|  |  |
| --- | --- |
| CSI - 5127 Applied Computational Geometry | Hieu Nguyen - 7234520 |

**Project**

Implementation of FastSplitTree, ComputeWSPD, and t-Spanner Construction from WSPD

horizontal line

### Code Overview

**Data Structures**

PointNode : node in a DoublyLinkedList which represents the point in

* Contains information about the following:
  + Previous and Next node in the DoubleLinkedList
  + Coordinates of the point it represents
  + The dimension it is sorted on in the DoublyLinkedList
  + Pointers (or references) to its counterparts in other DoublyLinkedLists
  + TreeNode pointer for step 4/5 of the partialSplitTree algorithm
* Comparator : Basic implementation of Java’s comparator, on PointNode. Compares 2 PointNodes at a given dimension. (used for sorting)

DoublyLinkedList :

* Contains information about the following:
  + Head and Tail of the list
  + Comparator object to sort the points
  + Dimension size, as well as the dimension that the current list is sorted on.
* Iterator : Basic implementation of Java’s iterator on DoublyLinkedList. Allows for traversal along the list

LS\_Collection : A collection of DoublyLinkedLists,

* Contains information about the following:
  + LinkedList of DoublyLinkedList (ie LSi )
  + Initial point set of size *n*

HyperRectangle :

* Contains information about the following:
  + Rectangle and Bounding Rectangle
  + Hyperplane dimension
  + Hyperplane split point

Pair : Pair of 2 DoublyLinkedList which are 2 sets of pointNodes that are well-separated

Edge : Pair of 2 List<Double> coordinates that make up an edge

SplitTree : Root of the FastSplitTree algorithm, where preprocessing occurs and the recursion is initialized

TreeNode : Nodes of the split tree

* Containing:
  + PointSet
  + Bounding Rectangle
  + PartialSplitTree Algorithm (Steps 1 - 6)

Ball : Hypersphere

* Ritter’s Algorithm

**Ritter’s Algorithm**

Ritter’s algorithm is used to compute the hypersphere which encapsulates the hyper rectangle, to determine if two pairs are well separated.

**Creation of DoublyLinkedLists and setting up cross pointers**

Creating the cross pointers between doublylinkedlists can be confusing when reading through the code. As such we will further illustrate the process here.

(1) The data points are stored into a doubly linked list, and copied over *d* times for each dimension

(2) For each list the cross pointers are set to the node in other doubly linked list at the same index

(3) Now each doubly linked list is sorted on a unique dimension, which gives us *LSi* *,*

Time Complexity : *O(dnlogn)* for the sorting

Now we discuss how to create the copy *CLSi* of *LSi*

(1) For each dimension *i* we iterate through the doubly linked list and for each point node *p* we create a

second point node *p’.* We connect these two point nodes at dimension *i*. Then add *p’* to a doubly linked list in *CLSi* at *i*

(2) For each dimension *i* in *CLSi*, and for each point node *p* we traverse the pointer to point node *q* in *LSi*

by the connection we created in step (1). For each cross pointer of *q* within *LSi* we traverse them to point node *q’* in *LSi*. Then from *q’* we traverse back to *CLSj* to *p’.* Now we connect *p* and *p’*

Time Complexity : *O(dn)*

**Numerical Imprecision, and rounding Errors**

**Java Double Types (64-bit)**

Java’s double type offers 16 decimal point precision, therefore any rounding will occur at that precision.

**Point Query**

Though Java double types are 64-bit and therefore offer 16 points precision, point queries can still suffer from rounding in extreme cases. This problem is non-trivial and so isn’t really handled. If the input data has close to 16 point precision, rounding may have an effect if points exists on the vertex of the hyper rectangle.

**Duplicate Points**

Duplicate points are not handled in this program. So assume that all input sets have no duplicates.