ALGORITHM TO PREVENT COLLISIONS BETWEEN ROBOTIC BEES

Juliana Lalinde
Universidad EAFIT
Colombia
jlalindev@eafit.edu.co

Isabel Urrego
Universidad EAFIT
Colombia
iurregog@eafit.edu.co

Mauricio Toro
Universidad EAFIT
Colombia
mtorobe@eafit.edu.co

ABSTRACT
Bees are fundamental for the proper development of an ecosystem. Their population decline is an alarming phenomenon we are currently facing, and it has brought popularity to the idea of developing robotic bees, that can fulfill their functions. A key factor in the execution of this project is constant and effective monitoring that ensures effective distribution.

This distribution determines the benefit that can be obtain, but it depends of a proper execution: there can not be collision between bees that move in the same three-dimensional space.

This work is an evidence of two possible time-efficient algorithms that warn about bees in collision risk, one based on the concept of spatial hashing and another one that uses heap trees, analyzing the data of bees distributed among the city of Bello in Colombia. The advantages and disadvantages of each algorithm analyzed in this work, helped us to define in which cases using one algorithm might be more efficient than using the other.

Keywords
Collisions, Hashing, Hash Table, Hash Functions, Static Hashing

ACM CLASSIFICATION Keywords
CCS → Theory of computation → Design and analysis of algorithms → Streaming, sublinear and near linear time algorithms → Bloom filters and hashing

1. INTRODUCTION
Bees have been categorized as one of the most important animals for ecosystems. They pollinate different plants, which contributes to the reproduction of different species and the production of aliment vital for living beings. There are cases in which an entire species reproduction depends on the behavior of bees.

Nowadays, various factors have led to their population decrease, putting at risk many species and ecosystems. This has alerted specialists and has brought the development of ideas to solve the problem. One key idea is the invention and use of robotic bees.

These robotic bees would complete the same functions of real bees, becoming a help to ensure the constant reproduction of different plants. Achieving this present a challenge: bees need to be properly programmed and evaluated to be actioned, a process that brings need factors to be considered.

One of the most important factors to take into account is the risk of collisions between the bees, and the fact that in order to optimize their functions bees need to have a significant distance while operating [1]

2. PROBLEM
This projects objective is to implement a time efficient algorithm that helps with one of the key factors in the development of robotic bees. The factor that is going to be taken as a base is the risk of collisions.

Knowing this, the main objective is to design a program that allows the detection and notification of bees that are less than one hundred meters apart. The input about the bees will be their coordinates in a three-dimensional space.

3. RELATED WORK
3.1 Spatial Hashing
This algorithm is used to detect collisions between objects on both two- and three-dimensional spaces with the help of a data structure known as a hash table. Having a set of elements, hash tables associate proper keys of each element to a value. Additionally, they have a hash function that return an index to be able to access an element in the table [2].

When two elements are saved in the same index in the table then there is a collision, and this is exactly the case that relates hash tables with our bee collision problem.

In Spatial Hashing, the table works as a subdivision of the space and if there is a collision then it means that there are at least two elements in the same subdivision.

This algorithm helps to avoid the use of a brute force solution that would have to compare the distance between every pair of elements that are part of the set given. Comparison is only made between subdivisions that contain more than one element, because the number of elements in the same location indicates a possible collision. Through this process the algorithm guaranties a greater efficiency in the same execution time [3].

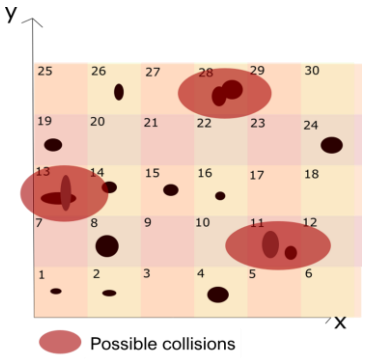


Figure 1: Example of Spatial Hashing in two dimensions

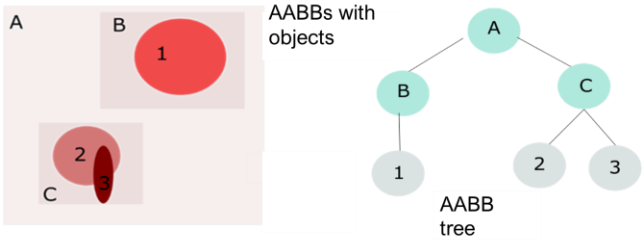
3.2 AABB trees

This method is used to verify if there are possible collisions between different objects, with the help of AABBs.

AABBs are basically boxes that contain objects, this way the axes between different objects align. Instead of taking an object with all its regularities

AABBs son básicamente cajas que contienen objetos y de esta manera los ejes entre diferentes objetos se alinean. Es decir, en vez de tomar un objeto con todas sus regularidades, se toma el objeto como algo que está contenido entre ciertas coordenadas en x, ciertas en y y otro rango en z si se habla de tres dimensiones. Esto es para disminuir notoriamente el número de comparaciones que se deben hacer para identificar una colisión, pues usando AABBs solo se tendría que tener en cuenta un rango en cada eje. [4]

Nevertheless, comparing each AABB within a huge amount of elements might not result very effective and that is why trees should be considered. These data structures are built associating elements (leafs) and connecting them with the same root that would be, in this case, an AABB. The advantage lies in the moment of verifying the possible collisions, it is only need to check if one object has risk with a root and if it is positive then it proceeds to check each leaf connected to the root to find where the collision is. Finally, there would not be comparisons between each pair of AABB, but only in a reduced amount of them based on the tree that was built.



por medio de asociaciones de AABBs. [2]

Figure 2: Example of the use or AABB trees

3.3 R-tree

It was formulated by Antonin Guttman in 1984. A tree is a data structure that simulates the shape of a real tree, an empty tree is used to build a R-tree, into which the elements that need to be saved are added.

In the R-tree structure the leaves correspond to the smallest rectangles that contain the geometrical object that need to be explored, an element is inserted on an R-tee with the following process: the object moves down the tree, from the root to the leaves, checking in which of the following rectangles belongs the object. This is done until the element reaches one of the leaves, where it is added. To search for a specific object the same process is followed, except that in the end it returns the leave that contains the searched objects. This procedure is used to identify possible collisions between objects, if by the end of the search there are no objects on the leave you reach then there are no collisions [6].

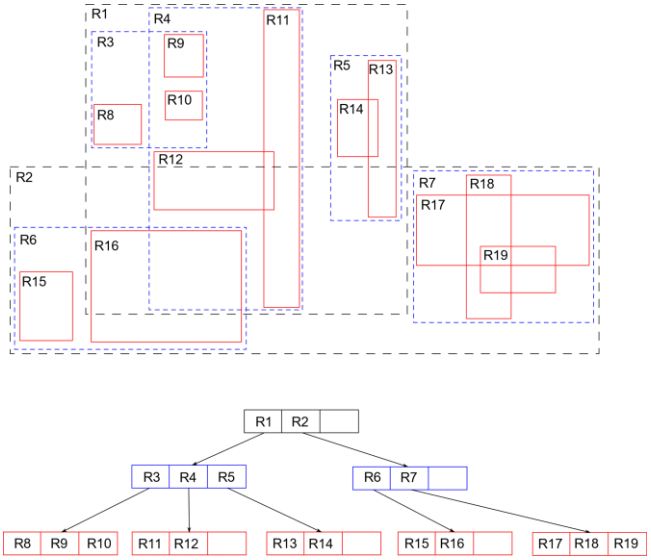


Figure 3: Example of the use of R-trees

3.4 Octree

An Octree is a hierarchical data structure that divides space, it is used for objects located in a three-dimensions. It works by dividing the space in eight cubes, which are also divided into eight cubes individually. This process of division is applied until a designated cube size is reached, if we are working with collisions we would want to reach the smallest object size or the minimal distance between two objects to avoid a collision. The objects that are contained within the same cube are at risk of collision and must be evaluated [7].

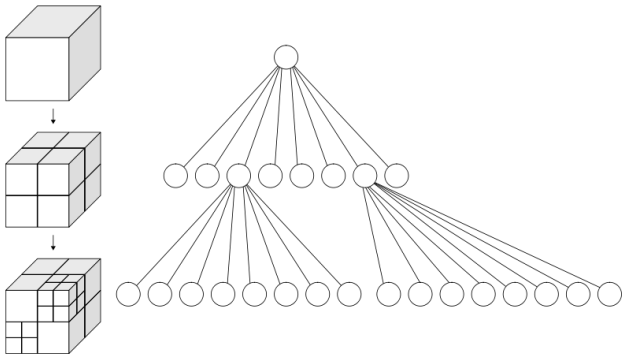


Figure 4: Example of the use of Octrees

3.5 Heap Tree

This data structure is often used as a sorting algorithm known as Heap Sort. To sort the values, you build a binary tree with the condition that the parent is always either bigger or lower than the children. Hence, the root of the tree always represents the maximum or the minimum value among the whole tree depending on the criteria chosen, leading to two different kinds of Heap Tree, the Minimum and the Maximum Heap Tree. [9]

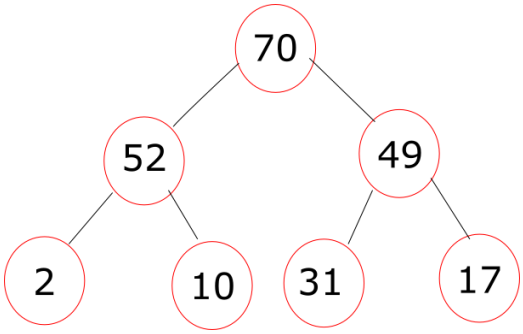


Figure 5: Example of Max Heap Tree

4. Spatial Hashing in three-dimensions

The first data structure that we implemented to solve the problem is Spatial Hashing, based on Hash Tables. It works by dividing the space into cube grids, in which each one has a diagonal of 100, and associating each of the bees into a cube based on their coordinates. This size guarantees that the biggest distance between two bees inside is 100 meters.

The different cubes are associated with an index, that helps identify the elements that are within its range. Every element, in this case the coordinates of a bee, that is added to the grid is referred to as a key and must go through a Hash Function that return a number that matches the index of one of the cubes. Using this division, we can then quickly affirm that all the elements that share the same key are in a collision risk, and we can also lower the number of comparisons between objects that are in different cubes using a simple strategy: we only need to compare the distance between bees that are in adjacent cubes if the number of bees in the cube is 1. Otherwise, there will be a warning of the bees sharing the same cube without comparisons needed.

Although the idea was based on the concept of Hash Tables, it was decided to use another implementation thinking the division of space as a ArrayList three-dimensional array to make the distribution of the bees easier. The linked list allows to save more elements inside one cube grid to collect the bees with collision risk.

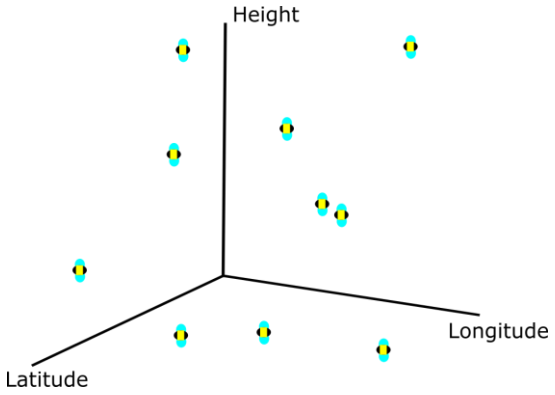


Figure 6: Example of the interpretation of the input

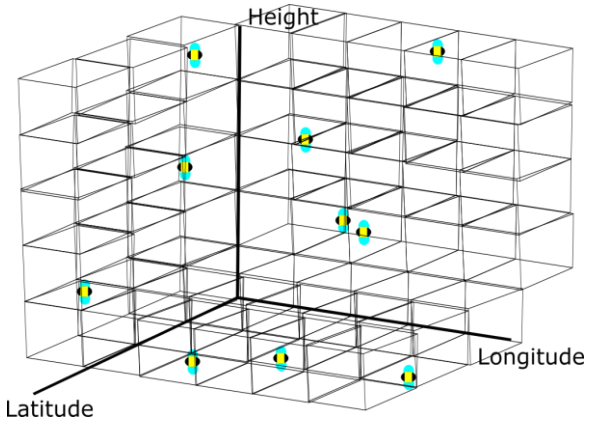


Figure 7: Bees classified by their coordinates using Spatial Hashing

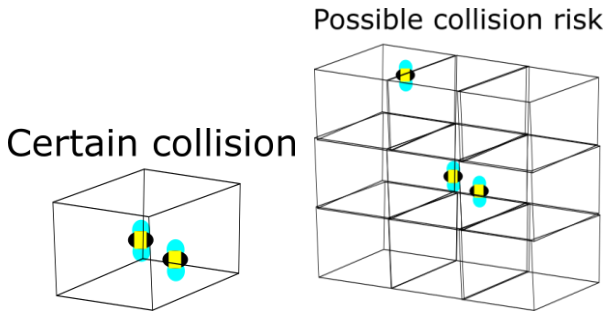


Figure 8: Example of the possible collisions between different cubes

4.1. Operations of the data structure

The main operations used with Spatial Hashing are [8]:

*The images of the operations show a two-dimensional plane (the bees have two coordinates). By the real implementation of the algorithm we will use a Euclid Plane, which adds another dimension, but the operations will do the same as shown.

- **buildMatrix:** used to create a new hash table. Each grid will represent a cube which has different keys representing the coordinates of the bees and a index associated. This function creates a table of the given size and the grids will be the divisions of the plane. Additionally. the creation of a new Hash Table includes a Hash Function which pairs a key (bee) with a grid (cube). To guarantee that the table has the needed size we first read the data and save the maximum and minimum values of each coordinate to determine the dimensions of our three-dimensional array, this with help of another method called readFile.

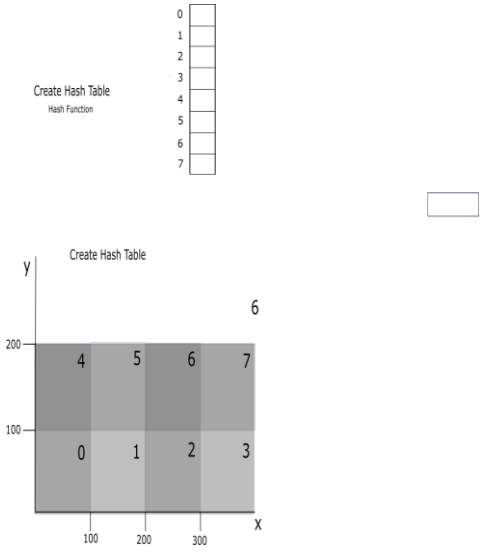


Figure 9: buildMatrix

- **detectCollision:** this function transverses the whole three-dimensional array to look for the bees in collision risk. If the number of bees in one cube grid is greater than 1, it automatically adds it to the list of bees in collision. (Figure 9) On the other hand, if there is only one bee in the cube it must verify the distance with bees on adjacent cubes. (Figure 10)

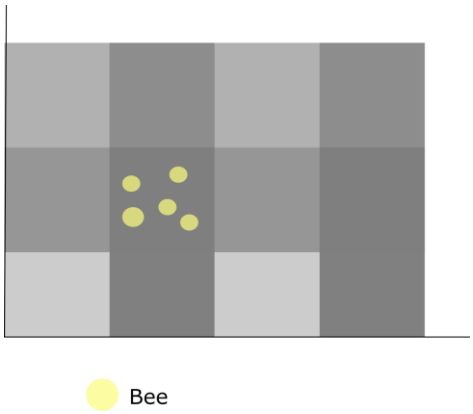


Figure 10: detectCollision

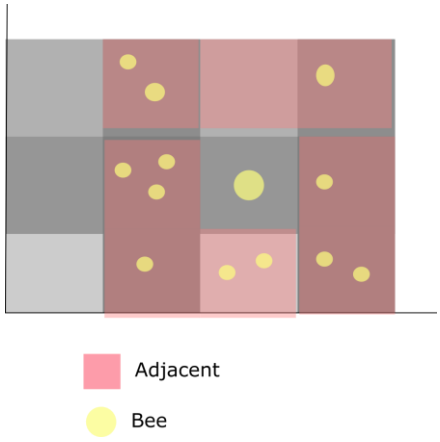


Figure 11: detectCollision

4.2. Design criteria of the data structure

While looking for the solution for the problem we put time efficiency as the main criteria. We read about the different data structures, always thinking how we could reduce the number of comparisons. Our first idea was to separate the bees into different groups defined by their coordinates. In that order, we would have different divisions of the Euclidean space and the comparisons needed would be limited by the number of bees sharing the same cube. That is how we realized that we could use the concept of hash tables to solve the problem.

Spatial Hashing allowed us to divide the space equally and place the bees considering their three coordinates, it also lowered significantly the number of comparisons that needed to be made. Additionally, we decided to work with Lists within the cubes to guaranty that any number of bees could be contained without a limit. This Lists also allow a more dynamic execution for the comparisons that still need to be executed.

4.3. Complexity analysis

Function	Best Complexity	Worst Case Complexity	Explanation of the variables
buildMatrix	O(n)	O(n)	n is the amount of bees
detectCollision	O(xyz)	O(xyz)=O(n^2)	x, y and z are the dimensions of the matrix

Table 1: Complexity Analysis of Spatial Hashing

4.4. Execution Time

DataSets	*ArrayList*	*Hash Table*
10	0 ms	8 ms
100	2 ms	15 ms
1000	24 ms	22 ms
10000	356 ms	46 ms
100000	23282 ms	46 ms
1´000000	--	2833 ms

Table 2: Execution Time of Spatial hashing

4.5. Memory used

DataSets	*ArrayList*	*Hash Table*
10	9.92 MB	11.26 MB
100	10.59 MB	12.61 MB
1000	11.27 MB	13. 95 MB
10000	32.82 MB	28.04 MB
100000	773.975 MB	70.73 MB
1´000000	--	474.59 MB

Table 3: Memory used

4.6. Result analysis

These results prove the use of Spatial hashing as optimal solution for this problem if we talk about memory and time in comparison to ArrayLists.

However, there are some cases where the memory and time complexity might increase. For example, if we have one bee on the position [x] [y] [z] and the rest is in [0][0][0] there would be a lot of memory wasted in cube grids that are empty. Additionally, the need to verify the adjacent cubes would lead to a worst-case time complexity of O(n^2) where n is the number of bees.

Even if our principal objective was time efficiency and not memory, we see that a hash table is not a good option if memory is wanted to be saved. It is only a good option if all the bees are in a small space and there is less memory wasted in amount of cube grids empty.

5. Heap Tree

Subsequently, we thought about another possible data structure that would lower the use of memory significantly, heap trees. Compared to hash tables, the memory used to store information in heap trees does not depend on the space occupied by the bees but on the number of bees. Therefore, it guarantees more efficiency in memory usage.

Heap trees are binary trees that have a very special property: the parent node is always either smaller or bigger than the child node. This property is also applied in heap sort, a very efficient sorting algorithm.

To solve the problem with heap trees you sort the information of the coordinates of the bees, first in x, then y and z. This way, you can relate the coordinates of bees by their distance and evaluate only a set of bees that are at risk of colliding. This search for collisions is done by three different stages (they are explained in the following subsection) and is done using personalized Priority Queues. Priority Queues are queues that work like a heap tree, when we talk about them being personalized we are referring to the comparator that is applied to the data.

The bees that make it successfully trough the three operations are considered collisions.

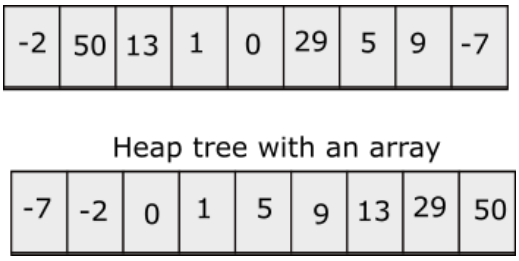


Figure 10: Example of the heap sort based on heap tree

5.1 Operations of the data structure

The data structure uses three main operations calculating collisions:

- **checkX:** This operation organizes the bees by their x coordinate, this way they are organized in an array in which the x increases as you move along. A reference bee is the picked and the following bees are compared to it until the distance between their x coordinate is bigger than 100 meters. After this process is done the method calls checkY.

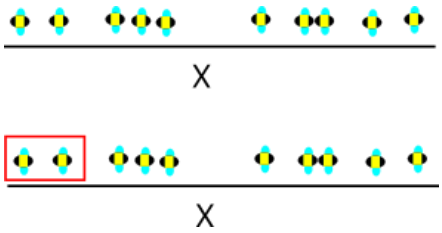


Figure 11: Example of the sorting of the bees with the x coordinate

- **checkY:** This operation allows the bees that are in possible risk of collision, based on their x coordinates, to be compared by their y coordinates. The heap tree that is used to organize the bees is based on the difference between the y coordinates, but the possible collisions are examined by both their x and y coordinates. After this process is done the method calls checkZ().

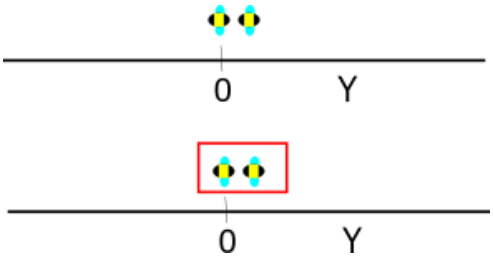


Figure 12: Example of the sorting of the bees with the y coordinate

- **checkZ:** This operation allows to check if the bees that are in risk of collision by their x and y coordinate are in actual risk of colliding. The heap tree used to organize the bees is based on the difference between the z coordinates, and the collision is evaluated by the three coordinates given.

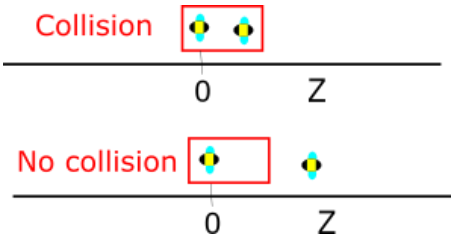


Figure 12: Example of the detection of the collision with the z coordinate

5.2. Design criteria of the data structure

When we began designing our program for the detection of possible collisions we were faced with a very interesting challenge: how well would spatial hashing work if the space the bees were in was very big? The key to the success of our first data structure was that the coordinates of the bees we were working with were limited to a very specific and small space, this allowed the efficient creation of a three-dimensional matrix that had almost no empty spaces.

Thinking of future related works, we thought of designing a new data structure that did not depend on the area of the space the bees were in, we decided to use heap trees to organize the bees and only compare those who were near each other based on their separate coordinates.

5.3. Complexity analysis

Even through there are different operations in the structure the series of calls between methods that is done in the algorithm binds them together in a single function, detectCollision.

Function	Best Complexity	Worst Case Complexity	Explanation of the variables
detectCollision	O(n log n)	O(n^2)	n is the amount of bees

Table 4: Complexity Analysis of Heap Tree

5.4. Execution Time

DataSets	ArrayList	Hash Table	Heap Tree
10	0 ms	8 ms	1 ms
100	2 ms	15 ms	1 ms
1000	24 ms	22 ms	35 ms
10000	356 ms	46 ms	395 ms
100000	23282 ms	46 ms	51109 ms

Table 5: Execution time of Heap Tree

5.5. Memory used

DataSets	ArrayList	Hash Table	Heap Tree
10	9.92 MB	11.26 MB	7.06 MB
100	10.59 MB	12.61 MB	7.38 MB
1000	11.27 MB	13. 95 MB	9.01 MB
10000	32.82 MB	28.04 MB	13.2 MB
100000	773.975 MB	70.73 MB	42 MB

Table 6: Memory used

5.6. Result analysis

These results prove that the use of heap trees allows a better use of the memory, but they also prove that in this specific case (where there are a big number of bees in a small space) the data structure is not the most appropriate to use. This solution can be explored and worked on for future and different case, for example, when there are fewer bees at larger distances.

6. CONCLUSIONS

To sum up, it was shown that there are optimal alternatives to verify the collision risks within a group of bees, more efficient than the trivial solution of comparing every pair of bees. At the beginning it was showed different data structures that were related to the aim of this work, but only two of them were implemented and deeply analyzed, spatial hashing and heap sort.

The hash table in one hand lowered significantly the number of comparisons needed and guarantees to be very efficient if the bees are distributed not to far from each other. Thus, if we have a huge area to cover the heap trees do not waste memory because the usage of memory depends on the number of bees, what would be more optimal to solve the problem within huge areas.

The data structure that we found most useful for the detection of possible collisions within a designated area was the Spatial Hashing. An improved version of the Heap Tree might help solve this problem, using new concepts and languages that we have yet to learn.

This work was an evidence of the huge impact that technology can have in the nature, showing a solution of how robotic bees that might ensure the reproduction of several species, should be programmed.

6.1. Future work

To improve the impact of the robotic bees, it would be better to find an algorithm in the future that has a worst case time complexity of O(n) or at least O(nlogn). That would help the bees to detect they collision risks faster and avoid them from colliding.

Additionally, this solution takes the data of bees in a specific instant and does not consider the fact that the bees are dynamical object constantly moving and a future work might look for a solution in real time.

ACKNOWLEDGMENTS

We thank Juan Guillermo Lalinde for his assistance and comments that greatly improved the manuscript.

REFERENCES

1. Klein, K. Robotic bee could help pollinate crops as real bees decline. *NewScientist*. Retrieved on August 26 2018: https://www.newscientist.com/article/2120832-robotic-bee-could-help-pollinate-crops-as-real-bees-decline/

2. Narasimha Karumanchi. 2018. *Data structures and algorithms made easy in Java: data structure and algorithmic puzzles*, Madinaguda, Hyerabad: CareerMonk Publications.

3. MacDonald, T. Spatial Hashing. Gamedev.net. Retrieved on August 26 2018: https://www.gamedev.net/articles/programming/general-and-gameplay-programming/spatial-hashing-r2697/

4. James. Introductory Guide to AABB Tree Collision Detection. AZURE FROM THE TRENCHES. Retrieved on August 25 of 2018: https://www.azurefromthetrenches.com/introductory-guide-to-aabb-tree-collision-detection/

5. Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In Proceedings of the 1984 ACM SIGMOD international conference on Management of data (SIGMOD '84). ACM, New York, NY, USA, 47-57. DOI=http://ezproxy.eafit.edu.co:2079/10.1145/602259.602266

6. Jane Wilhelms and Allen Van Gelder. 1992. Octrees for faster isosurface generation. ACM Trans. Graph. 11, 3 (July 1992), 201-227. DOI=http://ezproxy.eafit.edu.co:2079/10.1145/130881.130882

7. Hosam M. Mahmoud. 2000. *Sorting: A Distribution Theory*, Wiley- Interscience Publications.