

Writeup for third project of
CMSC 420: “Data Structures”
Section 0102 , Summer 2017

Theme: KD Trees and Priority Queues

Handout date: Wednesday, 07-12-2017

On-time deadline: 07-22-2017, 11:59pm

Late deadline (30% penalty): 07-24-2017, 11:59pm

1 Overview

For this project, you will be implementing two data structures that we’ve talked about in class: (Bounded) Priority Queues and K-D Trees. To that end, you will design and implement two Java classes: `BoundedPriorityQueue` and `KDTree`. The reason for having you implement two separate classes for each data structure is because you will be using the first one to solve a certain class of queries for the second one, and separating the interfaces allows us to write tests for both that will help you out with debugging!

The Priority Queue part of the project is somewhat straightforward to explain, yet has some interesting twists to make you think about Priority Queue implementations. You will have full freedom to implement `BoundedPriorityQueue` in whichever way allows you to pass our unit tests. This includes modifying Priority Queue libraries given by the Java standard library or a third-party implementation you trust.

The K-D tree part of the project is self-explanatory as well. You will implement K-D Trees for an arbitrary k , supplied as information by the user. `KDTree` will support standard structure operations (insertion, deletion, lookup) and spatial queries (range, (m -)nearest neighbor(s)). Our unit tests are very comprehensive in terms of code coverage and parameter ranges, so you will need to think about many corner cases of your algorithms! To establish a common interface between your code and ours, we define a class that describes a k -dimensional point in Euclidean space and provide it for you so that you can use it. More details are given in Section 2.

2 Provided code base and documentation

For this project, we supply you with an implementation of two Java classes called `KDPoint` and `KNNComparator`, as well as Javadocs for these classes. We also include tests for `KDPoint` both to give you ideas for quality unit testing and to pacify any concerns regarding the state of `KDPoints`. `KDPoint` is a simple abstraction over k -dimensional points in Euclidean space. `KNNComparator` is a class that implements a `Comparator` that sorts `KDPoints` based on the Euclidean distance towards a given “anchor” `KDPoint`.

Among those two classes, `KDPoint` is the only one that is really interesting. `KNNComparator` is only useful for testing (i.e we use it in our own unit tests, so your submission needs to contain it such that the tests don’t break). You are most welcome to use it for your own tests if you deem it worthwhile.

When it comes to `KDPoint`, you will notice this class’s objects have 8-byte accuracy and they can be constructed in various ways. **Make sure you read either the documentation for this class *before* you proceed with your project!** In particular, it is important to understand how to properly construct objects of this class. Make sure you examine the documentation for the constructors in order to avoid any pitfalls.

Furthermore, while you are most welcome to add to the functionality of `KDPoint` and `KNNComparator` if you so desire, we recommend that you do *not* alter their *existing* functionality. As you will see in section 3.2, some of your public methods will play around with `KDPoints`, and **our unit tests depend on the functionality provided to function smoothly**. When it comes to `KNNComparator`, while it is true that we do not require it anywhere on the interface, we use it on our tests, so its current functionality is also needed.

Lastly, you will notice that the implementation of those classes is contained within a sub-directory called `utils`. Please maintain this directory-file structure, since our unit tests treat the name `utils` as that of a package which is imported at compile-time. **Both `BoundedPriorityQueue.java` and `KDTree.java` will have to be in the default package**, as with our previous projects this summer session. Refer to 5 for more details.

3 Interface

As with project 1, we will need you to provide some **public** methods which will implement the desired interface. Those methods will be checked against through our unit tests. All **private**, **protected** or package access data members and methods that you use are your own business.

The code bundle mentioned in section 2 contains Javadoc that explains the expected behavior of the methods of both `BoundedPriorityQueue` and `KDTree`. This Javadoc only contains **public** method descriptions; as always, you are free to implement those in any way you wish, **subject to some constraints mentioned in section 3.2**.

*Important note: Read every method’s Javadoc **very carefully!!!** Some methods require that you throw appropriate exceptions when certain conditions are met; our unit tests depend that these exceptions are thrown wherever appropriate!*

3.1 BoundedPriorityQueue

`BoundedPriorityQueue`'s (hereafter called *BPQ* for brevity) interface is described in the docs. A BPQ is different from a classic PQ in that it only allows one to store a certain number m of elements. If it already does have m elements (so it's full) and a certain candidate element for insertion has a priority **smaller** than that of the *last* element in the BPQ, then that last element is removed from the queue and the candidate is inserted in the appropriate space, which is dependent on the implementation of the queue itself (linked / array-based heap, list of lists, etc).

3.2 KDTree

`KDTree`'s interface is also described in the docs. Notice that the method that implements the m -nearest neighbor queries returns a `BoundedPriorityQueue` over `KDPoints` to the caller. You are thus **required** to use your `BoundedPriorityQueue` to solve these queries, and of course you have to use our familiar *branch-and-bound* algorithm to fill in this `BoundedPriorityQueue`. That is, submissions that just fill in this queue by solving the m -nearest neighbor problem in a **brute-force** manner ¹ **will not be considered for credit with respect to the relevant unit tests! This is also true for range queries:** The naive approach of amassing all points and then only inserting those that are within the desired range in the `Collection` instance that is returned is **not** acceptable, and the relevant unit tests will **not** receive **any** credit!

4 Code Structure / Directives

Naturally, there are various different ways that you can approach this project with. You can handle the KD-Tree part first, assuming the Bounded Priority Queue (BPQ) implemented, switch to the BPQ before implementing KNN, and then implement KNN. Or you can handle the BPQ first, and then move on to the KD-Tree. There is no right or wrong way to approach the project, so the following should only be considered general directives rather than strict guidelines:

- Think about how a BPQ differs from a Priority Queue. Is there anything in a BPQ that warrants additional thought when compared to a standard Priority Queue? What would be a reasonable implementation of a BPQ?
- Your algorithms should really be agnostic towards the dimensionality of the space. More informally, you should be able to generalize everything you know about “2D-trees” to arbitrary-dimensional KD-Trees. For $k \geq 3$, it is unlikely that you will be able to even produce a meaningful visualization; that's not a problem, so long as you understand all corner cases of KD-Tree operations. You can be certain that our unit tests cover a number of various dimensions!

¹The brute-force algorithm in this case would be to recurse over all elements, sort them based on distance to the anchor element, and insert the first k ones into the `BoundedPriorityQueue` instance that is returned.

- Make sure that you avoid some common pitfalls associated with range and nearest neighbor queries:
 - The query points themselves should *not* be part of your answer. If they were, this would make nearest neighbor queries **for points that are part of your tree** trivial to implement; you would just need to return the point itself.
 - Range queries are “**range-inclusive**”; that is, points exactly within the specified range *should* be part of your answer.
 - In nearest neighbor queries, is there any possibility of ties? The answer is **yes**, but one needs to be careful to understand the nature of those tie-breakers. Consider the case of Figure 1. For the query point q , a **2** (two) -NN query would return points p_1 and p_2 , without any tie-breaker issues. For $m = 3$ (three), we need to decide which point among p_3 and p_4 we should pick as our **third** nearest neighbor. For $m \geq 4$, **both** of these points need to be in our solution set, and the question then is *which one we consider the closest to our query point such that we sort them appropriately!* In our unit tests, we assume that **the point that is first to be admitted to our solution set as we traverse the KD-Tree is the one that ends up being considered “closest”**. If you think about it, this is consistent with Priority Queue primitives, where tied elements are stored in FIFO order!
 - Recall that there is **absolutely no reason** for the query point of a range or nearest neighbor query to be **actually contained** by the tree itself.

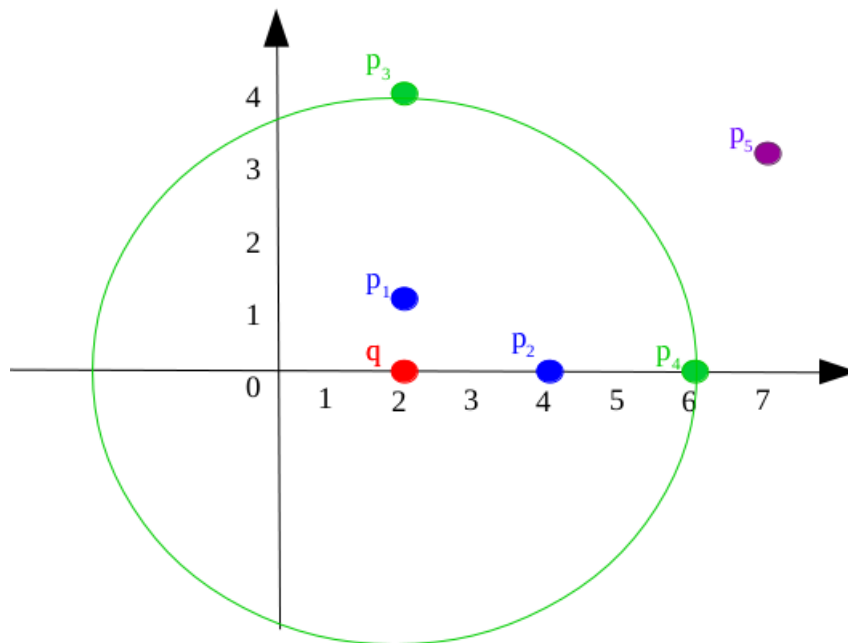


Figure 1: A **query point** and various near neighbors of the point. The **blue points** are the 2 nearest neighbors of the point. The **green points** are equi-distant to q , and p_5 is the furthest neighbor of all. Figure best viewed in color.

5 Submission

Submission of your project will occur in exactly the same fashion as the previous projects. Both `KDTree.java` and `BoundedPriorityQueue.java` will need to be in the default package. You will also need to leave the package `utils` **AS IS**. This package structure is required by our unit tests.

When you feel ready to submit, please create a ZIP,TAR.GZ or JAR archive off of your **ENTIRE PROJECT DIRECTORY** and upload it on the submit server as Figure 2 shows.

[illegible]

Making another submission

- use automatic submission tools,
- edit and submit code in the browser (discouraged)
- upload source files

Uploading a submission

You can submit a zip file or multiple text files.

file(s) for submission

File(s) to Submit
<input type="button" value="Choose File"/> No file chosen

Figure 2: Uploading your project on the submit server.

