CS2521 Assignment: Range Searching

Nir Oren

Version 1.1

Overview and Marking

This project will evaluate your ability to create, analyse, implement and evaluate algorithms. It will count for 25% of your overall course mark.

Hand-in

The assignment is due at 23:59:59 on the 25th of March 2019. Handing in up to 24 hours late will attract a 10% penalty, while handing in up to a 7 days late attract a 25% penalty, deducted as a percentage of the mark obtained. Work handed in more than a week late will be treated as a “no paper”.

Submission should take place via myAberdeen — upload a *ZIP* file containing a single *PDF* report, and source code to the appropriate submission area. There should be no subdirectories in your ZIP file *If you upload a RAR, ARC, ARJ, LZH, TGZ, or other format, your submission may not be marked. If you hand in your report as a non-PDF document e.g., a DOC, ODT, RTF, etc file), it will not be marked, and you will receive a 0 mark.*

Plagiarism

Plagiarism is a serious offence, and will not be tolerated. If caught, punishments ranging from a 0 mark to expulsion. If you are unsure about whether your work counts as plagiarised, please contact me before the submission deadline. For further details, please refer to the Code of Practice on Student Discipline (`https://www.abdn.ac.uk/staffnet/documents/academic-quality-handbook/Code%20of%20Practice%20in%20Student%20Discipline%20(Academic).pdf`).

Implementation details

Tasks with an associated implementation will specify the filename you should use, and the input and output format. All input will be provided on STDIN (as done in Codemarker), and should be output on STDOUT in the format specified. *Failure to adhere to the specifications may result in a 0 mark for that task.*

Introduction

A common problem in many domains, ranging from computer games to GIS databases, involves identifying a set of data-points within some interval or range. In this assessment, you will investigate data-structures to make such queries operate efficiently, and evaluate them.

1-D range finding

We begin by considering how a set of elements can be identified from a 1-dimensional range. For example, given the list [3, 10, 19, 23, 30, 37, 49, 59, 62, 70, 80, 89, 100, 105], and the range [18,77], the sublist [19,23,30,37,49,59,62,70] should be returned.

Basic Approaches

Tasks

Given a list containing n *unique* values, and a range x_s, x_f, describe algorithms which return all values within the range x_s, x_f (inclusive), and which run in worst-case

1.1 $\Theta(n)$ time for both adding n items to the list, and for querying the list. Implement your approach using the input and output described in Task 2.1. Call this file `Task11`.

1.2 $\Theta(n \log n)$ for adding a new element to the list, and $\Theta(k + \log n)$ time for querying, where k is the number of elements returned by the query. Implement your approach using the input and output described in Task 2.1. Call this file `Task12`.

For both of these, you should provide both an intuitive (English) description of your algorithm, as well as pseudocode. You should also prove the correctness of your algorithms and its complexity.

A Tree based approach

Now consider a balanced binary search tree whose leaves store elements in the list, and whose internal nodes store *splitting values* used to guide the search. For an internal node v, its splitting value x_v is set as the greatest value found below its left sub-tree. For this node, all values in the left sub-tree are smaller or equal to x_v, and the values in its right sub-tree are greater than x_v. Figure 1 illustrates one such tree.

The first operation we consider — given such a tree — is to find a *split node* for some query. That is, which roots the subtree which minimally covers the query interval. Figure 1 illustrates the subtree rooted at the split node in light grey for the interval between 18 and 77 (inclusive). Algorithm 1 describes how a split node can be found.

Algorithm 1 Algorithm to find the split node.

```
Require: A tree $T$, an interval $[x_s, x_f]$
 1: function FindSplitNode($T, x_s, x_f$)
 2:     $v \leftarrow root(T)$
 3:     $x_v \leftarrow$ value stored in $v$
 4:     while $v$ is not a leaf and ( $x_f \leq x_v$ or $x_s > x_v$) do
 5:         if $x_f \leq x_v$ then
 6:             $v \leftarrow$ left child of $v$
 7:         else
 8:             $v \leftarrow$ right child of $v$
 9:         end if
10:         $x_v \leftarrow$ value stored in $v$
11:     end while
12:     return $v$
13: end function
```

Observe that the nodes that should be returned in response to a query are always children of the split node (though some children must not be returned). Now consider the left subtree of the split node. If the lower limit of the interval is smaller than the split value stored at that node, then the right subtree

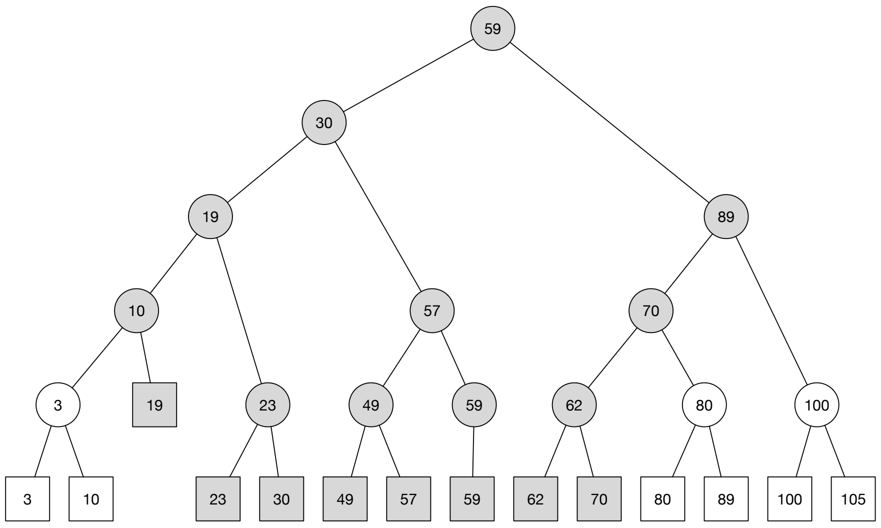

Figure 1: A search tree with splitting nodes.

of that subtree must fully be within the interval, and parts of the left subtree could also form part of the interval. Otherwise, the right subtree might contain elements of the interval. Finding the nodes to return can therefore be done recursively on the appropriate subtree.

We can apply the same argument to the right subtree of the split node by considering the upper value of the interval. This results in the pseudocode of Algorithm 2:

Tasks

2.1 Implement the `OneDRangeQuery` algorithm. The first line of input to your implementation will consist of two numbers, R Q, with R representing the number of elements over which search will take place, and Q the number of queries to be posed. The next R lines will each contain a single number representing an element to be stored. The final Q lines of the file will consist of pairs of numbers (separated by a space), S and F, which denote the start and end of the search range. Your output should consist of Q lines, with each line containing the space separated list of numbers that fall within the range of the qth query. For example, the input

```
8 2
3
10
23
30
62
47
105
89
7 47
10 89
```

will return the output

```
10 23 30 47
```

Algorithm 2 Performing a 1-D range query

```
Require: A tree $T$, an interval $[x_s, x_f]$
function OneDRangeQuery($T, x_s, x_f$)
    $v_{split} \leftarrow$ FindSplitNode($T, x_s, x_f$)
    if $v_{split}$ is a leaf then
        check if the point stored at $v_{split}$ must be reported (and if so, report it).
        return
    end if
    $v \leftarrow$ left child of $v_{split}$    $\triangleright$ Follow the path to $x_s$ and report the points in subtrees right of the path
    while $v$ is not a leaf do
        if $x_s \leq$ value stored in node $v$ then
            add the right subtree of $v$ to the report
            $v \leftarrow$ left subtree of $v$
        else
            $v \leftarrow$ right subtree of $v$
        end if
    end while
    check if $v$ should be reported
    Repeat lines 7-16 for the right subtree of $v_{split}$, going to $x_f$ and reporting subtrees to the left of the path.
    return reported nodes
end function
```

```
10 23 30 62 47 89
```

The file for this task should be called `Task21` (with an appropriate extension (e.g., `.py`, `.java`, or `.rb`)).

2.2 Describe/prove the complexity of initially building the tree from the list.

2.3 Describe/prove the complexity of performing a single query using the algorithm.

2.4 Finally, compare the average case complexity of the three approaches you have implemented (in tasks 1.1, 1.2 and 2.1). To do this, evaluate the time taken to construct lists of different sizes, and then do multiple random queries for each list, averaging the runtime. Plot these on a graph. Marks will be given for the depth of analysis undertaken here (e.g., error bars etc will provide additional marks). If the results differ greatly for each approach, discuss the type of workload that they might be appropriate for.

Scaling up to multiple dimensions

Consider how range reporting could work for multiple dimensions. For example, in 2 dimensions, one could have a set of points, and then query which of these points fall within the rectangle described by corner coordinates (1,1),(5,6). This is illustrated in Figure 2.

Implementation details

The input format for questions 3.1-5.4 is identical, and is as follows. The first line of input will consist of three numbers, R, D, Q, with R representing the number of elements over which search will take place, D the dimensionality of the space, and Q the number of queries to be posed. The next R lines will each contain

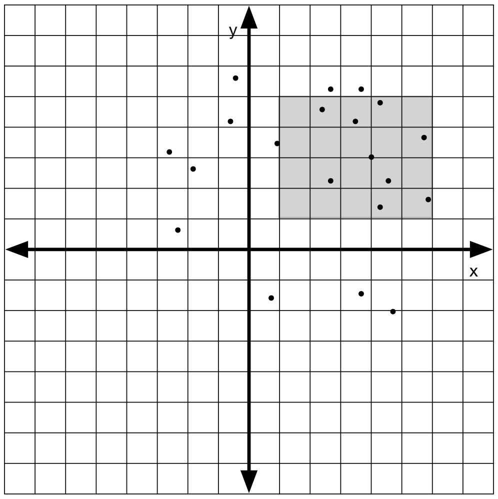

Figure 2: 2-D range search.

a D numbers per line, representing an element to be stored. The final Q lines of the file will consist of pairs of lists (each pair separated by a space), representing the range to be searched. The first element of the pair denotes the minimum coordinate for each dimension, and the second element of the pair denotes the maximum coordinate for each dimension.

Your output should consist of Q lines, with each line containing the space separated list of lists that fall within the range of the qth query. For example, the input

```
8 2 1
3 2
10 4
23 6
30 10
62 8
47 14
105 9
89 7
[7 2] [47 12]
```

will return the output

```
[10 4] [23 6] [30 10]
```

The file for each task should be called `TaskXY`, where `XY` is the task number (e.g., for task 3.1, the file should be called `Task31`. The extension is language dependent, and should have an appropriate extension (e.g., either `.py`, `.java`, or `.rb`)).

Input for tasks 5.5 and 5.6 are similar, changes are described within the tasks.

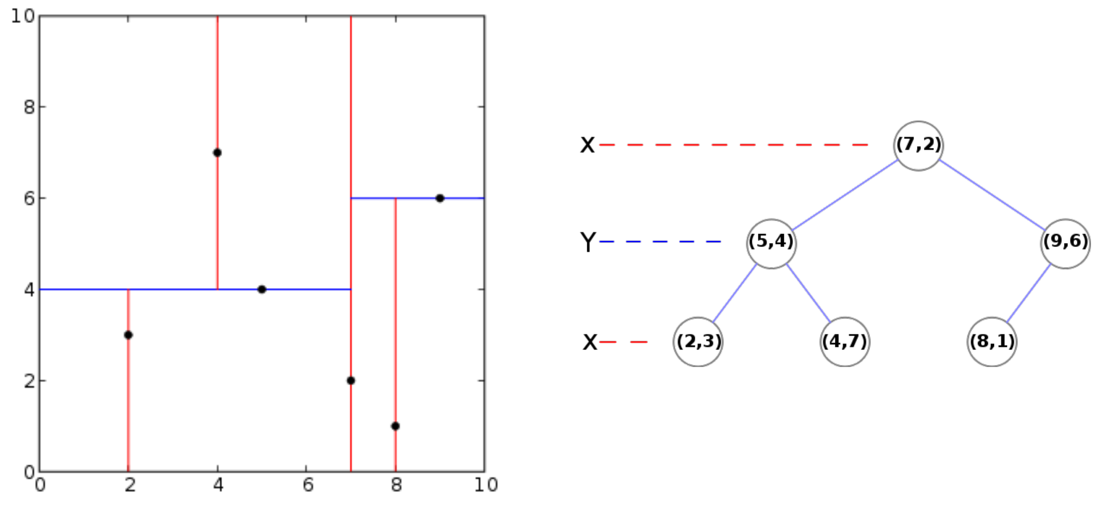

Figure 3: A 2-D split of a set of points, together with the resultant KD-tree. *Note that this tree was generated with a different median to that described in the text (due to being taken from a different source). It illustrates the general ideas, but will be different to the KD-tree you will create.*. Original source: KiwiSunset at the English language Wikipedia [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], via Wikimedia Commons.

Arrays again

Tasks

3.1 Describe and implement a simple algorithm to perform a multi-dimensional range query. More specifically, this algorithm should check if every point is within the range rectangle.

3.2 Prove the correctness of your algorithm, and derive its worst case complexity.

K-D Trees

We're going to focus on a simplified version of the range search problem, where no two points have the same coordinate in any dimension[1]. The first observation we make is that in (for example) the 2-D case, our query is actually composed of two 1-D sub-queries, one on the x-coordinate of the point, and one on the y-coordinate.

Now let's think about the data structure used for 1-D queries. What it actually does is split a set of 1-D points into two subsets of (roughly) equal size, with one subset containing points smaller or equal to the splitting value, and the other containing points bigger than it. This process is repeated in the subtrees.

For the n-dimensional case, we can do something similar, but split on a different coordinate at each level. For the 2-D case, we first split on (say) the x coordinate, and then on the y coordinate, and then on the x coordinate again, and so on. Figure 3 illustrates this, and also shows a tree obtained by splitting across the lines.

Such a tree, in k dimensions, is called a *KD-tree*. To build it, we can use Algorithm 3.

[1]Details of how to overcome this simplification can be found in [1].

Algorithm 3 Building a KD tree.

```
Require: A set of points $P$, a current depth, $d$
 1: function BuildKDTree($P, d$)
 2:     if $P$ contains only one point then
 3:         return a leaf storing this point
 4:     end if
 5:     $axis = d$ modulo the dimensionality of space
 6:     select a $medianPoint$ from pointlist on dimension $axis$
 7:     $P_l = \{p | p \in P$ and $p \leq medianPoint$ on dimension $axis\}$
 8:     $P_r = \{p | p \in P$ and $p > medianPoint$ on dimension $axis\}$
 9:     $V_l =$ BuildKDTree($P_l, d+1$)
10:     $V_r =$ BuildKDTree($P_r, d+1$)
11:     return a node with value $medianPoint$, left child $V_l$, right child $V_r$
12: end function
```

Tasks

4.1 Describe an algorithm to compute the median point of a list, and identify its complexity. For even cases, the median point should be rounded down.

4.2 By means of a recurrence, compute the time and space complexity of the `BuildKDTree` algorithm, taking into account the complexity of computing the median as described in the previous step.

4.3 Implement the `BuildKDTree` algorithm. This task will be used as part of a later task, but should be implemented in a separate file.

Range Searching in KD-trees

Having built the KD-tree, we must now consider how to search it. It is easiest to think about this by first considering the 2-D case. The splitting line stored at the root of the tree partitions the plane into two half-planes, with the points in the left half-plane stored in the left-hand subtree, and the points in the right stored in the right-hand subtree.

Nodes further down the tree are also regions in the plane. For example, the right child of the right child of the root corresponds to the region to the right of the first split, and below the second split. In general, any node in the KD-tree represents a region of the plane which may be unbound on one or more sides.

Now consider the region covered by a tree rooted at some point v. Any point within this region must be a leaf of the subtree of v. Therefore, given a region where we perform range checking, and some node of the tree, we have several possibilities.

1. If node is a leaf, we must check if it falls within the region.

2. If the region covered by a child of the node is fully contained within the region where range checking is performed, we can return all children of this node.

3. If the region covered by a child of the node is partially contained within the region where range checking is performed, we must perform range checking on the child nodes.

This leads us to Algorithm 4 for searching a KD-tree.

Note that the lines checking that a subtree is fully contained are optional; simply checking for the intersection will cover these cases too (but at a slight loss of efficiency). To implement the algorithm, we must be able to compute whether ranges intersect, which we consider next.

Algorithm 4 Searching a KD-tree

```
Require: v a (root) node of the KD tree; R the range or region for which points must be reported.
 1: function SearchKDTree($v, R$)
 2:     if $v$ is a leaf then
 3:         check if point stored in $v$ lies in $R$, and return it if so.
 4:     end if
 5:     if the region of the left child of $v$ is fully contained in $R$ then
 6:         report the subtree of the left child of $v$
 7:     else if the region of the left child of $v$ intersects $R$ then
 8:         SearchKDTree(left child of $v$,R)
 9:     end if
10:     if the region of the right child of $v$ is fully contained in $R$ then
11:         report the subtree of the right child of $v$
12:     else if the region of the right child of $v$ intersects $R$ then
13:         SearchKDTree(right child of $v$,R)
14:     end if
15: end function
```

Intersection

Rectangular regions can be defined by the minimum and maximum coordinates along each dimension. For example, the 2 dimensional rectangle with corners at (0,0),(1,0),(0,2) and (1,2) has a minimum/maximum x-coordinate pair (0,1) and y-coordinate pair (0,2). Finding the intersection between multiple d-dimensional rectangles can be achieved by applying Algorithm 5.

Algorithm 5 Algorithm for finding the intersection.

```
Require: minMax1,minMax2 a list with $d$ pairs (where $d$ is the dimension of the space). Each pair has the
    minimum and maximum values for that dimension for the region.
1: function intersect($minMax1, minMax2$)
2:     for all $d$ in the number of dimensions do
3:         if not ( $minMax2[d][0] \leq minMax1[d][0] \leq minMax2[d][1]$ or $minMax2[d][0] \leq minMax1[d][1] \leq minMax2[d][1]$ or $minMax1[d][0] \leq minMax2[d][0] \leq minMax1[d][1]$ or $minMax1[d][0] \leq minMax2[d][1] \leq minMax1[d][1]$ ) then
4:             return false
5:         end if
6:     end for
7:     return true
8: end function
```

Tasks

5.1 Implement the `SearchKDTree` algorithm (and as part of this, the intersection algorithm).

5.2 Provide pseudocode for, and implement the ”is fully contained within” portion of the `SearchKDTree` algorithm.

5.3 Compare the runtime of the naive implementation with the `SearchKDTree` algorithm both using, and omitting the ”is fully contained within” part of the code. You should evaluate this over different numbers of dimensions, and different numbers of points.

5.4 Describe, and implement, a way to overcome the restriction of each point having a unique coordinate in each dimension.

5.5 Extend your implementation to search for a circular/spherical/hyper-spherical region instead of a rectangular region. Provide pseudocode for your changes, and evaluate its runtime against rectangular regions. Input for this task is as above, except that queries consist of one list and a radius, e.g., `[3 3] 4` denotes a circle centred at (3,3) with radius 4.

5.6 Extend your implementation to search for points within a convex polygonal region of your space, and evaluate its runtime against rectangular and spherical regions. Input for this task is as above, except that queries do not consist of minimum and maximum coordinates, but rather an ordered list of points defining the polygon. For example input query `[3 3] [3 5] [5 5] [5 3]` denotes a square with corner coordinates $(3,3),(3,5),(5,5)$ and $(5,3)$.

Marking

The following table indicates the marks available for each task. In all cases, marks will be assigned for correctness. In addition, marks will be given for novelty, rigour, insight, and clarity for the report portions of the assessment, while good coding practices (e.g., comments) will be required to achieve full marks for implementation.

Task	Expected Deliverable	Marks Available
1.1	implementation, report - pseudocode, description, correctness, complexity	5
1.2	implementation, report - pseudocode, description, correctness, complexity	5
2.1	implementation	8
2.2	report - complexity	4
2.3	report - complexity	4
2.4	report - evaluation	7
3.1	implementation	6
3.2	report - correctness, complexity	4
4.1	report - pseudocode, complexity	4
4.2	report - complexity	5
4.3, 5.1	implementation	14
5.2	report - pseudocode, implementation	5
5.3	report - evaluation	6
5.4	implementation, report - approach	5
5.5	implementation, report - approach, pseudocode, evaluation	9
5.6	implementation, report - approach, evaluation	9

Changelog

14/03/19	Explained KD-tree figure would yield different results
14/03/19	Changed submission date
14/03/19	Explained extensions of files are examples.

References

[1] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.