NOMBRE: Diego Fluxá

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructura de Datos y Algoritmos — 2' 2020

Informe Tarea 1

1 Construcción del árbol

El pseudo-código utilizado para la formación del árbol corresponde a:

```
Node* kdtree(Segment* segmentos, int depth, int inicio, int fin)
{

    min_x <- x minimo de los segmentos;
    max_x <- x maximo de los segmentos;
    min_y <- y minimo de los segmentos;
    max_y <- y maximo de los segmentos;

    Vector min_vector;
    min_vector.x = min_x;
    min_vector.y = min_y;
    Vector max_vector;
    max_vector.x = max_x;
    max_vector.y = max_y;

    //Creamos nodo con bbox de min_vector y max_vector
    Node* node = node_init(min_vector, max_vector);

    // k es el n mero  m ximo de segmentos dentro de la caja de un nodo hoja
    if ((fin - inicio) < k)
    {
        node -> leaf = true;
        node -> num_seg = fin - inicio + 1;
        for (int i = 0; i < node -> num_seg; i++)
        {
            node -> segments[i] = &segmentos[i + inicio];
        }
    }
    else
    {
        //Depth nos otorga el eje (x = 0, y = 1)
        quicksort(segmentos, inicio, fin, depth%2);
        int mediana = (fin - inicio)/2;
        node -> left = kdtree(segmentos, depth + 1, inicio, mediana + inicio);
        node -> right = kdtree(segmentos, depth + 1, inicio + mediana + 1, fin);
    }
    return node;
}
```

Para la creación del árbol se utiliza la función *kdtree* la cual corresponde a una función recursiva que a partir de un array de segmentos, la profundidad actual, el índice inicio y el final del array, crea un árbol. para esto crea un nodo con la bbox correspondiente a los segmentos entre los índices dados y luego revisa si el número de segmentos es menor al k fijado y de serlo lo define como un nodo hoja y le adjudica dichos

segmentos. En otro caso, se define el nodo hijo izquierdo al llamar a la misma función (recursivamente) con los índices inicio a mediana y el derecho con los índices mediana + 1 a fin. Además, hay que recalcar que este algoritmo utiliza quicksort, algoritmo de ordenación visto en clases.

Definiremos el número de segmentos como $\mathcal{S}$. Además, podemos notar que num_seg sera siempre menor o igual a $\mathcal{S}$. Luego, podemos darnos cuenta que la complejidad de nuestro algoritmo estará dado por la complejidad de encontrar los minimos y maximos que corresponderá a $\mathcal{O}(\mathcal{S})$ por recorrer todos los segmentos para encontrarlo, la complejidad de quicksort y la complejidad de otorgar los segmentos al nodo hoja que será $\mathcal{O}(num\_seg) \leq \mathcal{O}(\mathcal{S})$.

Luego, en clases vimos que cualquier algoritmo de ordenación (como el que estamos analizando) en el peor de los casos tendrá que realizar como mínimo $n * log(n)$ comparaciones. Por nesta razón, la complejidad de crear nuestro árbol corresponderá a $\mathcal{O}(\mathcal{S} * log(\mathcal{S}))$

Esto se debe a que realizarán como máximo $log(\mathcal{S})$ iteraciones de nuestra función, por lo que la complejidad corresponderá al número de iteraciones multiplicado por las complejidades analizadas anetriormente.

Por último, es necesario mencionar que la complejidad de este algoritmo no afectará en gran manera al desempeño de nuestro programa debido a que se ejecutará solo una vez al comienzo para crear el árbol y posteriormente solo se recorrerá este.

2 Búsqueda en árbol

En el caso de búsqueda en el árbol se utiliza la siguiente función:

```
Segment* check_collision_tree(Node* node, Particle particle)
{
    if (node -> leaf)
    {
        for (int s = 0; s < node -> num_seg; s++)
        {
            // Si la particula choca con el segmento
            if (particle_segment_collision(particle, *node->segments[s]))
            {
                // Si es que no ha chocado con nada, o si no desempatamos por ID
                if(particle.intersected_segment == NULL || node->segments[s]->ID < particle.
    intersected_segment -> ID)
                {
                    particle.intersected_segment = node->segments[s];
                }
            }
        }
    }
    else
    {
        Segment* seg = NULL;
        if (particle_boundingbox_collision(particle, node -> left -> box))
        {
            seg = check_collision_tree(node -> left, particle);
            particle.intersected_segment = seg;
        }
        if (particle_boundingbox_collision(particle, node -> right -> box))
        {
            seg = check_collision_tree(node -> right, particle);
            particle.intersected_segment = seg;
        }
```

```
    }
    return particle.intersected_segment;
}
```

Esta función es llamada $\mathcal{F} * \mathcal{P}$ veces, pues se ejecuta dentro de un doble for, que recorren todos los frames y todos las partículas.

Además, podemos notar que corresponde a una función recursiva y la cantidad de veces que se ejecutará esta función corresponderá a la cantidad de nodos que existan en el árbol tal que su BoundingBox este en contacto con la partícula entregada. Finalmente para los nodos hoja se ejecutara un for num_seg veces cuyo valor de este corresponderá a uno menor a la constante k definida y menor a $\mathcal{S}$.

Luego, sabemos que el árbol lo estamos creando de tal manera que este quede lo más balanceado posible, por lo que tendremos la altura mínima de un árbol creado con esos datos, es decir $log(\mathcal{S}/k$ (Pues dependerá del número de segmentos máximos). Entonces, de acuerdo a lo visto en clases, toda función de búsqueda en un árbol binario tendrá una complejidad relacionada con la altura del árbol y para la altura mínima corresponderá a $\mathcal{O}(log(\mathcal{S}/k)$.

Además vimos que al llegar a un nodo hoja se ejecutará un for de num_seg veces.- Entonces la complejidad de la función $check\_collision\_tree$ rn rl prot de los casos corresponderá a:

$$\mathcal{O}(log(\mathcal{S}/k) + \mathcal{O}(\mathcal{S})$$

$$\mathcal{O}(log(\mathcal{S}/k)$$

$$\mathcal{O}(log(\mathcal{S})$$

Finalmente, la complejidad de todo el algorítmo de búsqueda corresponderá a la de ejecutar la función dentro de los 2 for mencionados en un principio. Es decir:

$$\mathcal{F} * (\mathcal{P} * \mathcal{O}(log(\mathcal{S})))$$

$$\mathcal{O}(\mathcal{F} * \mathcal{P} * log(\mathcal{S}))$$

3 Ejecución del programa

Para la ejecución del programa nos podemos dar cuenta fácilmente al analizar $main.c$ que su complejidad estará dada por la complejidad de crear el árbol y recorrer este, pues el resto de lo que se realiza posee complejidad constante (se considera constante a las funciones entregadas para facilitar el análisis en la parte de interés). Entonces, la complejidad de ejecución corresponderá a:

$$\mathcal{O}(\mathcal{S} * log(\mathcal{S})) + \mathcal{O}(\mathcal{P} * \mathcal{F} * log(\mathcal{S}))$$

$$Max\{\mathcal{O}(\mathcal{S} * log(\mathcal{S})), \mathcal{O}(\mathcal{P} * \mathcal{F} * log(\mathcal{S}))\}$$

Aquí, podemos observar que en los casos estudiados $\mathcal{P} * \mathcal{F} >> \mathcal{S}$, es decir el número de partículas multiplicado por el número de frames es considerablemente mayor al número de segmentos. Por esta razón, al menos en los casos estudiados, tendremos que la complejidad final de ejecución corresponderá a:

$$\mathcal{O}(\mathcal{P} * \mathcal{F} * log(\mathcal{S}))$$

4 Velocidades de ejecución

Para este análisis consideraremos los tiempos reales de ejecución para el programa base y el final de todos los tests pares provistos.

Es importante mencionar que para el programa final se consideró un $k = 50$ siendo k el número máximo de segmentos que tendrá un nodo hoja.

Test	Programa Base	Programa Final
test_02.txt	0m19.281s	0m0.280s
test_04.txt	0m34.715s	0m0.397s
test_06.txt	0m54.082s	0m0.438s
test_08.txt	1m36.270s	0m0.645s
test_10.txt	2m34.657s	0m0.975s
test_12.txt	4m48.344s	0m1.107s
test_14.txt	6m33.373s	0m1.310s

Podemos observar claramente que los tiempos de ejecución del Programa Final son notoriamente menores a los del Programa Base para todos los tests.

En el primer test el Programa final es aproximadamente 69 veces más rápido que el Programa Base, y a medida que va aumentando la complejidad de los tests, la diferencia de velocidad también va aumentando llegando hasta aproximadamente 300 veces más rápido en el test 14.

Esta diferencia se debe principalmente a que en el Programa Base para cada frame se revisa cada particula y para cada una de estas se revisa la totalidad de segmentos para revisar si la particula ha chocado con alguno. En cambio, en el Programa Final, se crea un árbol binario que contiene BoundingBoxes de segmentos determinados donde sus hijos dividiran esos segmentos en otra BoundingBoxes hasta llegar a un número menor a k, donde se crearán los nodos hoja que tendrán la referencia a esos segmentos y solo se recorreran aquellos segmentos donde se detecte una colisión de la articula con la BoundingBox de ese nodo hoja.

Entonces, se comenzará revisando el nodo raiz con una bbox con todos los segmentos y se verá si la partícula choca con esta, y de hacerlo se procederá de igual manera con ambos hijos hasta llegar a uno o más nodos hojas donde se revisarán las colisiones con los segmentos. De esta manera, la cantidad de segmentos recorridos disminuye drásticamente en comparación con el Programa Base debido a que se eliminan rápidamente todos aquellos segmentos que se encuentren lejos de la partícula de interés.