**COMP3210/6210: Report for Assignment 2**

1. **Group Member Information**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Student ID** | **Email:** | **Assigned Task** |
| Coordinator: Sia Kaur Tervinder Singh Duggal | 48390321 | siakaurtervindersingh.duggal@students.mq.edu.au | Report |
| Abiola Ishola Olatunji |  |  | Task 2 |
| Gursant Singh |  |  | Task 1 |

1. **Program Execution Requirements**
   1. **Program Environment (e.g., OS, database, CPU, etc.)**

The requirements for the types of search:

* CPU
* Python supporting environment(Notepad++ , Anaconda Navigator)
* Existing database (Example: shop.txt , city1,txt)
* Visual Studio (download respective libraries of python to run the code)
  1. **Input Files and Parameters (directory settings and other parameters)**

|  |  |  |
| --- | --- | --- |
| **Input Files** | **Output files** | **Process Files(src folder)** |
| * Data(folder) * City1.txt * Parking\_dataset.txt * Query\_points.txt | * Output\_files(folder) * Task\_2\_bbs.txt * Task\_2\_bbs\_divide\_and\_conquer.txt * Task\_2\_sequential.txt * Task1\_bf\_search\_divide\_conquer.txt * Task1\_BF\_search.txt * Task1\_sequential.txt | * Utilities.py * Tree.py * Sequential\_search.py * Bbs\_search.py * Bf\_search.py * Main.py * Report.docx * Ppt.pptx |

* 1. **Additional Requirements**

1. **Program Documentation**
   1. **Program Organization**

If your assignment consists of multiple files and/or classes, please provide brief, high-level descriptions of each file/class within your program, as illustrated below.

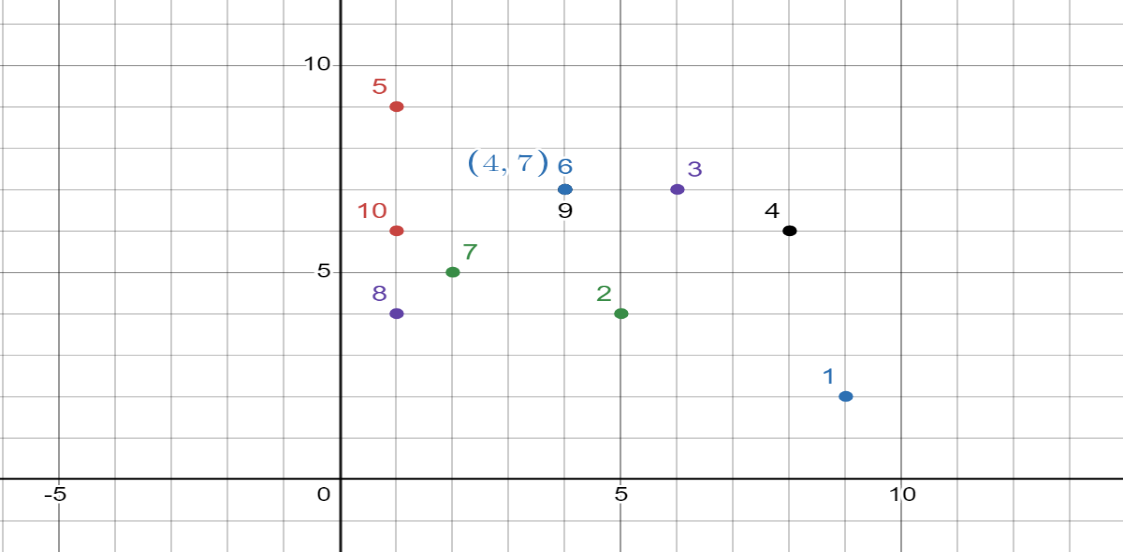
|  |  |
| --- | --- |
| **Class/File Name** | **Description (detailed information)** |
| **Utilities.py** | This python file has the basic function load\_data() , write\_to\_file() and get\_distance\_km() . These function are responsible for accessing the data and splitting it , writing in respective file, calculating the Euclidean distance between 2 points respectively. |
| **Tree.py** | This python file is responsible for creating a basic tree structure for the different kinds of search. It uses the node class to initialize the attributes of the basic node. Theses both classes have multiple methods that are explained below |
| **Sequential\_search.py** | This python file performs the sequential search on the given dataset. It consists of several function like sequential\_task\_two() , dominates\_meters(), dominates\_cords()and revalidate\_skyline(). |
| **Main.py** | This python file is responsible for calling all the methods from other file in main class like utilities.py , bf\_search.py , sequencial\_search.py, tree.py , bbs\_search.py |
| **Bf\_search.py** | This python file is responsible for the first task of finding the nearest neighbour of the query\_points.txt dataset using the best first algorithm. We are using the parking\_dataset.txt as our source dataset to search through. It also uses the approach of divide & conquer in order to give an efficient and quicker result. |
| **Bbs\_search.py** | This python file is responsible for conducting the skyline search on the selected dataset. It also uses the approach of divide & conquer in order to give an efficient and quicker result. |

* 1. **Function Description**

|  |  |  |
| --- | --- | --- |
| **Python File** | **Function Name**  **(parameters)** | **Description (detailed information)** |
| **Sequential\_search.py** | sequential\_task\_two() | This function is responsible for first sorting the data in the list and then finding the skyline points using a nested for loop. It uses a for statement to check the skyline points are correctly selected. The result is stored in ‘output\_files/task\_2\_sequential.txt’ |
| dominates\_meters() | This function is responsible for calculating the distance in kilo meters |
| dominates\_cords() | This function is responsible to calculate the coordinates to check if one point is dominated in both axis by the other point. |
| revalidate\_skyline() | The function reevaluates that all the points in the skyline array are correct or not. It checks if any other point is dominating those points. It uses a nested for loop and two if statements to check the requirements. |
| **Tree.py : Node class** | \_\_init\_\_() | This function is responsible for initialization or the parameters required for rtree construction. We are initializing id , child nodes , data opoints , parents and the MBR |
| perimeter() | This function is used to calculate the perimeter of the MBR using the coordinates ( x1, x2, y1, y2). |
| is\_overflow() | The functions is responsible to check the data in nodes is not over flowing. We use a nested if loop to check the following. The overflow is checked on the condition of leaf node or not. |
| Is\_leaf() | The function is calculating the leaf nodes of the r-tree using a f condition to check. |
| is\_root() | The is\_root is responsible to calculate the root of R-tree by checking if the node has any parent or not. |
| **Tree.py : R-tree class** | \_\_init\_\_() | This is responsible to creat a base node for the rtree structure by accessing the root node we calculated above in the function. |
| Insert() | This function is responsible for inserting the datapoints in node to calculate their position in r-tree structure on the bases of their MBR. The function uses if loop and checks mehods to determine like is\_leaf , is\_overflow. This then creates a subtree is in the else statement to move to the next node. |
| Choose\_subtree | This method is used by the insert function to determine the position at which the node must be inserted. This method uses an if- else statement to determine the suitable child where our node can be inserted. |
| **Tree.py : R-tree class** | Peri\_increase() | The function is responsible for calculating the perimeter increase from the initial point to the current point. The calculation is done using the original MBR that we initialized in the node class an the current coordinates. |
| Handle\_overflow() | This function is responsible to handle situations when a node datapoints in the r-tree become overloaded. They exceed the predefined capacity. We are calling the split function to divide the data point into two new nodes. The conditions are on the bases of is a root node or a non-root node. |
| Split() | This function is responsible for splitting an overloaded node into two new nodes. The goal is to create two new nodes with a split that has minimize overall perimeter. The splitting can be done on the bases od x-coordinates or y-coordinates. We divide them by using 4 different kinds of divides. |
| Add\_child() | The function takes the nodes and assigns a child node to the current parent node. It uses the if statement to check for the conditions suitable of the addition of the node. |
| Add\_datapoints() | This function is similar to the above function but instead of nodes it is responsible to add the datapoints to the node and update it. |
| Update\_mbr() | This is responsible for updating the initial MBR with the new MBR coordinates. It is used to check the number combination that are formed and also to update the root node for the balancing. |
| **Bbs\_search.py** | Bbs\_search() | This function is responsible to conduct a recursive depth first search on the r-tree we constructed to find the skyline points. We use a heap to store the points and sort them according to their distance. In the while loop we deque the point with the minimum distance. Then we check if it is dominated by any point using a nested if statement. The loop checks differently for leaf nodes and non-leaf nodes. Returns the skyline points. |
| **Bbs\_search.py** | Bbs\_search\_divide\_conquer() | This function implements the method of divide & conquer in the bbs\_search algorithm in order to make it more efficient in searching the skyline points. The main logic is to divide the search space into two r-tree and searching through the. After the search we have to merge the skyline array to gives us the final skyline points. |
| Bbs\_search\_async\_caller() | This asynchronous function is a helper for the divide and conquer approach to simply schedule the execution within thread pool executor to return the results. |
| Is\_dominated() | This function is responsible for checking the final array has all the skyline points or not. It checks for an point that is dominated by another point it returns true else it gives a false result. |
| **Bf\_search.py** | heuristic() | This function is responsible for calculating the Euclidian distance to the MBR nodes we have constructed and return the final calculation. |
| Bf\_search\_divide\_conquer() | This function is responsible for accessing the two R-trees we have constructed and apply the best first algorithm to them to find the nearest neighbour. The final distance is compared in the if statement to get the nearest value. |
| Bf\_search() | This function is responsible for accessing the R-trees we have constructed and search the nodes using priority queue. It calculates the Euclidian distance when it reaches a leaf node or it calculates the heuristics when it reaches a non-leaf node. |
| **Utilities.py** | Load\_file() | This function is responsible for accessing the selected dataset and loading the datapoints from the file. The process uses an empty list where points will be stored. The split function is to gather the datapoints. In our case id , x , y is accessed using the method. |
| Save\_file\_loaction() | This function is responsible for storing the path of the file , the program information and comments to a file. It uses two empty strings to store the data. Used to read the data written and find the runtime for the program |
| Write\_to\_file() | This function uses a loop to write the data for each point. It uses the simple open statement to write in the file given to us |
| Revalidate\_skyline() | This function us responsible to removing any datapoints from the skyline array that are dominated by another point. It uses an iterative search for each point and validates the is\_dominated Boolean variable. |
| Dominates\_cord | This function is used by the above function to check if any point is dominated by another. |

1. **Analyzing BF Algorithm based NN Search**

We tried to find the nearest neighbor search using the Best First algorithm. The process is based on the index of the query points and divided into clusters. The clusters are aggregated together to a hierarchical structure called R-tree. The task min agenda is to find the nearest location of either restaurant, shops and parking for a specific query point. The main algorithm we have tried to use is best-first algorithm, which requires a r-tree construction as its initial step.

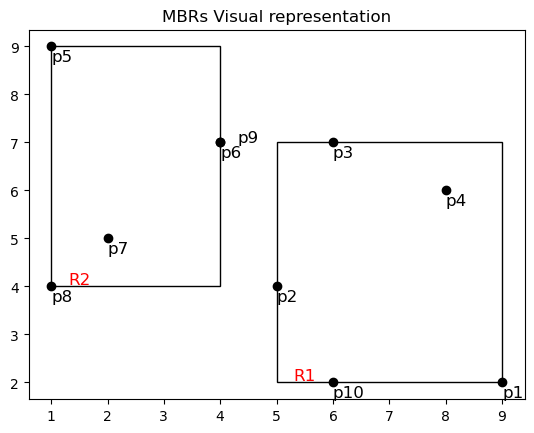


The graph is a pictorial representation of a Sample dataset

* 1. **The Process of R-Tree Construction**

The R-tree construction plays a vital role in this algorithm. It is a hierarchical structure that gives us a 2-dimentional representation of the dataset. The tree structure is designed with a regression format of the distance between the query point and each point. It helps use to navigate through the spatial data. The process to create an R-tree is as follows:

* We start by gathering the data points with locations (Example: shop\_dataset.txt, restaurants\_dataset.txt, parking\_dataset.txt)
* We then create a minimum bounding rectangle around each data point. The minimum bound rectangle is the smallest rectangle one which covers all the points.
* Group MBRS into larger non-overlapping MBRS. Continue this grouping process until we find a root node.
* During the grouping process we can select a splitting strategy in order to conclude the maximum number of children each node can have.



* We start by creating root node method which helps us to create the nodes of the tree.

We can create a node by defining it property:

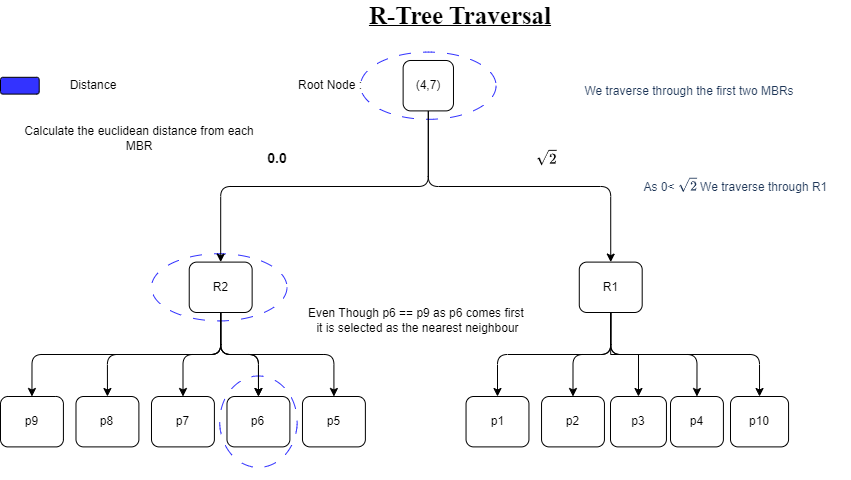
Self.id: it used to determine the id of the node

We can create two lists one can store node values

Where as the other can store datapoints

We have to create a self.mbr. It is intially the value of the current coordinated of the rectangle to the points.

* We must create a method to construct the R-tree. For example: add\_data\_points() , update\_data\_point(), insert\_data\_points() in order to construct a R-tree by adding each node to a R-tree structure



**The Process of BF Algorithm**

The process of Best First algorithm is searching for the optimal point in the through the R-tree. We can get started by creating a priority queue. The queue contains all the points that stand in the minimum distance from the query point and the nodes of MBR.

We add the root node of the MBR to the priority queue.

We go through the queue to dequeue the query points with highest distance and shortlist the points closest.

If the node is a leaf node

* We must calculate the distance using Euclidean distance formula.

For the sample dataset we get this kind of calculation:

7.0710678118654755 between (9, 2) and (4, 7)

3.1622776601683795 between (5, 4) and (4, 7)

0.0 between (6, 7) and (4, 7)

4.123105625617661 between (8, 6) and (4, 7)

3.605551275463989 between (1, 9) and (4, 7)

between (4, 7) and (4, 7)

2.8284271247461903 between (2, 5) and (4, 7)

4.242640687119285 between (1, 4) and (4, 7)

0.0 between (4, 7) and (4, 7)

5.385164807134504 between (6, 2) and (4, 7)

* We add these in the priority list and then accordingly sort them
* Then find the point with the minimum distance and identify it as the nearest neighbor

If it is an internal node

For each childe MBR code

* We must calculate the minimum distance from the query point and the child
* Add the childe to updated priority queue with the new distance .

After checking through the priority list we can determine the nearest neighbour without visiting the complete data nodes like Branch and Bound Algorithm.

The results we get for the tests run on the sample space are :

Facility ID: 6, Distance: 0.0

Nearest neighbor found!

The time taken for the complete search is using Best First Search: 0.11418008804321289 seconds

* 1. **The Process of Divide-and-Conquer**

Divide and conquer is a very ancient concept that is quite effective and reliable. It helps to give faster results for the Best first nearest neighbor algorithm.

* As the name suggests we must start by dividing the dataset into two subsets based on their dimension (in our case x, y).
* We are required to built a r-tree for each subset.
* Then we can perform the Best first search on each tree using out query point dataset. This provides us a much smaller candidate queue.

The results using Divide and conquer:

Facility ID: 6, Distance: 0.0

Nearest neighbor found in second subset!

The time taken for the complete search is Best First Search with Divide-and-Conquer: 0.02745532989501953 seconds

* After that we can do the final comparison of the distance from the query points in order to get the nearest neighbor.

For the subset given to us we have applied both logics and here are the results:

The time taken for the complete search is using Best First Search: 0.11418008804321289 seconds

The time taken for the complete search is Best First Search with Divide-and-Conquer: 0.02745532989501953 seconds

We can conclude from this even though Divide and Conquer is a very complicated algorithm but when applied correctly it helps to achieve adequate and lucrative results. The time has been decreased by 4 times which makes it very dependable algorithm

Some Insights:

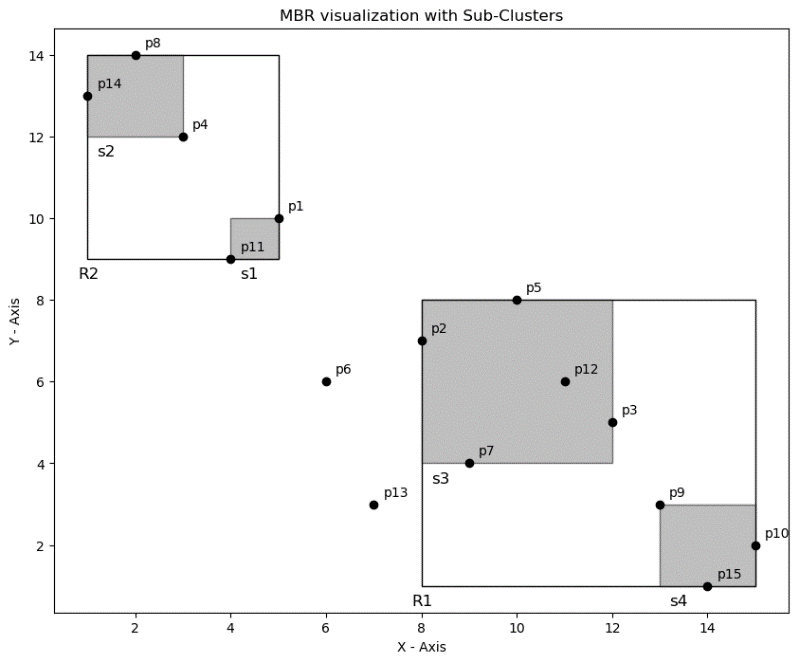
While working on the dataset we noticed there was an overlap between two points. You can refer to the point graph above where you can notice the point 6 and 9 have the same coordinated (4,7). As (4,7) is the point closest to the query\_point how can we conclude which one do we select?

The use of priority list helps us decide that, It selected the point which was inserted first in the list therefore point 6 was selected as the nearest neighbor in this case.

1. **Analyzing the BBS Algorithm based Skyline Search**

Skyline search is a dataset technique that is used to analyses a complete dataset without checking each point. The technique uses the dominates rule which is that if Point A is dominating Point B by comparing it coordinates (xa>xb or ya > yb). We can conclude that point A is clearly dominating point B. If Point is not dominated by any other point, it can be selected as a Skyline point. These Skyline points provide a more concise and accurate dataset of solution for the query.

This algorithm is mainly focused on getting an efficient set of skyline points that can be further used in the problem-solving queries to make decisions. This algorithm can be used in several test-case solutions for data visualization, spatial database analyses, allocating problems, etc.



* 1. **The Process of R-Tree Construction**

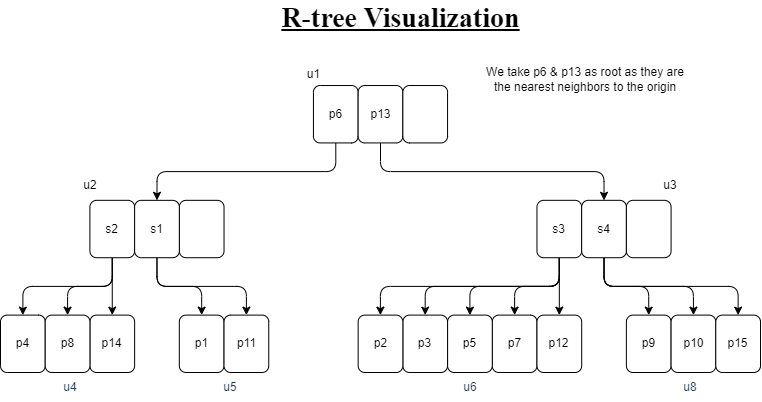
The construction of R-tree is the base of this searching algorithm. The R-tree can

created from scratch using a base node class and a r-tree class. The steps are as follows:

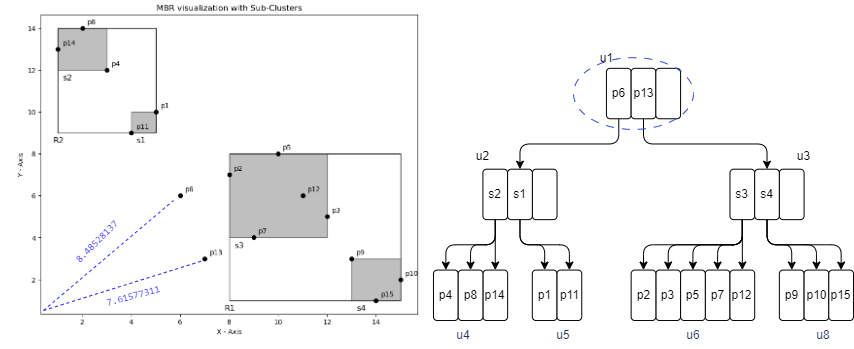
Step 1: Construct a base node. Initialize all the parameters in your dataset. In my case the sample dataset contains three columns id, x and y. I have also initialized a parameter to find dominate points.

Step 2: Construct a Rtree class is used add characteristics to the structure. The \_\_init\_\_ function is to read the nodes. The insert function is to add the nodes to the R- tree.

Step 3: We insert a method to implement the BBS algorithm to find the skyline points.



* 1. **The Process of BBS Algorithm**
* The search is implemented by initializing an empty skyline array.
* We use two for loops to iterate through each node of the R-tree.
* We can compare both the points using the if loop with the condition: (p.x <= q.x and p.y < q.y) or (p.x < q.x and p.y <= q.y). This condition compares the coordinates to find the dominating point.
* We add the q point to out dominates variable
* If by the end of the inner for loop if the point is not dominated by any point, we can conclude it as a skyline point and add it into the skyline array.



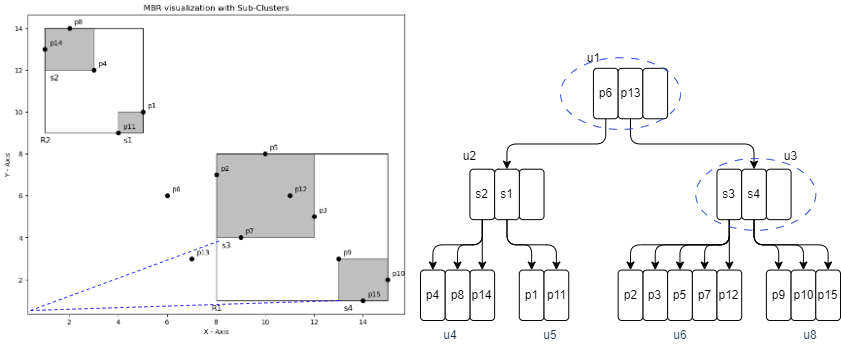
Priority List ={[p13, 7.61] ,[ p6 , 8.48]}

As point 6 and 13 are not dominated by any points

Skyline points ={[p13, 7.61] ,[ p6 , 8.48]}

Priority List = { [p6, 8.48] , [s3 , 8.94], [s4, 13.34] }

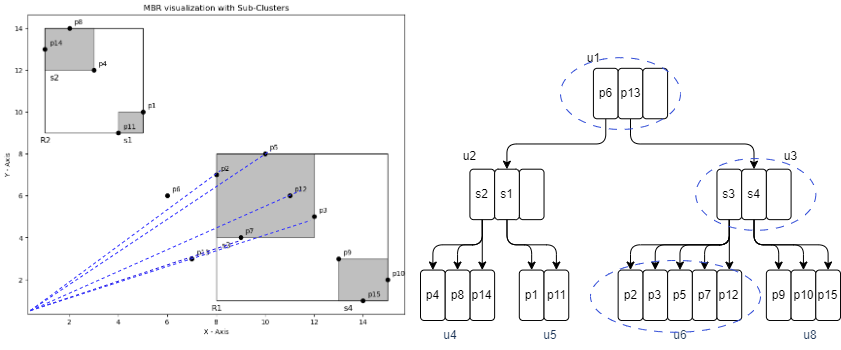
Priority List = { [s3 , 8.94] ,[s4, 13.34] , [s1, 13.45] ,[ s2, 17.69]}



Priority List={ [p7, 9.48] , [p2 , 10.63] , [p12 , 12.526] , [p5, 12.80] , [p3, 13.00] , [s4, 13.34 , [s1, 13.45] , [ s2, 17.69]}

P7 , p2 , p5, p12, p3 is completely dominated by p6 and p13 therefore not a skyline point.

Skyline points ={[p13, 7.61] ,[ p6 , 8.48]}



Priority List={ [p9,13.34] , [s1, 13.45] , [p15, 14.03 ], [ p10, 15,13 ] , [ s2, 17.69]}

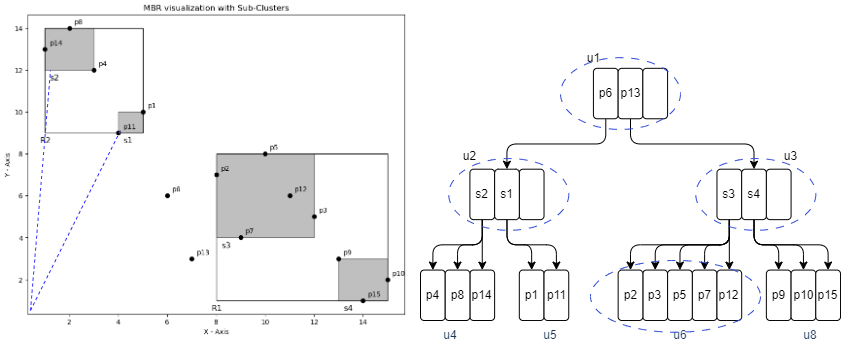
P9 is dominated by p15

Priority List= {[p11, 9.84] ,[p1, 11.18] , [p15, 14.03 ], [ p10, 15,13 ] , [ s2, 17.69]}

P1 is dominated p11 and p10 is also dominated by p15

Therefore p11 and p15 are skyline points

Skyline points ={[p13, 7.61] ,[ p6 , 8.48], [p11, 9.84] , [p15, 14.03 ]}



Priority list = { [p4 , 12.36 ] , [p8 , 14.14] ,[ p14 , 13.03]}

Both p4 and p14 are skyline points as they dominate point p8

**Final skyline points= { [p13, 7.61] , [ p6 , 8.48] , [p11, 9.84] , [p4 , 12.36 ] ,[p15, 14.03 ] , [p8 , 14.14] }**

Results For The Sample Dataset:

Skyline Points

ID: 4, X: 3, Y: 12

ID: 6, X: 6, Y: 6

ID: 11, X: 4, Y: 9

ID: 13, X: 7, Y: 3

ID: 14, X: 1, Y: 13

ID: 15, X: 14, Y: 1

Time taken to perform the BBS skyline search: 0.07417465209960938 seconds

Skyline point with ID: 6, X: 6, Y: 6 is dominated by another point.

* 1. **The Process of Divide-and-Conquer**

The divide and conquer approach is to get more efficient answers in lesser time though the merging of the two R-trees can be a difficult task.

* The divide and conquer method help us to divide the nodes evenly
* The nodes are then accessed into a merge skyline method that uses several conditional statements to compare the left array with right array points.
* The comparison is used to find the skyline point easily.
* We then merge the point dominating in the merger\_skyline array

Result using Divide & Conquer:

Skyline Points

ID: 14, X: 1, Y: 13

ID: 4, X: 3, Y: 12

ID: 11, X: 4, Y: 9

ID: 6, X: 6, Y: 6

ID: 13, X: 7, Y: 3

ID: 15, X: 14, Y: 1

Time taken to perform the BBS skyline search using Divide & Conquer: 0.06579780578613281 seconds

Skyline point with ID: 14, X: 1, Y: 13 is dominated by another point.

As observed you can see that the taken has reduced making the algorithm very helpful in cases of a larger dataset.