Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2133 - Estructuras de Datos y Algoritmos

Tarea 1

21 de abril de 2018

1er semestre 2018 - Profesor F. Yadran

Ignacio Tomas Hermosilla Cornejo - 1362072J

Analisis empírico

Para realizar el análisis empírico de tiempo, lo realizaré con el número de llamadas a la función *Euclidian Distance* ya que es una medidad concreta que no variará de PC en PC donde se realize la medición y es un buen reflejo del tiempo que demora el algoritmo y su complejidad

Creación del Kdtree

En teoría, para crear el kdtree debemos esperar que tenga una complejidad de $O(n \times log(n)$ en promedia ya que al inicio, tendremos que usar qselect() que toma en promedio $O(n$ para cada uno de los niveles, los que como siempre dividen por la mediana son $log(n)$. Ahora, qselect() toma en el peor de los casos $O(n^2)$ por lo que el peor caso estamos hablando de una complejidad para construir el algoritmo de $n^2 \times log(n)$. En la práctica, lo que sucede es que el arreglo de coordenadas tiende a irse ordenando, por lo que una optimización que se realizó es que cambiamos el elemento del medio con el del final antes de iniciar la ordenación, como se muestra a continuación.

```
1 void qselect(...)
2 {
3   ...
4  SWAP(len - 1, (len - 1) / 2);
5   ...
6 }
```

Si medimos la función usando el relok del computador, podemos ver que

n	tiempo (segundos)
100	0.000076
1.000	0.000561
10.000	0.005709
100.000	0.081744
1.000.000	0.773655
10.000.000	7.90027

Como podemos ver de la figura 1, podemos ver que más que una linea recta (el eje X esta en escala logarítmica) tenemos más bien un comportamiento exponencial lo que nos índica que nuestra implementación del algoritmo se aproxima más de lo que quisieramos a nuestro peor caso que a nuestro caso promedio.

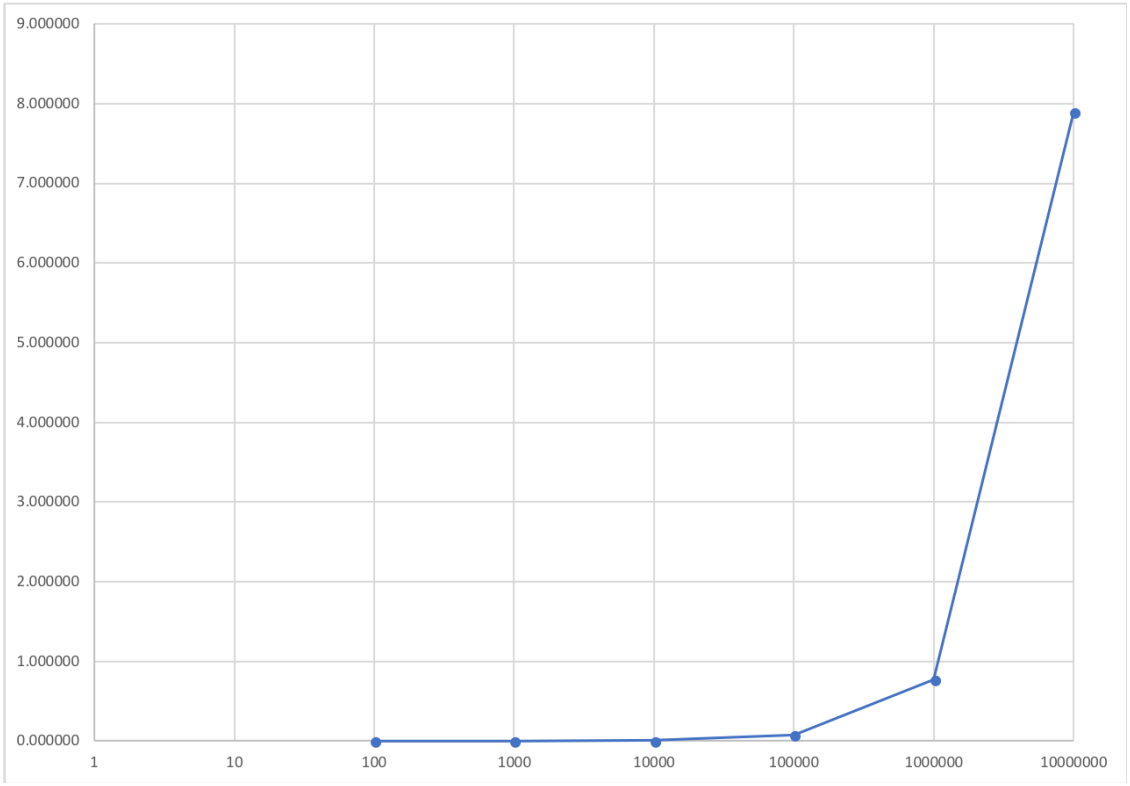

Figura 1: Tiempo de construcción de kdtree en base a número de puntos

Entonces decidí probar elegir simplemente elegir 3 elementos y retornar el del medio

```
void qselect(...)
{
  ...
  SWAP(len - 1, middle(len - 1, (len - 1)/2, 1));
  ...
}
```

pero de nuevo los tiempos obtenidos fueron más cercanos al peor caso que al caso promedio como se puede apreciar en la figura 2

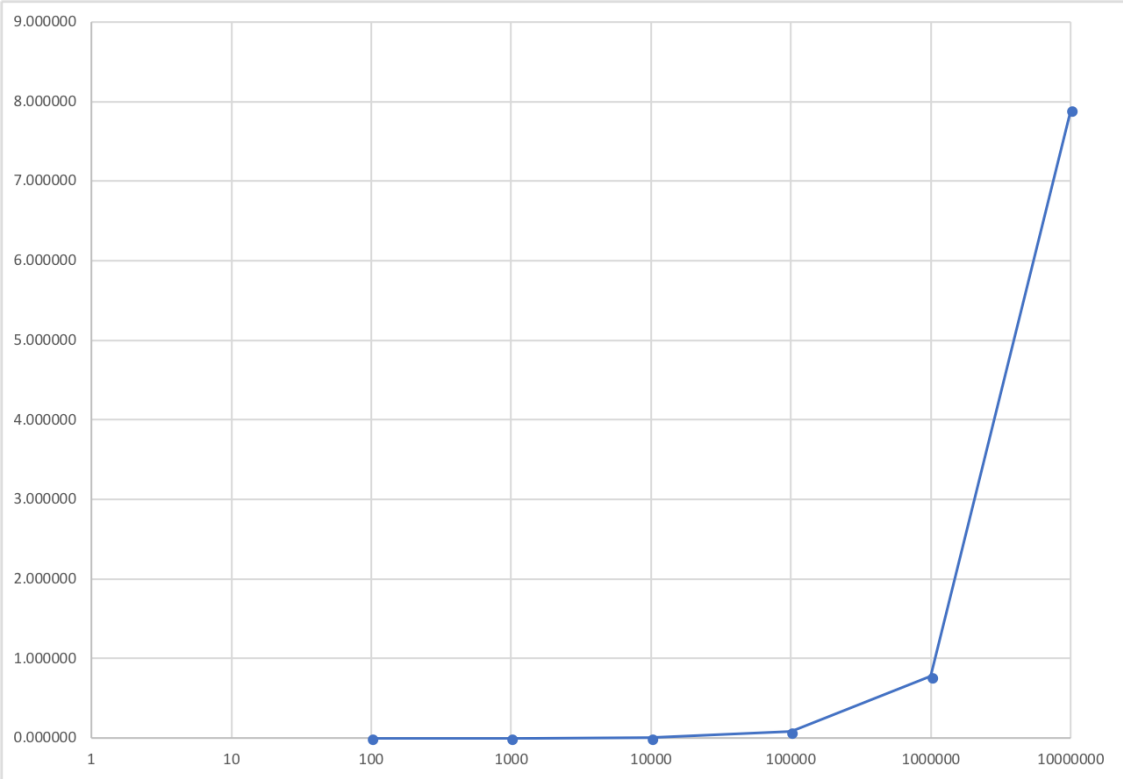

Figura 2: Tiempo de construcción de kdtree en base a número de puntos, segundo metodo

Concluyendo debo decir que no se cumplieron mis expectativas téoricas sobre los tiempos. Esto se debe principalmente a que los datos se tienden a ordenar despues de varias llamadas, expecto que acá estamos haciendo llamadas alternadas en dos dimensiones. Es decir, ordenar los puntos por el eje Y los puede desordenar en el eje X, por lo que al siguiente qselect(), este va a tener los datos menos ordenados.

Cantidad de Nucleos

Con la implementación *naive*, tenemos que la cantidad de operaciones a realizar es de $O(\text{pixeles} * \text{nucleos})$. De hecho la cantidad de operaciones es exactamente $= \text{pixeles} * \text{nucleos}$

Podemos comprobar esto revisando la cantidad de veces que la función *Euclidian Distance* fue llamada y tenemos que para

tamaño imagen	número nucleos	operaciones esperadas	operaciones realizadas
360.000	100	36.000.000	36.000.000
360.000	1.000	360.000.000	360.000.000
360.000	10.000	3.600.000.000	3.600.000.000

Como vemos la implementación *naive* se comporta exactamente como esperamos que se comporte, con una complejidad $O(\text{pixeles} * \text{nucleos})$.

Ahora, en la teoría, la implemnentación usando el kdtree debiera reducír esto a un complejidad de $O(p * log(n))$ ya que para cada pixel, solo tengo que revisar los nucleos dentro de la caja correspondiente al pixel, y los nucleos de las cajas donde hayan colisiones. Si revisamos esto, variando el numero de nucleos, tenemos que

tamaño imagen	número nucleos	operaciones realizadas
360.000	100	2.529.851
360.000	1.000	3.156.658
360.000	10.000	3.553.208
360.000	100.000	2.866.322
360.000	1.000.000	3.234.236
360.000	10.000.000	3.537.413

Lo que es un comportamiento bastante extraño dado que la cantidad de operaciones realizadas aumenta, de manera bastante logarimica hasta llegar a algun punto entre 10.000 y 100.000 para luego decrecer cerca de los 10.000 y finalmente aumentar nuevamente. Este comportamiento se explica por la naturaleza del kdtree. Si tenemos 10 nucleos, entonces relativo a la cantidad de pixeles, hay pocos nucleos por lo que la distancia promedio de estos a sus pixeles será mayor que con digamos, 100, por lo que hay más chances de que al revisar el circulo con las otras cajas, se produzcan más colisiones. Finalmente, si se tienen demasiadas cajas, empieza a pesar más el hecho de que uno se va a demorar mucho más en encontrar la caja donde esta el pixel, lo que recordemos toma $O(log(n))$ por lo que nos hace sentido que haya un valore óptimo que minimize la cantidad de colisiones de caja y a la vez minimize la cantidad de cajas necesarias. Interesantemente este comportamiento se puede apreciar ligeramente en la figura 3, donde vemos que el numero de llamadas

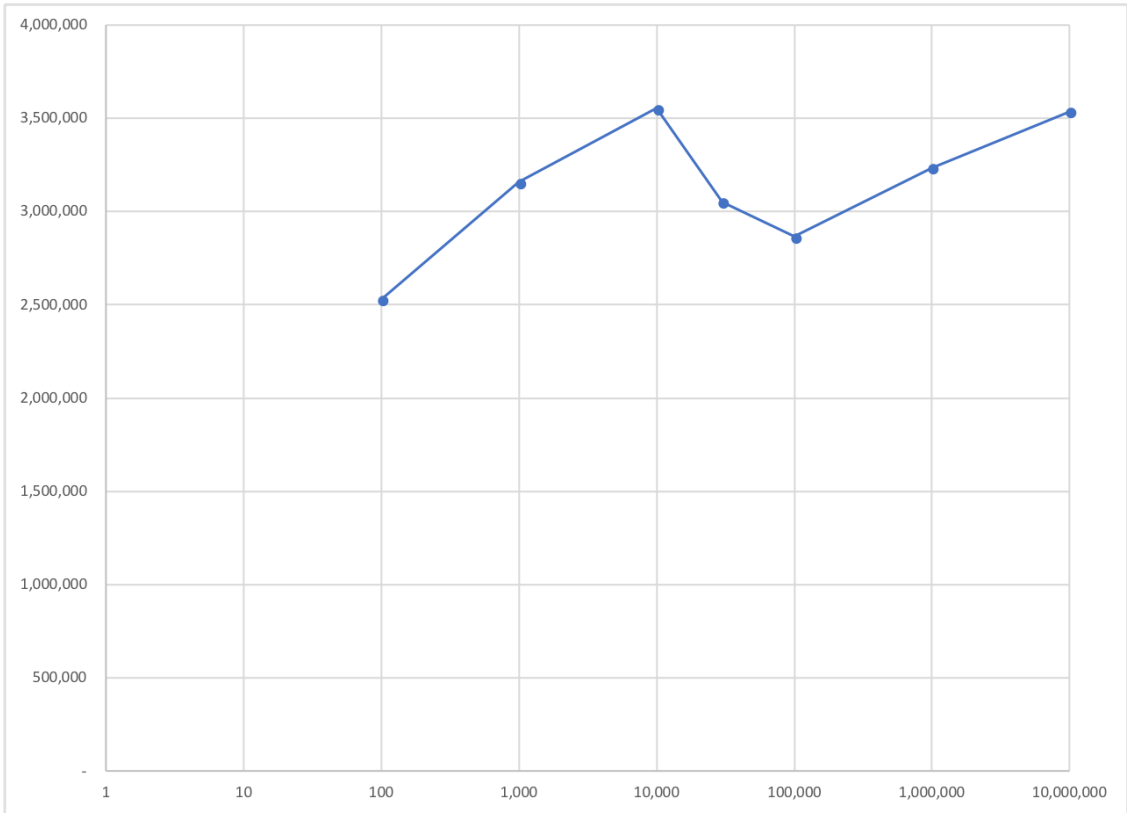

Figura 3: Número de llamadas vs número de nucleos

Número de pixeles

El tamaño de la imagen debiera tener un impacto lineal en la cantidad de llamadas dado que hemos afirmado que la cantidad de llamadas a *Euclidian Distance* es $O(p * log(n))$. Manteniendo constante el numero de nucleos en 10.000, tenemos que:

tamaño imagen	número nucleos	operaciones realizadas
262.144	10.000	2.589.915
360.000	10.000	3.553.208
409.600	10.000	4.045.825

Graficado, podemos ver en la figura 4 que se aprecia una clara relación lineal entre las variables, por lo que en este caso se cumple en la práctica lo que predijimos en la teoría.

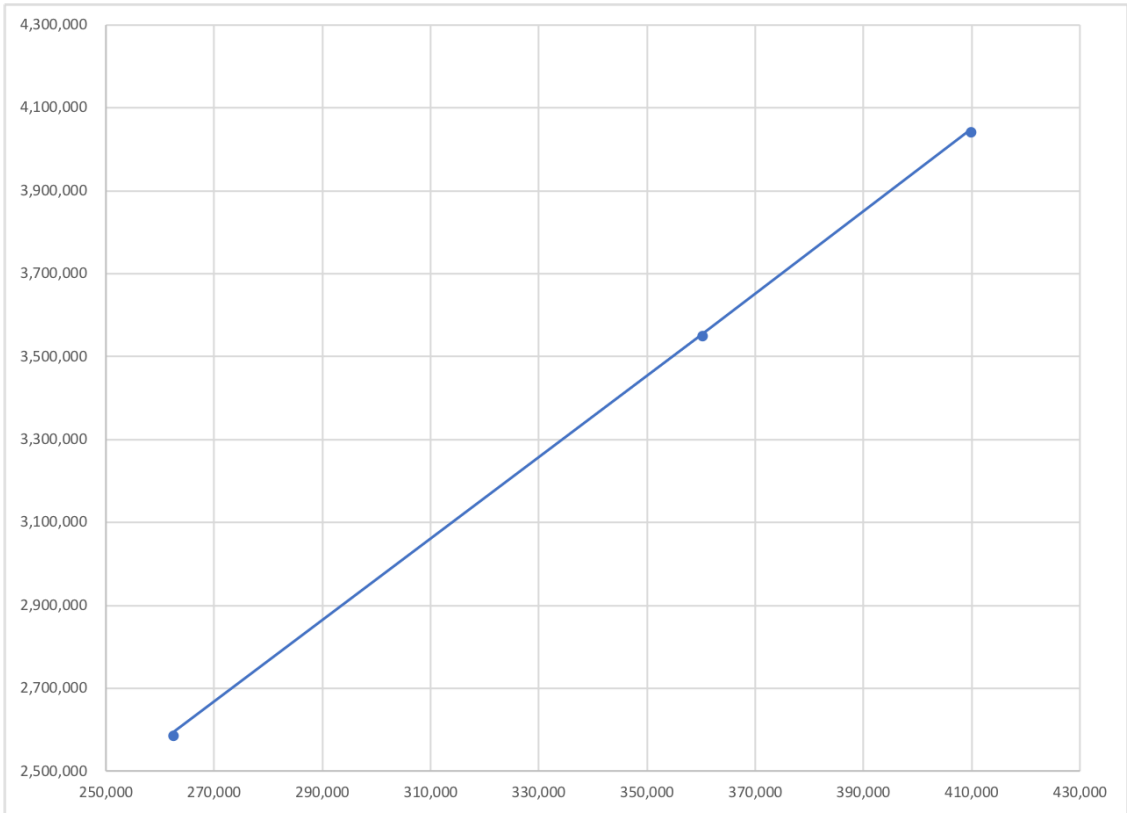

Figura 4: Número de llamadas vs número de pixeles

Tamaño de las cajas

Finalmente, comprobaremos dos criterios para cuando para de dividir el Kdtree. El primero es con el que hemos trabajado hasta ahora en este documento, el cual son 2 o menos elementos. Es decir, cada caja tiene 1 o 2 nucleos. Probaré además varios otros criterios. Si vemos en mi código, dentro de main hay un parametró que se usa para esto, por lo cuál se pueden probar facilmente como se muestra acontinuación

```
void main(...)
{
  ...
  int points_per_box = 2;
  ...
}
```

Ahora, manteniendo fija la imagen, y el numero de nucleos, obtenemos los siguientes resultados:

tamaño imagen	número nucleos	nucleos por caja	operaciones realizadas
360.000	10.000	1	2.704.073
360.000	10.000	2	3.553.208
360.000	10.000	3	3.553.208
360.000	10.000	4	4.027.664
360.000	10.000	5	5.743.876
360.000	10.000	6	5.743.876
360.000	10.000	7	5.743.876
360.000	10.000	8	5.743.876
360.000	10.000	9	7.552.505

Si graficamos la información, tenemos lo que se ve en la figura 5, lo que nos dice que

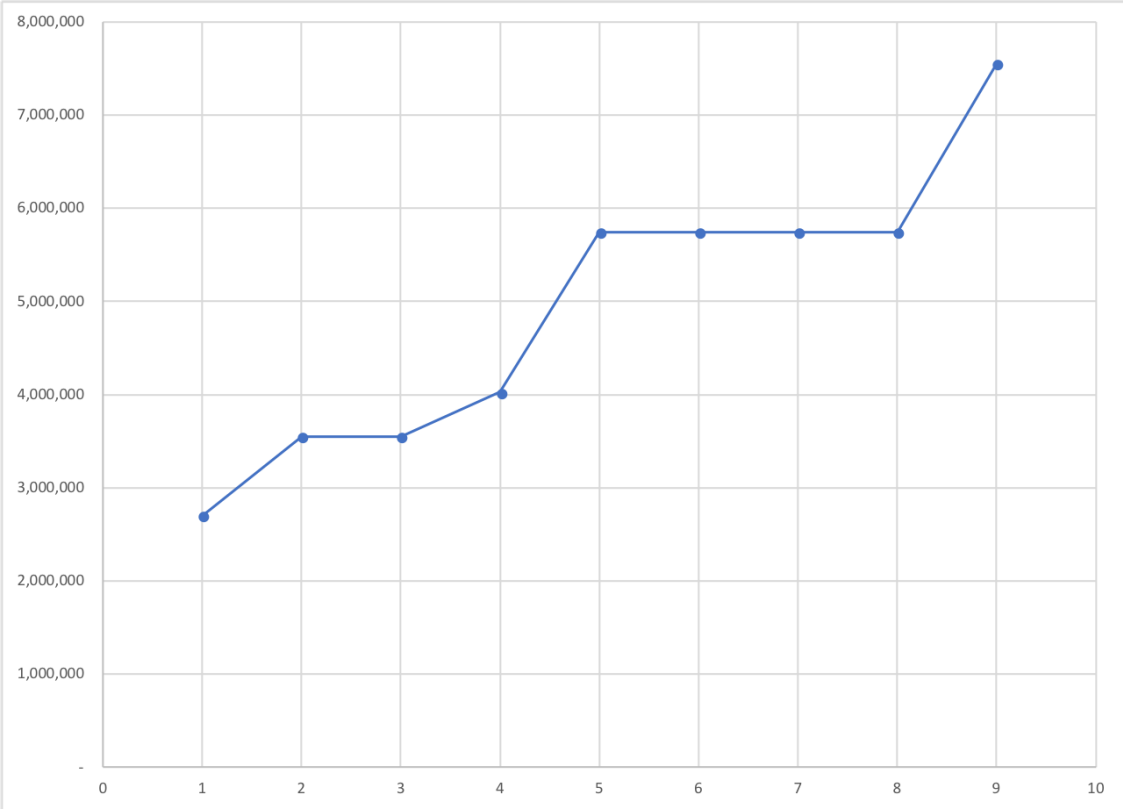

Figura 5: Número de llamadas vs número de nucleos por caja

Se puede a preciar claramente que hay una tendencia lineal entre el numero de cajas y el número de llamadas para estos nucleos. Lo que pasa acá es que aumentar el numero de nucleos por caja no hace que disminuya mucho la profundidad del Kdtree, esto debido a que es necesario duplicar la cantidad de nucleos por caja para poder eliminar un solo nivel del Kdtree. Ahora, esto hace duplicar la cantidad de comparaciones que tenemos que hacer a

nivel caja, por lo que es esperable que aumente el número de llamadas si aumentamos el número de nucleos en cada caja.