**COMP6210**

**Big Data**

****

**Report for** **Assignment 2**

**Submitted by:**

|  |  |  |
| --- | --- | --- |
| Name | Student ID | Email |
| Anuj Adhikari | 48547743 | anuj.adhikari@students.mq.edu.au |
| Bharat Poudel | 48687448 | bharat.poudel@students.mq.edu.au |
| Gopal Datt Bhatt | 48489131 | gopaldatt.bhatt@students.mq.edu.au |

**1. Group Member Information**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Student ID** | **Email:** | **Assigned Task** |
| Anuj Adhikari | 48547743 | anuj.adhikari@students.mq.edu.au | Task 2 |
| Gopal Datt Bhatt | 48489131 | gopaldatt.bhatt@students.mq.edu.au | Task 1 |
| Bharat Poudel | 48687448 | Bharat.poudel@students.mq.edu.au | Project Report |

**2. Program Execution Requirements**

**2.1 Program Environment (e.g., OS, database, CPU, etc.)**

1. **Operating System Compatibility**:

The program is designed to run on **Windows, macOS** and platforms that support **Python version 3.6 or higher**. Task 2 was developed and tested on **macOS using Spyder IDE**.

1. **Python Version**:

The code requires **Python 3.6 or newer** so that it can make use of built-in libraries available in the python version and above.

1. **Required Libraries**:

**TASK 1: Nearest Neighbor Search and Task 2: Skyline Search**

For both of the task, the script uses **time,math, sys, os, and heapq**, which are pre-installed standard libraries in python. There are no external dependencies that require manual installations for the project.

1. **Hardware Recommendations**:

Since the dataset is very large, the hardware requirement for smooth performance are as follows:

* **4 GB of RAM and above**
* **Any Dual-core processor and above.**

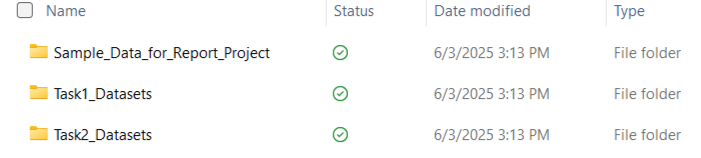
1. **Development Environment**:

**TASK 1: Nearest Neighbor Search and Task 2: Skyline Search**

Both of the Task 1 and Task 2 were developed in different IDE/environment. Task 1 was developed in **VS code** while Task 2 in **Spyder IDE** which means both tasks were extensively tested and debugged in these respective IDE, but the program executes flawlessly in other IDEs such as **PyCharm, or command-line** etc. which supports python**.**

**2.2 Input Files and Parameters (directory settings and other parameters)**

To ensure correct execution, the project requires specific input files organized within a structured directory. Essential parameters should also be set within the code for accurate processing. Below is an overview of the directory structure for both Task 1 and Task 2, followed by a screenshot illustrating this setup.

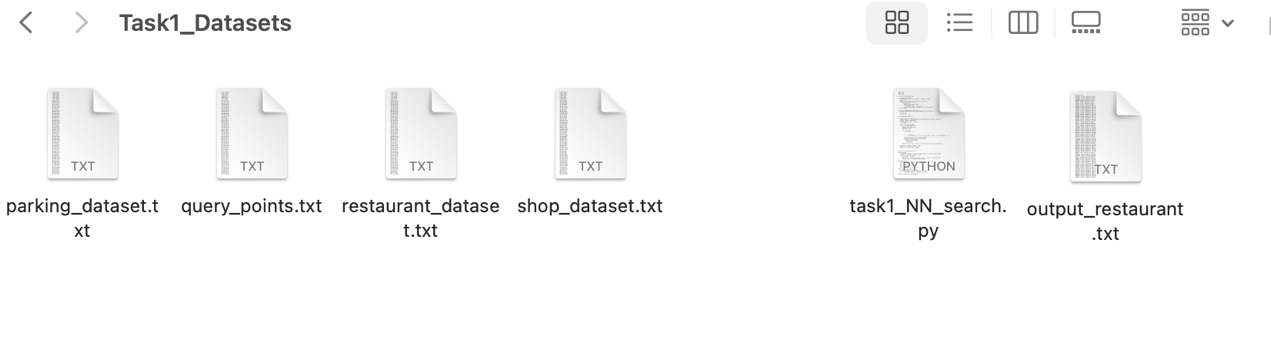
**Figure 1: Screenshot of Main Directory Structure of Tasks**

To ensure correct execution, both Task 1 and Task 2 rely on specific input files located within a structured directory setup. Below are details on the directory structure for each task, along with screenshots for easy reference.

**Task 1: Nearest Neighbor Search**

For Task 1, the input files, output file, and code file are in the following structure. This setup includes the dataset, query file, and the necessary Python script to execute the nearest neighbor search algorithms.

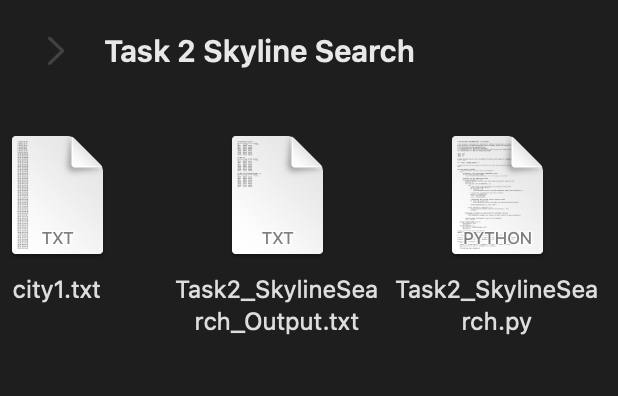
* Query\_points.txt contains the user location in the format [ID, longitude (x), and latitude (y)].
* Parking\_dataset.txt records parking lot locations in the format [ID, longitude (x), and latitude (y)].
* Restaurant\_dataset.txt Contains restaurant locations in 2D format in the format [ID, longitude (x), and latitude (y)].
* Shop\_dataset.txt stores shop locations in the format [ID, longitude (x), and latitude (y)].
* task1\_NN\_search.py contains the code to implement NN search algorithm using all three methods.
* output\_restaurant.txt file contains the output queries for three methods with total and average execution time.



**Task 2: Skyline Search**

For Task 2, the input file, output file, and code file are in the following structure. This setup includes the dataset and the necessary Python script to execute the skyline search algorithms. The screenshot below provides a visual representation of the directory structure for Task 2.

* Input file for city datasets is in city1.txt.
* Output files are in Task2\_SkylineSearch\_Output.txt.
* Python code is located in Task2\_SkylineSearch.py.



**Figure 3: Directory Structure for Task 2 - Skyline Search**

**Task 3: Project Report**

Project Report Compilation Task 3 involves compiling the final project report, which includes descriptions, diagrams, requirements, and an analysis of results for both Task 1 and Task 2.

**2.3 Additional Requirements**

**1. Input File Format Compliance**

All dataset and query files for both the Nearest Neighbor Search (Task 1) and Skyline Search (Task 2) must strictly follow the specified formats. Particular attention should be given to Task 1 query files to ensure attribute alignment with the main dataset, avoiding mismatches during search operations.

**2. Output Storage Management**

Designated output folders (Task1outputs and Task2outputs) must be created with proper write permissions. This ensures smooth saving of results and prevents file write errors caused by access restrictions.

**3. Robust Error Monitoring**

Integrated error handling captures common issues like file access problems or invalid input formats. A debug mode is also available to track program execution step-by-step, making troubleshooting and performance analysis more efficient.

**4. Optimized Resource Utilization**

To prevent memory overuse on low-resource systems, batch processing of queries is advised. Smaller processing loads per cycle ensure better CPU and memory management without compromising accuracy.

**5. Pre-Execution Quality Checks**

Before running the full system, trial runs were conducted using reduced-size datasets. This validated directory setup, data integrity, and algorithm behavior—especially for Skyline Search—ensuring consistent and accurate outputs under test conditions.

**3. Program Documentation**

**3.1 Program Organization**

The brief description of files used in assignments is given below in the table:

|  |  |
| --- | --- |
| **File Name** | **Description (detailed information)** |
| **Task 1** | Nearest Neighbor Search: Identifies the closest data points to a set of query points using three methods: Sequential Scan, BF Algorithm (Best First with R tree), and Divide-and-Conquer. Each method is structured to process and output results for nearest neighbor searches. |
| restaurant\_dataset.txt | Task 1 is performed using the restaurant dataset which contains the location of restaurants in the format [ID, longitude (x), and latitude (y)]. |
| task1\_NN\_search.py | This is the main program file containing functions, class and main program which implements NN search algorithm using **sequential scan, BF search using custom rtree and BF search with Divide-and-Conquer.** |
| output\_restaurant.txt | This is the output file containing 200 output queries for each method which shows the nearest restaurant for different query points. The file gives ID and location coordinates for each query along with the total and average time of execution for each method. |
| **Task 2** | Skyline Search: Determines optimal points (skyline points) from a dataset based on criteria such as minimizing cost and maximizing size, using three methods: Sequential Scan, BBS Algorithm (Branch and Bound with R-tree), and Divide-and-Conquer. |
| **Task2\_SkylineSearch.py** | Contains the main functions for Task 2. Implements Sequential Scan, BBS Algorithm with R-tree, and Divide-and-Conquer approaches to identify skyline points in the dataset. |
| **city1.txt** | Dataset containing city points with attributes like cost and size, used as the input data for Task 2 skyline calculations. |
| **Task2\_SkylineSearch\_Output.txt.** | Output file that stores results from each skyline search algorithm (Sequential Scan, BBS, and Divide-and-Conquer). It includes skyline points’ IDs, attributes, and execution times for performance comparison. |
| **Task 3** | Project Report Compilation: Compiles the final report with descriptions, diagrams, requirements, and result analysis for Tasks 1 and 2. |
| **report.docx/pdf** | Document file for the final project report, stored in the main directory. Includes a comprehensive overview of Tasks 1 and 2 with all relevant findings and performance comparisons. |

**3.2 Function Description**

**For the TASK1, the function and class table is below:**

| **Name & Parameters** | **Type** | **Description** |
| --- | --- | --- |
| read\_points(file\_path) | Function | Reads a file containing spatial points. Each line is parsed into a tuple (id, x, y) and returned as a list. Skips lines with missing or malformed data. |
| euclidean\_distance\_sq(x1, y1, x2, y2) | Function | Computes **squared Euclidean distance** between two 2D points. Square root is omitted for speed since it's not needed for comparison. |
| sequential\_scan(dataset, queries) | Function | Performs brute-force nearest neighbor search: For each query point, it scans all dataset points to find the closest one based on Euclidean distance. |
| RTreeNode(is\_leaf) | Class | Represents a **node in the R-tree**. Contains entries (points or children) and an MBR (minimum bounding rectangle). Uses \_\_slots\_\_ for memory efficiency. |
| get\_mbr(points) | Function | Computes the **Minimum Bounding Rectangle** for a list of points: returns (min\_x, min\_y, max\_x, max\_y). |
| enlarge\_mbr(mbr1, mbr2) | Function | Merges two MBRs and returns the **smallest rectangle** that contains both. Used to expand node MBRs during tree building. |
| build\_rtree(points, max\_entries=12) | Function | Constructs an **R-tree spatial index** from raw data points. Groups points into leaf nodes, then builds upper layers using build\_level. |
| build\_level(entries, is\_leaf) | Nested Function | Groups entries into fixed-size nodes for the R-tree. Computes and assigns MBRs to each node. Used recursively to build tree layers. |
| mbr\_distance\_sq(mbr, x, y) | Function | Computes the **squared distance** from a point (x, y) to the closest point within an MBR. Returns 0 if the point is inside the MBR. |
| best\_first\_search\_rtree(rtree, query) | Function | Performs **nearest neighbor search** using **best-first traversal** of the R-tree. Uses a priority queue (min-heap) to explore the closest MBRs first and prune distant branches. |
| bf\_rtree\_search(dataset, queries) | Function | Wrapper that builds an R-tree from the dataset and performs **best-first nearest neighbor search** for all query points. Returns results and timing. |
| bf\_divide\_and\_conquer(dataset, queries) | Function | Optimized search: splits dataset into left/right halves based on x-coordinate, builds two R-trees, and compares both results for each query. |
| write\_output(all\_results, output\_path) | Function | Writes all algorithm results to a text file. Includes query results, total time, and average query time for each method. |
| if \_\_name\_\_ == "\_\_main\_\_" | Script Entry | Main script logic: prompts user for dataset, loads data, runs all algorithms (sequential, R-tree, divide-and-conquer), and saves results. |

**For TASK 2: Skyline Search**

|  |  |
| --- | --- |
| Function Name (parameters) | Description (detailed information) |
|  |  |
| load\_dataset(filename) | Loads , validates dataset from text file and parses each line into (ID, cost, size) tuples while validating data types and finally returns dataset as list of tuples with inclusion of error handling. |
| is\_skyline\_point(point, skyline) | Checks to see if a point belongs in skyline by testing domination against other skyline points. The logic where point p dominates q if p has lower/equal cost AND higher/equal size is also implemented. It also iterates through current skyline for non-domination check. The result is false if point is dominated else true. |
| skyline\_sequential(dataset) | Implements O(n²) sequential scan algorithm where every point is compared against other using nested loops and returns a list of non-dominated points. |
| Node.\_\_init\_\_(self, entries, is\_leaf=True) | This is the constructor for R-tree node class. It stores entries which are data points and also sets node type flag. This calculates MBR. |
| Node.compute\_mbr(self) | Calculates Minimum Bounding Rectangle (MBR) by using spatial boundaries of entries. The cost and size values are taken from provided data points for leaf nodes or combines child MBRs for internal nodes. It returns (min\_cost, max\_cost, min\_size, max\_size). |
| Node.mindist(self) | Calculates minimum distance heuristic for priority queue ordering by performing sum on minimum cost and size whole also estimating the likelihood of containing skyline points where lower values indicate higher processing priority. It is used by heapq for node expansion order in BFS. |
| Node.\_\_lt\_\_(self, other) | Enables Node comparison for heapq operations by implementing less-than operator. It compares mindist() values for priority ordering. It makes the automatic processing of nodes in increasing mindist order for optimal search possible. |
| bulk\_load(dataset, max\_entries=100) | Constructs balanced R-tree where bulk loading method is used. This function also sorts dataset by cost for spatial locality and chunks into leaf nodes. It recursively builds tree levels bottom to up by grouping nodes until a single root remains. |
| mbr\_dominated(mbr, skyline) | Determines if entire MBR can be skipped by checking skyline point domination and verifies if any skyline point has cost ≤ MBR minimum cost AND size ≥ MBR maximum size. It returns true if region can be skipped. |
| skyline\_bbs(dataset) | Implements Branch and Bound Skyline algorithm using R-tree spatial indexing. It builds R-tree, initializes priority queue, and performs best-first traversal with mindist approach. It returns skyline points. |
| skyline\_bbs\_divide\_and\_conquer(dataset) | Implements divide-and-conquer BBS variant splitting dataset into subspaces for separate processing. Sorting via cost is done and splitting is done at median into leftand right halfs. It applies BBS to each half and merges with dominance. |
| run\_all\_algorithms() | It executes skyline algorithms with evaluation metrics and error handling. It measures the execution times also has try-catch blocks. |
| write\_results\_to\_file(results, filename="Task2\_SkylineSearch\_Output.txt") | The main result of the algorithms implemented are written into a file named Task2\_SkylineSearch\_Output I. text format. |

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

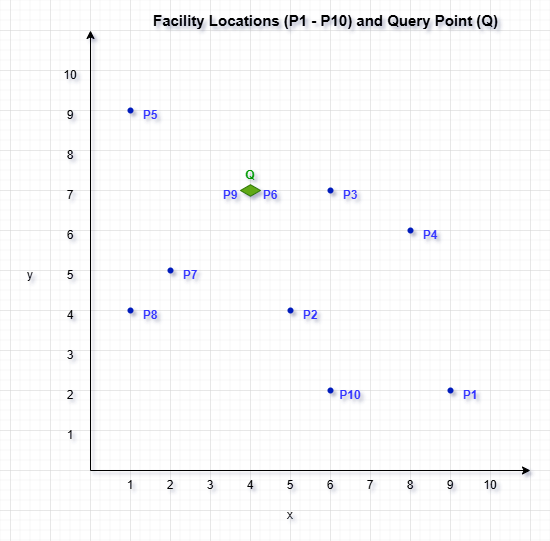
The objective of Task 1 is to identify the nearest neighbor to a query point (4, 7) from a given set of facility location points using an efficient spatial data structure, the R-tree, and the Best-First (BF) algorithm.

Here’s the table for the **Facility Location** dataset, including the points and query points.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| X | 9 | 5 | 6 | 8 | 1 | 4 | 2 | 1 | 4 | 6 |
| Y | 2 | 4 | 7 | 6 | 9 | 7 | 5 | 4 | 7 | 2 |

|  |  |  |
| --- | --- | --- |
| Query Point |  |  |
| ID | **X** | **Y** |
| 1 | 4 | 7 |

This table is organized to list each facility location with its ID, X, and Y coordinates. The query point is listed separately to clearly indicate it as the point of reference for the nearest neighbor search.



**Figure 4 : Visualization of Facility Locations and Query Point (Q) on an X-Y Grid**

The figure shows the spatial distribution of facility locations (P1 to P10) and the query point (Q) on an X-Y coordinate grid. Each facility location is represented by a blue dot labeled with its unique ID, while the query point (Q) is highlighted in red for easy identification. The X and Y axes range from 1 to 10, providing a clear layout of each point’s position on the grid. This visual layout is the starting point for constructing the R-tree, as it illustrates the spatial relationships between points, which will inform the grouping of points into Minimum Bounding Rectangles (MBRs) for efficient nearest neighbor search.

* 1. **R-Tree Construction**

**Step by Step R-Tree Construction**

**Step 1: Insert Point P1**

- This is the first point being inserted into an empty R-tree.

- P1 is inserted into a newly created leaf node u1.

**MBR (Minimum Bounding Rectangle):**

- Since P1 is the only point, the MBR becomes a degenerate rectangle at its coordinates:

Xmin = Xmax = 9

Ymin = Ymax = 2

- The MBR simply wraps the point P1 with no expansion needed.

A graph with numbers and points

AI-generated content may be incorrect.

**Figure 5 :Initial Insertion of P1 into the R-Tree**

- u1 acts as both the **root and a leaf node** at this stage.

**Step 2: Insert Point P2**

- Point P2 (5, 4) is inserted into the existing leaf node u1, which already contains P1 (9, 2).

- Since u1 still has capacity (fan-out > 2), no split is required.

**Updated MBR for Node u1:**

- After adding P2, the MBR must now cover both P1 and P2:

Xmin = 5 (from P2)

Xmax = 9 (from P1)

Ymin = 2 (from P1)

Ymax = 4 (from P2)

- New MBR spans from **(5, 2)** to **(9, 4)**

**A graph with a rectangle and points

AI-generated content may be incorrect.**

**Figure 6 : Insertion of P2 and MBR**

- u1 remains the root and a leaf node.

- Still under capacity (2 entries).

**Step 3: Insert Point P3**

- Point P3 (6, 7) is inserted into the existing leaf node u1, which currently holds P1 and P2.

- Since this brings the total entries to 3 (equal to the fan-out limit), insertion proceeds without a split.

**Updated MBR for Node u1:**

- After inserting P3, the MBR is recalculated to include all three points:

Xmin = 5 (from P2)

Xmax = 9 (from P1)

Ymin = 2 (from P1)

Ymax = 7 (from P3)

- New MBR spans from (5, 2) to (9, 7)

A graph with a square and a rectangle with red dots

AI-generated content may be incorrect.

**Figure 7 : Insertion of P3 and Updated MBR**

- Node u1 is now at full capacity (3 entries).

- The tree remains balanced and simple at this stage.

**Step 4: Insert Point P4 and Split Node u1**

- Point P4 (8, 6) is inserted into node u1, which already holds P1, P2, P3.

- Since u1 exceeds its fan-out of 3, a split is triggered**.**

**Perform Node Split:**

- To split, we create two new child nodes, u2 and u3 and distribute the points between them to minimize the total MBR perimeter.

To determine the best way to split, we compare the MBR perimeters by sorting on both the X-axis and Y-axis, and then choose the option that minimizes the total perimeter. This approach helps to provide a balanced distribution and minimizes the overall perimeter.

**Process for Each Option:**

**1. Sorting on the X-axis (P2, P3, P4, P1):**

Split as u2 = (P2, P3) and u3 = (P4, P1).

**Calculate the MBR perimeters:**

For u2: Xmin = 5, Xmax = 6, Ymin = 4, Ymax = 7 = Perimeter = 2 × ((6 - 5) + (7 - 4)) = 8

For u3: Xmin = 8, Xmax = 9, Ymin = 2, Ymax = 6 = Perimeter = 2 × ((9 - 8) + (6 - 2)) = 10

**Total Perimeter = 8 + 10 = 18**

**2. Sorting on the Y-axis (P1, P2, P4, P3):**

Split as u2 = (P1, P2) and u3 = (P4, P3).

**Calculate the MBR perimeters:**

For u2: Xmin = 5, Xmax = 9, Ymin = 2, Ymax = 4 = Perimeter = 2 × ((9 - 5) + (4 - 2)) = 12

For u3: Xmin = 6, Xmax = 8, Ymin = 6, Ymax = 7 = Perimeter = 2 × ((8 - 6) + (7 - 6)) = 6

**Total Perimeter = 12 + 6 = 18**

Since both sorting approaches result in the same total perimeter (18), either option could be chosen. However, to maintain a more balanced distribution of points between the two child nodes, sorting along the X-axis would be prefe rable, as it splits the points more evenly. Therefore, X-axis sorting is chosen as the optimal split for creating the two child nodes.

****

**Figure 8 : Insertion of P4 and Split of Root Nod e(X-Axis)**

- u1 is now an internal node

- u2 and u3 are new leaf nodes

**Step 5: Insert Point P5**

- Point P5 (1, 9) is evaluated for insertion.

- Both child nodes (u2 and u3) are not full (each has 2 points).

- We compute the MBR expansion for each to determine optimal placement.

**MBR Expansion Comparison:**

u2 (P2, P3):

- Current MBR: (5, 4) to (6, 7)

- New MBR with P5: (1, 4) to (6, 9)

-Perimeter = 2 × ((6−1) + (9−4)) = 20

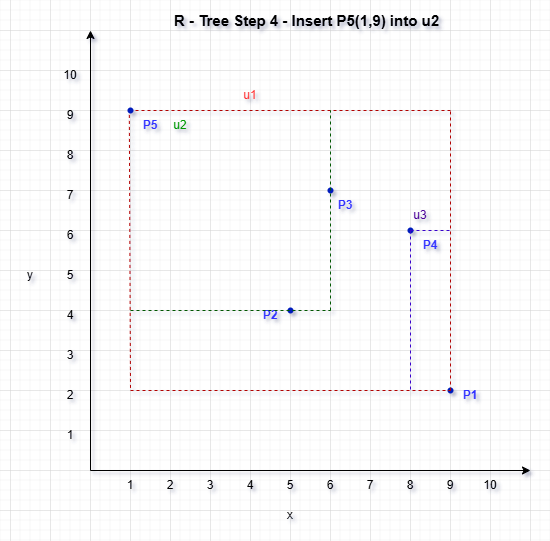
u3 (P4, P1):

- Current MBR: (8, 2) to (9, 6)

- New MBR with P5: (1, 2) to (9, 9)

- Perimeter = 2 × ((9−1) + (9−2)) = 30

**u2 chosen (less expansion)**

****

**Figure 9 : Insertion of P5 and the updated MBR**

- u2 is now full.

- u3 still has 1 slot.

**Updated MBRs:**

- u2 (Green): P2, P3, P5 → MBR: (1, 4) to (6, 9)

- u3 (Purple): P4, P1 → remains unchanged

- u1 (Red): Encloses u2 and u3 → updated to (1, 2) to (9, 9)

**Step:6 Insert P6 (4,7) and Node Split in R-tree :**

R-tree Before Insertion

- Root node: u1

Two child nodes: u2 and u3

u2 contains: P2 (5,4), P3 (6,7), P5 (1,9)

u3 contains: P1 (9,2), P4 (8,6)

**Insertion Strategy**

Since **u2 is already full** (contains 3 points and has a max fan-out of 3), we must check if **u3 can accommodate P6**.

**u3’s MBR Before Insertion:**

Xmin = 8, Xmax = 9

Ymin = 2, Ymax = 6

Area/perimeter is relatively tight

**If We Insert P6 (4, 7) into u3:**

- MBR expands to:

Xmin = 4 (from P6), Xmax = 9 (P1)

Ymin = 2 (P1), Ymax = 7 (P6)

**- Perimeter = 2 × ((9−4) + (7−2)) = 20**

**u3 has room for 1 more entry → Valid insertion**

**A graph with lines and dots

AI-generated content may be incorrect.**

**Figure 10 : Insertion of P6 and the updated MBR**

**Diagram Explanation:**

Blue Dots: Represent data points P1 to P6

Red Box (u1): Root MBR that spans all child MBRs

Green Box (u2): MBR enclosing P2, P3, P5

Purple Box (u3): MBR enclosing P4, P1, P6

- P6 was inserted into u3 to avoid splitting full node u2

- This decision minimizes MBR expansion while maintaining balance

-The R-tree now has 6 points organized in a 2-level hierarchy with both child nodes full

**Step:7 Insert Point 7 (2, 5) into the R-tree :**

Both u2 and u3 are currently at full capacity (3 points each).

We must:

- Compute MBR expansion if P7 were inserted into either node

- But since both nodes are full, any choice will require a split

So we compute which one would result in less MBR expansion, and then split that node.

**MBR Expansion Comparison**

**Inserting into u2:**

- Current MBR (P2, P3, P5): (1, 4) to (6, 9)

- P7 = (2, 5) → already inside this MBR

**- MBR does not need to expand** (excellent spatial fit!)

- Best spatial choice, but **u2 is full → must split**

**Inserting into u3:**

- MBR: (4, 2) to (9, 7)

- P7 = (2, 5) → Expands Xmin from 4 → 2

- New MBR = (2, 2) to (9, 7)

**More expansion**

- Also full → would split

**Conclusion**: Insert P7 into **u2** and **split u2**

**Split Node u2**

**-u2 contains: P2 (5,4), P3 (6,7), P5 (1,9), and now P7 (2,5)**

We apply **linear split using X-axis** for clarity:

- Sort: P5 (1,9), P7 (2,5), P2 (5,4), P3 (6,7)

- Split:

**New u4** = {P5, P7}

**New u5** = {P2, P3}

**Update Tree Structure**

We now need to:

- Replace u2 in root u1 with new nodes **u4** and **u5**

- u1 now has 3 children: u4, u5, u3 (no overflow in root yet)

**A graph with lines and numbers

AI-generated content may be incorrect.**

**Figure 11 : Insertion of P7 and the updated MBR**

**Diagram Explanation :**

u4 (Green)

Contains: P5 (1, 9), P7 (2, 5)

MBR spans X: 1–2, Y: 5–9

u5 (Purple)

Contains: P2 (5, 4), P3 (6, 7)

MBR spans X: 5–6, Y: 4–7

u3 (Orange)

Contains: P4 (8, 6), P1 (9, 2), P6 (4, 7)

MBR spans X: 4–9, Y: 2–7

u1 (Root)

Encloses all child MBRs

**Step: 8 Insert Point P8 (1, 4) into the R-tree :**

Insert the data point P8 (1, 4) into the current R-tree structure while minimizing the expansion of Minimum Bounding Rectangles (MBRs) and maintaining tree balance.

**Insertion Strategy:**

The R-tree at this point has three leaf nodes under the root (u1):

u4: P5, P7

u5: P2, P3

u3: P4, P1, P6

To determine the optimal insertion node for P8, we evaluate the MBR expansion required by each candidate node.

**MBR Expansion Analysis:**

**- u4** (P5, P7):

Original MBR: (1, 5) to (2, 9)

Inserting P8 (1, 4) extends Ymin → (1, 4) to (2, 9)

New Perimeter = 2 × ((2−1) + (9−4)) = **12**

u4 has room (only 2 entries) → Ideal choice

**- u5** (P2, P3):

Would require expanding Xmin to 1

Perimeter increases to **16**

Full → would require split

**- u3** (P4, P1, P6):

Already full

Would expand Xmin to 1

Perimeter increases to **26**

Not suitable

**Decision:**

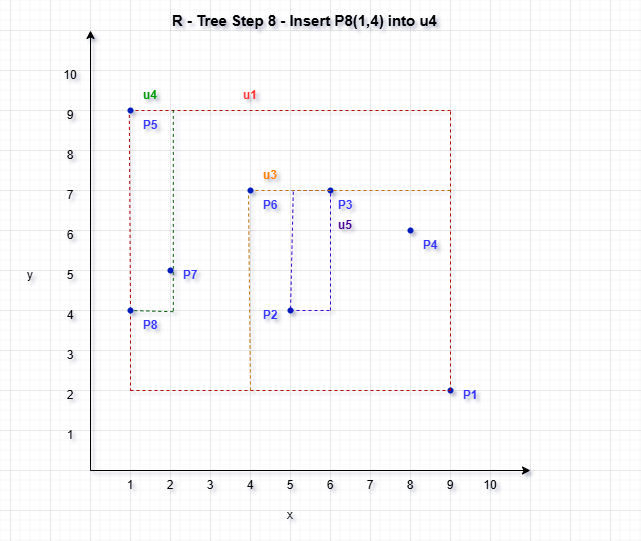
**- P8 is inserted into u4**, which now contains:

**P5 (1, 9)**

**P7 (2, 5)**

**P8 (1, 4)**

- MBR of **u4** updates to: (1, 4) to (2, 9)

****

**Figure 12 : Insertion of P8 and the updated MBR**

**Diagram Explanation:**

Blue Dots: Represent points P1 through P8

u4 (Green MBR):

Points: P5, P7, P8

MBR spans X: 1–2, Y: 4–9

u5 (Purple MBR):

Points: P2, P3

MBR spans X: 5–6, Y: 4–7

u3 (Orange MBR):

Points: P4, P1, P6

MBR spans X: 4–9, Y: 2–7

u1 (Red MBR):

Root node MBR

Spans all three children u4, u5, and u3

Each MBR is accurately represented by dashed rectangles and labeled.

**Step : 9 Insert Point P9 into the R-tree :**

To insert P9 (4, 7) into the existing R-tree while maintaining minimal MBR expansion and avoiding unnecessary splits.

**MBR Expansion Evaluation:**

**Option A: Insert into u4**

Full → would require a split

**Option B: Insert into u5**

Current MBR: (5,4) to (6,7)

Adding P9 = (4,7) expands Xmin to 4

New MBR: (4,4) to (6,7)

New Perimeter: 2 × ((6−4)+(7−4)) = 10

u5 has space → optimal

**Option C: Insert into u3**

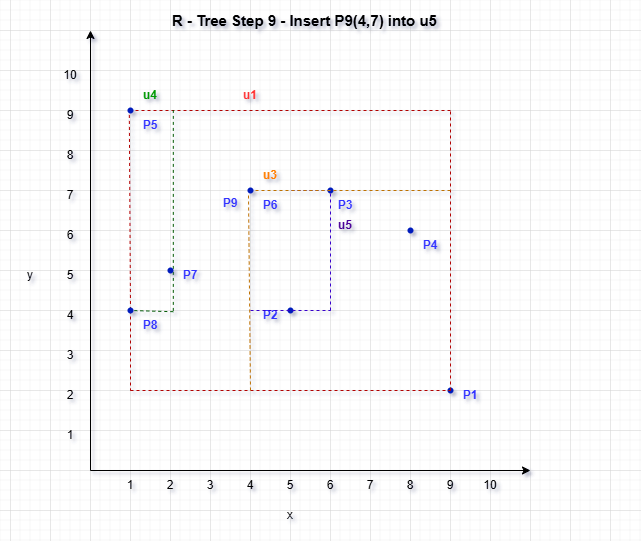
Full → would require a split

**Decision:**

P9 inserted into u5

u5 now holds: P2, P3, P9

u5 MBR updated to: (4,4) to (6,7)

****

**Figure 13 : Insertion of P9 and the updated MBR**

**Diagram Explanation:**

**- Blue Dots: Represent points P1 to P9**

**- Green MBR (u4):**

Points: P5, P7, P8

MBR spans X: 1–2, Y: 4–9

- **Purple MBR (u5):**

Points: P2, P3, P9

Updated MBR spans X: 4–6, Y: 4–7

- **Orange MBR (u3):**

Points: P4, P1, P6

MBR spans X: 4–9, Y: 2–7

- **Red MBR (u1):**

Root MBR encapsulating all three child MBRs

**Each rectangle is labeled, and spatial distribution is balanced and clear.**

**Step : 10 Insert Point 10 (ID 10: X=6, Y=2) :**

Insert point P10 (6, 2) into the R-tree while maintaining balance and minimizing the impact on spatial indexing performance.

**Insertion Decision:**

All existing leaf nodes (u4, u5, u3) were already full with 3 entries each. Thus, inserting P10 into any of them would trigger a split.

We need to:

- Calculate which node incurs the **least MBR expansion**

- Insert and **split that node**

**MBR Expansion Cost**

**Option A: u4 (P5, P7, P8)**

MBR: (1, 4) to (2, 9)

P10 = (6, 2) → massive X and Y expansion

New MBR: (1, 2) to (6, 9)

**Perimeter = 2 × ((6−1) + (9−2)) = 28**

**Option B: u5 (P2, P3, P9)**

MBR: (4, 4) to (6, 7)

P10 = (6, 2) → Ymin expands from 4 → 2

New MBR: (4, 2) to (6, 7)

**Perimeter = 2 × ((6−4) + (7−2)) = 20**

**Option C: u3 (P4, P1, P6)**

MBR: (4, 2) to (9, 7)

P10 = (6, 2) → already inside

**- No MBR change**

-**Perimeter remains the same (22)**

-But u3 is **already full**

**Decision:**

**- u5** is the best choice:

Causes less expansion than u4

u3 would be ideal spatially, but it's already full

- u5 is full → we must **split it**

**Split u5:**

- u5 currently has: P2 (5,4), P3 (6,7), P9 (4,7), P10 (6,2)

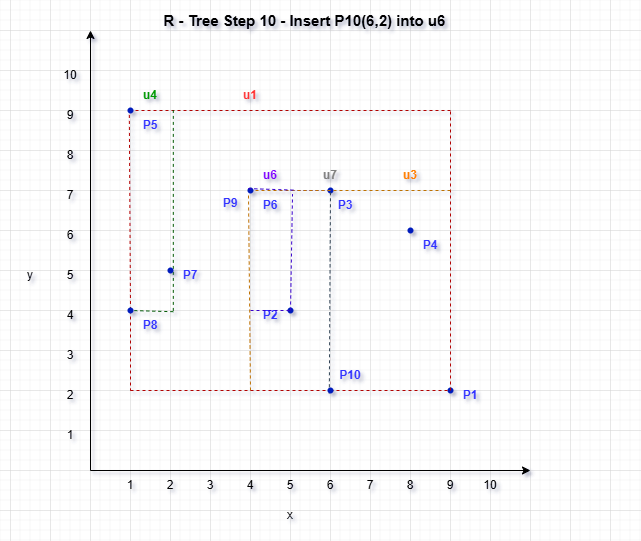
**Sort by X-axis:**

→ P9 (4,7), P2 (5,4), P3 (6,7), P10 (6,2)

**Grouping:**

**u6** = {P9, P2}

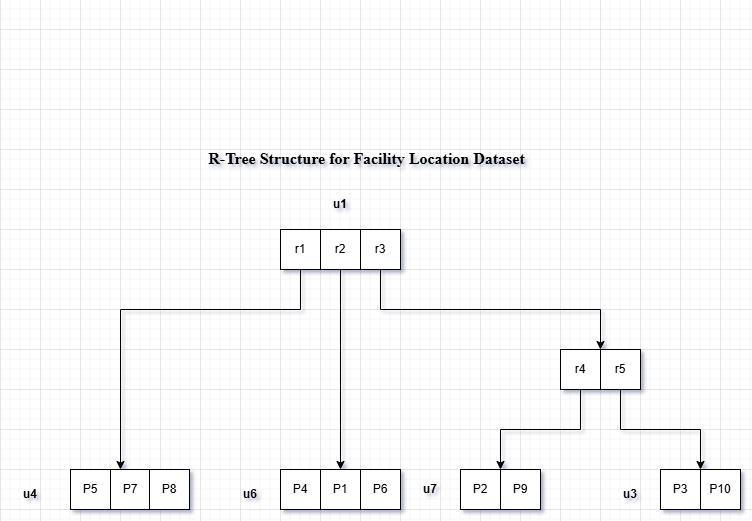
**u7** = {P3, P10}

****

**Figure 14 : Insertion of P10 and the updated MBR**

The final R-tree diagram illustrates a spatial indexing structure built over ten facility location points (P1 to P10). At the top level, the root node u1 encapsulates four leaf nodes: u4, u6, u7, and u3, each of which holds a subset of the data points. Specifically, u4 contains P5, P7, and P8; u6 contains P9 and P2; u7 contains P3 and P10; and u3 contains P1, P4, and P6. These groupings ensure that no node exceeds the maximum fan-out limit of 3. Each leaf node is enclosed within its own Minimum Bounding Rectangle (MBR), and the root node’s MBR spans the full spatial extent of all points. The R-tree is two levels deep, efficiently balancing point distribution while supporting fast nearest neighbor and range queries. The structure is both compact and optimized for spatial access, reflecting the incremental insertion and split decisions taken during the construction process.

**Final R-Tree Structure for Facility Location Dataset**



**Figure 15 : Final R-Tree Structure for Facility Location Dataset**

This R-tree structure provides a balanced, compact representation of the dataset. It was built incrementally using insertion and node splitting based on spatial criteria (MBR expansion), resulting in a final structure where all points are efficiently indexed across 4 leaf nodes under a single root node.

**Figure for R-tree after inserting the query point.**

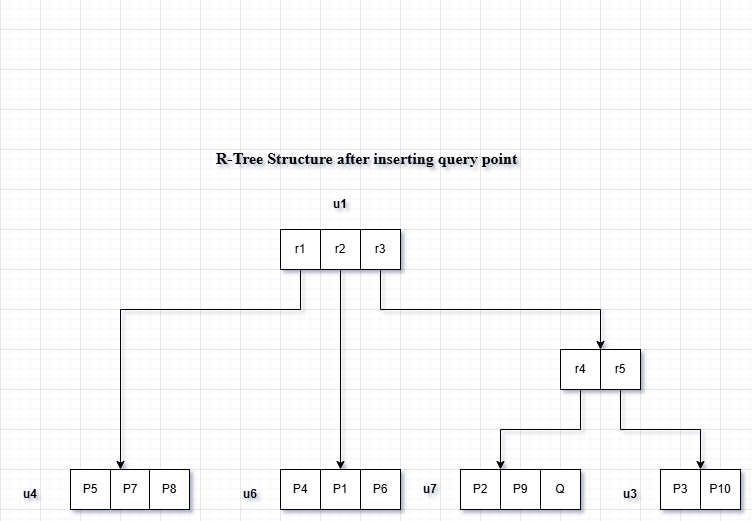
**A graph with lines and dots

AI-generated content may be incorrect.**

**Figure 16 : Final R-Tree after adding Query Point (4,7)**

Lastly, here is the completed R-tree structure after the insertion of all facility location points and the query point Q at coordinates (4,7). This addition demonstrates how the R-tree adjusts to accommodate new points within the existing structure, with updates to the Minimum Bounding Rectangles (MBRs) at various levels as necessary. The root node and leaf nodes collectively define the hierarchical organization of spatial data, facilitating efficient access and query processing. The inclusion of the query point Q exemplifies the dynamic nature of the R-tree, capable of expanding to handle additional spatial data without compromising search efficiency.

**Final R-Tree Structure after inserting query point**



**Figure 17: Final R-Tree Structure after inserting query point**

The final R-tree structure for the facility location dataset consists of a root node (u1) and four leaf nodes (u4, u6, u7, and u3), each containing a group of spatially close data points. The query point Q(4, 7) is inserted into leaf node u6, which already contains P9 (4, 7) and P2 (5, 4). This decision is based on spatial alignment and efficiency—since Q matches P9 exactly, inserting it into u6 avoids expanding the existing Minimum Bounding Rectangle (MBR), whereas inserting it into other nodes like u7 would result in unnecessary MBR growth. The R-tree maintains a balanced, compact structure with each leaf node containing no more than three entries, ensuring efficient spatial indexing and query performance.

**4.2 The Process of BF Algorithm :**

The Best-First (BF) algorithm is a method to find the nearest neighbor to a query point in an R-tree structure.

The goal of the BF algorithm is to quickly locate the closest point by prioritizing nodes based on their distance to the query point. Instead of exploring all nodes, it focuses on those likely to be closest.

A graph with lines and dots

AI-generated content may be incorrect.

**Figure 18 : Initial Steps of the Best-First (BF) Algorithm in an R-tree Structure**

In this illustration, we demonstrate the initial steps of the Best-First (BF) algorithm in an R-tree structure to locate the nearest neighbor to a query point Q= (4,7). The BF algorithm begins by prioritizing nodes based on the minimum distance (mindist) to the query point, focusing on nodes that are likely to contain the closest points.

The figure shows the spatial layout of facility locations within their Minimum Bounding Rectangles (MBRs). By calculating the mindist from Q to each MBR and updating the priority queue, the BF algorithm narrows down the search.

**Step-by-Step Plan for BF Algorithm:**

**1. Initialize the Priority Queue**

- Insert the root node u1 into the priority queue.

- Use the minimum distance between the query point Q(4, 7) and each MBR as the priority.

**2. Traversal Process**

- Pop the MBR (or point) with the lowest distance from the queue.

- If it’s a non-leaf node:

Insert all its children MBRs with their distance to Q.

* If it’s a leaf node:

Compute exact distances from Q to each data point in that node.

Track the closest point found so far.

**3. Termination**

- Once the nearest data point is popped from the queue, return it as the result.

**Initial Setup (Iteration 0)**

Query Point: Q(4, 7)

**Priority Queue (Min-Heap):**

We start by inserting the root node u1:

In a Min-Heap–based priority queue for R-tree search, we begin by inserting the root node u1 because the query point Q lies within its Minimum Bounding Rectangle (MBR). Therefore, the distance to Q (MinDist) is 0, making u1 the first and highest-priority entry in the queue.

**Iteration 1: Expand u1**

We now expand root u1 and enqueue its children:

After inserting the root node u1 into the priority queue, we expand it by enqueuing its child nodes. These are all internal nodes representing groups of data points enclosed within their respective Minimum Bounding Rectangles (MBRs). Specifically:

* u4 is added to the queue, which encloses points P5, P7, and P8. Its distance to the query point Q (MinDist) is calculated based on the MBR boundary.
* u6 contains points P9 and P2, and is also enqueued with its corresponding MinDist to Q.
* u7, representing P3 and P10, is included in the queue after computing its MBR’s distance from Q.
* Lastly, u3, which holds P1, P4, and P6, is added to the queue after its MinDist is computed.

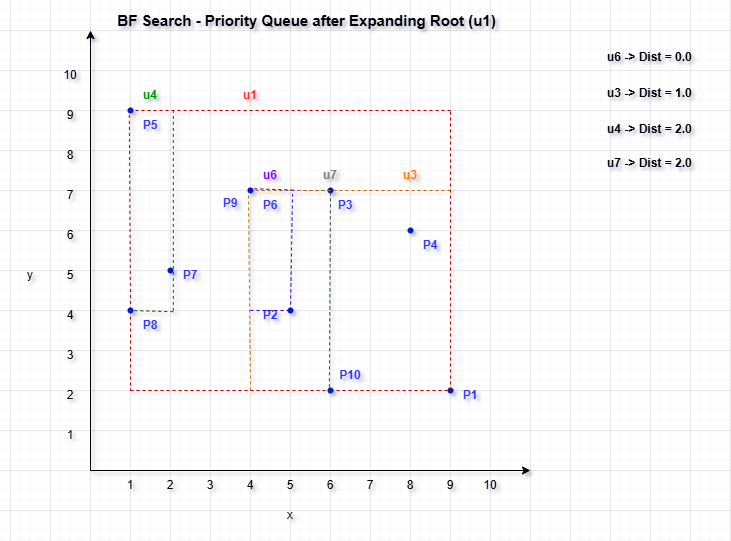
Each of these entries is now prioritized in the queue based on their distance from the query point, ensuring that the closest regions are explored first in the R-tree traversal.

I will now compute the distances from Q(4, 7) to each of these MBRs and update the queue.

**Priority Queue – Iteration 2**

Based on the calculated minimum distances (MinDist) from the query point Q(4,7) to the MBRs of the child nodes, we prioritize the nodes in the queue as follows:

* 1st Priority – u6 (MinDist = 0.0): The query point lies *inside* the MBR of u6, which contains points P9 and P2. In fact, Q matches P9 exactly, making this node the most relevant and nearest for exploration.
* 2nd Priority – u3 (MinDist = 1.0): Although Q is not inside u3's MBR (containing P1, P4, and P6), it lies just 1 unit away from its closest edge or corner, making it the next closest region.
* 3rd Priority – u4 (MinDist = 2.0): The query point lies outside the MBR of u4 (which includes P5, P7, and P8), with a closest edge distance of 2 units.
* 4th Priority – u7 (MinDist = 2.0): Similarly, Q is also 2 units away from the MBR of u7, which holds P3 and P10. Since it has the same distance as u4, it's ranked equally in terms of distance but listed after u4 due to queue order.



**Figure 19 : Visualization of Iteration 2 of the Best-First (BF) Search algorithm**

**Iteration 3 :**

**Since u6 has MinDist = 0, we expand it next in Iteration 3.**

* Expand u6 and insert its contents (P9, P2, Q) into the priority queue.
* Compute distances from Q(4, 7) to:

P9 (4, 7) → 0.0

P2 (5, 4) → √[(5−4)² + (4−7)²] = √10 ≈ 3.16

Q (4, 7) → 0.0 (we skip re-checking Q itself)

The first point dequeued will be P9, the nearest neighbor.

After expanding node u6, we examine its data points and insert them into the priority queue:

P9 has a distance of 0.0 from the query point Q(4, 7)—it is an exact match.

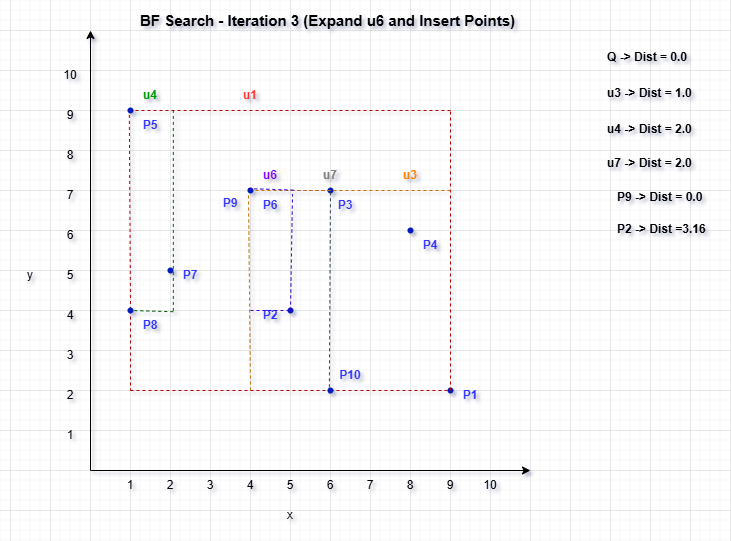
P2 lies farther away, with a calculated distance of approximately 3.16 units from Q.

Since P9 has the smallest distance and is at the top of the priority queue, it is identified as the nearest neighbor. The algorithm terminates at this point because the first data point retrieved from the queue is the closest possible match.

**Final Result:**

Nearest Neighbor to Q(4, 7): P9 (4, 7)

Distance: 0.0 (Exact match)

****

**Figure 20 : Visualization for Iteration 3 of the Best-First Search algorithm**

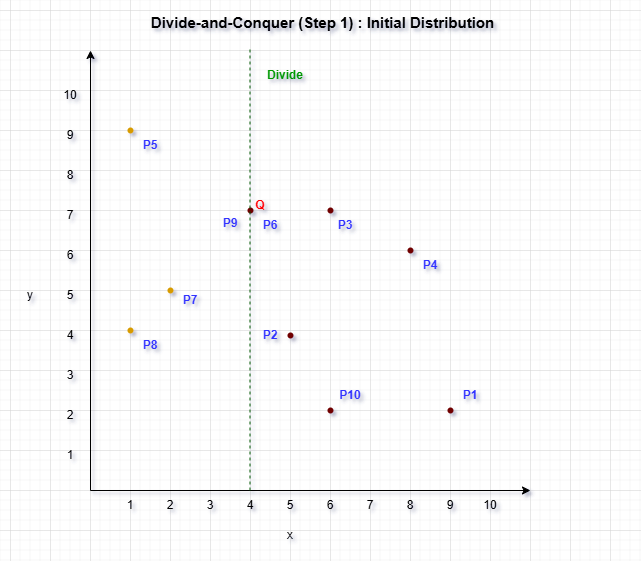
**Conclusion :**

The Best-First Search (BF) algorithm was executed on the constructed R-tree to locate the nearest facility to the query point Q(4, 7). The search began at the root node u1 and expanded nodes based on the minimum distance (MinDist) between the query point and the MBRs of child nodes. Upon expanding node u6, which had a MinDist of 0.0, its child points were inserted into the priority queue. Among them, P9 (4, 7) matched the query point exactly, yielding a distance of 0.0. Since the BF algorithm guarantees that the first data point dequeued is the nearest neighbor, the search terminated immediately. Therefore, P9 was identified as the nearest neighbor to Q, efficiently and accurately, using the spatial indexing structure of the R-tree.

**4.3 Divide-and-Conquer :**

The divide-and-conquer strategy can significantly enhance the efficiency of locating the nearest neighbor in an R-tree structure. By dividing the search area into manageable regions and analyzing these sections independently, the algorithm reduces the number of points it needs to consider. This approach allows the BF search algorithm to zero in on the areas closest to the query point, minimizing unnecessary evaluations.

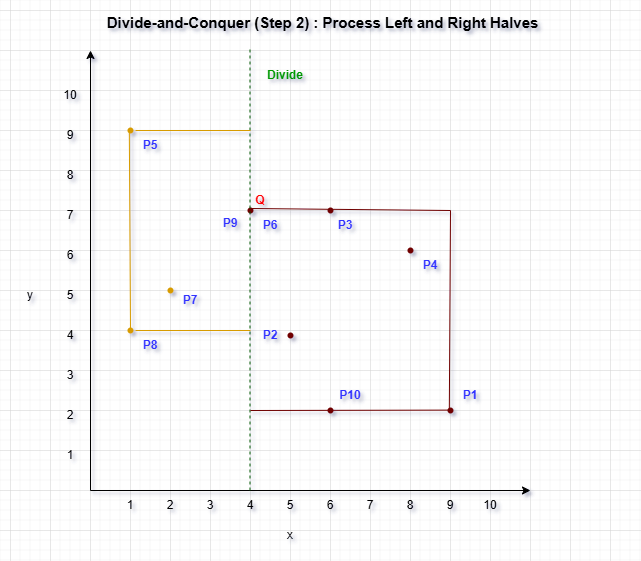
The image illustrates the **first step of the Divide-and-Conquer algorithm**, which aims to find the nearest neighbor to a given query point. Here, all facility points from **P1 to P10** along with the **query point Q(4, 7)** are plotted on a 2D plane. A **green dashed vertical line** at **x = 4** acts as the dividing line, splitting the space into two subregions: left and right. This division is crucial for recursively narrowing down the search space.

****

**Figure 21: Step 1 of the Divide-and-Conquer implementation**

To create this division, the algorithm sorts all points by their x-coordinate and then splits them into two equal halves. The left subset, shown in orange, contains points that fall to the left of the vertical line — specifically P5, P8, and P7. These points have x-values less than 4. The right subset, shown in purple, contains the query point Q, as well as P2, P10, P3, P4, and P1, all of which have x-values greater than or equal to 4.

This setup forms the foundation of the Divide-and-Conquer approach. By separating the dataset into two balanced parts, the algorithm can independently process each half and later combine results, checking across the dividing line only if necessary. Including the query point in the right half ensures it will be considered early in the nearest neighbor comparison.

****

**Figure 22 : Step 2 of the Divide-and-Conquer approach**

After dividing the dataset in Step 1 using the median x-coordinate, Step 2 focuses on processing each half independently to search for the nearest neighbor. The image highlights two clearly separated rectangular regions—orange for the left half and purple for the right half—representing the recursive processing phase of the Divide-and-Conquer algorithm.

The orange rectangle encloses the left subset of points: P5, P8, P7, P6, and P9. These are the points whose x-coordinates are less than the dividing line (x = 4). Meanwhile, the purple rectangle represents the right subset, which includes the query point Q, along with P2, P10, P3, P4, and P1—all with x-coordinates greater than or equal to 4. These regions are processed recursively to determine local nearest neighbors.

The green dashed line at x = 4 serves as the division between the two subsets. Since the query point Q(4, 7) lies exactly on this boundary, it is important to consider potential nearest neighbors from both sides of the divide. Therefore, even after finding a candidate in one region, the algorithm may still need to compare across the boundary to ensure global optimality.

**A graph with lines and dots

AI-generated content may be incorrect.**

**Figure 23 : Step 3 of the Divide-and-Conquer approach**

After identifying the nearest neighbors within each half, the Divide-and-Conquer algorithm proceeds to Step 3, which involves constructing a vertical strip region centered on the dividing line to check for potential closer pairs across the boundary.

In this case, the minimum distance (δ) found so far—between the query point Q(4, 7) and P9—is 1.0. Using this δ, a gray shaded vertical strip of width 2δ (i.e., 2 units) is drawn around the green dividing line (x = 4). This strip spans from x = 3 to x = 5, as shown by the dashed gray boundaries.

Only the points within this shaded strip are considered for cross-boundary comparisons because any point outside the strip must be at least δ units away in the x-direction alone, and thus cannot be closer than the current best. Since the query point Q lies inside this strip, it may be compared with nearby points from both sides—left and right—to ensure the global nearest neighbor is correctly identified.

**Conclusion :**

The Divide-and-Conquer approach for nearest neighbor search begins by sorting all facility location points by their x-coordinates and dividing them into two halves—left and right—at the median x-value. This initial division, illustrated in Step 1, creates two balanced spatial regions for recursive processing. In Step 2, each region is processed independently to determine the closest point to the query point Q(4, 7) within that half. The query point lies directly on the dividing line, indicating potential relevance to both regions. After computing the closest distances from both halves, Step 3 introduces a vertical strip of width 2δ around the divide, where δ is the current minimum distance found. This strip focuses the search to include only those points close enough to possibly yield a better match across the boundary. In this case, the query point Q falls within the strip, and its comparison with points in both sides—particularly P9—reveals that P9 (4, 7) is the nearest neighbor with a distance of 0.0. This method efficiently narrows the search space and demonstrates how divide-and-conquer techniques can enhance nearest neighbor queries, especially when combined with spatial locality awareness.

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

We have determined the Skyline points from a set of city location data using an efficient spatial indexing method—R-tree combined with the Branch and Bound Skyline (BBS) Algorithm. The primary objective was to identify the best city locations that are not dominated by any other point across all dimensions, meaning they represent optimal choices in terms of spatial attributes. The R-tree structure allowed for organized spatial data storage, significantly speeding up the search for potential skyline candidates.

The dataset, known as "City1," comprises 15 points labelled P1 through P15. The dataset points were plotted on a scatter graph to visually examine their distribution. The skyline points, represented as green circles, stand out as they are not outperformed by any other point in both X and Y dimensions. This visual representation provides a clear and intuitive understanding of which points are most significant within the dataset, emphasizing their dominance-free positioning and relevance in skyline analysis.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ID** | **1** | **2** | **3** | **4** | **4** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **X** | **5** | **8** | **12** | **3** | **10** | **6** | **9** | **2** | **13** | **15** | **4** | **11** | **7** | **1** | **14** |
| **Y** | **10** | **7** | **5** | **12** | **8** | **6** | **4** | **14** | **3** | **2** | **9** | **6** | **3** | **13** | **1** |

A graph with dots and numbers

Description automatically generated

**Figure 24 : Skyline Search Dataset Visualization**

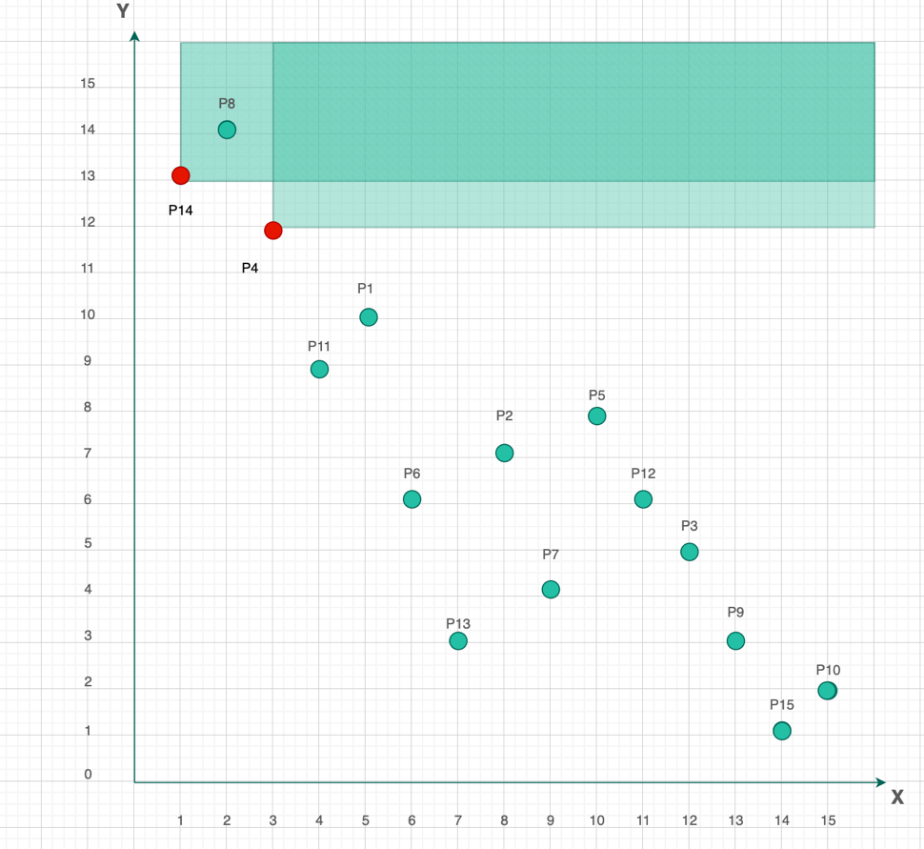
**5.1 R-Tree Construction Process**

The sequence of diagrams illustrating R-tree construction highlights a systematic approach to organizing spatial data for efficient query performance, especially in skyline queries. In the early stages, potential skyline points—such as P1, P2, P5, P12, and P15—are marked in green, indicating that they are not dominated in terms of key attributes like cost and distance. As the tree continues to build, these green points remain enclosed within green shaded bounding rectangles, reflecting their ongoing importance to the skyline. These rectangles adapt dynamically to tightly enclose only the relevant non-dominated points, helping to narrow the search to meaningful regions during queries.

Conversely, red points like P6, P11, P13, and P14, which appear in later stages of the diagrams, represent dominated entries. As the R-tree structure evolves, these points are excluded from further skyline consideration, demonstrating the algorithm's ability to effectively prune unqualified data. This pruning capability is key to the R-tree’s efficiency, as it limits the processing to only those points that may contribute to the final skyline set.

**A graph with green dots and red points

Description automatically generatedFigure 25 : Step 1 - Skyline Coordinates from Left to Right**

****

**Figure 26 : Step 2 - Skyline Coordinate points from left to right**

A graph with green and red dots

Description automatically generated**Figure 27 : Step 3 - Skyline Coordinates from left to right**

A graph with green squares and red dots

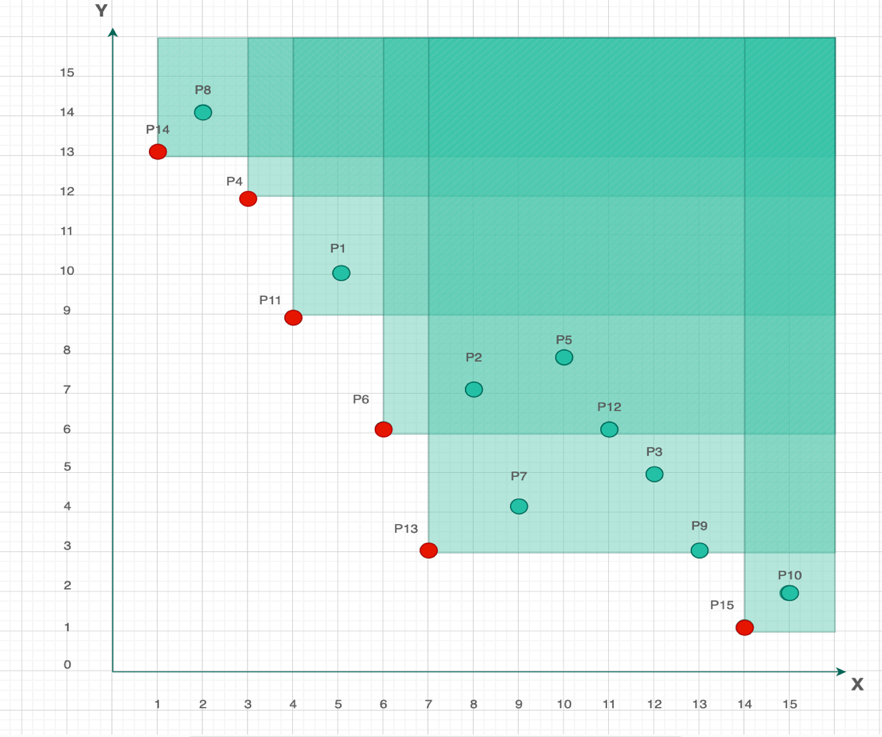
AI-generated content may be incorrect.

**Figure 28 : Step 4 - Skyline Coordinates from left to right**

A graph with green squares and red dots

AI-generated content may be incorrect.

**Figure 29 : Step 5 - Skyline Coordinates from left to right**

****

**Figure 30 : Step 6 - Skyline Coordinates from left to right**

A graph with green squares and red dots

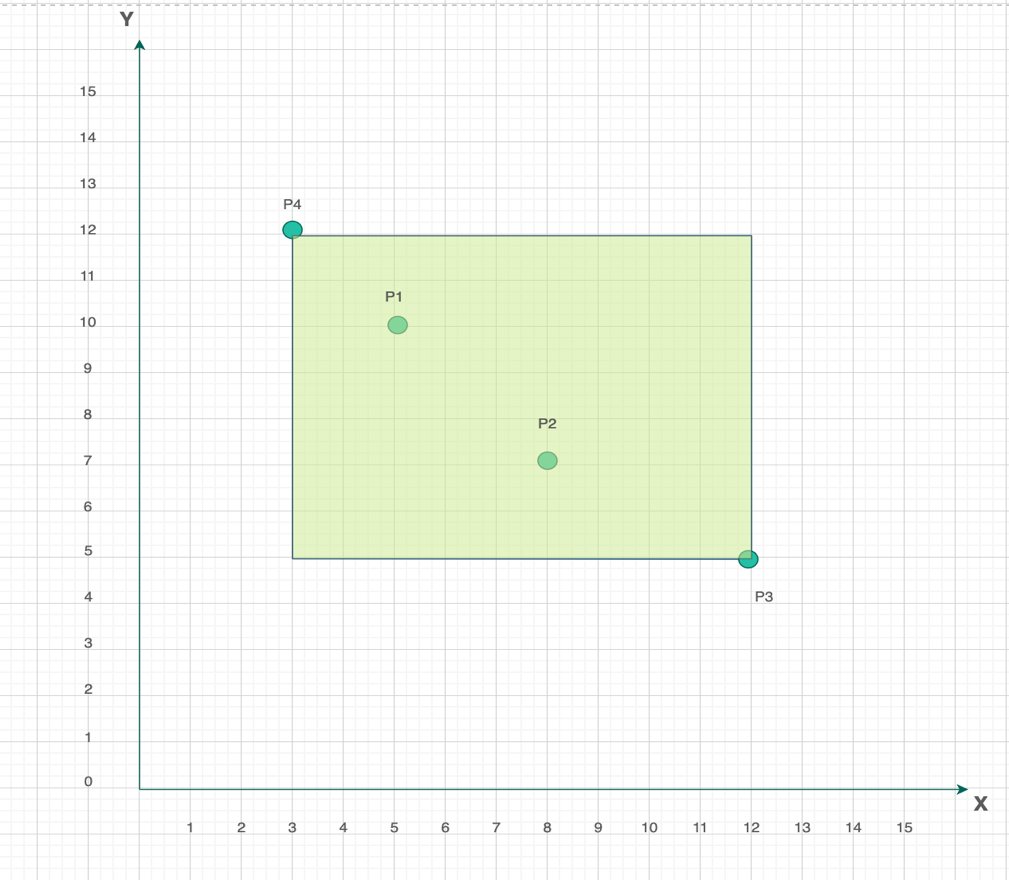
AI-generated content may be incorrect.

**Figure 31 : Step 7 - Skyline Coordinates from left to right**

Each diagram incrementally updates the R-tree structure, with red dotted lines in the final step indicating adjustments made to refine node boundaries. These refinements aim to optimize space partitioning and coverage, ensuring the tree structure remains efficient. By clearly defining boundaries and selectively including only the most relevant points, the R-tree maintains a balanced hierarchy with minimal overlap between nodes. This focused organization enhances the performance of spatial queries by directing attention to the most promising, non-dominated regions, thereby reducing unnecessary computations.

**5.2 BBS (Branch and Bound Search) Algorithm Process**

In the figure below, the early stage of R-tree construction is shown using the Minimum Bounding Rectangle (MBR) approach. At this point, points P1, P2, P3, and P4 are grouped together within a single, large MBR. This initial grouping is essential as it lays the groundwork for further spatial subdivision in later stages. The large green rectangle illustrates how the R-tree begins by encompassing all data points in one broad region, providing a unified structure from which it will later refine. This step is especially important for the Branch and Bound Search (BBS) algorithm, as it defines the initial spatial limits within which the skyline (non-dominated) points will be searched. By starting with all points enclosed in one MBR, the R-tree demonstrates its efficiency—reducing the need for excessive spatial checks by treating all enclosed points as initial skyline candidates within a single bounding region.



**Figure 32 : Minimum Bounding Rectangles and R-Tree Process**

**A diagram of a graph

AI-generated content may be incorrect.**

**Figure 33 : MBR and R-Tree Process**

In the figure above, the R-tree structure begins to refine itself by dividing into smaller MBRs, each containing subsets of spatially closer points. These new rectangles, labeled r4, r5, r6, and r7, reflect the hierarchical nature of the R-tree, where broader regions are systematically broken down to capture spatial proximity more accurately. For example, rectangle r4 now encloses points P4, P8, and P14, which are located near each other—this grouping effectively narrows the search area during spatial operations like nearest neighbor or skyline queries.

The creation of each smaller rectangle is a deliberate strategy to improve query efficiency by reducing unnecessary comparisons. Instead of evaluating every point globally, the structure allows queries to focus on localized clusters, streamlining the process. This segmentation is crucial for the Branch and Bound Skyline (BBS) algorithm, as it enables faster identification of skyline points by prioritizing relevant regions and avoiding repeated evaluations across the entire dataset.

**A screenshot of a computer

AI-generated content may be incorrect.**

**Figure 34 : Root Node, Internal Node and Leaf Node**

Figure above presents the finalized R-tree structure, showcasing a clear hierarchy from the root node down to the leaf nodes, which efficiently organizes spatial data for optimized querying. The root node branches into internal nodes such as u2 and u3, which in turn manage larger MBRs like r4, r5, r6, r7, and r8, each encompassing specific groups of points. At the lowest level, leaf nodes such as u4 contain actual data points like P14, P8, and P4, while u5 includes P11 and P1. Similarly, r6 contains points like P6, P2, and P13, with other rectangles organizing their own respective subsets.

This hierarchical structure plays a vital role in supporting the Branch and Bound Skyline (BBS) algorithm, enabling it to quickly zero in on relevant regions while efficiently ignoring areas that do not contribute to potential skyline points. Leaf nodes, which represent the most granular level in the tree, are considered during the final stage of query execution, allowing precise comparisons only where necessary.

By significantly reducing the search space, this design improves both speed and accuracy in skyline and spatial queries. The integration of the BBS algorithm with the R-tree structure ensures fast and scalable retrieval, making it particularly useful in spatial databases and geographic information systems (GIS). It streamlines the process of identifying skyline points by narrowing the focus to the most promising data regions, thus boosting overall query performance.

**A graph with green and yellow squares

AI-generated content may be incorrect.**

**Figure 35 : Distance from Origin to Root Node**

The first step of the Branch and Bound Skyline (BBS) algorithm involves accessing the root node of the R-tree and creating a sorted priority list of its child Minimum Bounding Rectangles (MBRs) based on their Euclidean distances from the origin (or the reference point used in the skyline computation).

The distances are computed using the standard Euclidean distance formula:

Distance to r2 = √(1² + 9²) = √82

Distance to r3 = √(6² + 1²) = √13

Based on these calculated values, the initial sorted list of MBRs becomes:

{ (r3, √13), (r2, √82) }

This ordering ensures that the BBS algorithm explores the most promising (i.e., closest) regions first, improving the efficiency of skyline point detection.

**A graph with green and yellow squares

AI-generated content may be incorrect.**

**Figure 36 : Distance from Origin to u3 Child Node**

Next, the BBS algorithm visits node u3, which leads to the addition of more MBRs into the sorted list based on their Euclidean distances from the reference point. The distances for the newly encountered MBRs are calculated as follows:

Distance to r6 = √(6² + 3²) = √45

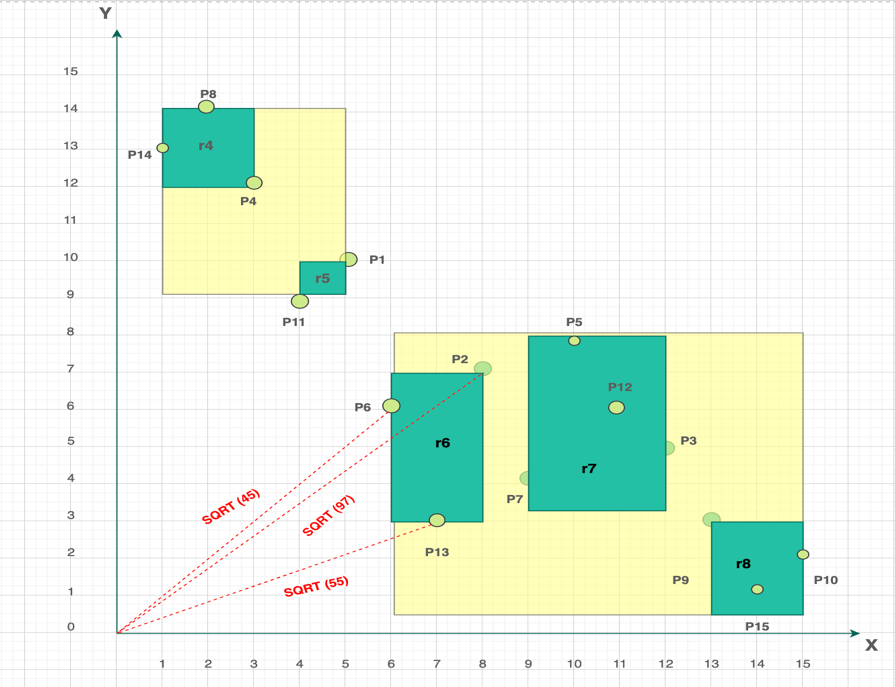
Distance to r7 = √(9² + 4²) = √97

Distance to r8 = √(13² + 1²) = √170

With these new entries, the updated sorted list becomes:

{ (r6, √45), (r2, √82), (r7, √97), (r8, √170) }

This approach allows the algorithm to prioritize MBRs that are spatially closest, effectively narrowing the search space. By visiting the nearest bounding rectangles first, the BBS algorithm avoids unnecessary comparisons and speeds up the identification of skyline points, enhancing the overall efficiency of the skyline query.

****

**Figure 37 : Distance from Origin to r6 Child Node**

As the BBS algorithm proceeds to the leaf node of r6, it encounters three data points and computes their Euclidean distances from the origin (or reference point). These distances are:

P13 = √(7² + 3²) = √58

P6 = √(6² + 6²) = √72

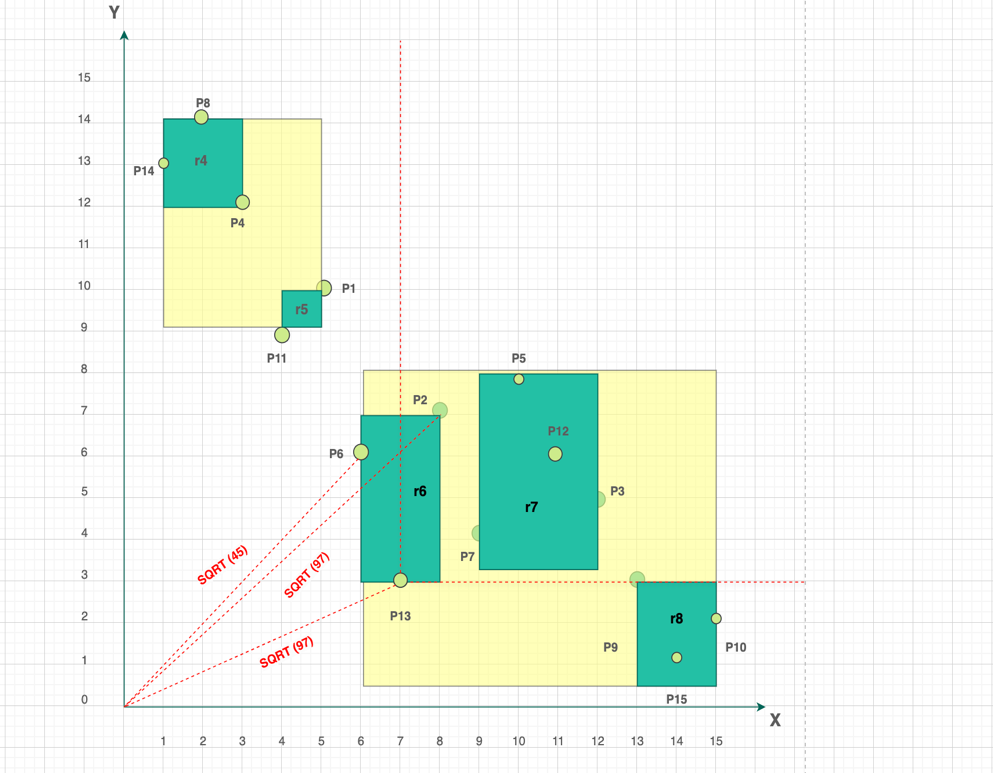
P2 = √(8² + 7²) = √113

The updated sorted list now becomes:

{ (P13, √58), (P6, √72), (r2, √82), (r7, √97), (P2, √113), (r8, √170) }

From this list, P13 and P6 are selected as the first and second skyline points, as they are not dominated by any earlier points in terms of both dimensions (e.g., lower cost and better proximity). When comparing with MBR r7, it is determined that all points within r7 are dominated by either P6 or P13, leading to the pruning of r7 from further consideration.

Additionally, P2 is also removed from the candidate list because it is dominated by both P6 and P13. This step refines the skyline candidate set by eliminating any non-promising entries, ensuring the algorithm continues with only the most relevant points. This pruning mechanism significantly improves performance, making the skyline evaluation more efficient in the subsequent steps.

****

**Figure 38 : X-coordinates and Y-coordinates (after P13)**

The algorithm then moves on to decompose internal node r2, which is the next closest MBR in the sorted list based on Euclidean distance. Upon expanding r2, its child nodes r5 and r4 are evaluated and inserted into the sorted list. The distances are calculated as:

Distance to r5 = √(4² + 9²) = √107

Distance to r4 = √(1² + 12²) = √145

This results in an updated sorted list:

{ (r5, √107), (r4, √145), (r8, √170) }

The algorithm continues this recursive process, evaluating and decomposing remaining internal nodes in the order of their proximity. Each step involves checking whether a node or point contributes to the skyline or is dominated and can be pruned. Through this efficient prioritization and elimination strategy, the algorithm ultimately identifies the final skyline points: P14, P4, P11, P6, P13, and P15.

By systematically breaking down only the most promising areas and excluding dominated entries early, this approach ensures that the Branch and Bound Skyline (BBS) algorithm efficiently narrows the search to the most relevant, non-dominated points—optimizing both accuracy and performance in skyline query processing.

**5.3** **Divide and Conquer Process :**

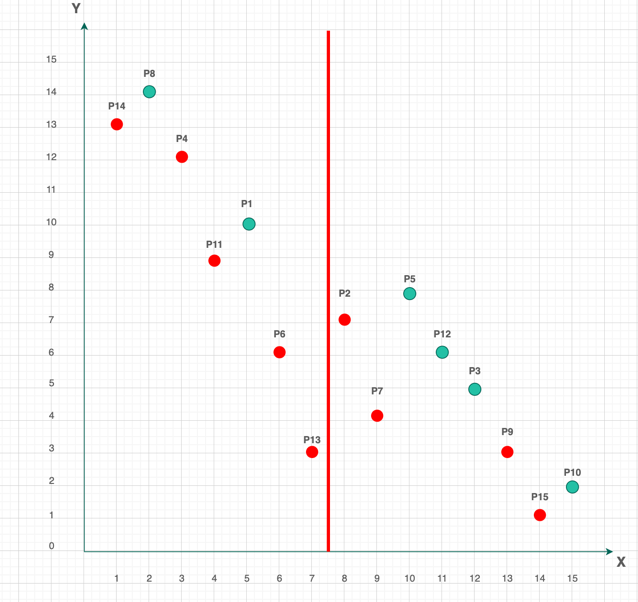
The divide-and-conquer approach is used to efficiently compute skyline points by reducing the overall computational complexity. In the first step, the dataset is split into two parts by dividing the maximum x-axis value (15) in half, resulting in a threshold of 7.5. Based on this division:

The left subset (A) includes points: {P14, P8, P4, P11, P1, P6, P13}

The right subset (B) includes points: {P2, P7, P5, P12, P3, P9, P15, P10}

Skyline points are then calculated independently within each subset, and the results are later merged to form the final skyline. This strategy significantly reduces processing time and complexity, as it limits comparisons to smaller groups before integrating the results—a process clearly illustrated in the accompanying figures.

**A graph with dots and lines

AI-generated content may be incorrect.**

**Figure 39 : Skyline points shown using Divide and Conquer Approach in X-axis**

After dividing the dataset and computing skylines within each subset, the dominant points are identified from both groups. For the left subset (A), the resulting skyline candidates are: {P14, P4, P11, P6, P13}. These points are retained as potential final skyline members since they are not dominated within their group.

For the right subset (B), the intermediate skyline set is: {P2, P7, P9, P15}. To finalize the skyline, a 1D dominance check is performed along the y-axis, comparing points from subset B with those in subset A. During this screening, all points in B are dominated by P13, which has a lower y-coordinate, except for P15, which remains undominated.

As a result, P15 is the only point from the right subset that qualifies as a final skyline point, alongside the left subset's skyline points.

**A graph with numbers and symbols

Description automatically generated**

**Figure 40 : Skyline Points sorting in Y-axis**

Therefore, the final set of skyline points is determined to be (P14, P4, P11, P6, P13, P15).