 10.6c

Construct a binary search tree of the endpoints.
For each leaf in the Binary Search Tree, store the number of overlapping intervals.

Construction algorithm:

```
-Make a Binary Search Tree of Endpoints
-Sort the list of endpoints
-Initialize currentCount to zero
-For each endpoint in ascending order:
-     If left endpoint:
-          Add one to currentCount
-     If right endpoint:
-          Subtract one from currentCount
-     Put currentCount into endpoint node in Binary Search Tree
```

Query algorithm to get number of intervals at point q:
1. Traverse binary search tree until you find the greatest endpoint that is less than q
2. Report its count.

Analysis:
Construction will be $O(n \cdot logn)$
Query will be $O(logn)$ since it is still a binary search tree
Storage will be $O(n)$ since it is just a binary search tree with an added data field in the nodes

Problem 14.6

In this chapter we called a quadtree balanced if two adjacent squares of the quadtree subdivision differ by no more than a factor two in size. To save a constant factor in the number of extra nodes needed to balance a quadtree, we could weaken the balance condition by allowing adjacent squares to differ by a factor of four in size. Can you still complete such a weakly balanced quadtree subdivision to a conforming mesh such that all angles are between 45 and 90 by using only O(1) triangles per square?

There is no way to do a triangulation that will fit that criteria. In the worst case, we will have a square and it will be next to a square that is one fourth its size and the vertices need to be triangulated. We will thus have a point P that is one fourth the way along one of the sides of the triangle.

If P is connected to the center point, then an obtuse angle ends up being formed with the side. Here is an illustration of how it happens:

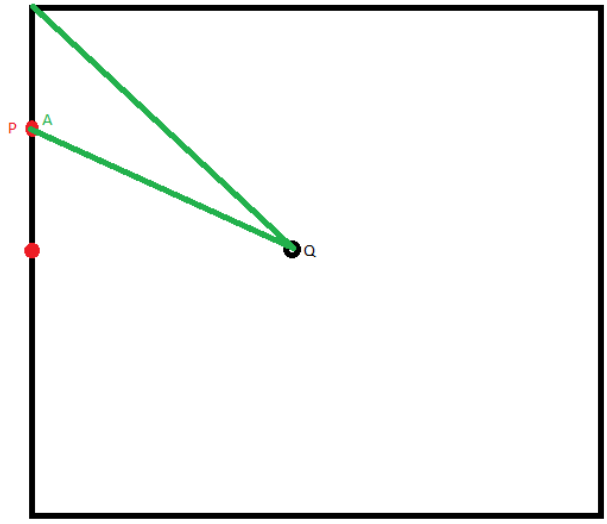

Figure 1: Proposed Triangulation. Angle A will be obtuse, violating requirements

If we have an interior point that is not the center, then its triangulation has an obtuse angle, violating the constraints. This is because it has to be connected to the 4 corner points, meaning there are 4 triangles whose angle sum needs to add to 360. Since they are not all 90 degrees if it is not the center point, at least one of them must be greater than that.

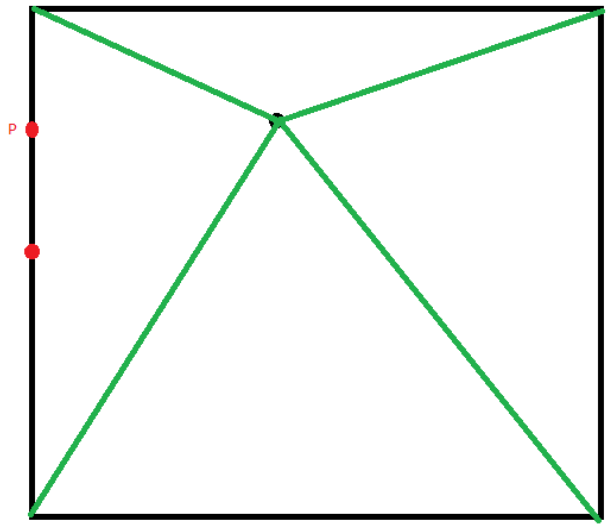

Figure 2: Proposed Triangulation. At least one interior angle will be obtuse

Finally, we cannot connect P to any of the vertices on its own side since it is already connected there. Therefore, the last remaining vertices it might be possible to connect it to are the vertices on the opposite side. Lets call them R and S where R is the one that lies closer to P. If we connect P to S or P to R then in both cases we end up with right triangles that violate our constraints.

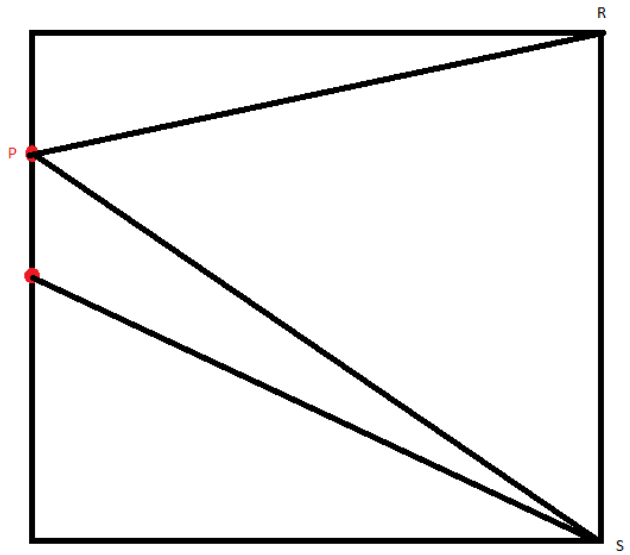

Figure 3: Proposed Triangulation. The triangles formed will violate the constraints

Since we have run out of places to connect P to, it is not possible to do the triangulation.

Problem 14.12

For this algorithm, a vertex v in the quad tree will have this data:
- $subtree(i)$ for i=1,2,3,4 will be pointers to subvertices if this square is split up
- $value$ will be intensity value if this vertex is a leaf.
J_1, J_2 will denote the quad trees for the images and will point to the root vertex

The idea behind this algorithm is that we can traverse the two quad trees concurrently to compute the AND and OR quad trees and potentially take shortcuts. If we are computing the AND tree and we encounter a 0, then we already know that the entire square in the result image will be 0. Similarly, if we are computing the OR tree and we encounter a 1, then we already know that the entire square in the result image will be 1. If we are computing the AND tree and encounter a 1, then the result will be entirely dependent on the other tree. Similarly, if we are computing the OR tree and encounter a zero, then the result is entirely dependent on the other tree.

Here are the algorithms in detail:
Run GetANDQuadTree(J_1,J_2) and GetORQuadTree(J_1,J_2)

```
GetANDQuadTree(I,J):
-     Initialize vertex v with no subtrees or values
-     if both I,J have subtrees:
-          for each subtree i from 1 to 4:
-               v.subtree(i) = GetANDQuadTree(I.subtree(i),J.subtree(i))
-          return v
-     else if both I,J are leaves:
-          v.value = (I.value and J.value)
-     else:
-          let K_1 be the one that is a leaf
-          let K_2 be the one that has a subtree
-          if K_1.value is 0:
-               v.value = 0
-               return v
-          else:
-               for each subtree i from 1 to 4:
-                    v.subtree(i)=K_2.subtree(i)
-               return v
```

```
GetORQuadTree(I,J):
-     Initialize vertex v with no subtrees or values
-     if both I,J have subtrees:
-          for each subtree i from 1 to 4:
-               v.subtree(i) = GetORQuadTree(I.subtree(i),J.subtree(i))
-          return v
-     else if both I,J are leaves:
-          v.value = I.value or J.value
-     else:
-          let K_1 be the one that is a leaf
-          let K_2 be the one that has a subtree
-          if K_1.value is 1:
-               v.value = 1
-               return v
-          else:
-               for each subtree i from 1 to 4
-                    v.subtree(i)=K_2.subtree(i)
-               return v
```