PREVENCIÓN DE COLISIONES DE ABEJAS ROBÓTICAS

Santiago Albisser
Cifuentes Universidad
Eafit Colombia
salbisserc@eafit.edu.co

Juan Pablo Leal
Universidad Eafit
Colombia
jplealj@eafit.edu.co

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

RESUMEN

En la actualidad debido a distintos factores como lo viene siendo los pesticidas, la deforestación excesiva y el cambio climático, ha generado una reducción drástica en la población de las abejas lo que conlleva un problema grave para los agricultores y para el medio ambiente.

1. INTRODUCCIÓN

"Si las abejas desaparecieran, a la humanidad le quedarían cuatro años de vida". Esta frase fue utilizada por el famoso Albert Einstein a pesar de que pueda sonar como algo drástico lo cierto es que esto conllevaría un enorme desafío para la humanidad. Las abejas cumplen un papel fundamental en el equilibrio ambiente debido a su trabajo que tienen de polonizar, esto es vital para que las especies vegetales puedan subsistir. Últimamente debido a los factores que se dijeron previamente, las abejas han reducido su población, por eso mismo genera problemas a los agricultores y se intenta buscar una solución. Aunque se esté hablando actualmente de sustituir esas abejas que se han" desaparecido".

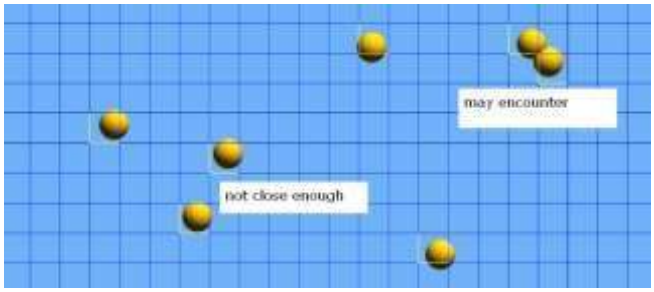
2. PROBLEMA

Con base en la solución dada de implementar drones que puedan cumplir esa función de polonizar, logramos ver ciertos problemas como lo podrían ser un encuentro negativo por parte de las abejas con los drones. Que vayan generando la extinción de una parte mayor de la población restante. Lo que nos lleva a buscar una manera de prevenir que se presenten colisiones. Nosotros percibimos este problema como una curita, ya que a largo plazo se debe hacer énfasis en algo para evitar que las abejas que existen sigan desapareciendo. Pero para esto habría que evaluar distintos factores.

3. TRABAJOS RELACIONADOS

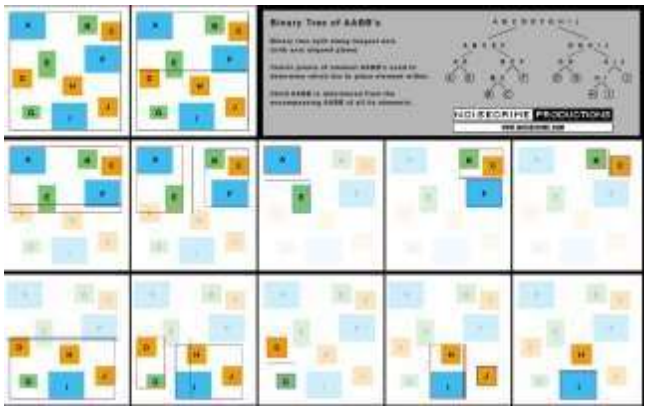
3.1 Spatial hashing

Es una estructura de datos que puede ser trabajada tanto en 2D como en 3D, la cual consiste en una forma de ir almacenando los objetos de tal manera que se puedan manejar grandes cantidades de objetos sin tener que estar recorriendo todo cada vez que queramos comprobar algo. En este método se crean varias tablas de "picar", donde cada llave es una coordinada 2D y contiene el valor de una lista de objetos dentro de esa área (la llave). Para comprobar las coaliciones, las llamadas tablas de "picar" también son conocidas como celdas, donde se ubica dentro de cada una un objeto, y para encontrar la coalición se comprueba el desplazamiento de los objetos dentro de las celdas. Si un objeto ocupara varias celdas debido a su tamaño se debe coger cada esquina y crear un ciclo para comprobar el tamaño de esta y si una de las esquinas se ha movido de celda se puede ir comprobando debido al valor. Por medio de un cuerpo que es guardado, este método funciona para manejar grandes cantidades de objetos sin afectar el desempeño de esta.



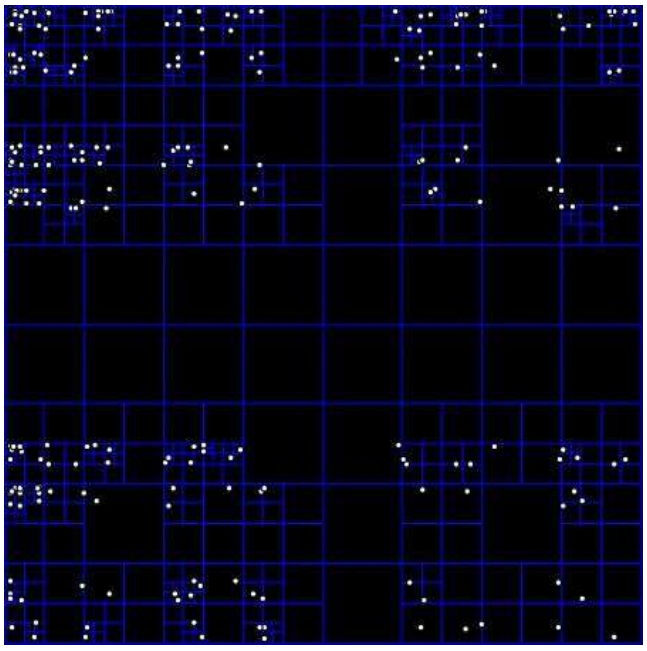
3.2 Dynamic AABB Tree:

Consiste en un arreglo de nodos que es óptimo para el cache que algunos montículos. Es una estructura de datos que consiste de 3 pasos: insertar, eliminar y actualizar. En el método de insertar se crea un AABB grueso para unir datos con un nuevo nodo de hoja. Luego de que se crea el nodo de hoja se hace un recorrido por el árbol buscando un hermano con el cual emparejarse y se vuelve a crear otra ramificación. En el paso de eliminación, los únicos nodos que requieren ser eliminados son las hojas ya que cada ramificación debe de tener 2 hijos válidos. También se puede retirar el padre de la hoja y se puede reemplazar con un hermano y se repite el paso de la inserción. En el paso de actualización se comprueba que la ramificación que se está revisando esté contenida dentro del árbol que inicialmente se creó.



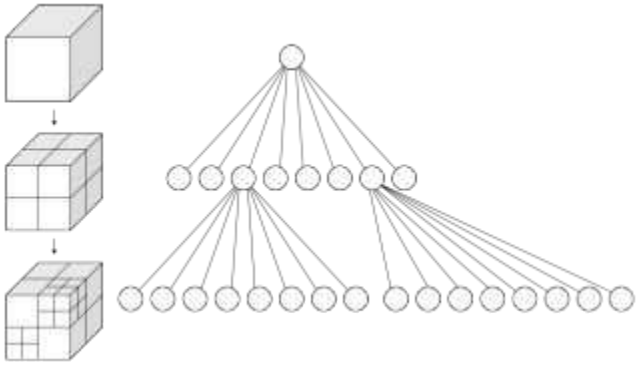
3.3 Quadtree:

Es una estructura de datos que descompone el espacio, se utiliza para descomponer objetos en 2D. Este funciona dividiendo el espacio en 4 nodos en forma de rectángulo o cuadrado, los cuales a la vez se van descomponiendo en 4, para calcular si hay una coalición los objetos que se utilizan representados en círculos, se debe ver si la distancia entre 2 círculos es menor que la suma de sus radios significaría que colisionarían, esta estructura de datos seria eficaz dependiendo de la cantidad de objetos que sean utilizados, pero si se utilizan muchos objetos esto se va a volver un gasto de datos innecesarios, ya que cada vez se van dividiendo los nodos en más nodos. Lo que sería demasiada memoria utilizada.

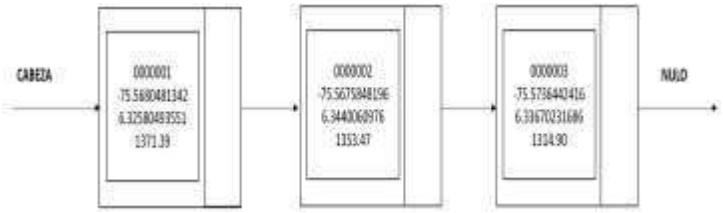


3.4 Octree

Esta estructura de datos es similar al quadTree en algunos aspectos, pero se caracteriza por desarrollarse en una dimensión 3D y se parte un nodo dentro de 8 hijos, dentro de los cuales están los objetos. Para encontrar el riesgo de colisión entre los objetos lo que se debe hacer es que los objetos que queden dentro de un mismo hijo se analiza su riesgo, de esta manera se obtiene una estructura de datos mucho más eficaz.



4. DISEÑO DE LA ESTRUCTURA DE DATOS



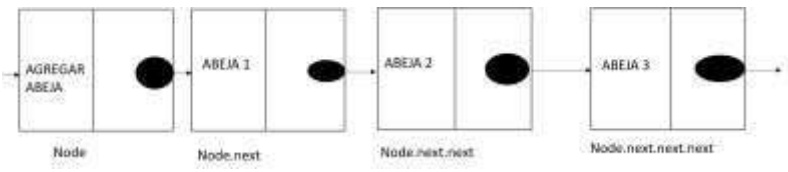
Es una lista enlazada de abejas robóticas. Una abeja es una clase que tiene código de la abeja, coordenada x de la abeja, coordenada y de la abeja y coordenada z de la abeja.

En el código del repositorio se puede evidenciar el uso de esta estructura.

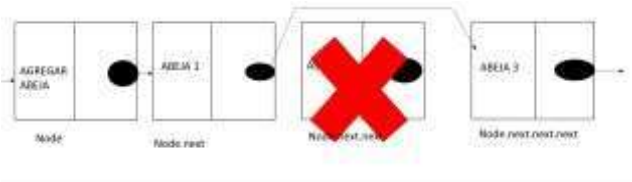
https://github.com/salbisser105/ST0245-032/blob/master/proyecto/PrevencionColisiones.java

5. DISEÑO DE LAS OPERACIONES DE LA ESTRUCTURA DE DATOS

Para el método agregar abeja la lista funcionaría así:



Para el método eliminar abeja la lista funcionaría así:



6. CÁLCULO DE COMPLEJIDADES

Método	Complejidad
leerArchivo()	$O(n)$
detectarColisiones()	$O(n^2)$
guardarArchivo()	$O(n)$

7. CRITERIOS DE DISEÑO DE LA ESTRUCTURA DE DATOS

Cambiamos en el código que ya estaba escrito el ArrayList por un LinkedList. Primero que todo, estas dos estructuras de datos son similares entonces permiten que se trabaje de forma similar y sencilla con cualquiera de las dos. Además, nuestra idea fue reducir el tiempo de ejecución y al usar la lista enlazada nos reducía el tiempo de ejecución y la complejidad del algoritmo ya que inicialmente se encontraba $O(n^3)$ debido a que el método agregar del ArrayList tiene complejidad de n y al cambiarlo por una lista la complejidad se volvió una constante y el algoritmo entero quedo con una complejidad de $O(n^2)$. Debido a esta razón, el algoritmo entero se vuelve más efectivo aunque no lo suficiente ya que sigue siendo un tiempo de ejecución muy alto. Para la entrega #3 esperamos poder mejorarlo y llegar a dejar la complejidad en $O(n * logn)$.

8. RESULTADOS OBTENIDOS DE LA PRIMERA SOLUCIÓN

8.1 Para LinkedList

	Para 10 abejas	Para 100 abejas	Para 1000 abejas	Para 10000 abejas
leerArchivo()	0 ms	0 ms	10,47 ms	19,7 ms
detectarColisiones()	0 ms	0 ms	4,7 ms	646,1 ms
guardarArchivo()	0 ms	0 ms	16,35 ms	1048,9 ms

Tabla 1

	Mejor tiempo	Peor tiempo	Tiempo promedio
leerArchivo()	0 ms	32 ms	7,54 ms
detectarColisiones()	0 ms	719 ms	162,7 ms
guardarArchivo()	0 ms	1078 ms	266,31 ms

Tabla 2

8.2 Para ArrayList

	Para 10 abejas	Para 100 abejas	Para 1000 abejas	Para 10000 abejas
leerArchivo()	0 ms	0 ms	7,05 ms	21,05 ms
detectarColisiones()	0 ms	0 ms	7,05 ms	625,2 ms
guardarArchivo()	0 ms	0 ms	17,1 ms	1044,4 ms

Tabla 3

	Mejor tiempo	Peor tiempo	Tiempo promedio
leerArchivo()	0 ms	32 ms	7,025 ms
detectarColisiones()	0 ms	672 ms	158,06 ms
guardarArchivo()	0 ms	1094 ms	265,37 ms

Tabla 4

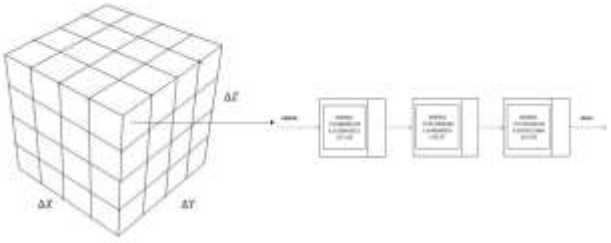
*No se pudo realizar la medida del tamaño de memoria consumida ya que la página web en la que lo ibamos a realizar se encontraba caída.

8.3 Explicación de los resultados

En los resultados se pudo evidenciar que el ArrayList tuvo tiempos menores de ejecución en comparación con la LinkedList. Esto nos lleva a pensar que se eligió mal la estructura de datos ya que en vez de optimizer el tiempo, se

está aumentando y está dejando de ser eficiente. Hay una contradicción ya que según la complejidad, la complejidad máxima de este algoritmo es O(n^2), la cual es menor que la que existía antes que era de 0(n^3). Esto significa que no se obtuvieron los resultados esperados y que en la entrega #3 debemos trabajar con una estructura de datos que logre ser más eficiente y que tenga tiempos menores a los que se tienen actualmente.

9. DISEÑO DE LA ESTRUCTURA DE DATOS FINAL



Para esta entrega final escogimos hacer una matriz tridimensional de listas de adyacencia. Las tres dimensiones de la matriz son Δx,Δy y Δz, que son la distancia que hay entre la posición más grande y la más pequeña para cada dimensión. La diagonal del cada cubito es 100m que es la distancia mínima a la que colisionan. En cada cubo hay una lista enlazada, dentro de esta lista enlazada y en esa lista se almacenan las abejas que colisionan, es decir, las que se encuentran en ese cubo pequeño de la matriz grande.

10. CÁLCULO DE COMPLEJIDADES

Método	Complejidad
leerArchivo()	$O(n)$
detectarColisiones()	$O(n)$
guardarArchivo()	$O(n^3)$

Tabla 5

11. CRITERIOS DE DISEÑO DE LA ESTRUCTURA DE DATOS

Escogimos esta estructura de datos ya que nos permitía tener una complejidad más baja ya que era más fácil de recorrer las posiciones en las que estaban las abejas y agregarlas a esta matriz de listas. Usamos las listas ya que la complejidad para agregar es constante y hacía que no se multiplicara por el número de veces que ya se repetía el ciclo. Los tiempos de ejecución bajaron y nos permiten afirmar que esta estructura de datos que usamos fue clave para reducir la complejidad del algoritmo.

12. RESULTADOS OBTENIDOS DE LA SOLUCIÓN FINAL

	Para 10 abejas	Para 100 abejas	Para 10000 abejas
detectarColisiones()	0 ms	0 ms	2,55 ms

Tabla 6

	Mejor tiempo	Peor tiempo	Tiempo promedio
detectarColisiones()	0 ms	16 ms	0,63 ms

Tabla 7

13. CONCLUSIONES

De este trabajo obtuvimos unos resultados satisfactorios ya que cumplimos con el objetivo de reducir las complejidades de los algoritmos y de bajar el tiempo de ejecución. Nos llevamos la enseñanza de una nueva estructura de datos que permite almacenarlos de forma más eficiente.

Al comparar las tablas de cuando hicimos el código con ArrayLists y LinkedLists podemos ver que la complejidad del algoritmo bajó notablemente y que la reducción de los tiempos de ejecución también fue muy buena.

14. AGRADECIMIENTOS

Nosotros agradecemos por la ayuda con nuestro proyecto a Daniel Mesa, monitor, Universidad Eafit por los comentarios que nos hizo para mejorar nuestro código.

REFERENCIAS

http://www.randygaul.net/2013/08/06/dynamic- aabb-tree/

https://www.gamedev.net/articles/programming/ge neral-and-gameplay-programming/introduction-to- octrees-r3529/

https://www.azurefromthetrenches.com/introducto ry-guide-to-aabb-tree-collision-detection/

http://javaprogrammernotes.blogspot.com/2015/01/why-algorithms-matter-quad-tree-example.html

https://github.com/salbisser105/ST0245-032/blob/master/proyecto/PrevencionColisiones.ja va

https://matematicascercanas.com/2016/03/11/solucion-problema-cubos-pintados/