Project 3 Analysis

Bharath (Jai) Chintagunta, Harsha Jonnavithula, Tarun Nadipalli

April 30, 2018

1 General Overview of Problem

Consider the problem where we are given a set of 3-d points that exist in $\mathbb{R}^3$, and we want to find out how many points are within a certain space Q where $Q \subseteq \mathbb{R}^3$. For the sake of simplicity the input Q takes the form of a rectangular prism.

2 General Overview to our Approach

Our team targeted to write an algorithm that followed the limitations specified by Tier 3 of the Project 3. Recall the bounds of Tier 3 are as follows Query Time: $O(\sqrt{n})$, Space: $O(nlogn)$, Construction Time: $O(nlogn)$

Our approach utilizes a 2-dimensional KD Tree partition on x and y medians.Each node in the KD Tree maintains a list of KD Points which are sorted in z order. All sorts utilized in our approach sort by composite costs (if $x_1 = x_2$ then compare y_1 and y_1 and so on). We utilized fractional cascading to achieve a fast query time.

2.1 Building KD Tree

Our procedure is as follows:

1. Sort the points by x, y, and z with composite cost
2. Find the median and partition a list of point P by the median and recursively build the tree on the sub divisions
    - The median is either the x median or the y median depending on the depth, (if the depth is even x median, otherwise y median)
3. The base case is when a division only has one elment in which we make that a leaf node of the KD Tree.

We will further discuss how our approach to building is within the constraints of Tier 3 and specifically how the partitioning in the build process works in Section 4.

2.2 Querying the KD Tree

The basic idea of the query algorithm is that as we traverse through the KD Tree we are building smaller and smaller subspaces of query Q,C, which contain the points in that subtree. We want to compare this subspace to our query; recall that our query, Q, is a rectangular prism. If C is fully contained with in Q (formally, $C \subseteq Q$)then we know that we must check points from our 1-d range tree utilizing Fractional Cascading. If none of C is within Q(formally, $C \cap Q = \{\emptyset\}$) then we know that none of the points of C fulfill our query requirements. Now, if C is partially in Q (formally, $C \cap Q = \{a_1, a_2, \ldots, a_n\}$) then we must split C into two rectangles and proceed recursively. We also have a special case when we are at a leaf node in our KD Tree, where we simply check if the point is in Q or not (Note this is also the base case for recursion).

3 Description of Class and some Main Methods

3.1 KDPoint

KDPoint is a class that stores a point in $\mathbb{R}^3$. A KDPoint also points to a an array of z values that are always sorted.

3.2 RecPris

RecPris is a class that stores information, specifically the bounds of prisms that exist in $\mathbb{R}^3$

3.3 KDTree

The purpose of the KDTree class is really just to build the KDTree, the bulk of this class lies in the method BuildKDTree.

3.3.1 BuildKDTree

The method signature is as follows:

```
buildKDTree(KDPoint[] P, KDPoint[] xsorted, KDPoint[] ysorted,
        KDPoint[] zsorted, int depth)
```

where `KDPoint[] P` is an array of points (not necessarily sorted).
`KDPoint[] xsorted` is `P` sorted by the x values
`KDPoint[] ysorted` is `P` sorted by the y values
`KDPoint[] zsorted` is `P` sorted by the z values
`depth` is the depth of the KDTree

3.3.2 Procedure

```
buildKDTree:
if length(P) = 1
    return(P[0])
else
    if depth is even
        medIndex ← index of the median of xsorted
        div1,xsortediv1 ← xsorted[0...medIndex]
        div2,xsortediv2 ← xsorted[medIndex+1...end]
        ysorteddiv1,zsorteddiv1 ← Elements of ysorted,zsorted ≤ xmedian
        ysorteddiv2,zsorteddiv2 ← Elements of ysorted,zsorted > xmedian
    else
        medIndex ← index of the median of ysorted
        div1,ysortediv1 ← ysorted[0...medIndex]
        div2,ysortediv2 ← ysorted[medIndex+1...end]
        xsorteddiv1,zsorteddiv1 ← Elements of xsorted,zsorted ≤ xmedian
        xsorteddiv2,zsorteddiv2 ← Elements of xsorted,zsorted > xmedian

left ← buildKDTree(div1, xsorteddiv1, ysorteddiv1, zsorteddiv1, ++depth)
right ← buildKDTree(div2, xsorteddiv2, ysorteddiv2, zsorteddiv1, ++depth)
```

3.4 Query

The method signature is as follows:
`int rangeCount(RecPris Q, KDPoint t, RecPris C, int depth)`
`Q` is a rectangular prism where $Q \subseteq \mathbb{R}^3$
`C` is a rectangular prism where $C \subseteq \mathbb{R}^3$
`depth` is the depth of the KDTree

3.4.1 Procedure

```
if (t is not a leaf)
    if(Q contains t) return(1)
    else return(0)
else
    if(C ∩ Q = {∅}) return(0)
    else if(C ⊂ Q) return( z_min ≤ # of points in C ≤ z_max)        (*)
    else
        C_1 ← Left or Bottom Rectangle in Intersection
        C_2 ← Right or Top Rectangle in Intersection
        return(rangeCount(Q,t.left,C_1,++depth)+rangeCount(Q,t.right,C_2,++depth))
```

In (*), note that since C is constructed with the constraints of our 2-d KD Tree. Thus, we must check to make sure that our nodes/points in our 2-d Kd tree satisfy the z value constraints from the query. This is why in our construction each KDNode points to a 1-d range tree on the z dimension. One efficient way to count the number of points which satisfy our z constraint from the query is simply to binary search for the min z constraint and max z constraint and return the number of elements between these two in our 1-d range tree. We will see that this leads to a $O(\sqrt{n} \cdot log(n))$ query time. We will see an improvement to this using Fractional Cascading next.

4 Fractional Cascading

Notice that as we build the K-d tree level by level our corresponding 1-d range tree for dimension z gets smaller, specfically it is a subset its corresponding KDNode's parent's range tree. This leads to the question if we do binary searches on n sets where $k_1 \supset k_2 \supset \ldots \supset k_n$, where k_i represents a set (specifically the values stored in our 1-d range tree). Lets look at a simplification of the problem in the context of sets of integers rather than with geometrical data structures.

Consider the following problem: Given n sets where $k_1 \supset k_2 \supset \ldots \supset k_n$, where k_i denotes each consequent subset. Furthermore the sets $k_1, k_2, \ldots, k_n$ are sorted. What is the most efficient way given a query number Q to find elements $a_j \in k_i \mid 1 \leq j \leq m_i$ such that $a_j \geq Q$ where m_i is the size of the particular subset k_i.

Consider an example for $n = 2$, Let $Q = 2$,

$$k_1 = \{1, 2, 3, 4\},$$

$$k_2 = \{2, 3, 4\}$$

Instead of binary searching for Q twice in k_1 and k_2 we can simply map elements from k_1 to k_2 in a way such that we only have to do one binary search on k_1 and follow the mappings to k_2 to find our answer. We define this mapping by having elements from k_1 point to an element in k_2 which is in the smallest position/index that is greater than or equal to the element from k_1. This can be generalized to n sets and can be further generalized to optimize our binary search of 1-d range trees in our query implementation. We generalized the procedure below for the mappings. In our Project we defined a function setArrayPointers to do exactly this.

The method signature is as follows:

```
void setArrayPointers(KDPoint zarr, KDPoint zdiv, Char c)
```

where

`zarr` is the set

`zdiv` is the subset of `zarr`

`c` specifies whether we are setting a left pointer or right pointer

4.1 Procedure

$i, j \leftarrow 0$
while(`i` $<$ `length(zarr)`)
    if($z_{arr}[i] == z_{div}[j]$)
        $z_{arr}[i]$ points to $z_{div}[j]$
        $i, j \leftarrow i+1, j+1$
    else if($z_{arr}[i]$ ¡ $z_{div}[j]$)
        $z_{arr}[i]$ points to $z_{div}[j]$
        $i \leftarrow i+1$

Now instead of binary searching everytime we go into **(*)** from Procedure 3.4.1 for the elements in our 1-d range tree for dimension z, we only do it once when $depth = 0$, then every subsequent time we simply traverse the pointers defined by Procedure 4.1 to find the number of elements which satisfy **(*)** from Procedure 3.4.1

5 Analysis of Time and Space

5.1 Construction

Space:

`buildKDTree:`
if `length(P)` $= 1$
    `return(P[0])`
else
    if `depth` is even
        `medIndex` $\leftarrow$ index of the median of `xsorted`
        `div1,xsortediv1` $\leftarrow$ `xsorted[0...medIndex]` (1)
        `div2,xsortediv2` $\leftarrow$ `xsorted[medIndex+1...end]` (2)
        `ysorteddiv1,zsorteddiv1` $\leftarrow$ Elements of `ysorted,zsorted` $\leq$ `xmedian` (3)
        `ysorteddiv2,zsorteddiv2` $\leftarrow$ Elements of `ysorted,zsorted` $>$ `xmedian` (4)
    else
        `medIndex` $\leftarrow$ index of the median of `ysorted`
        `div1,ysortediv1` $\leftarrow$ `ysorted[0...medIndex]`
        `div2,ysortediv2` $\leftarrow$ `ysorted[medIndex+1...end]`
        `xsorteddiv1,zsorteddiv1` $\leftarrow$ Elements of `xsorted,zsorted` $\leq$ `xmedian` (5)
        `xsorteddiv2,zsorteddiv2` $\leftarrow$ Elements of `xsorted,zsorted` $>$ `xmedian` (6)

left $\leftarrow$ `buildKDTree(div1, xsorteddiv1, ysorteddiv1, zsorteddiv1, ++depth)`
right $\leftarrow$ `buildKDTree(div2, xsorteddiv2, ysorteddiv2, zsorteddiv1, ++depth)`

Every single time we divide partition P into two divisions we must create two arrays for the divisions. Assume that the size of P is n thus we get two arrays resulting in sizes

$medIndex+1$ and $n-(medIndex+1)$. Notice that:

$$(medIndex+1)+(n-(medIndex+1))=n$$

. Thus, every single time we divide partition P into two divisions we get Space Complexity: $O(n)$. Notice that we partition P, 8 times in (1),(2),(3),(4),(5),(6); also we do this for *logn* levels as our KD Tree is a 2-ary tree that is balanced.

$$T(n)=8nlogn$$

$\implies$

$$O(nlog(n))$$

Time: First we divide P into two partitions. Note that partitioning our y,z sorted arrays when we have an *xmedian* and partitioning our x,z when we have a *ymedian* takes linear time. Without loss of generality this is due to the fact that we must iterate through our y,z lists when we have an *xmedian* and partition into left and right subarrays by marking elements in y,z as left and right. Thus we get

$$T(n)=2T(\frac{n}{2})+O(n)$$

. Recall Masters Theorem is of the form,

$$T(n)=aT\left(\frac{n}{b}\right)+f(n)$$

$$\therefore a=2,b=2,c=1,k=0$$

. Follows the following case of Masters Theorem:

$$c=log_a(b)\implies O\left(n^{\log_b a}\log^{k+1}n\right)$$

$$\therefore O(n)=nlog(n)$$

5.2 Query

Notation: Let Q be a rectangular prism, and let C be a rectangle.

Let grey nodes represent nodes in our KDTree which may or may not be in the solution (meaning we use Q and C to decide how to proceed). Recall the Lemma from Chapter 11 of the *Computational Geometry* by *David M. Mount*. The Lemma is as follows:

Lemma: Given a balanced kd-tree with n points using the alternating splitting rule, any vertical or horizontal line stabs $O(\sqrt{n})$ cells of the tree.

In plain English the Lemma claims that there are $O(\sqrt{n})$ grey nodes, a proof can be found in the textbook. We will use this to show that the query time complexity is $O(\sqrt{n} \cdot logn)$.

Claim: The worst case query time complexity utilizing fractional cascading is $O(\sqrt{n})$.
Proof: Recall that the number of nodes that we visit is $O(\sqrt{n})$, further recall that the worst case time complexity of a binary search is $O(log(n))$. Utilizing fractional cascading we have the overhead of computing a binary search once so $T(n) = log(n)$ We potentially have to visit $\sqrt{(n)} - 1$ nodes now from (*) of Procedure 3.4.1; however the computation at each grey node is only $O(1)$ as we only have to subtract the index of the MaxPointer and the index of the MinPointer implemented in Fractional Cascading. Thus

$$T(n) = log(n) + \sqrt{n}$$

$\therefore$ the worst case time complexity for query time is

$$O(\sqrt{n})$$

Remark Note that fractional cascading does have the overhead of setting up the pointers from a set to its subset. However, this is done in $O(n)$ as in Procedure 4.1 we iterate through the *zarr* and utilizes some extra space (for pointer arrays) which is $O(n)$ space, and thus does not change our construction time complexity or space complexity.