

LOCALIZACIÓN DE ABEJAS ROBÓTICAS EN 3D:

Optimización considerando el tiempo de algoritmo que verifica la posible colisión de abejas robóticas

Pablo Alberto Osorio Marulanda
Universidad Eafit
Medellín, Colombia
paosoriorom@eafit.edu.co

Verónica Mendoza Iguarán
Universidad Eafit
Medellín, Colombia
vmendozai@eafit.edu.co

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

RESUMEN

En el presente documento se muestra la implementación de una serie de estructuras de datos, donde se verifica la eficiencia de estos en relación a su velocidad de ejecución y gasto de memoria para la prevención de colisiones de abejas robóticas.

Para la ejecución de lo anterior se usan las coordenadas de las abejas en el espacio y una implementación del algoritmo Octree, que verifica cuales objetos, encerrados en una esfera de 100 m, chocarán. La eficiencia del algoritmo utilizado depende en gran medida del método de almacenamiento de las abejas y la comparación de coordenadas.

Palabras clave

Teoría de la computación -- análisis de algoritmos -- análisis de estructura de datos -- comparación de modelos existentes -- diseño de algoritmos -- diseño de estructura de datos -- Colisiones

1. INTRODUCCIÓN

Dada la gran necesidad de una nueva forma de preservación del medio ambiente, mediante la polinización de cultivos con abejas robóticas, que suplen las necesidades de los sistemas globales al igual que abejas reales, nos vemos en la necesidad de prever y anticipar posibles problemas futuros que estas puedan causar.

Un problema a considerar son las posibles colisiones entre las abejas, es decir, aglomeraciones y mala distribución de estas, generando así, posiblemente otros graves daños futuros. Este proyecto se realiza con el fin de implementar un algoritmo que permita solucionar el problema mencionado, mediante la búsqueda de problemas algorítmicos similares y sus soluciones, que proporcionen diferentes percepciones para una mejor y más completa solución de este.

2. PROBLEMA

El problema que se busca solucionar mediante la ejecución de este proyecto es la instauración y optimización (priorizando el tiempo) de un algoritmo que, a través de las

coordenadas geoespaciales de cada abeja, prevenga las colisiones entre ellas, evaluando las abejas que se encuentran a 100 o menos metros entre ellas. Este problema sugiere la solución a los posibles que puedan surgir al momento de polinización de cultivos y otros procesos ejecutados por las mismas.

3. TRABAJOS RELACIONADOS

3.1 Quadtree

Su idea básica es dividir sucesivamente un plano en ambas direcciones para obtener cuadrantes, llegando así a una subdivisión recursiva del espacio[4]. Cuando un cuadrante contiene datos se denomina como área negra, los vacíos como área blanca y las áreas que contienen a ambos son áreas grises[4]. De esta manera, se verifica si cada cuadrante se encuentra lleno, vacío, o con contenidos de ambos tipos. Así, los cuadrantes parcialmente llenos se subdividen recursivamente de nuevo en cuadrantes llenos y vacíos, todo esto hasta un límite determinado. Lo anterior es ejecutado mediante una subdivisión arbórea, donde las estructuras jerárquicas y las propiedades de cada nodo describen el cuadrante que estas representan.

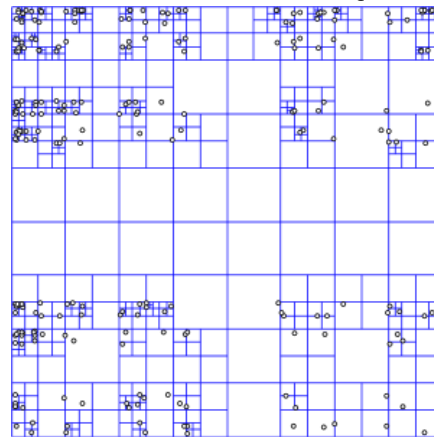


Gráfico 1. Visualización gráfica de los puntos con datos en el Quadtree

3.2 Bounding Boxes-Sprites

Este algoritmo se desarrolla mediante la sobre-inscripción de figuras geométricas sobre los objetos, desarrollando una

especie de "hit-boxes". De esta manera, cada figura representan las dimensiones del objeto en el algoritmo de detección. Este es un algoritmo basado en la serie de McLaurin, donde se calcula la distancia entre dos puntos mediante derivadas aproximaciones matemáticas, con una precisión cercana al 97%[4].

Tales figuras tienen entonces lados paralelos a los ejes x e y (z para 3 dimensiones), que son lo suficientemente grandes para que el modelo completo quede dentro. El centro de estos determinan su posición. Con la posición de cada objeto en el espacio se puede hallar la distancia entre ellos y teniendo en cuenta que , para que no hayan chocado, esta distancia debe ser mayor a la suma de las distancias del centro a una arista en cada figura. La selección de si cada objeto vendrá representado por círculos o rectángulos no es arbitraria, por lo que se recomienda que :*Usa rectángulos para aquellos sprites que no necesiten rotaciones, usa círculos para todo lo demás*[5].

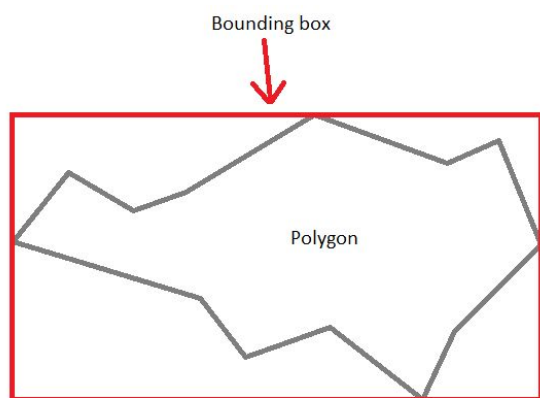


Gráfico 2. Caja con estructura regular que determina un límite para un polígono irregular

3.3 Octree

Un Octree es una estructura de datos en "árbol" en la cual cada nodo interno tiene exactamente 8 "hijos". Las estructuras octree son las análogas tridimensionales de los quadtree bidimensionales [2]. En esta, la región envolvente va a ser un rectángulo tridimensional (comúnmente un cubo), donde se aplica la lógica de subdivisión y se corta la región envolvente en ocho rectángulos más pequeños. Si un objeto encaja completamente dentro de una de estas regiones subdivididas, se arroja hacia abajo hasta la región que contiene ese nodo. Seguidamente, se continúa subdividiendo recursivamente cada región resultante hasta que se cumpla una de las condiciones de interrupción [4].

Las estructuras octree dividen alrededor de un punto. En una región punto octree, el nodo almacena un punto tridimensional explícito, el cual es el "centro" de la

subdivisión para ese nodo; el punto que define una de las esquinas para cada uno de los ocho hijos [3].

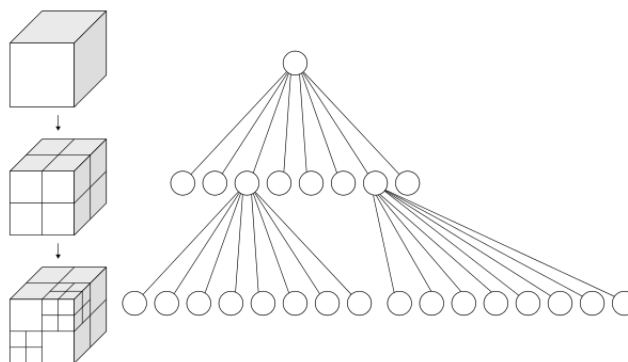


Gráfico 3. Diseño de la estructura arbórea de el Octree depende de la estructura tridimensional relacionada

3.4 Árboles BSP (Binary Space Partitioning)

Este es un método eficiente para calcular las relaciones de visibilidad entre un grupo de polígonos (estáticos) observados desde una perspectiva arbitraria[6]. El árbol se genera tomando cualquier polígono como la raíz . Desde ese subespacio se divide el espacio total en dos cuadrantes, donde uno contiene los polígonos delante de él y otros los que se encuentran tras él . Si un polígono pertenece a ambos subespacios es dividido por el plano, y si otro polígono pertenece al plano raíz se pone en cualquier subespacio[6]. De esta manera, los subespacios "hijos" se dividen nuevamente de forma recursiva para la creación de nuevos subespacios. El nodo tiene llamados recursivos hasta que cada polígono tiene un nodos, es decir, un subespacio único. Así se podrían evaluar los nodos que no son únicos en cierto momento de la recursión, verificando que estos, dado que comparten un subespacio específico, por lo tanto, no les corresponde un subespacio único.

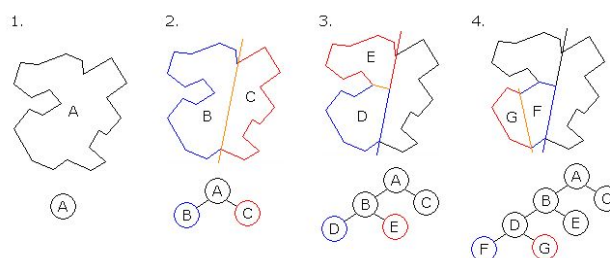


Gráfico 4. Partición del plano ejecutada por el BSP

4. Diseño de algoritmo para prevenir colisiones

Comparación de algoritmo que tiene implementación con ArrayList en colación con uno que implementa LinkedList, ambos usando como forma de búsqueda la fuerza bruta.

Lista Peligro (abejas con riesgo de colisión)

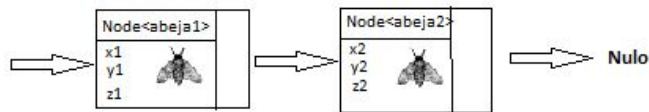


Gráfico 5. Diseño de la LinkedList para almacenar las abejas

Para el desarrollo del algoritmo se va a comparar cada abeja con el resto de abejas en el arreglo, de esta manera se verificarán todas las posibles colisiones para cada una de las que pertenecen al conjunto. Cuando se determine cuál es el conjunto de abejas que posiblemente se van a chocar, estas se almacenarán en una linkedList, para que los procesos posteriores sean más eficientes que como lo serían con arraylist.

4.1 Operaciones de la estructura de datos

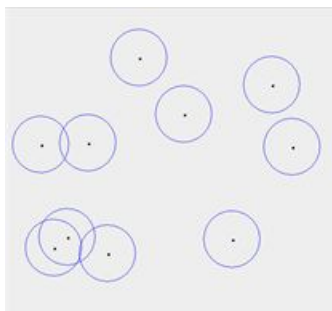


Gráfico 6. Conjunto de 10 abejas

Operación 1:

abeja1	abeja2	abeja3	abejan
x1	x2	x3	xn
y1	y2	y3	yn
z1	z2	z3	zn

Gráfico 7. Este gráfico muestra la operación que hace el programa cuando lee el archivo que tiene un conjunto de ubicaciones de abejas, y almacena cada una de las coordenadas en un arraylist de este tipo.

Operación 2: Esta operación conlleva la comparación de cada una de las abejas con las posibles colisiones.(Fuerza bruta)

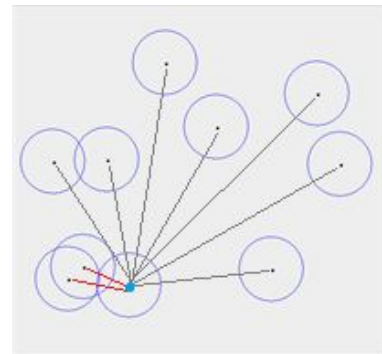


Gráfico 8. Operación de comparación

(En el gráfico anterior se muestran las posibles abejas con riesgo de colisión señaladas con color rojo)

Operación 3: Para esta operación se seleccionan las abejas con posible riesgo de colisión y se almacenan en un linkedlist, de esta manera podemos determinar cuáles son las posibles abejas que van a chocarse sin omitir ninguna colisión.

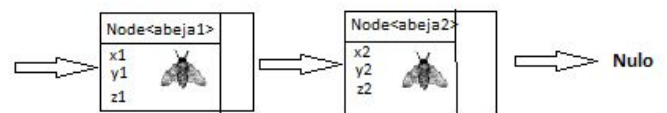


Gráfico 9. Se insertan las abejas con riesgo de colisión en una lista.

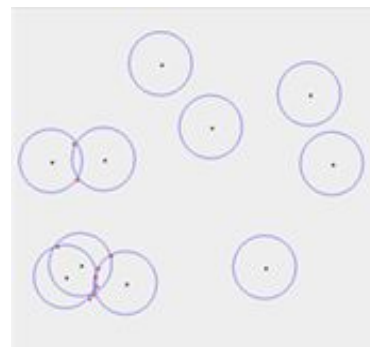


Gráfico 10. Posibles riesgos de colisión para un conjunto de 10 abejas, mediante puntos rojos en las intersecciones de los campos (100 m) a los que pertenecen.

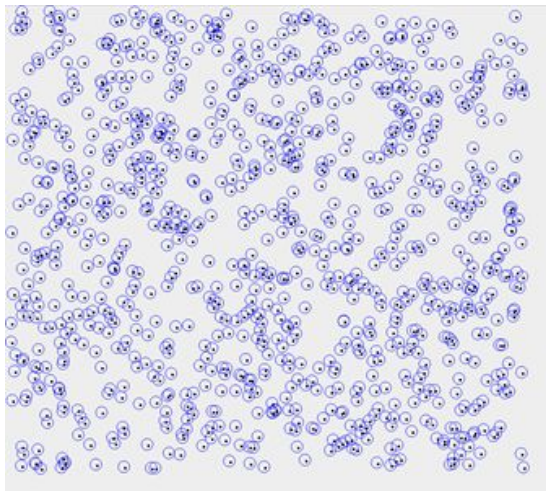


Gráfico 12. Conjunto con 1000 abejas

4.2 Criterios de diseño de la estructura de datos

En el momento de determinar la estructura de datos más eficiente se debe decidir cual método de almacenamiento y comparación será el más eficiente para el programa.

La estructura que tiene el programa para determinar las colisiones es esencial para el buen desarrollo de las operaciones que este ejecuta. Cuando los archivos se almacenan, por primera vez en el arraylist, al momento de ejecutar la operación de comparación es esencial que cada una se compare con el resto para prevenir que se omita alguna colisión. Para el almacenamiento de las abejas que se encuentran ya en peligro, se optó por insertarlas en una lista enlazada, de esta manera la complejidad es constante para esta operación. Notar que siempre se almacena al inicio, dado que el orden final de las colisiones que se almacenan en el archivo es irrelevante para los términos que ahora estamos trabajando. Así que, justamente el almacenamiento de las abejas determina que la complejidad del algoritmo en comparación a si se ejecuta tal acción con un ArrayList.

4.3 Análisis de Complejidad

Método	Complejidad
distancia (entre abejas)	$O(1)$
detectarColisiones	$O(n^2)$
añadirInicio (añade abeja al inicio de una lista)	$O(1)$

insert (insertar ubicaciones de una abeja en un lugar específico de la lista)	$O(1)$
---	--------

Tabla 1. Tabla para reportar la complejidad de los métodos principales.

4.4 Tiempos de Ejecución

Los siguientes tiempos de ejecución corresponden a la estructura de datos **LinkedList**. Por operación se ejecutó un total de 100 repeticiones de las cuales se sacó el promedio de tiempo, esto para los diversos tamaños del problema.

LinkedList	10	100	1000	10000
Lectura archivo	0,06 ms	0,19 ms	1 ms	9,83 ms
Deteccion de colisiones	0,02 ms	0,06 ms	5,99 ms	650,44 ms
guardar archivo	0,24 ms	0,01 ms	0,05 ms	10,9 ms

Tabla 2. Tiempo en milisegundos para las diferentes operaciones usando LinkedList

La siguiente tabla recopila el tiempo promedio de ejecución de cada operación usando **ArrayList**, con, de igual manera al anterior, el tiempo en función de milisegundos.

ArrayList	10	100	1000	10000
Lectura archivos	0,63ms	1,01ms	3,3ms	14,6ms
deteccion de colisiones	0,02ms	0,25ms	28,74ms	3200,73ms
guardar archivo	4,58ms	6,72ms	45,15ms	6917,82ms

Tabla 3. Tiempo en milisegundos para las diferentes operaciones usando ArrayList

4.5 Memoria

En las tablas posteriores se determinará cuál es el consumo de memoria por operación para cada uno de los diferentes conjuntos para cada una de la estructura de datos.

Consumo de memoria(LinkedList)	10	100	1000	10000
Lectura archivo	0,00006 MB	0,00019 MB	0,001 MB	0,00983 MB
Deteccion de colisiones	0,00002 MB	0,00006 MB	0,00599 MB	0,65044 MB
guardar archivo	0,00024 MB	0,00001 MB	0,00005 MB	0,0109 MB

Tabla 4. Consumo de memoria por operación para LinkedList

Consumo de memoria(ArrayList)	10	100	1000	10000
Lectura archivo	0,00063 MB	0,00101 MB	0,0033 MB	0,0146 MB
Deteccion de colisiones	0,00002 MB	0,00025 MB	0,02874 MB	3.20073 MB
guardar archivo	0,00458 MB	0,00672 MB	0,04515 MB	6.91782 MB

Tabla 5. Consumo de memoria por operación para ArrayList

4.6 Análisis de los resultados

Ahora, sacando los valores máximos y mínimos por operación obtenemos las siguientes tablas para LinkedList y ArrayList Respectivamente

LinkedList	Mejor tiempo	Peor Tiempo	Tiempo promedio	Mejor Memoria	Peor Memoria	Memoria promedio
Lectura archivo	0 ms	16 ms	8 ms	0 MB	0,016 MB	0,008 MB
Deteccion de colisiones	0 ms	836 ms	418 ms	0 MB	0,836 MB	0,418 MB
guardar archivo	2 ms	1090 ms	545 ms	0 MB	1,09 MB	0,545 MB

Tabla 6. Mejores y peores tiempos con su promedio respectivo por operación con LinkedList

ArrayList	Mejor tiempo	Peor Tiempo	Tiempo promedio	Mejor Memoria	Peor Memoria	Memoria promedio
Lectura archivo	0 ms	49 ms	24,5 ms	0 MB	0,049 MB	0,0245 MB
Deteccion de colisiones	0 ms	3736 ms	1868 ms	0 MB	3,736 MB	1,868 MB
guardar archivo	3 ms	8537 ms	4270 ms	0,003 MB	8,537 MB	4,27 MB

Tabla 7. Mejores y peores tiempos con su promedio respectivo por operación con ArrayList

La tabla anterior nos da entonces un panorama general del problema, con su gasto de memoria y tiempo de ejecución por operación para cada estructura de dato que se está analizando.

De esta manera, y con los datos anteriores, podemos concluir y llegar a la generalización de los resultados con las siguientes tablas:

General(Tiempo de ejecución)	ArrayList	LinkedList
Lectura archivo	24,5 ms	8 ms
Deteccion de colisiones	1868 ms	418 ms
guardar archivo	4270 ms	545 ms

Tabla 8. Tiempos de ejecución para cada estructura de datos por operación

General(Gasto de memoria)	ArrayList	LinkedList
Lectura archivo	0,0245 MB	0,008 MB
Deteccion de colisiones	1,868 ms	0,418 MB
guardar archivo	4,27 ms	0,545 MB

Tabla 9. Gasto de memoria para cada estructura de dato por operación

Con todo esto, llegamos, finalmente a la siguiente gráfica



Gráfico 13. Tiempo de ejecución promedio para cada estructura de dato

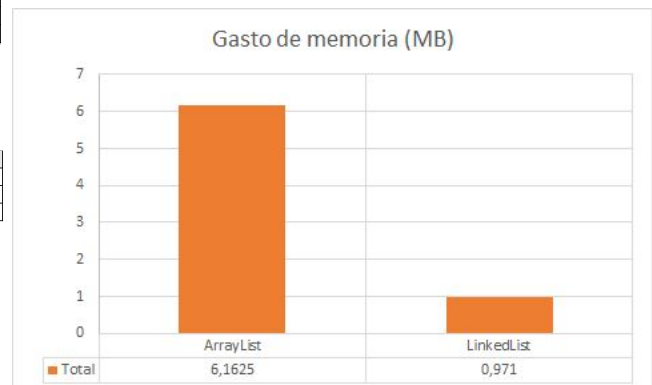


Gráfico 14. Gasto de memoria promedio para cada estructura de dato.

Así entonces concluimos que, para el almacenamiento de las abejas en peligro y las posibles operaciones con este conjunto es preferible el uso de la LinkedList.

5. Implementación de Octree para prevenir colisiones entre abejas.

Para el desarrollo del algoritmo, se divide el espacio en el que se encuentran las abejas en ocho subespacios, de forma similar a la división de un árbol; esto con el fin de que la búsqueda sea más sencilla, y de esta manera minimizar el tiempo de operación.

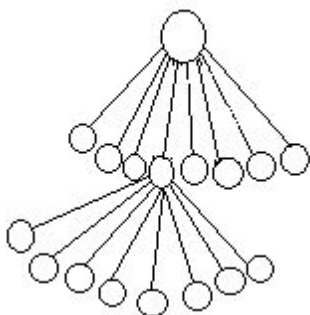


Gráfico 15. Representación de la división del espacio para la búsqueda de posibles abejas a colisionar.

5.1 Operaciones de la estructura de datos

Operación 1: En primer lugar, el programa lee un archivo de texto, donde se encuentran las coordenadas de las abejas. Al almacenar las abejas en forma de puntos 3D (Point3D). a medida que esto se ejecuta, se busca la posición mínima y máxima de una abeja, para utilizar esta información como coordenadas del cubo que se formará para el análisis del espacio.

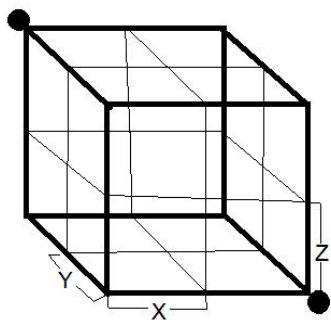


Gráfico 16. Este gráfico muestra la forma en que se toman las coordenadas, y cada punto negro representa la posición mínima y máxima que se toma del espacio.

Operación 2: Con el cubo que representa el espacio ya formado, se procede a agregar cada abeja a un espacio de área mínima. Para esto, se divide el árbol (es decir, el espacio) hasta que el área de los círculos que contienen a cada abeja sea aproximadamente de $100 \times 100 \times 100$.

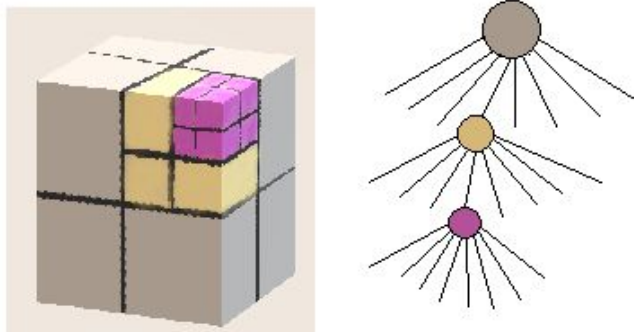


Gráfico 17. Abstracción de la división del espacio.

Operación 3: A continuación, se agrega a cada abeja al área correspondiente, teniendo en cuenta sus coordenadas. Esto es, se agrega la abeja a un espacio que corresponde a un nodo del árbol y cada nodo tiene una LinkedList, entonces la abeja se agrega a la linkedlist que tiene ese nodo de acuerdo a su posición en el espacio.

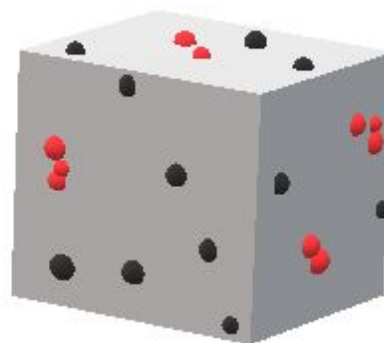


Gráfico 18. Modelo de las abejas en el espacio. Los círculos rojos representan las abejas con peligro de colisión.

Operación 4: Ya teniendo las abejas integradas en el árbol, se procede a introducir en una lista enlazada las listas enlazadas que, para el final, contienen más de una abeja, lo que significa que hay más de una abeja en un espacio asignado.

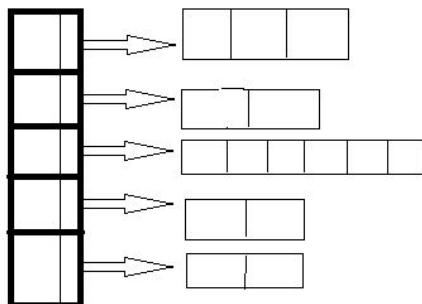


Gráfico 19. Representación de la linkedList que contiene linkedLists con las abejas.

5.2 Criterios de diseño de la estructura de datos

La estructura de datos elegida, divide el espacio recursivamente en ocho subespacios; como la ramificación de un árbol en el que cada nodo equivale a un espacio menor al anterior. Esto permite que la búsqueda sea más exacta pues se analiza cada sitio sin excepciones, además de que la búsqueda es más rápida, el espacio de memoria puede ser menor pues se limita la búsqueda a las dimensiones de un cubo que se crea con las coordenadas de la abeja en el punto mínimo y máximo.

5.3 Análisis de Complejidad

La siguiente tabla muestra la complejidad de las operaciones principales en la implementación de Octree.

Método	Complejidad(O(n))
leerArchivo()	O(n)
detectarColisiones()	O(n) para el mejor de los casos(Abejas en un espacio pequeño) O(nlog(n)) en promedio y para el peor de los casos
guardarArchivo()	O(n)
inicio()	O(m)(Con m el número de veces que se debe sumar 100 a 0 para llegar a un entero arbitrario)
insert()	O(log(n))
returnig()	O(k)(Con k el número de espacios que tengan más

	de dos abejas dentro de ellos)
ListaPeligro()	O(l)(Con l<=n)

Tabla 10: Complejidad de los métodos anexos a Octree

5.4 Tiempos de Ejecución

Los siguientes tiempos de ejecución corresponden a la estructura de datos implementada.

Octree	10	100	1000	10000	100000	1000000
Lectura archivo	0,26 ms	0,27 ms	1,09 ms	10,33 ms	124,14 ms	1143,05 ms
Deteccion de colisiones	0,08 ms	0,08 ms	0,29 ms	2,14 ms	23,27 ms	237,02 ms
guardar archivo	0,14 ms	0,27 ms	0,39 ms	12,13 ms	117,98 ms	1186,36 ms

Tabla 11: Tiempos correspondientes para cada uno de los métodos para cada una de las colecciones de abejas.

5.5 Memoria

En las tablas posteriores se determina el consumo de memoria por operación para diferentes conjuntos de datos.

Octree	10	100	1000	10000	100000	1000000
Lectura archivo	0,00026 MB	0,00027 MB	0,00109 MB	0,001033 MB	0,12414 MB	1,14305 MB
Deteccion de colisiones	0,00008 MB	0,00008 MB	0,00029 MB	0,000214 MB	0,02327 MB	0,23702 MB
guardar archivo	0,00014 MB	0,00027 MB	0,00039 MB	0,001213 MB	0,11798 MB	1,18636 MB

Tabla 12: Memoria usada por cada uno de los métodos para cada colección de abejas

5.6 Análisis de los resultados

Después de haber analizado todos los tiempos obtenidos por cada uno de los métodos primordiales, obtenemos la siguiente tabla promedio, claro está, para la estructura del octree:

Octree	Mejor tiempo	Peor Tiempo	Tiempo promedio	Mejor Memoria	Peor Memoria	Memoria promedio
Lectura archivo	0 ms	1389 ms	694,5 ms	0 MB	1,389 MB	0,6945 MB
Deteccion de colisiones	0 ms	338 ms	169 ms	0 MB	0,338 MB	0,169 MB
guardar archivo	0 ms	1320 ms	660 ms	0 MB	1,32 MB	0,66 MB

Tabla 13: Datos que representan el mejor y peor tiempo del algoritmo mostrado de manera general.

Así, se concluye primero que, el tiempo obtenido por el algoritmo diseñado es, en efecto, mucho menor que por los algoritmos anteriores, esto, sobre todo, por el método de análisis de colisiones diseñado, que, aunque tiene un mayor porcentaje de error, es mucho más eficiente.

Esta eficiencia se ve reflejada no solo en el tiempo, sino en la memoria, debido a que para el caso de la lista enlazada y comparación por fuerza bruta, el heap del compilador usado para ejecutar el código se llenaba completamente, por lo que, no se pudieron tomar los tiempos y memoria correspondientes a el conjunto con 100000 y con 1000000

6. CONCLUSIONES

El desarrollo del proyecto verificó conceptos teóricos tales como la ineficiencia de la fuerza bruta, importancia de la baja complejidad, importancia del uso de estructuras de datos e igualmente lo crucial que es la implementación de algoritmos para el desarrollo y solución de problemas a gran escala.

Para el proyecto, aunque las listas enlazadas parecían ser una opción viable, la intervención del desarrollo de árboles tales como el Octree otorgaron la visión de una mejora al desarrollo del problema.

Es por ello que con los resultados obtenidos, el Octree, por cuestiones de tiempo y memoria (aunque no exactitud) es una opción superiormente viable.

6.1 Trabajos futuros

Una de las opciones para continuar con el desarrollo de este proyecto es mejorar el algoritmo en razón de tener un menor índice de error, esto es, construir un cubo con medidas no proporcionales de largo ancho y alto y desglosar de allí un octree.

Otra posible continuación del proyecto es un cambio del enfoque principal, es decir, una aplicación del mismo concepto del algoritmo pero a un ámbito diferente, tal como el desarrollo de videojuegos, modelación de aerolíneas, e incluso, modelación y simulación del comportamiento de ecosistemas biológicos.

AGRADECIMIENTOS

La finalización de este proyecto nos hace sentir realmente agradecidos por todos los conocimientos que nos han atribuido tanto docentes como compañeros; pues sin ellos la solución obtenida, posiblemente hubiera sido menos satisfactoria. Dichas personas han favorecido nuestro aprendizaje y permitieron que el curso realizado fuera realmente fructuoso. Agradecemos con sinceridad todo acto benévolo hacia nosotros.

REFERENCIAS

1. Artículo tomado de :
<https://es.wikipedia.org/wiki/Quadtree#Uso>
2. Artículo tomado de
<https://es.wikipedia.org/wiki/Octree>
3. Nevela, E. *Introducción a los octrees*, Gamedev, 20 de enero 2014
4. Gallego, J and Pradas, S *Colisiones 2D en los videojuegos*, DCCLA (Departamento de ciencia de la computación e Inteligencia artificial) , 2002.

5. Nickname: ADRIGM , *Teoría de colisiones 2D: Conceptos básicos* , Genbeta (weblog), 29 de junio 2013
6. Universidad de las américas, *Árboles BSP*, Cap. 5, Puebla, México.

