UNIVERSITAT POLITÈCNICA DE CATALUNYA

ALGORÍTMICA

Consultas del vecino más cercano en árboles k-dimensionales aleatorios

Q1 23/24

GRUPO 15

Pol Fonoyet González

Yassin El Kaisi Rahmoun

Gorka Parra Ordorica

Tahir Muhammad Aziz

ÍNDICE

1. INFORME TÉCNICO

1.1. DISEÑO E IMPLEMENTACIÓN DE LOS ÁRBOLES K-DIMENSIONALES

La implementación completa en formato código del árbol se puede ver en el apéndice I al final del documento. En este apartado se comentarán únicamente los puntos más vitales de la implementación, por lo que los métodos auxiliares o con menor importancia para entender el funcionamiento se omitirán.

1.1.1. Estructura y atributos

La implementación del árbol *k*-dimensional estándar la hemos basado en un árbol binario. Por lo que por cada árbol hay una única dimensión *k*, y una única raíz. La raíz y los subárboles que cuelgan de esta, están estructurados por nodos. Estos nodos almacenan dos punteros hacia los nodos de sus hijos, un vector para almacenar la *k*-tupla (que identifica la coordenada del nodo) y el discriminante de dicho nodo.

1.1.2. Inicialización de un árbol vacío

El constructor del árbol `BinaryTree` toma como parámetro el número de dimensiones *k* y establece el nodo raíz como nulo.

1.1.3. Inserción en un árbol estándar, relajado y cuadrado.

Para los **árboles estándares** se ha definido un método `insert` que inicia el proceso de inserción recursiva llamando a una función privada `insertRecursive`. Esta segunda función inserta un nodo en el árbol comparando el discriminante de cada nodo para decidir su posición final. El discriminante de este nodo se calculará haciendo el residuo entre la altura a la que se inserta el nodo y la dimensión del árbol.

Además de la inserción estándar, también se han implementado métodos para insertar nodos en versiones relajadas y cuadradas del árbol *k*-dimensional, que son los `insertRelaxed` y `insertSquarish`.

Los **árboles relajados** son una variante de los árboles *k*-dimensionales que introducen cierto grado de flexibilidad al proceso de construcción. En lugar de determinar estrictamente en qué dimensión dividir en función de la profundidad del nodo, los árboles relajados permiten divisiones en cualquier dimensión elegida de forma aleatoria.

En la implementación, la función `insertRelaxed` comienza el proceso de inserción en un árbol relajado llamando a otra recursiva. Esta función, también llamada `insertRelaxed` inserta un nodo de manera recursiva en el árbol. Sin embargo, el discriminante de cada nodo se elige de manera aleatoria. La elección aleatoria del discriminante proporciona una estructura más "relajada" al árbol, lo que puede tener implicaciones en su equilibrio y eficiencia en operaciones de búsqueda, como veremos en los resultados de la experimentación.

Los **árboles cuadrados** buscan mantener una estructura equilibrada dividiendo el espacio de manera que se corte la “bounding box” con el lado más grande.

En la implementación, de forma análoga a los otros dos, una función, `insertSquarish` en este caso, inicia la inserción en un árbol cuadrado. Luego, se usa la inserción recursiva privada, `insertSquarish`, que crea un vector con los valores mínimo y máximo que hay en cada dimensión. Así podremos saber cuál es el lado más largo de la “bounding box”. Esto tiende a dividir el espacio en formas que se aproximan a un "cuadrado", es decir, intenta mantener la uniformidad en todas las dimensiones, lo que puede conducir a búsquedas más eficientes al reducir el número de regiones que necesitan ser exploradas durante la búsqueda del vecino más cercano.

1.1.4. Inicialización de árboles k-dimensionales aleatorios estándares, relajados o cuadrados

El segundo constructor `BinaryTree(int k, int n, int treeType)` genera un árbol *k*-dimensional con tamaño *n*. La coordenada de cada punto se determina de manera aleatoria y el atributo *treeType* especifica el tipo de árbol aleatorio que se va a generar: standard, relaxed o squarish.

1.2. ALGORITMO PARA LA BÚSQUEDA DEL VECINO MÁS CERCANO

La idea detrás de la búsqueda del vecino más cercano en un árbol k-dimensional es recorrer el árbol comparando las coordenadas del punto de consulta con las de los nodos del árbol. Al hacerlo, se explota la estructura del árbol para descartar rápidamente grandes porciones del espacio que no pueden contener al vecino más cercano.

El método `nearestNeighbor` inicia la búsqueda del vecino más cercano para un punto de consulta dado. Si el árbol está vacío, devuelve un vector vacío; de lo contrario, inicia una búsqueda recursiva utilizando otro método del mismo nombre `nearestNeighbor`.

La función recursiva realiza la búsqueda de la siguiente manera::

1. Compara la distancia entre el punto de consulta y el nodo actual (*actualDistancia*). Si esta distancia es menor que la mejor distancia encontrada hasta ahora (*millorDistancia*), actualiza el vecino más cercano y la mejor distancia.
2. Determina qué hijo del nodo actual debe explorar primero. Esto se decide basándose en el discriminante y comparándolo para el punto de consulta y el nodo actual.

3. Una vez que ha explorado el primer hijo, verifica si hay alguna posibilidad de que el vecino más cercano se encuentre en el otro hijo. Esto se hace usando la desigualdad triangular. Si el vecino más cercano potencialmente se encuentra en el otro hijo, explora ese subárbol también. De lo contrario, lo descarta y continúa.

Por lo tanto, la clave de la eficiencia del algoritmo radica en cómo utiliza la desigualdad triangular para descartar subárboles que no pueden contener al vecino más cercano. Si la diferencia entre la coordenada de consulta y la coordenada del nodo actual en la dimensión de división es mayor que la mejor distancia encontrada hasta ahora, entonces no es necesario explorar el otro hijo, ya que no puede contener un punto más cercano.

Es por poder hacer búsquedas de esta forma, por lo que los árboles *k*-dimensionales son una estructura de datos mejor que, por ejemplo, un vector que almacene todos los puntos del espacio y buscar el más cercano comparándolo uno a uno, que tendría un coste lineal, ya que se exploran todos los puntos.

1.3. EXPERIMENTACIÓN

1.3.1. Configuración inicial

El propósito de este experimento es analizar la evolución del número de consultas sobre el vecino más cercano hechas en un árbol *k*-dimensional aleatorio estándar, relajado y cuadrado a partir de ciertos parámetros que nos ayudan a definir los árboles. Estos parámetros se recogen a través de la entrada del usuario y son los que se detallan en el siguiente apartado.

1.3.2. Casos estudiados

Detallando más sobre los cuatro parámetros o variables, tenemos que:

- ***k**:* representa la dimensión.
- ***n**:* indica el tamaño del árbol, que son los nodos, también se puede ver como el número de puntos del espacio.
- ***q**:* es el número de consultas sobre el vecino más cercano que se quieren hacer, los puntos de origen sobre los que aplicar la búsqueda son seleccionados aleatoriamente.
- ***t**:* informa sobre el número de árboles para los que repetir el proceso y así sacar un resultado más convincente.

Para los valores posibles sobre la cantidad de dimensiones, optamos por $\mathbf{2 \leq k \leq 6}$, ya que 1 dimensión no aporta nada nuevo en este estudio y 6 es suficiente como para considerarla una dimensión alta y ver cómo actúa nuestro algoritmo sobre ella.

Dado que la obtención del número medio de nodos visitados lo da el propio programa, no es ningún fastidio establecer un número alto, siempre y cuando este no sea ridículamente alto. Se decidió que sacar la media a partir de $\boldsymbol{t}$**=50** árboles aleatorios para cada *k* y *n* es suficiente.

De momento no se ha especificado el valor rango de *q* ni *n* porque se detalla más en el siguiente apartado, en la parte de "Preliminares" en "Resultados Obtenidos", ya que su justificación deriva de la experimentación inicial.

Para agilizar el proceso, se hizo uso de la paralelización con la biblioteca OpenMP:
#pragma omp parallel for num_threads(omp_get_max_threads()) reduction(+:sum)
Con esta línea de código repartimos las iteraciones del bucle "for" entre todos los hilos disponibles que tenga el dispositivo que ejecuta el programa. Además, crea subvariables de suma para que no haya "data race" y hace el sumatorio de estas al acabar el "for".

1.3.3. Resultados obtenidos

1.3.3.1. Preliminares

El límite de n se puso a partir de los casos prácticos, es decir, que fuera un tamaño máximo representativo de un conjunto de datos considerablemente grande, pero aún manejable desde una perspectiva computacional. También era importante que los valores fueran aumentando de manera constante para obtener resultados lo más precisos posible independientemente de la escala. Estas dos condiciones se unieron para formar que $\mathbf{0 \leq n \leq 10^5}$**, con incremento constante de 5000**.

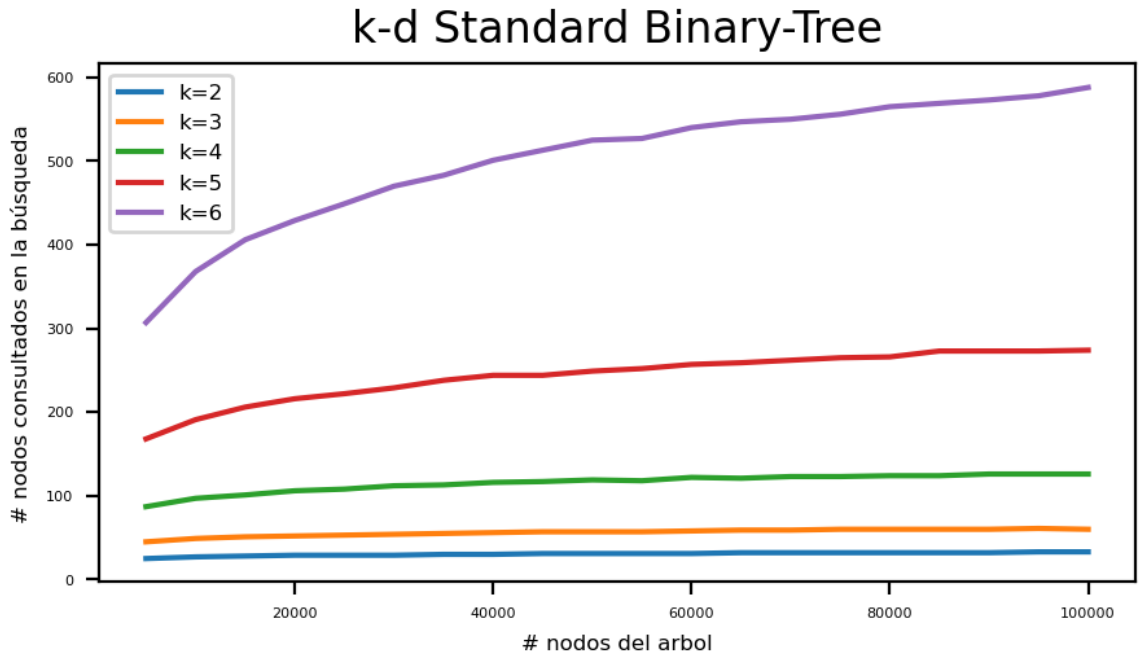
Para q, es lógico que los valores se repitieran dada una dimensión k y un tamaño n. Esto se debe a que la aleatoriedad no influye tanto en una media sobre t=50 árboles como para obtener resultados muy diferentes. Lógicamente, en alguno de ellos el coste individual será bastante diferente a algún otro; sin embargo, como se trata de la media, estos valores se normalizan.

Por lo que a partir de aquí decidimos darle un valor fijo y no repetir el mismo proceso una y otra vez para diferentes valores de q, esta elección se basó en las siguientes dos consideraciones parecidas a las de n. Primeramente, para reflejar el comportamiento promedio y evitar casos atípicos, es fundamental tener un número significativo de consultas. Pero, por otro lado, mientras que un valor muy alto para q puede proporcionar más precisión, también aumenta significativamente el coste computacional de la experimentación.

Nuestra decisión fue elegir un punto intermedio, estableciendo $\boldsymbol{q=n/2}$, ya que nos permite mantener una precisión más que razonable sin quemar recursos.

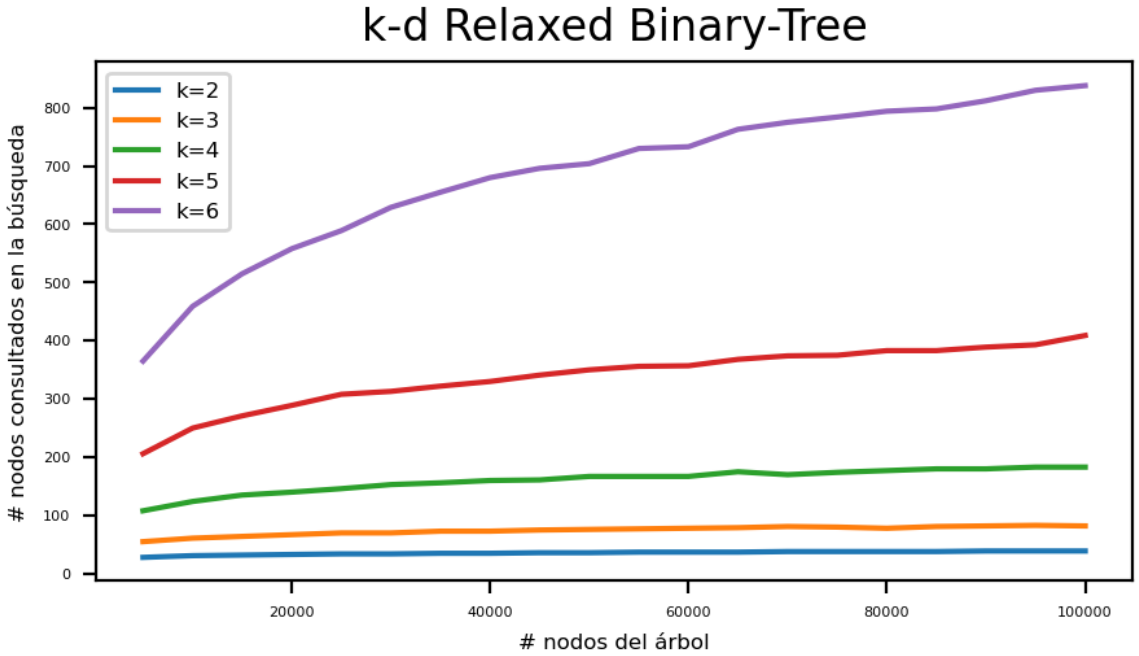
Con todo esto, sabemos que el coste a calcular está en función de k y n: C(k, n).

1.3.3.2 Resultados sobre árboles estándares



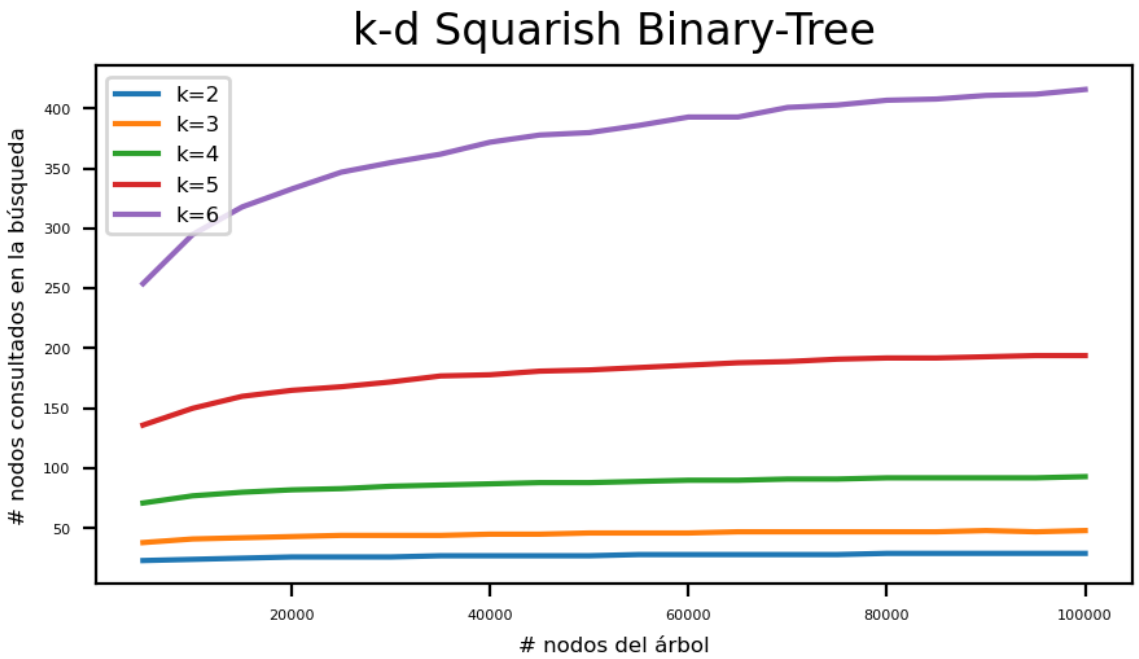
Gráfica 1. Costes para k-d Standard Binary-Tree

1.3.3.3. Resultados sobre árboles relajados



Gráfica 2. Costes para *k*-d Relaxed Binary-Tree

1.3.3.4. Resultados sobre árboles cuadrados



Gráfica 3. Costes para *k*-d Squarish Binary-Tree

1.3.3.5. Interpretación y tendencia de los resultados

Los resultados se presentan en gráficos de líneas donde el eje de las abscisas representa "*n"* y el de las ordenadas C(n, k). Obviamente, como C está en función de dos variables, no nos basta únicamente con el eje X para representar todos los resultados, es por eso que para diferenciar las dimensiones se han usado colores diferentes.

Los resultados de nuestra experimentación muestran que los costes de realizar una búsqueda del vecino más cercano en nuestros árboles k-dimensionales **tienden a comportarse de forma logarítmica**. Esto implica que, a medida que el tamaño del árbol crece, el número de nodos a visitar para encontrar el vecino crece mucho más lentamente.

De manera más detallada tenemos que para las dimensiones bajas, el comportamiento logarítmico no es tan pronunciado en nuestros resultados. Esto se debe a que, con menos dimensiones, hay menos espacio para dividir y, por lo tanto, menos eficiencia inherente en la estructura *k*-dimensional. Aun así, se puede observar una tendencia logarítmica leve.

Para las dimensiones más altas, la ventaja de usar los árboles *k*-dimensionales se hace más evidente. La capacidad de estos árboles para dividir el espacio en múltiples dimensiones significa que pueden excluir rápidamente grandes porciones del espacio durante la búsqueda. Esto se traduce a menos nodos visitados y, por lo tanto, en un comportamiento más claramente logarítmico.

1.3.4. Cálculo del coste medio

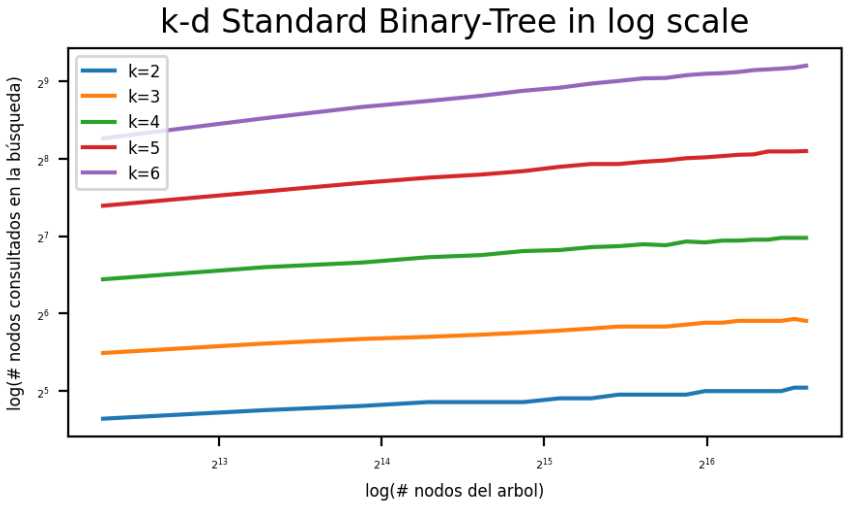
Acorde al enunciado, sabemos que el coste de las búsquedas en los árboles *k*-dimensionales es de la forma $n^Z + \log n$ donde *n* es el número de nodos en el árbol y *Z* puede ser un valor muy pequeño que tiende a 0. Para encontrar este valor de *Z* experimentalmente hemos seguido las instrucciones vistas en clase, usando una regresión lineal.

La idea general es la siguiente: sabemos que las funciones lineales son de la forma $mx + n$ (donde *m* representa la pendiente y *n* el término independiente) y que el logaritmo del coste tiene la forma de $\log(n^Z + \log n) = Z \cdot \log(n) + O(\log\log n)$, según lo visto en clase. De esta manera podemos trabajar con que *Z* es la *m* de la función lineal, es decir, la pendiente.

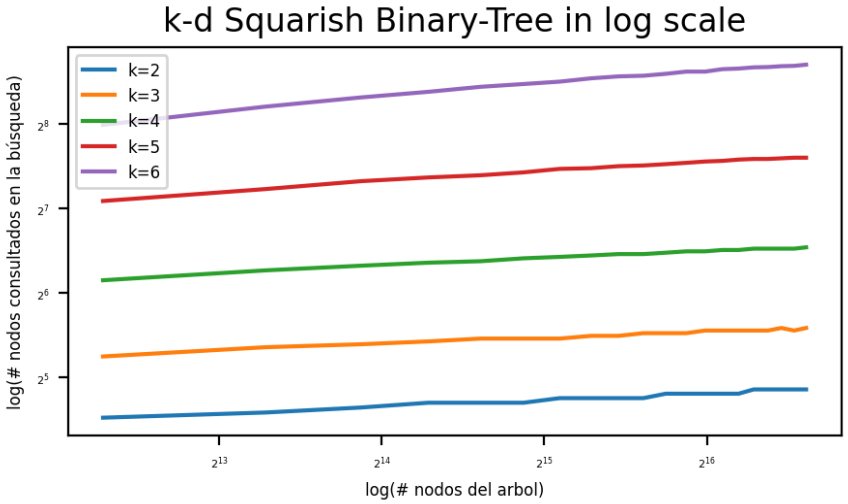
Con tal de encontrar dicha pendiente hemos escalado los ejes a valores logarítmicos, es decir, que observamos una línea recta en vez de una curva logarítmica en el eje de ordenadas, que nos dice el coste C. Con los métodos de regresión lineal de la librería SciPy de Python podemos encontrar la pendiente de estas "rectas", lo que nos dará nuestro estimador de *Z*. Cabe recalcar que la *Z* encontrada no es más que un estimador adquirido de nuestras muestras, y no se puede asumir que ese sea el valor exacto de *Z*. Los resultados obtenidos para cada *k* y tipo de árbol son:

Valores de Z	k=2	k=3	k=4	k=5	k=6
Standard	0.08694	0.09871	0.12209	0.16226	0.20885
Relaxed	0.10716	0.13512	0.17206	0.21236	0.26859
Squarish	0.07969	0.07211	0.08590	0.11638	0.15663

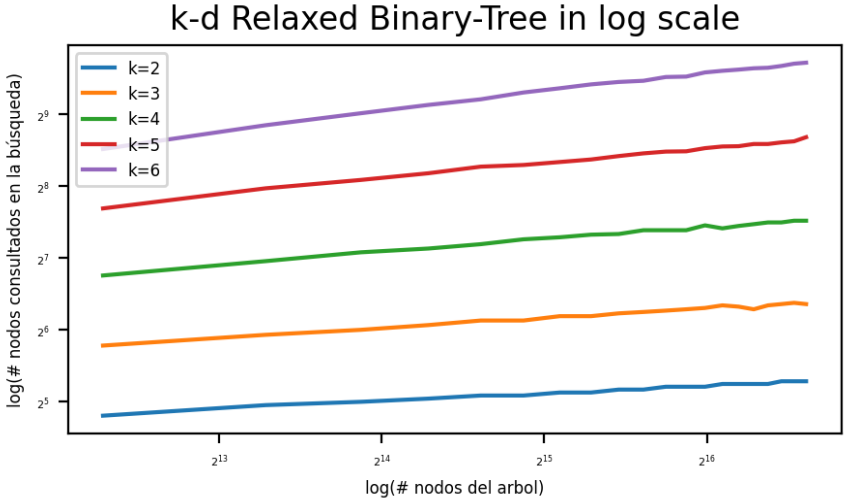
*Para ver como han sido calculados los valores de Z acceda la parte final del archivo "resultados.ipynb"



Gráfica 4. *k*-d Standard Binary-Tree in logarithmic scale



Gráfica 5. k-d Squarish Binary-Tree in logarithmic scale



Gráfica 6. *k*-d Relaxed Binary-Tree in logarithmic scale

1.4. Conclusiones

En cuanto a la búsqueda del vecino más cercano a un punto, se inicia una exploración utilizando un enfoque recursivo basándonos en la distancia euclidiana entre los nodos y el punto de interés. Lo que hace especial a esta búsqueda es que se explotan las propiedades de los árboles *k*-dimensionales y la desigualdad triangular para evitar explorar áreas del árbol que no contienen al vecino más cercano, ahorrándonos un tiempo de cálculo importante.

Respecto a ese tiempo de cálculo, hemos observado que es logarítmico respecto n y que varía dependiendo de la versión de árbol que usemos. Esto es porque, aparte del árbol estándar, tenemos el relajado y el cuadrado. Estas adaptaciones o variaciones de la estructura *k*-dimensional son interesantes porque ofrecen diferentes compromisos entre el tiempo de construcción, el equilibrio del árbol y la eficiencia de la búsqueda.

Por lo tanto, y a partir de los datos que hemos obtenido reflejados en *Gráfica 1* en adelante, podemos afirmar que dependiendo la necesidad, es conveniente hacer un estudio previo sobre qué tipo de árbol debería de implementarse, porque cada uno aporta algo diferente.

1. Si se necesitan crear o modificar árboles regularmente, nosotros elegiríamos los árboles ***k*-dimensionales relajados** sobre las otras opciones porque se construyen más rápidamente (con menos instrucciones). En contra, tenemos que su eficiencia a la hora de hacer ciertas búsquedas, es menos destacable por el equilibrio del árbol respecto a las otras dos versiones, como en el caso de las búsquedas del vecino más cercano. Un ejemplo práctico de esto serían los mapas procedurales en videojuegos, que se van generando a medida que pasa el tiempo, el espacio u otro medio.

2. Siguiendo el ejemplo anterior, en un mapa que no va a variar, como puede ser una representación de Barcelona, para una empresa de repartos sería conveniente usar un árbol ***k*-dimensional cuadrado** para aprovecharse de su eficiencia sobre el resto. Estos árboles a la hora de construir y modificar son peores, ya que se intenta "cuadrar" el espacio lo máximo posible. Esto, a su vez, es lo que nos aporta la gran eficiencia respecto a los otros dos. En nuestros datos, esta eficiencia resalta más para $C(k, n) = C(6, n=10^5)$, que es el valor máximo calculado. Concretamente, obtenemos que el coste C para esos valores es un 30% menor que el estándar y un 50% respecto al relajado, que son diferencias muy notorias.

3. Para situaciones intermedias no habría problemas a no decantarse a alguna de las otras dos opciones y elegir los ***k*-dimensionales estándares**. Esto es porque, como se ve en las tres gráficas, el árbol estándar tiene un coste promedio entre los otros dos y ofrece tanto.

2. DESCRIPCIÓN Y VALORACIÓN DEL PROCESO DE AUTOAPRENDIZAJE

2.1. METODOLOGÍA

Para llevar a cabo el proyecto de la mejor forma posible, lo primero que hicimos fue crear un sistema de comunicación para ir informando sobre el proceso y los avances a medida que hubiera nueva información. Para ello utilizamos las siguientes herramientas: "WhatsApp" para comunicar los pequeños cambios y debatir los nuevos problemas o retos que van surgiendo; "Discord" para hacer reuniones y poder organizar el desarrollo y distribución del trabajo; y finalmente "GitHub" para tener un controlador de versiones y poder trabajar todos los integrantes de equipo con comodidad. Teniendo esto en cuenta, durante el proceso de autoaprendizaje, adoptamos un enfoque sistemático y estructurado para abordar la implementación y el análisis de los árboles *k*-dimensionales.

- **Recopilación de Recursos**: Iniciamos el proceso identificando y recopilando recursos relevantes que incluyen tutoriales para aprender git [1], webs especializadas para comprender los distintos tipos de árboles [2][3][4], y estudios para entender en profundidad cómo trabajar con este tipo de árboles [5]. Los recursos de las clases particulares y el libro en la bibliografía, especialmente, proporcionaron una base sólida para comprender los conceptos fundamentales.

- **Estudio y Discusión**: Organizamos sesiones regulares de estudio para digerir el material recopilado. Estas sesiones eran seguidas de discusiones grupales para clarificar dudas y consolidar el conocimiento. Una vez hecho esto, se avanzaba en los diferentes campos que estuvieran por hacer, ya sea implementar una función, corregir alguna, documentar alguna parte o cualquier otra cosa del proyecto. Mientras que algunas sesiones se hicieron presencialmente entre horas muertas, otras se hicieron vía online por Discord, que es como Slack, pero donde estamos más acostumbrados.

- **Práctica y Evaluación**: Implementamos conceptos teóricos en pequeños segmentos de código, permitiéndonos recibir retroalimentación inmediata sobre nuestro entendimiento. También creamos funciones auxiliares para comprobar y visualizar que los árboles y los algoritmos de búsqueda funcionaran adecuadamente. Además, evaluamos constantemente nuestro progreso a través de pruebas y experimentación. Lógicamente, estas pruebas no están disponibles en el anexo porque iban evolucionando a medida que implementamos cada función cuando necesitábamos comprobar su corrección. Este proceso fue intercalando con el anterior.

2.2. VALORACIÓN DEL PROCESO DE AUTOAPRENDIZAJE

El proceso de autoaprendizaje demostró ser desafiante porque nunca ninguno de nosotros había oído hablar de los árboles *k*-dimensionales, pero también fue gratificante porque teníamos la sensación de que estábamos aprendiendo algo muy útil y aplicable. Algunas reflexiones sobre el proceso incluyen:

- **Desafíos Enfrentados**: Uno de los principales desafíos fue entender bien las diferencias y las ventajas entre los diferentes tipos de árboles *k*-dimensionales y cómo implementarlas correctamente. Sin embargo, el enfrentarnos a ello nos proporcionó una comprensión más profunda del tema.

Otro gran desafío ha sido tener que utilizar nuevas herramientas con las que alguno de los miembros del equipo no estábamos familiarizados, como podría ser “jupyter notebook”, algunas librerías de python e incluso la herramienta que estamos estudiando en la asignatura de PAR para paralelizar código: “OpenMP”.

- **Fortalezas descubiertas**: Reforzamos la importancia de comunicar cada punto de vista, ya que, a pesar de que a veces da la impresión de que entre el grupo nos estamos entendiendo sobre un punto en particular, nos damos cuenta de que no una vez nos ponemos las manos a la obra. Un ejemplo fue con el algoritmo de búsqueda del vecino, donde todos nos pusimos de acuerdo en el concepto, pero a la hora de implementarlo teníamos visiones algo diferentes. A base de ello, también hemos aprendido que mezclar y modelar nuestras ideas es una buena opción para seguir adelante.

- **Áreas de mejora**: Una posible mejora podría haber sido buscar algún servicio en la nube para ejecutar el código, ya que nos tardaba entre 10 y 20 minutos para cada tipo de árbol. Además, podríamos haber creado árboles de más de 100.000 nodos si hubiéramos usado este servicio de ejecución online.
  Por esa misma razón, la precisión de nuestros experimentos se vio afectada, teniendo que hacer tests con N siendo cada múltiple de 5000 hasta llegar a 100000. Idealmente, con un equipo suficientemente rápido, podríamos haber hecho tests cada 1000 nodos en vez de cada 5000.

3. BIBLIOGRAFÍA

[1] Learn Git Branching (2023)
Disponible en: https://learngitbranching.js.org/
(Último acceso: 08 Octubre 2023)

[2] Wikipedia: Árbol k-d
Disponible en: https://es.wikipedia.org/wiki/Árbol_kd
(Último acceso: 08 Octubre 2023)

[3] Geek for Geeks: Search and Insertion in k-dimensional Tree (2023)
Disponible en: https://www.geeksforgeeks.org/search-and-insertion-in-k-dimensional-tree/
(Último acceso: 13 Octubre 2023)

[4] Wikipedia: Relaxed k-d tree.
Disponible en: https://en.wikipedia.org/wiki/Relaxed_k-d_tree
(Último acceso: 08 Octubre 2023)

[5] [DJM14] A. Duch, R. M. Jimenez, and C. Martnez. Selection by rank in k-dimensional binary search trees. Random Structures & Algorithms, 2014.

4. APÉNDICE

4.1. MAIN.cc

Aquí está el código del “main automático” que hemos creado para generar los resultados en forma de “array” que hay en “resultats.ipynb”.

C/C++

```
#include <iostream>
#include <random>
#include <omp.h>
#include "Tree.hh"
using namespace std;

int main() {
    // Método para crear números aleatorios de 0 a 1
    random_device myRandomDevice;
    unsigned seed = myRandomDevice();
    uniform_real_distribution<double> Uniforme(0.0, 1.0);
    default_random_engine RNG(seed);

    int t, q, typeTree;

    cout << "Elige tipo de arbol (0-kd, 1-Relaxed, 2-Squarish): ";
    cin >> typeTree;

    t = 50;

    // Creamos árboles de 5.000 hasta 100.000 nodos
    cout << endl << "[";
    for (int n = 5000; n <= 100000; n += 5000) {
        q = n / 2;
        cout << endl << "[";

        // Creamos árboles de 2 a 6 dimensiones
        for (int k = 2; k <= 6; ++k) {
            int sum = 0;

            // Paralelizamos el código para que se puede ejecutar más rápidamente
             #pragma omp parallel for num_threads(omp_get_max_threads()) reduction(+:sum)

            // Creamos t árboles
            for(int i = 0; i < t; i++) {
                BinaryTree Arbol(k, n, typeTree);

                for(int j = 0; j < q; j++) {
                    vector<double> consulta(k);
                    for(int l = 0; l < k; l++) consulta[l] = Uniforme(RNG);

                    vector<double> vecino = Arbol.nearestNeighbor(consulta);

                    sum += Arbol.checkNumNodes();
                }
                 Arbol.BorrarInit();
            }
            cout << sum/(t*q);
            if (k != 6) cout << ",";
        }
        cout << "],";
    }
    cout << "]" << endl;
}
```