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
paosoriom@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, que, para este caso particular serán: abejas robóticas.

Para la ejecución de lo anterior se usan las coordenadas de las abejas en el espacio y el programa que resulte de la estructura de datos 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 --

1. INTRODUCCIÓN

Dada la gran necesidad de una nueva forma de preservación del medio ambiente y de la supervivencia del ser humano, 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, las aglomeraciones y la 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

El *Quadtree* es un algoritmo que usa un formato de representación 2D para codificar imágenes. 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, todo esto dependiendo que tanto el área intersecta el cuadrante. Así, los cuadrantes parcialmente lleno se subdivide 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.Con esto, conteniendo un grupo de puntos se pueden ejecutar tareas tales como determinar si un punto selecto se encuentra contenido en un grupo dado o finalmente seleccionar un grupo de puntos que se relacionen con algún criterio[1].

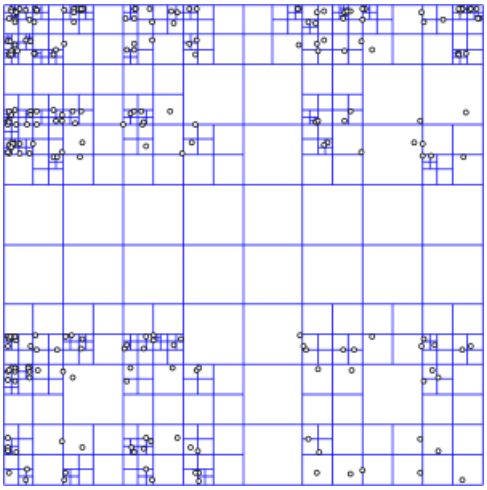

Gráfico 1: Visualización gráfica de los puntos con datos en el Quadtree

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 (por lo general círculos y rectángulos) 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].

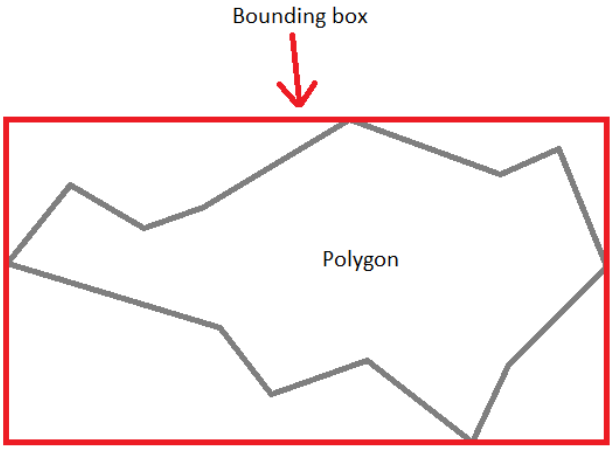

Gráfico 2: Caja con estructura regular que determina un límite para un polígono irregular

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 (empezando en la parte de arriba del árbol) 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].

Los octrees son útiles cuando se tiene que buscar un espacio tridimensional. En particular, se utilizan a menudo en la detección de colisiones eficiente, pero también se producen en algoritmos que funcionan en un espacio 3D más abstracto [4].

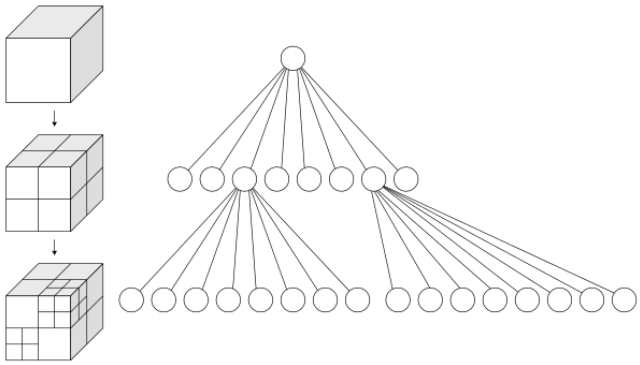

Gráfico 3: Diseño de la estructura arbórea de el *Octree* depende de la estructura tridimensional relacionada

3.4 Árboles BSP (Binary Space Partitioning)

Los árboles de partición binaria del espacio se desarrollaron gracias a Fuchs y Naylor.[6] Este es un método eficiente utilizado 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 de divide el espacio total en dos cuadrantes, donde un 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.

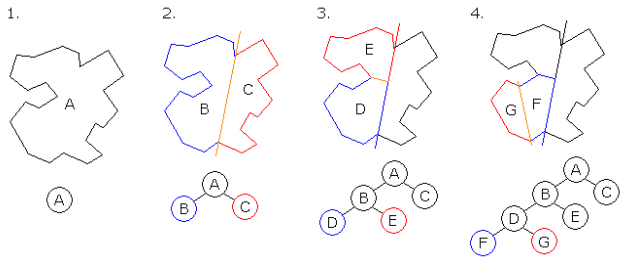

Gráfico 4: Partición del plano ejecutada por el *BSP*

4. Diseño de algoritmo para prevenir colisiones

ListaPeligro abejasConRiesgoDeColision

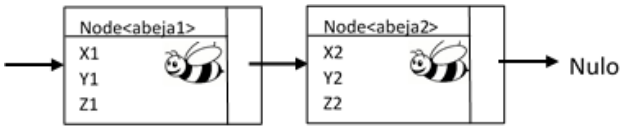

Gráfico 5:Diseño de la LinkedList para almacenar las abejas

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 abejas 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

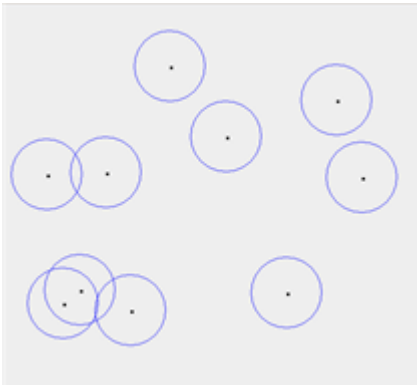

Gráfico 6: conjunto de 10 abejas

operación 1:

abeja1	abeja2	abeja3	abejan

x1 y1 z1	x2 y2 z2	x3 y3 z3	xn yn zn

Gráfico 7: este gráfico explica 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.

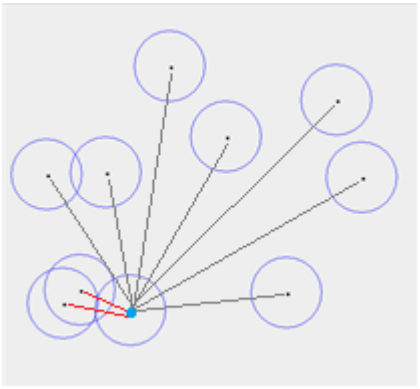

Gráfico 8: Operación de comparación

(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.

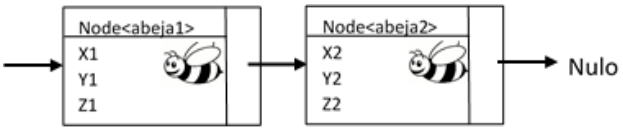

Gráfico 9: se insertan las abejas con riesgo de colisión en una lista.

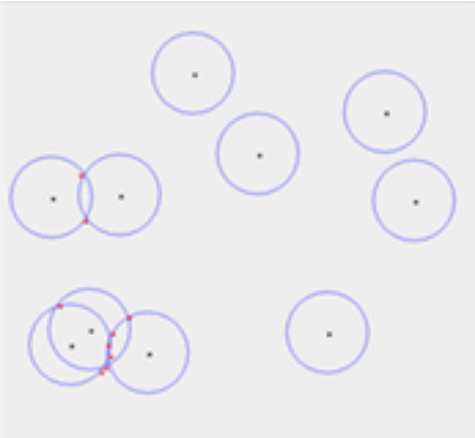

Gráfico 10: Posibles riesgos de colisión para un conjunto de 10 abejas, mediante puntos rojos en las intersecciones de los campos (100 m) a los que pertenecen.

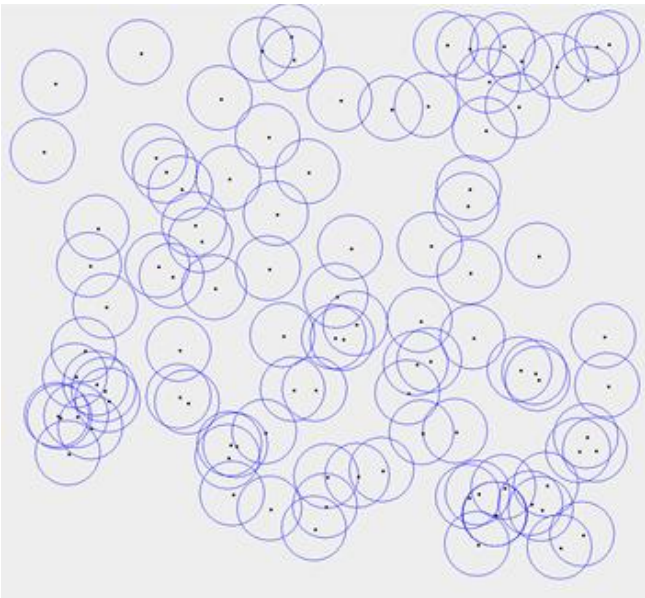

Gráfico 11: Conjunto con 100 abejas

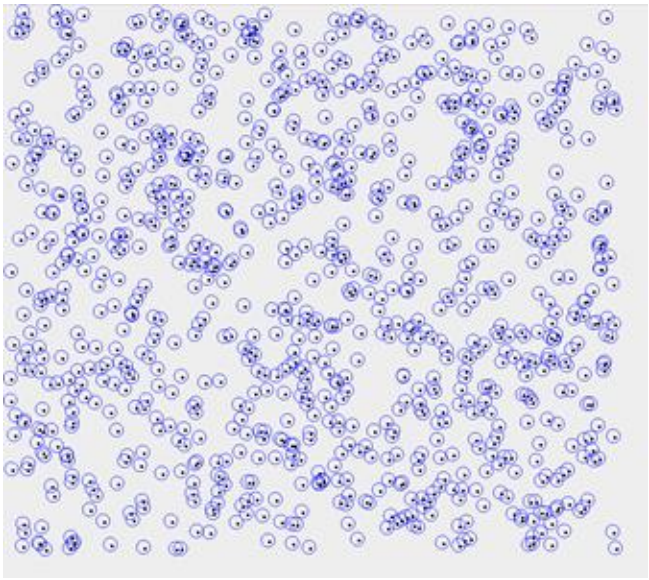

Gráfico 12: Conjunto con 1000 abejas

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)
leerArchivo (con ubicaciones de todas las abejas)	O(n)
detectarColisiones	O(n^2)
guardarArchivo (de ubicaciones de abejas con riesgo de colisión)	O(n)
añadirInicio (añade abeja al inicio de una lista)	O(1)
size (obtener el tamaño de la lista de abejas con riesgo de colisión)	O(n)
insert (insertar ubicaciones de una abeja en un lugar específico de la lista)	O(1)
get (obtener el dato de un, esto es, ubicación de una abeja)	O(1)
removeFirst(Borrar el primer dato de una lista)	O(1)

Tabla 1: Tabla para reportar la complejidad de cada método.

4.4 Tiempos de Ejecución

Los siguientes tiempo de ejecución corresponden a la estructura de datos ejecutada con una LinkedList, los tiempos aquí mencionados están en función de milisegundos. 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 coliciones	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 del programa 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 coliciones	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 coliciones	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 promed
Lectura archivo	0 ms	16 ms	8 ms	0 MB	0,016 MB	0,008 MB
Deteccion de coliciones	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 4: Mejores y peores tiempos con su promedio respectivo por operación con LinkedList

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

Tabla 5: 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 coliciones	1868 ms	418 ms
guardar archivo	4270 ms	545 ms

Tabla 6: 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 coliciones	1,868 ms	0,418 MB
guardar archivo	4,27 ms	0,545 MB

Tabla 7: Gasto de memoria para cada estructura de dato por operación

Con todo esto, llegamos, finalmente a la siguiente gráfica

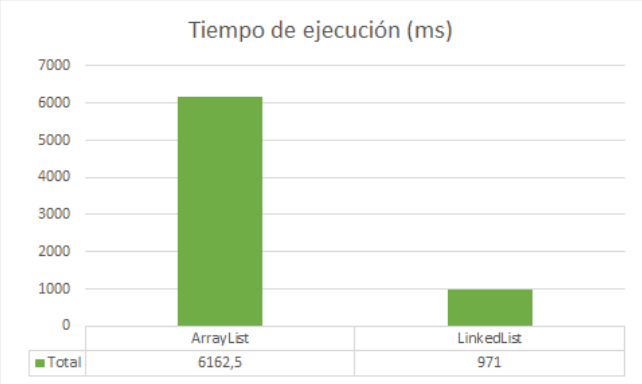

Gráfico 13: Tiempo de ejecución promedio para cada estructura de dato

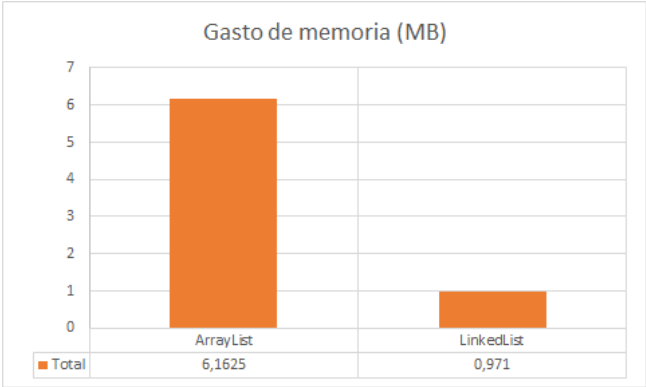

Gráfico 14: Gasto de memoria promedio para cada estructura de dato.

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.

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, DCCIA (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.*