TÍTULO (DESCRIPCIÓN CORTA DEL PROYECTO. ENTRE 8 Y 12 PALABRAS)

Andrés Felipe Tamayo Acevedo
Universidad EAFIT
Colombia
aftamayoa@eafit.edu.co

Jamer José Rebolledo Quiroz
Universidad EAFIT
Colombia
jjrebolleq@eafit.edu.co

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

RESUMEN

Cerca de tres cuartos de las especies que se utilizan en cultivos, desde manzanas hasta almendras, necesitan de la polinización de abejas y otros insectos. Infortunadamente, los pesticidas, la deforestación y el cambio climático han causado que disminuya la población de abejas, causando graves problemas a los agricultores. Un dron que pueda polinizar flores puede funcionar, en un futuro no muy lejano, para mejorar el rendimiento de los cultivos. Como un ejemplo, Eijiro Miyaco del instituto avanzado de ciencia y tecnología industrial de Japón, y sus colegas, han creado un dron que puede transportar polen entre flores." (Tomado de https://www.newscientist.com/article/2120832-robotic-bee-couldhelp- pollinate-crops-as-ral-bees-decline/)

1. INTRODUCCIÓN

Es la justificación de las condiciones en el mundo real que llevan al problema. En otras palabras, es hablar sobre qué va a tratar el documento e incluir la historia de este problema.

Hoy en día, la población de abejas se ha visto muy afectada y reducida por el uso de pesticidas en los cultivos de frutas y vegetales, frente a esta problemática se diseñó un prototipo de dron que hace las funciones de una abeja para así solucionar de una manera rápida y eficaz la falta de abejas.

2. PROBLEMA

En pocas palabras escriban cuál problema que están resolviendo, además de responder ¿para qué resolver este problema?

El problema que vamos a solucionar es el de las colisiones entre estos drones – abejas, desarrollaremos una estructura de datos que detecte colisiones entre las abejas robóticas.

3. TRABAJOS RELACIONADOS

Aquí deberán explicar 4 problemas algorítmicos similares que se encuentren documentados en libros, artículos científicos o sitios web, y dar al menos 1 solución para uno de ellos. NO poner soluciones de tecnología.

3.1 Hash espacial

Un hash espacial es una extensión de 2 o 3 dimensiones de la tabla hash, que debería serle familiar desde la biblioteca estándar o el libro / curso de algoritmos de su elección. Por supuesto, como ocurre con la mayoría de estas cosas, hay un giro.

La idea básica de una tabla hash es que tomes un dato (la 'clave'), lo ejecutes a través de alguna función (la 'función hash') para producir un nuevo valor (el 'hash'), y luego usar el hash como un índice en un conjunto de ranuras ('cubos').

Para almacenar un objeto en una tabla hash, ejecuta la clave a través de la función hash y almacena el objeto en el depósito al que hace referencia el hash. Para encontrar un objeto, ejecuta la tecla a través de la función hash y busca en el depósito al que hace referencia el hash.

Normalmente, las claves para una tabla hash serían cadenas, pero en un hash espacial usamos 2 o 3 puntos dimensionales como claves. Y aquí es donde entra el giro: para una tabla hash normal, una buena función hash distribuye las teclas lo más uniformemente posible a través de las cubetas disponibles, en un esfuerzo por mantener corto el tiempo de búsqueda. El resultado de esto es que las claves que están muy cerca (lexicográficamente hablando) entre sí, es probable que terminen en cubos distantes. Pero en un hash espacial estamos tratando con ubicaciones en el espacio, y la localidad es muy importante para nosotros (especialmente para la detección de colisiones), por lo que nuestra función hash no cambiará la distribución de las entradas.

3.2 QuadTree

El término Quadtree, o árbol cuaternario, se utiliza para describir clases de estructuras de datos jerárquicas cuya propiedad común es que están basados en el principio de descomposición recursiva del espacio. En un QuadTree de puntos, el centro de una subdivisión está siempre en un punto. Al insertar un nuevo elemento, el espacio queda divido en cuatro cuadrantes. Al repetir el proceso, el cuadrante se divide de nuevo en cuatro cuadrantes, y así sucesivamente.

Una gran variedad de estructuras jerárquicas existen para representar los datos espaciales. Una técnica normalmente usada es Quadtree. El desarrollo de éstos fue motivado por la necesidad de guardar datos que se insertan con valores idénticos o similares. Este artículo trata de la representación de datos en el espacio bidimensional. Quadtree también se usa para la representación de datos en los espacios tridimensionales o con hasta 'n' dimensiones.

El término Quadtree se usa para describir una clase de estructuras jerárquicas cuya propiedad en común es el principio de recursividad de descomposición del espacio.

Estas clases, basan su diferencia en los requisitos siguientes:

El tipo del dato en que ellas actúan.
El principio que las guías del proceso de descomposición.
La resolución (inconstante o ninguna).
La familia Quadtree se usa para representar puntos, áreas, curvas, superficies y volúmenes. La descomposición puede hacerse en las mismas partes en cada nivelado (la descomposición regular), o puede depender de los datos de la entrada. La resolución de la descomposición, en otros términos, el número de tiempos en que el proceso de descomposición es aplicado, puede tratarse de antemano, o puede depender de las propiedades de los datos de la entrada.

El primer ejemplo de un Quadtree se relaciona a la representación de un área bidimensional. La región Quadtree que representa las áreas es el tipo más estudiado. Este ejemplo es basado en la subdivisión sucesiva del espacio en cuatro cuadrantes del mismo tamaño. El subcuadrante que contiene datos simplemente se denomina área Negra, y los que no contienen datos se denominan área Blanca. Un subcuadrante que contiene partes de ambos se denomina área Ceniza. Los subcuadrantes Ceniza, que contienen aéreas Blancas y Negras (Vacío y Datos), deben subdividirse sucesivamente hasta que solo queden cuadrantes Negros Y Blancos... (Datos y Vacíos).

Cada cuadrante representa un nodo del Quadtree, los espacios negros y blancos siempre están en las hojas, mientras todos los nodos interiores representan los espacios grises.

3.3 Arbol dinámico AABB

After studying Box2D, Bullet and some slides an alumnus (thanks to Nathan Carlson!) from DigiPen created, I put together a dynamic AABB (axis aligned bounding box) tree broadphase. A Dynamic AABB Tree is a binary search tree for spatial partitioning. Special thanks to Nathanael Presson for the original authoring of the tree within the Bullet engine source code (see comments of this post).

A broadphase in a physics simulation has the job of reporting when bodies are potentially colliding. Usually this is done through cheap intersection tests between simple bounding volumes (AABBs in this case).
There are some interesting properties of a dynamic AABB tree that make the data structure rather simple in terms of implementation.

There are a couple main functions to implement outlined here:

Insert

Remove

Update

The underlying data structure ought to be a huge array of nodes. This is much more optimal in terms of cache performance than many loose heap-allocated nodes. This is very important as the entire array of nodes is going to be fetched from memory every single time the broadphase is updated.

The node of the AABB tree can be carefully constructed to take up minimal space, as nodes are always in one of two states: branches and leaves. Since nodes are stored in an array this allows for the nodes to be referenced by integral index instead of pointer. This allows the internal array to grow or shrink as necessary without fear of leaving dangling pointers anywhere.

The idea of the tree is to allow user data to be stored only within leaf nodes. All branch nodes contain just a single AABB enveloping both of its children. This leads to a short description of invariants for the data structure:

All branches must have two valid children

Only leaves contain user data

The first rule allows for some simplification of operations like insert/remove. No additional code branching is required to check for NULL children, increases code performance.

3.4 Kinetic data structures (KDS)

A kinetic data structure is designed to maintain or monitor a discrete attribute of a set of moving objects. A KDF therefore contains a set of certificates that constitutes a proof of the property of interest. Certificates are generally simple inequalities that asserts facts like “point c is on left

of the directed line through a and b". These certificates are inserted in a priority queue.