HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Fall 2017

T-301-REIR, Reiknirit

# S3: KD-Tree


Birgitta Feldís Bjarkadóttir
kt. 180395-3189
Edda Steinunn Rúnarsdóttir
kt. 241095-2909

Group 2


September 29, 2017

TA: Hannes Kristján Hannesson

# 1 Implementation

The project implements two different algorithms that both determine location of a set of points in a unit square and stores them in an ordered manner. The implementations have a method to find a given point's nearest neighbor point in the space and a method to list points within a given rectangle. The first implementation relied on a red-black tree data structure and the second on a 2D-tree data structure, that is a Binary Search Tree with two-dimensional keys. Since the red-black tree structure is a less efficient, less complex (brute-force) algorithm, this chapter of the report focuses entirely on implementation and methods of the 2D tree structure.

## Node

The Node data type that was used to implement the 2D-tree data structure was created in the class **KdTree.java** and is a tiny private class with a few parameters and no methods apart from a constructor. The node has a **Point2D** object as key representing a point in a 2D plane. The key of course determines node's location in the tree. In addition to the key, the node has two links; right link and left link representing the node's right and left child nodes. The node also has it's own **RectHV** object (a rectangle object) that corresponds to the rectangle that forms when a perpendicular line is drawn from node's point to parent's rectangle object. This was done to facilitate the nearest neighbor search and range determination.

## Range Search

The *KDtree_find_range( )* private function calls for another private recursive function to find points that fit a range within a given rectangle. It only takes in a single parameter; the rectangle that we should find all points in. The function initializes an empty queue object, then calls for the recursive function The recursive function, *KDtree_find_range recursive*, that fills the queue with appropriate points. The recursive function takes in three parameters:

- A queue object that will by the end of function execution include points within specified rectangle; is the returned object
- A node object which tells us were we are in tree in each recursive execution
- The rectangle in which we are searching for points

When the non-recursive function calls the recursive one the node parameter is always the root node since we want to start walking through the tree from there (it denotes the beginning of the tree).

The base case of the function represents the pruning rule and improves the algorithm's efficiency; execution will stop when the query rectangle does not intersect the current node rectangle. In those cases there is no need to explore that node or it's sub-trees further as there is no possibility that node's point is within rectangle.

If the base case does not apply the function checks whether or not a node is inside or on the bearer of the rectangle by using the RectHV object's function *intersects( )* which checks whether or not the node rectangle and the given rectangle intersect. Then the function checks if the current node's point is within the rectangle using RectHV's function *contains( )*. If this is the case that points is added to the queue object.

The left and right sub-trees are searched in that order since this order is usual for binary trees, but no particular sub-tree search order is preferable for this functionality. The pruning rule will take care of sub-tree pruning when searching. When there are no more nodes with rectangles that intersect the parameter rectangle, the recursive function returns and the non-recursive function returns the queue now filled with each point in range of the query rectangle. As expected the queue is returned empty when no points are within range of rectangle.

### Nearest Neighbor Search

The the nearest neighbor search is implemented by a function called *KDtree_find_nearest( )*. This function starts by checking special cases of a neighbor search. There are two special cases. The first one is if the tree is empty the function will return null (this is achieved by using the *isEmpty( )* private function for tree). The second one is if the tree contains a point equal to the query point it is it's own nearest neighbor, hence that point is returned (this is achieved using the function *contains( )* for tree).

If neither of the special cases apply, the function makes a call to the recursive function *KDtree_find_ nearest_recursive( )* to find the nearest neighbor of point. That function has four parameters:

- A node object that keeps track of location in tree each time
- A nother node object containing the "champion point". That is, the point closest so far to the query point.
- A point object as the query point
- Boolean value that's true for when we are on a horizontal depth of tree and false for when we are on vertical depth of tree - important for knowing whether to compare points horizontally or vertically

When the non-recursive function calls the recursive one, these parameters are initialized as the **root** (to start walking tree at beginning, the **root** (we have not checked any other point so root must be the nearest neighbor of point), the **query point** (never changes through execution) and **true** (true for horizontal; the first depth of tree (root depth) is considered a horizontal depth in our implementation). All the parameters correspond to the key of the given query point that is being checked.

The recursive function has two base cases. The first base case returns the champion when the end of a sub-tree has been reached. The second base case represents the pruning rule and improves the algorithm's efficiency; if the rectangle of the current node is less than or equal to the champion point there is no need to explore this node or it's sub-trees any further and the function will return the current nearest node. The pruning rule effectively prunes sub-tree searches when appropriate.

If neither of the base cases apply the current node and it's sub-trees are checked. First we check whether the current node point is nearer to query point than the champion using the *distanceSquaredTo( )* function. The node is chosen as new champion point if appropriate. Then a comparison is made for the query point and the current node's point to see on which side of the node's splitting line the query point is located. This determines whether the left or the right sub-tree of current node is explored first. Choosing which sub-tree to explore first enhances the efficiency of the pruning rule as it explores the more likely sub-tree first and is therefore more

likely to prune the sub-tree not containing the query point faster. This therefore improves overall time complexity.

When the entire tree has been walked through and/or appropriate sub-trees have been pruned recursively, the node containing the nearest neighbour point to query point has been found and is returned to the non-recursive function which returns the node's point.

# 2 Analysis

## Memory

(In this section, B is used as an abbreviation for bytes)

- Bytes per Point2D: **32B**
- Bytes per RectHV: **48B**
- Bytes per Node object: Overhead 16B + padding 8B + Point2D object 32B + RectHV object 48B + 2 Node references 2*8B = **120B in total**
- Additional variables in KD-tree structure:

  root: one node object so **120B**

  2 boolean variables (determine horizontal/vertical comparisons): so **2B**

  Integer num_of_points: **4B**

bytes per KDTree of N points (using tilde notation): KD-variables + Memory Usage per Node = root node + 2 boolean variables + integer + N Node = 120B + 2B + 4B + N*120B = 126B + N*120B which gives us a tilde notation of: $\sim$ **120N bytes**

## Running Time

Table 1 gives the running times to build a 2d-tree on N random points in the unit square for different sizes of N. Tests in table 1 were conducted on operating system OS X 10.9.5 running on a 1,3 GHz Intel Core i5 processor with Intel HD 5000 1536 MB graphics. The expected running time in seconds is a tilde notation of $\sim 7.5 * 10^{-7} * N$ and our hypothesis states therefore that the running time can be semi-precisely estimated with the equation $T(N) = 7.5 * 10^{-7} * N$ where T(N) denotes the running time for N and N denotes number of random points.

Table 1: *Running time to build a 2d-tree on N random points*

| *N* | *time to create 2d-tree* |
|---|---|
| 1000 | 0.008s |
| 2000 | 0.008s |
| 4000 | 0.008s |
| 8000 | 0.009s |
| 16000 | 0.016s |
| 32000 | 0.024s |
| 64000 | 0.032 |
| 128000 | 0.052 |

Our hypothesis seems to be valid as when entering N = 16000 into the equation $T(N) = 7.5 * 10^{-7} * N$ we get a value close to 0.016s (0.012s) and we get exactly 0.024s when entering the value N = 32000 into the equation.

### Nearest Neighbor

Table 2 gives how many nearest neighbor calculations the brute-force implementation performs per second for input100K.txt which include 100,000 points and input1M.txt which includes 1 million points. All the points are random points in the unit square. Tests in table 2 were conducted on operating system OS X 10.9.5 running on a 1,3 GHz Intel Core i5 processor with Intel HD 5000 1536 MB graphics.

In order to conduct the experiment a stopwatch variable was initialized using the Stopwatch-method from the algs4 library. The stopwatch variable was initialized after building a tree and generating random points to an array to keep as a parameters for the nearest neighbor calls. We then measured calls to *nearest( )* per second using a loop that terminated when the stopwatch had reached 1s and had a counter integer that counted calls to *nearest( )* and determined results in table 2.

Table 2: *Calls to* ***nearest()*** *per second*

| | *brute force* | *2d-tree* |
|---|---|---|
| input100K.txt | 257 \s | 791.308 \s |
| input1M.txt | 22 \s | 478.046 \s |

Larger tree meant greater time-complexity for the *nearest( )* function. Obviously, the 2D-tree is much more optimized than the brute force algorithm and has significantly less time complexity. In fact, the 2D-tree implementation could call the function 2000-3000 more times than the brute force implementation.

# 3 About This Solution

Have you taken (part of) this course before: **No, neither one of us has.**
Hours to complete assignment (optional): **15+**

## 3.1 Known Bugs/Limitations

There are no known bugs and/or limitations for functionality, although the system cannot draw itself properly.

## 3.2 Help Received

No help was received.

## 3.3 Problem Encountered

No serious problems were encountered when solving the project. The assignment went OK as a whole although implementation and understanding concepts was time consuming. The Nearest Neighbor analysis was a bit time consuming due to problems with understanding the instructions, they could have been more thorough.

## 3.4 Comments

The 2D tree was implemented separately to the specified public API in project. That is, the 2D tree and each of it's methods is constructed entirely from private functions and are stored in it's own section below the public API. The public API functions therefore have no real functionality: their only purpose is to alter the tree that the project implements, then call for the tree's methods. This type of design was for us to further understand the KD-tree as a data type in isolation and it seemed like a more generalized implementation of the concept and therefore more understandable.