R-Trees: Performance and Extentions

JakeV

I. Introduction

An R-Tree, or 'rectangle tree' is a spatial data structure that can store the bounding boxes of polygons in a tree structure that allows for efficient searching of intersecting or enclosing rectangles. The R-Tree data structure is primarily used in 2d applications, but can be generalized to 3 or more dimensions. It can also be generalized to work with more than rectangles, or can be used as a wide-phase collision detection for polygons.

II. Preliminaries

To define an R-Tree, we will first define a few concepts.

A. Minimum Bounding Rectangle (MBR)

The minimum bounding rectangle of a set of rectangles 'R', is defined as the rectangle with the minimum area that encloses every rectangle within R.

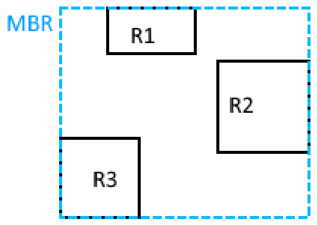

Fig. 1. MBR Demo

B. Area / Volume

The area of an MBR in 2d space is defined as its area (Width x Height). In 3d space, we use volume as a metric to bound 3d boxes (Length*Width*Height).

III. R-Tree Basics

A. Tree Structure

The nodes in an R-Tree represent a bounding rectangle that encloses all of its children within it. The tree has 2 variables that determine its structure, the M-value, and the m-value. M represents the max number of children any single node can have, leaf nodes also have these rules. The little m represents the minimum amount of rectangles that can be in a node, an m-value of 2 means no less than 2 children can exist for any node. However, the root node need not obey the little-m value rule.

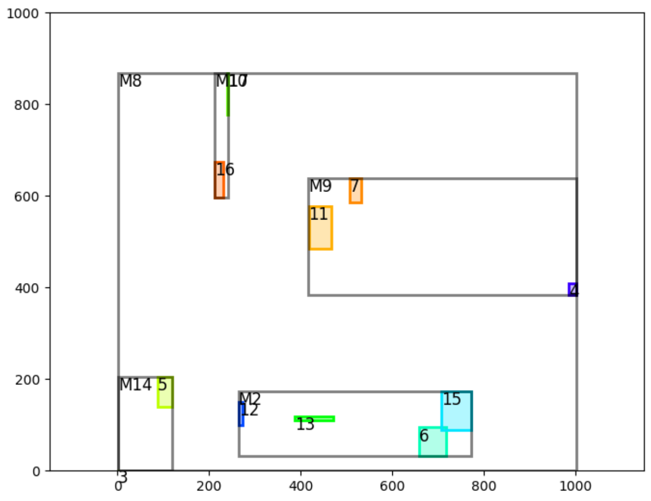

Fig. 2. Rectangular 2d Data

B. Visual

In Fig. 2, there is some 2d rectangle data plotted out. Using M=4, m=2, a tree is constructed.

In Fig. 3, we have our root node M8, which encloses all rectangles. M2 is a child of the root node and encloses rectangles 6, 12, 13, 15. Note that M2 is at maximum capacity for M=4, thus any more rectangles inserted into M2 will cause a split, which will be discussed later. Also note that every node has a minimum of 2 nodes, and a maximum of 4 nodes (corresponding to the large-M and little-M values). M8 does not need to obey this rule, but does in this case since it has 4 children.

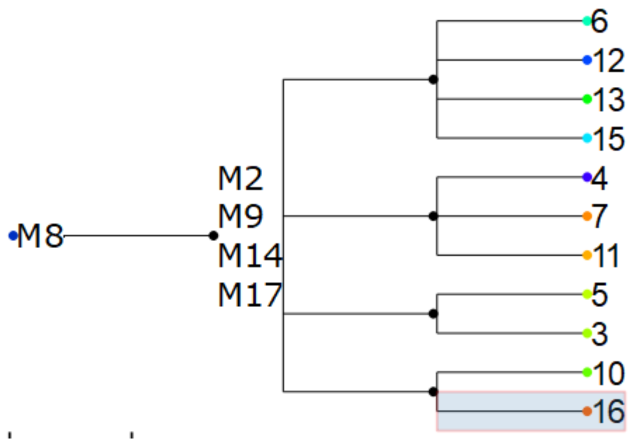

Fig. 3. Bounding Graph

C. *Querying*

The function **Search(S)** is used to determine all of the rectangles that intersect with the search rectangle S. The function descends the tree, starting at the root node, and checks intersection with each nodes children, traversing downward until a leaf is reached. It then checks the intersection with each of the data entries. In summary, an R-Tree reduces query time by eliminating whole groupings of rectangles by comparing the search rectangle with their MBR.

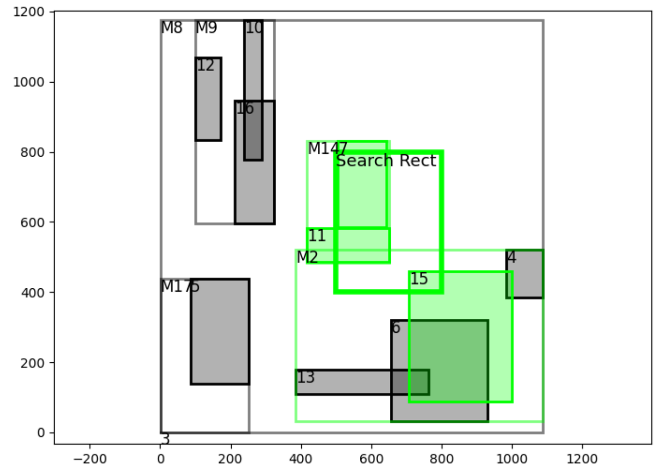

Fig. 4. Enter Caption

D. *Node Insertion*

Inserting a node into an RTree consists of 4 main steps

1) ChooseLeaf: Traverse the tree, starting at the root node. Select the child who's minimum bounding rectangle needs to be enlarged the least in order to fit the new rectangle. Recursively do this until a leaf node is reached. Insert the rectangle into it.

2) SplitNode: If the leaf node now has M+1 entries, it needs to be split in a way such that there are now 2 new nodes containing the split entries. The split should be done in a way such that the area of the two rectangles in minimized. The original paper outlines 3 different algorithms to do this, which will be discussed in the implementation section of the paper.

3) Recursive Step: Then, the newly created 2 nodes should be inserted into the parent, which may now have M+1 entries, thus needing SplitNode to be recursively called until the root node is reached.

4) Grow tree: If the root node needs to be split, we may need to create a new root.

E. *Node Insert Visual*

To visualize the node insert process, take Fig. 5. It is a valid R-Tree with m=2 and M=4, node M2 is full, so we will insert a new entry into it to cause a split.

Now, we will insert a red rectangle into M2, which will result in a split, since there is M+1 (5) entries in it now. Refer to figure 6, rectangles 18 and 13 have been added to their own

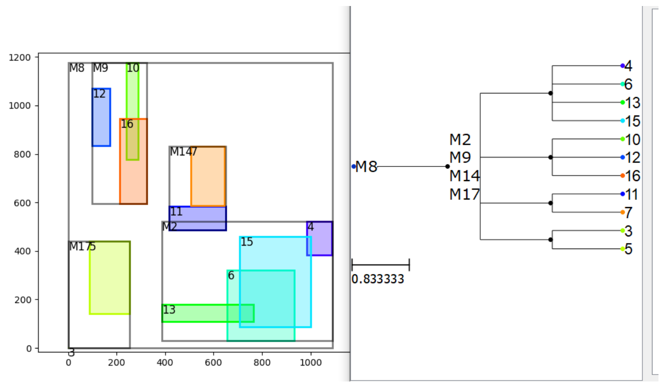

Fig. 5. A balanced RTree

node, and M2 has been shrunk. Also, the split caused M8 (the root node) to split.

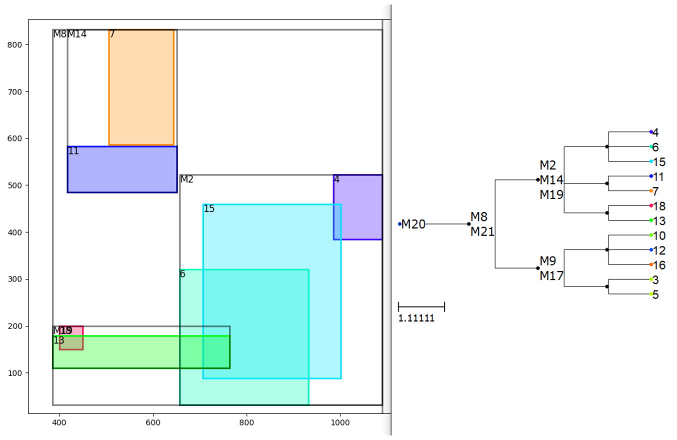

Fig. 6. Balanced Tree After Insert

IV. Differences from the Paper

Compared to the original paper, there are a few minor differences in my implementation that exist for the sake of simplifying the code, or because something is easier in python than it was in C. The first change is that MBRs are stored on a different level of the tree. In the original implementation, each node would store an MBR for each of its children, but now the MBR is stored so that each node contains the MBR of its children. This makes no real difference to the implementation besides the fact that there is now a single large rectangle that encloses all others. This can be useful in applications where the search rectangle may often be outside the data range.

A. *ChooseLeaf*

ChooseLeaf is unchanged

B. *Insert*

Insertion remains largely the same, however step L3 and L4, which involve propagating node splits upward, as well as growing the tree taller are now handled by SplitNode. When calling SplitNode, and it is the case that a parent node now has M+1 entries and needs to be split, It makes sense for SplitNode to handle this, rather than Insert.

C. AdjustTree

Similar to Insert, propagating node splits upward is handled by SplitNode now.

V. SplitNode Analysis

The original paper describes 3 algorithms for SplitNode.

A. Exhaustive Method

In the end, the author concludes that the exhaustive 2^{M-1} algorithm is completely unusable on real world datasets with a large M-value. This algorithm tries every possible node combination to determine the split with the least size. The algorithm is left in the code as an unused method, but further analysis of this method is pointless. An interesting point to note about the exhaustive method is that, although it guarantees the best solution for a *node split*, it does not guarantee an optimal R-Tree. This is due to the fact that inserting a new node does not always involve a node split, even though splitting may decrease overall area.

B. Quadratic Method

For the quadratic method, the two furthest rectangles in a node are selected, then, each remaining node is assigned to a group based on its distance to it, more precisely, how much each groups MBR needs to be enlarged to fit it.

C. Linear Method

The linear method only differs by the initial selecting of the two nodes. The quadratic brute-force method grantees the best solution (For the 2 furthest rectangles, not necessarily the optimal split, which is only guaranteed by the exhaustive method), however the linear method is faster.

1) Linear PickSeeds: The algorithm for linear PickSeeds is as follows:

1) Find the most extreme rectangles along each axis. (X and Y, in our case)
2) Normalize the distance by dividing the distance by the length of the rectangle.
3) Pick the pair with the maximum separation.

D. Linear vs Quadratic Seed Comparison

The linear and quadratic solutions often differ in terms of what seeds they select. Fig. 7 shows an example of this. Between 10 randomly generated rectangles, the linear solution picks the two in blue, and the quadratic chooses the two green rectangles. It is clear that the quadratic solution will result in a better split. It is often the case that the linear and quadratic solutions pick the same rectangle along a single side, as shown in Fig. 8 To compute the accuracy of the linear method compared to the quadratic solution, I ran 100000 tests of comparing the two for 10 randomized rectangles within an [0,100]x[0,100] range. The results are as follows.

Perfect	1 Side Correct	Completely Wrong
47.9%	28.4%	23.6%

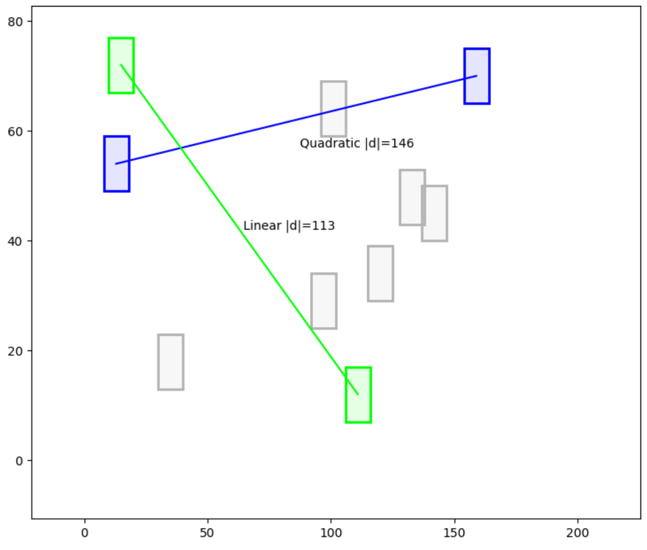

Fig. 7. Quadratic vs Linear Seeds

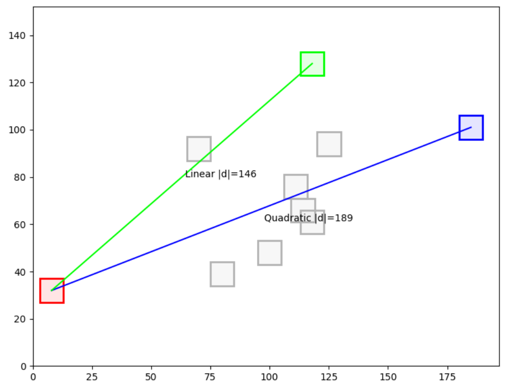

Fig. 8. A Typical Split

As you can see... The linear solution falls short to the quadratic solution. However, this is not the full story, as we are aiming to have the minimum area between the two rectangles, so a wrong solution may still produce an almost perfect result.

E. 4 Axis Split

To improve the accuracy of PickSeeds, I will propose a new linear algorithm that has an accuracy closer to Quadratic PickSeeds while still being linear. The quadratic algorithm is biased to pick rectangles near the corner since the nature of a rectangle means that points near corners can have a larger distance between each other. In our new linear algorithm, we will draw 4 lines through the rectangle, 2 diagonal, and 2 straight. Then, calculate the distance of reach rectangle to the line. Then, choose the min and max entries for each axis. Lastly, select the entry with the greatest separation.

The 2 lines that cut straight through the rectangle on the X and Y axis will yield the same result as the original linear pickseeds.

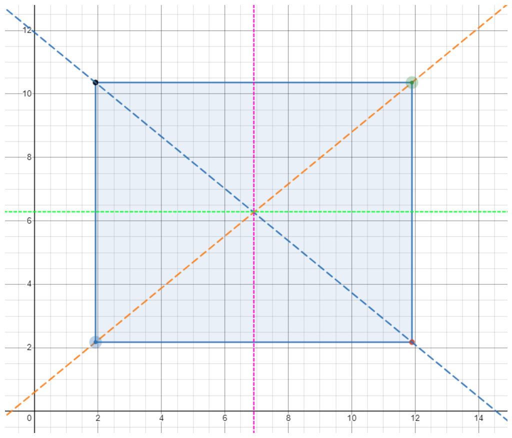

Fig. 9. 4 Line Example

The distance from a point to one of the two diagonal lines can be determined from this formula, where P is the point, R1 and R2 are opposite corners of the rectangle:

$$\frac{a \cdot P_x - P_y + c_1}{\sqrt{a^2+1}} \tag{1}$$

$$a = \frac{R1_y - R2_y}{R1_x - R1_y}$$

$$c = R1_y - R1_x$$

This is derived through the formula for a line, as well as the formula for a point's distance to a line:

$$y = mx + b$$

$$d = \frac{|ax + by + c|}{\sqrt{a^2+b^2}}$$

However, using this formula is actually equivalent to simply adding X+Y together, since we are only looking for the min/max, and the exact distance doesn't matter. By using this solution, we get a new accuracy chart:

Perfect	1 Side Correct	Completely Wrong
91.5%	5.1%	3.4%

This is a substantial increase over the original algorithm, in the next section, we will see how this affected the area.

F. Area Comparison

After our seeds are chosen by any of the above methods, the rest of the rectangles need to be assigned into 2 groups. First, iterate through all the remaining entries and determine the entry that needs the least enlargement to fit into each group. Assign them and repeat. If we have so few entries in a group that ¿m is not satisfied, assign the rest of the rectangles to this group. This results in 2 groups of rectangles.

We can compare the area accuracy of our new diagonal 4-axis split algorithm to the quadratic solution. By running 100000 simulations with 10 rectangles each, m=2, the quadratic algorithm has a total area of around 836.37M, while the 4-axis algorithm has a total area of 836.93M. This means

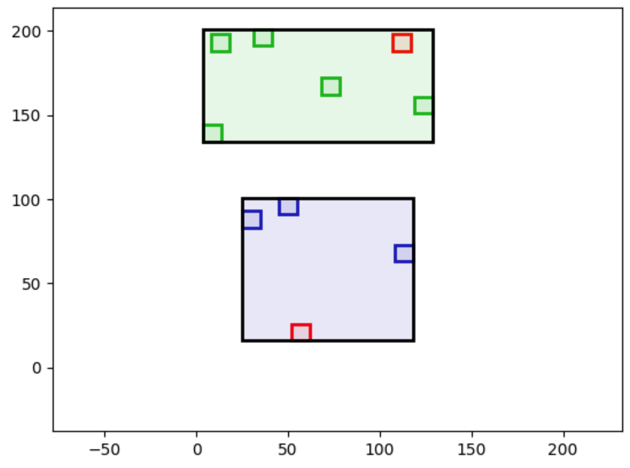

Fig. 10. SplitNode result

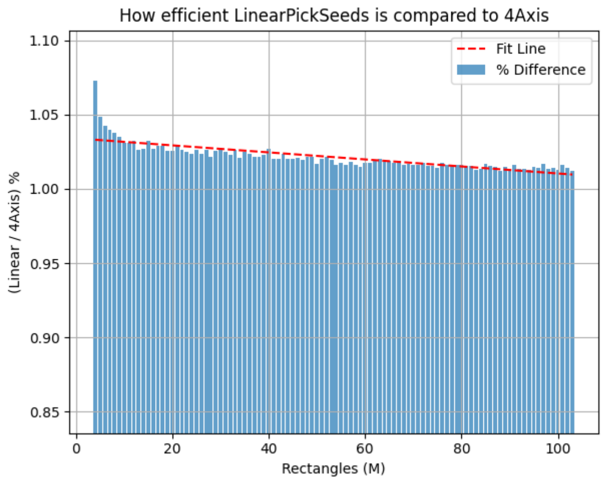

Fig. 11. 4 Axis to Linear comparison

that our new algorithm is 99%+ as efficient as the quadratic one. Comparing to the original linear algorithm, it is actually only 2% more efficient. This small advantage degrades even more when larger M-values are chosen. Fig. 11 demonstrates how efficient the 4-axis algorithm is compared to the linear one. As you can see, at low M-values 4Axis splits are worth it, but as M increases, the difference between efficiencies decrease. In conclusion, the 4 axis split algorithm is efficient for low M-values, but yields no practical benefit on large M-values.

VI. Performance

In the original paper, the authors concluded that the linear algorithm was nearly as efficient as the quadratic one with faster performance. In my tests, I want to determine the optimal M and m values for different distributions of data. We will try on 3 different datasets, one with an even distribution of

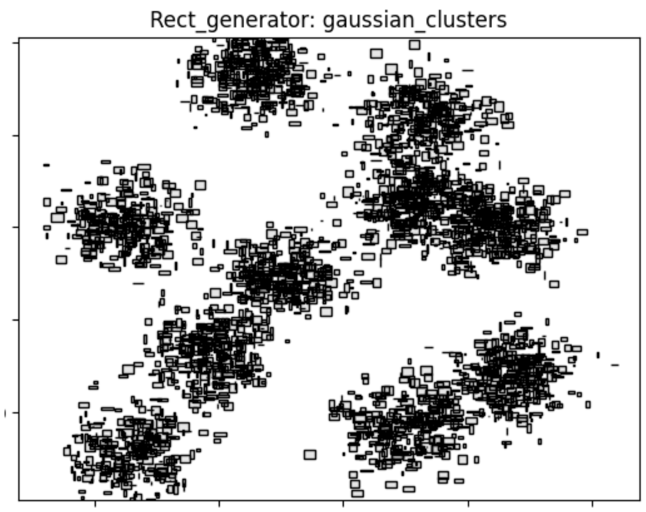

Fig. 12. Gaussian Cluster Distribution

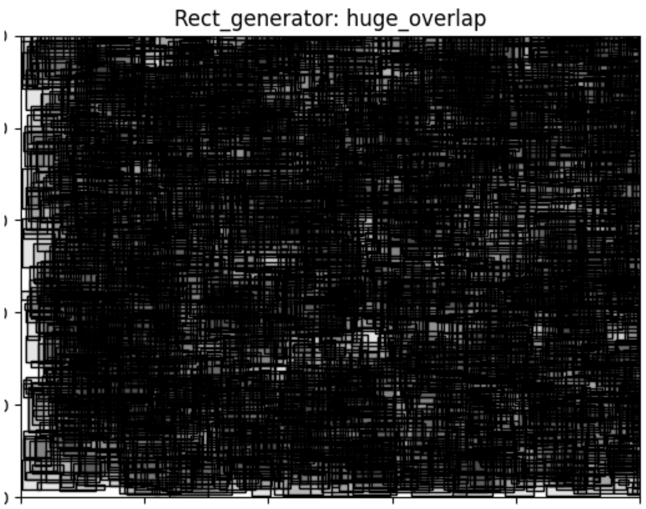

Fig. 13. Huge Overlap Distribution

rectangles, one with huge rectangles, and one where rectangles are clustered in 10 zones.

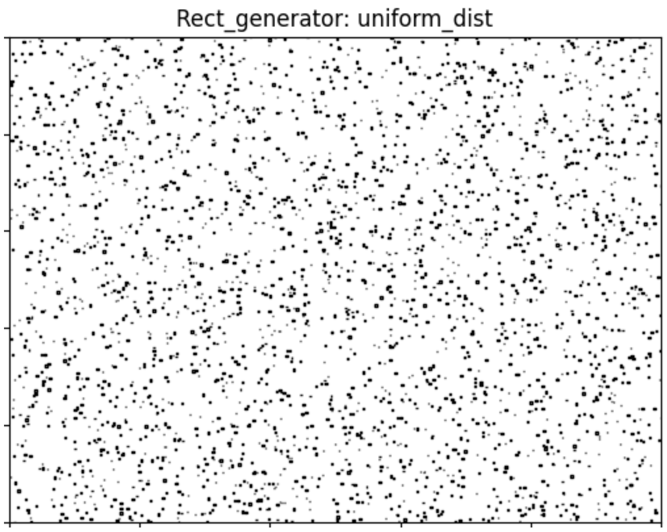

Fig. 14. Uniform Distance Distribution

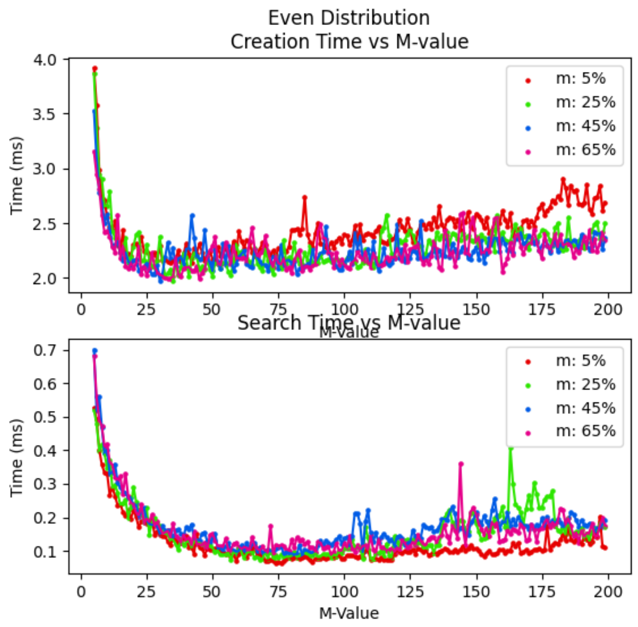

Fig. 15. Even Distribution Performance

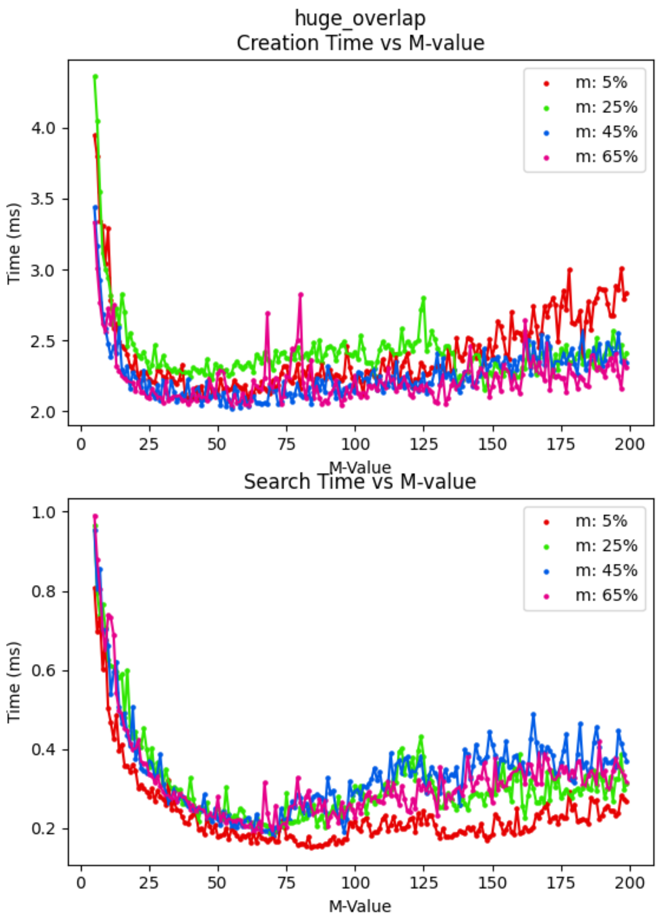

Fig. 16. Huge Overlap Performance

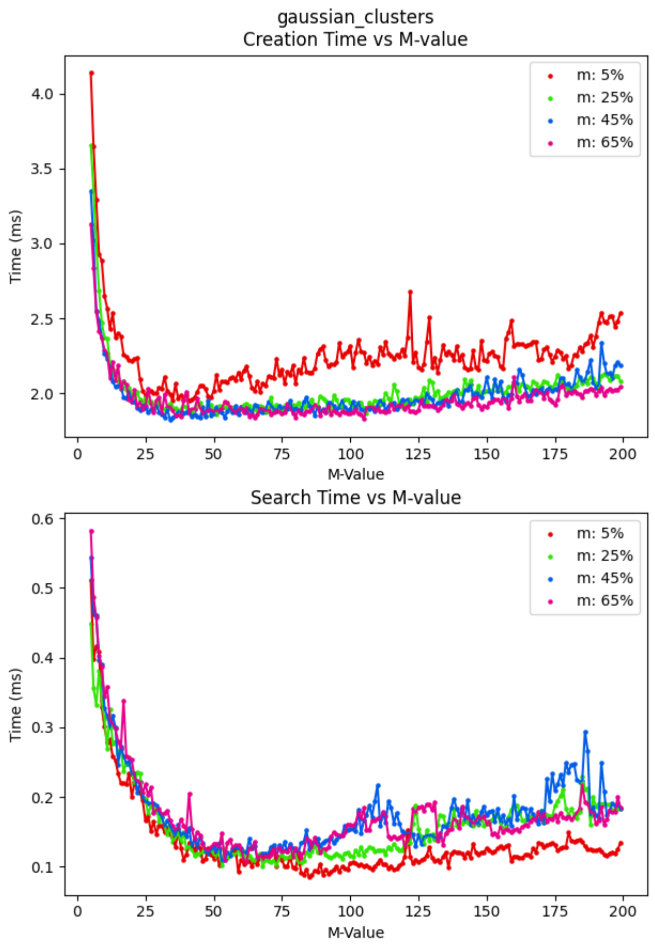

Fig. 17. Gaussian Clusters Performance

VII. Performance Conclusion

For each different distribution of data, very low M values

$$0 < M < 25$$

result in poor search and creation results. A reasonable M value for each of these datasets could be

$$50 < M < 150$$

Anything above 150 loses performance. When Little-m values are low (5%), it results in excellent search performance. This is likely because high m-values assign rectangles in a sub-optimal way, whereas low m-values do not usually need to assign rectangles to a group to fulfill the m-value minimum requirement. High m-values, such as 65%, result in the best creation performance, but fall behind on search performance. In conclusion, low m-value are good for search performance, and high m-values are good for creation performance.

VIII. Generalization to 3 dimensions

R-Trees are an n-dimensional data structure, so extention to 3 dimensions is very easy. There are only a few methods that need to be changed.

- Area: Area is Length*Width*Height, rather than Length*Width
- Intersection: Includes Z axis
- MBR: Includes Z axis
- LinearPickSeeds: Check for far away rectangles along Z axis too.

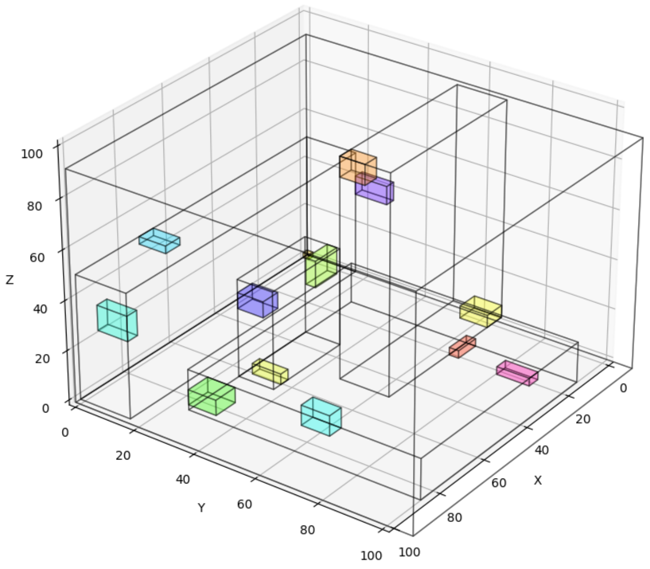

Fig. 18. 3d Box R-Tree

A. Circle Trees

It is also possible to modify an R-Tree to use other minimum bounding shapes instead of rectangles. Only the intersection and minimum MBR calculator function need to be changed.

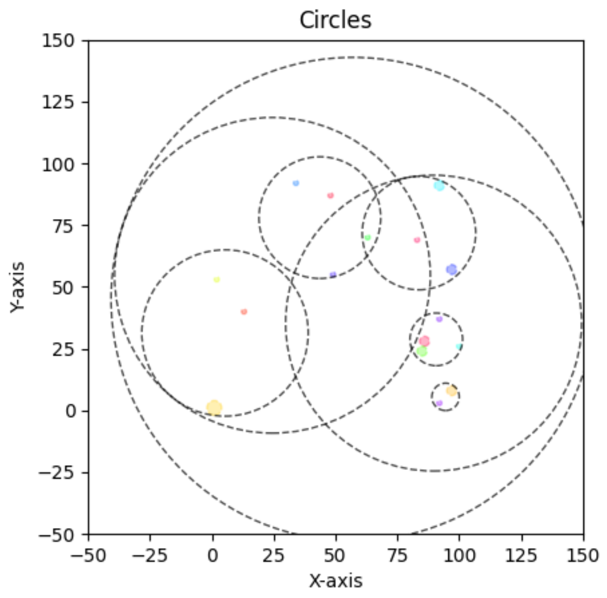

Fig. 19. Circular R-Tree demo

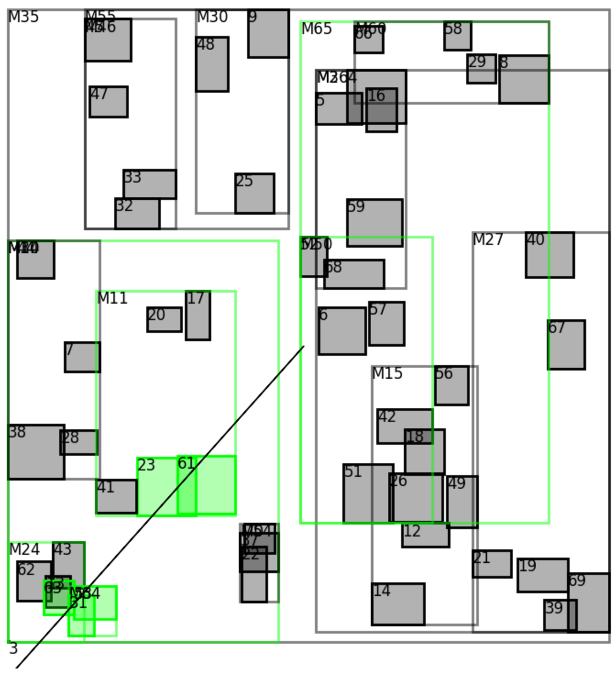

Fig. 20. Raycast-Rectangle Example

IX. Raycast-Rectangle Collision

The search function can also be modified to any generic object-rectangle or object-minumum bounding object collision, such as raycast-rectangle.