UNIVERSITAT POLITÈCNICA DE CATALUNYA

ALGORÍTMICA

Consultas del vecino más cercano en árboles k-dimensionales aleatorios

Biel Altés, Rubén Catalán, Ismael El Basli y Alex Garcés

Q1 2023-24

Índice

1. Introducción

El problema de las búsquedas asociativas (*associative retrieval*) es un problema informático muy presente en las aplicaciones actuales. En este problema se considera una colección F de n registros, donde cada registro tiene una llave que es una k-tupla ($k \geq 2$) $x = (x_0, \ldots, x_{k-1})$ de valores (llamados atributos o coordenadas) extraídos de un dominio $D = \prod_{0 \geq j < k} D_j$, donde cada D_j está totalmente ordenado. El objetivo de una búsqueda (o consulta) sobre F es recuperar todos los registros de F las llaves de los cuales cumplan ciertas condiciones dadas. La consulta se considera asociativa cuando incorpora al menos dos de los atributos de sus llaves. Hay muchos ejemplos de consultas asociativas, pero este proyecto se centra en las consultas del vecino más cercano, para recuperar el registro de F con la llave más cercana a una llave dada. Es obvio que para tratar eficazmente las consultas asociativas, el almacenamiento de los registros de F es crucial. Así, las estructuras de datos multidimensionales de propósito general —como los árboles k-dimensionales— son los métodos de almacenamiento adecuados para apoyar una amplia gama de consultas asociativas.

Este proyecto se centra precisamente en llevar a cabo un análisis experimental del coste en caso medio de consultas del vecino más cercano en tres variantes de los árboles k-dimensionales: *standard*, *relaxed*, y *squarish*. Previamente a ese análisis experimental, se muestra nuestra implementación de la estructura de datos y del algoritmo para realizar las consultas.

2. Implementación de los árboles k-dimensionales

2.1. Variante *standard*

Antes de que se muestre la implementación es conveniente recordar la definición formal de la estructura de datos:

Definición 1. *(Bentley75). Un árbol de búsqueda k-dimensional estándar T (k-d tree estándar) de tamaño $n \geq 0$ es una estructura de datos que almacena un conjunto de n registros, donde cada uno contiene una llave que consiste en una k-tupla $x = (x_0, \ldots, x_{k-1}) \in D$, donde $D = D_0 \times \cdots \times D_{k-1}$, y donde cada $D_j, 0 \leq j < k$, es un dominio totalmente ordenado. Asumiremos que $D_j \in [0, 1]$ para toda $j, 0 \leq j < k$. El árbol T es un árbol binario tal que:*

- *si $n = 0$ es vacío o bien,*
- *si $n > 0$ su raíz almacena un registro con llave x y tiene asociado un discriminante j tal que $j =$ nivel de la raíz mod k $(0 \leq j < k)$. Los $n - 1$ registros restantes se almacenan en los subárboles izquierdo y derecho de T, L y R respectivamente, de tal forma que tanto L como R son k-d trees y se cumple que para cualquier llave $u \in L$ $u_j < x_j$ y que para cualquier llave $v \in R$ $x_j < v_j$.*

Se ha usado programación orientada a objetos, dónde se considera la existencia de dos clases principales: *k-d tree* y *k-d node*. Se ha pensado que puede llegar a ser útil separar el nodo k-dimensional del árbol que los almacena. La clase *k-d tree* tiene tres atributos principales: el tamaño (número de registros), la dimensión k y un puntero a un objeto de la clase *k-d node*, que corresponde a la raíz del árbol. A parte de la constructora y destructora de la clase, se ha implementado un método *insert*, que dado un registro k-dimensional lo inserta en la estructura de datos:

Algorithm 1: Insert

```
Data: k-dimensional key
Result: key has been inserted into the k − d tree
if root is null then
    root = new node(key);
else
    root → insert_node(key);
end
```

Se ve como en caso de que el árbol esté vacío y la raíz tenga el valor de puntero nulo, se crea una instancia de nodo y se asigna a la raíz un puntero a él. Por otro lado, si ya hay nodos en el árbol, se llama a un método del nodo raíz, de forma que se delega el trabajo al nodo.

Algorithm 2: Insert Node

```
Data: k-dimensional key
Result: key has been inserted into the k − d tree
if n → key[n → disc] > key[n → disc] then
    if left son is null then
        left son = new node(key)
        left son → discr = (n → discr + 1)  mód  k
    else
        left son → insert_node(key)
    end
else
    if right son is null then
        right son = new node(key)
        right son → disc = (n → discr + 1)  mód  k
    else
        left son → insert_node(key)
    end
end
```

El nodo compara el valor de su clave con la que se quiere insertar, concretamente compara las coordenadas que corresponden a su discriminante. Se sigue por su hijo izquierdo si su coordenada discriminante es mayor que la de la clave a insertar, por el derecho en caso contrario. Si el hijo por el que se sigue no existe (vale *null*), se crea un nodo con el valor *key* que se quería insertar, se asigna como hijo del nodo

con el que se hizo la comparación previa y se le da un valor de discriminante igual a $(d + 1)$ *mod* k, dónde d es el discriminante del padre. Por ejemplo, para un *k-d tree* de 3 dimensiones, si el padre tuviese por discriminante $d = 0$, el del nuevo nodo seria $d' = d + 1 = 1$. Si fuera $d = 3$ entonces $d' = 0$. Por otro lado, en caso de que el hijo sí existiera, se seguiría con una llamada recursiva de *insert node* a dicho nodo.

2.2. Variantes *relaxed* y *squarish*

Se ha trabajado con dos versiones ligeramente distintas a los *stadard k-d trees*. Se trata de las variantes *relaxed* y *squarish*. La principal diferencia se encuentra en el cálculo del discriminante en cada nodo, que como se ha mencionado previamente, es el que rige hacia donde se inserta un nuevo nodo. Como se ha dicho, en el caso de los *standard k-d trees* la dimensión que se escoge como discriminante está directamente relacionada con la profundidad del árbol donde se pretende insertar el nodo. Dadas d dimensiones, el discriminante para una profundidad p es p *mod* d.

Algorithm 3: Insert Node (Squarish Variant)

```
Data: k-dimensional key, k-bounding box
Result: key has been inserted into the k − d tree
if n → key[n → disc] > key[n → disc] then
    bounding box → prune right(n → key, n → disc)
     if left son is null then
        left son = new node(key)
         left son → discr = arg máx_disc{bounding box}
    else
        left son → insert node(key, bounding box)
    end
else
    bounding box → prune left(n → key, n → disc)
     if right son is null then
        right son = new node(key)
         right son → disc = arg máx_disc{bounding box}
    else
        left son → insert node(key)
    end
end
```

En cuanto a los *squarish k-d trees*, cuando un rectángulo se divide por un punto recién insertado, en lugar de alternar los discriminantes, se corta el lado más largo del rectángulo. Por lo tanto, el corte es siempre un hiperplano $k-1$ dimensional que pasa por el nuevo punto como siempre, pero ahora es siempre perpendicular al lado más largo del rectángulo. Como resultado, estos *k-d trees* tienen regiones de aspecto más cuadrado, y de allí viene el nombre de la variante. Para ello, el algoritmo parte de una *bounding box* lo mas grande posible y la va podando conforme recorre las ramas del *k-d tree*. Para cuando toque insertar el nodo, el discriminante que se le asigna es igual al de aquel que tenga el valor mas grande en la *bounding box* resultante. Por

otro lado, los *relaxed k-d trees* son una variante propuesta por Duch, Estivill-Castro y Martínez en [1]. La construcción de un *relaxed k-d tree* elige los discriminantes de forma puramente aleatoria: cuando se inserta un nuevo nodo, se genera un número de forma pseudoaleatoria entre 0 y k - 1 y se elige como discriminante.

3. Búsqueda del vecino más cercano

Como ya se ha mencionado previamente, este proyecto se centra en las consultas por el vecino más cercano -conocidas en inglés como *nearest neighbor queries*. Para realizar las consultas, se usa la distancia euclídea entre puntos k-dimensionales, conocida también como norma L_2.

Es obvio que con un recorrido entero del árbol k-dimensional se podría obtener fácilmente el resultado deseado, pero entonces no se estaría aprovechando el potencial de la estructura de datos usada. Se ha pensado en intentar evitar hacer llamadas recursivas a los subárboles izquierdo o derecho cuando no sea necesario, minimizando así el número de nodos visitados.

Para ello, el algoritmo que se plantea explota la desigualdad triangular que satisface la función de distancia. La idea principal es intentar insertar el punto *query* en el árbol, y calcular en cada punto k-dimensional visitado la distancia entre él y el punto *query*, de forma que si la distancia es menor que la mínima encontrada hasta el momento, ésta es actualizada. Con intentar insertar nos referimos a recorrer el árbol descartando siempre uno de los dos subárboles, escogiendo visitar en base a la coordenada discriminante. El problema está en que no se puede descartar de forma sistemática uno de los dos subárboles (en dicho caso y si el árbol estuviera razonablemente equilibrado el coste del algoritmo seria $\theta(\log n)$). Entonces, al hacer *backtracking*, se debe evaluar si es necesario hacer la llamada recursiva al subárbol que previamente no había sido visitado. En este punto es donde se saca partido a la distancia mínima encontrada, ya que solo es necesario hacer la llamada al otro subárbol en caso que, siendo j la coordenada discriminante del punto visitado, la distancia que hay entre las coordenadas j del punto *query* y el punto actual sea estrictamente menor que la mínima. Es fácil ver que si esa distancia es mayor que la mínima, es imposible encontrar un punto más cercano en la *bounding box* correspondiente al subárbol.

En la Figura 1 se puede ver lo descrito. El punto negro es la *query*. Tal y como funciona nuestro algoritmo, se recorre hasta llegar al punto C (recorriendo $A \rightarrow B \rightarrow C$), pero al volver hacia atrás hay que decidir si explorar más. En el punto B, se ve como la distancia entre la coordenada discriminante de B y la misma de *query* (llamada $dmin'$ en la figura), es menor que la distancia mínima hasta el momento (la que hay con C). Por lo tanto, es necesario visitar la *bounding box* que corresponde al subárbol derecho de B, ya que se podría encontrar un punto a distancia menor, que de hecho existe y es el punto D. Si esa distancia fuera mayor, es obvio que no se podrían encontrar puntos mas cercanos. Si se siguiera la ejecución del algoritmo, se vería que también habría que visitar el subárbol derecho de A, porque seria posible encontrar puntos aún mas cercanos en esa *bounding box*.

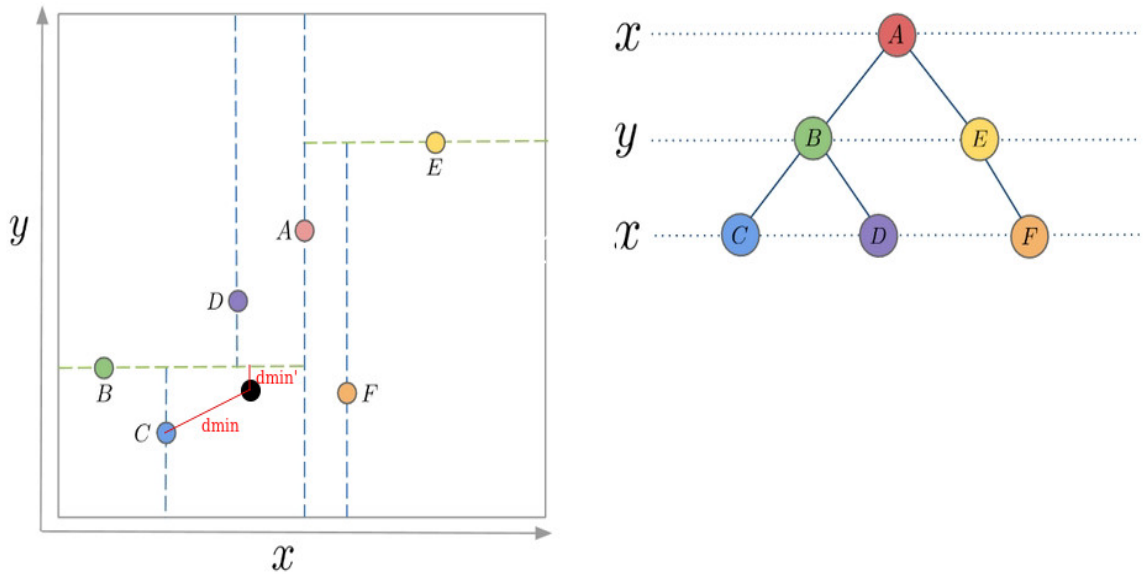

Figura 1: Ejemplo de búsqueda del más cercano

A continuación, se muestra el pseudocódigo correspondiente al algoritmo:

Algorithm 4: get_nearest_neighbor

```
Data: query key, k-d node* n, k-d node*& nn, double& min_dist
Result: nn points to the nearest neighbor of query, and min_dist stores the
        distance value
if n is null pointer then
    return;
end
double dist = distance(n, query);
if dist < min_dist then
    min_dist ← dist; nn ← n;
end
int j = n→getDiscriminant();
if query_j < n_j then
    get_nearest_neighbor(query, n→getLeftNode(), nn, min_dist);
    double diff = |query_j − n_j|;
    if diff < min_dist then
        get_nearest_neighbor(query, n→getRightNode(), nn, min_dist);
    end
else
    get_nearest_neighbor(query, n→getRightNode(), nn, min_dist);
    double diff = |query_j − n_j|;
    if diff < min_dist then
        get_nearest_neighbor(query, n→getLeftNode(), nn, min_dist);
    end
end
```

A parte de lo mostrado, hay otra función que se encarga de hacer la llamada inicial dando un valor inicial a los atributos que se pasan por parámetro, pasando como n el nodo *root* y un valor muy grande como *min_dist*. Además, con el objetivo de poder contar el número de nodos visitados para el análisis experimental del coste, cada vez que se visita un nodo, se incrementa en uno el valor de una variable pasada por referencia, que antes de la primera llamada vale 0.

4. Configuración experimental

Para la experimentación, era necesario escoger unos parámetros N y Q (siendo N el número de árboles a crear y Q el número de *queries* de vecino más cercano a realizar en cada árbol) que permitieran encontrar el valor de ζ dentro de $\theta(n^{\zeta}+\log n)$, que es la función que determina el coste asintótico de las búsquedas del vecino más cercano. Además, para conseguir suficientes datos como para inferir los resultados que se buscan, cada uno de estos experimentos se realizaría para arboles con un número variable de nodos $n \in [1000, 2000, 3000, ..., 100000]$ y dimensiones $k \in [2, 6]$.

Se termina escogiendo $N = 100, Q = 1000$, ya que, si bien valores más altos permitirían mayor precisión, para valores de n cercanos a los 10^5 nodos, la creación de árboles se vuelve costosa y lenta. Con 100 árboles y 1000 consultas, para cada combinación de nodos y dimensiones, conseguimos un número más que suficiente de datos para generar medias respecto al número de nodos visitados por consulta y extraer resultados.

Para facilitar la experimentación, se ha creado un programa *tester* que recibe los cuatro parámetros ya explicados, en este orden: k, n, N, Q. Así, para cada una de las tres variantes de *k-d tree* vistas anteriormente se generan N árboles de n nodos de k dimensiones y se realizan Q consultas del vecino más cercano, después de lo cual se imprimen por pantalla las medias de nodos visitados y las varianzas.

5. Resultados experimentales

El objetivo de este apartado es mostrar los resultados experimentales. Para cada $k \in [2, 6]$ (siendo k el número de dimensiones) se muestra una gráfica del coste medio de las búsquedas del más cercano $C(n)$ (número de nodos visitados) en función del número de registros n, para cada una de las tres variantes de *k-d tree*. Además, se muestran gráficas que incluyen barras verticales que representan la variabilidad de nuestros datos.

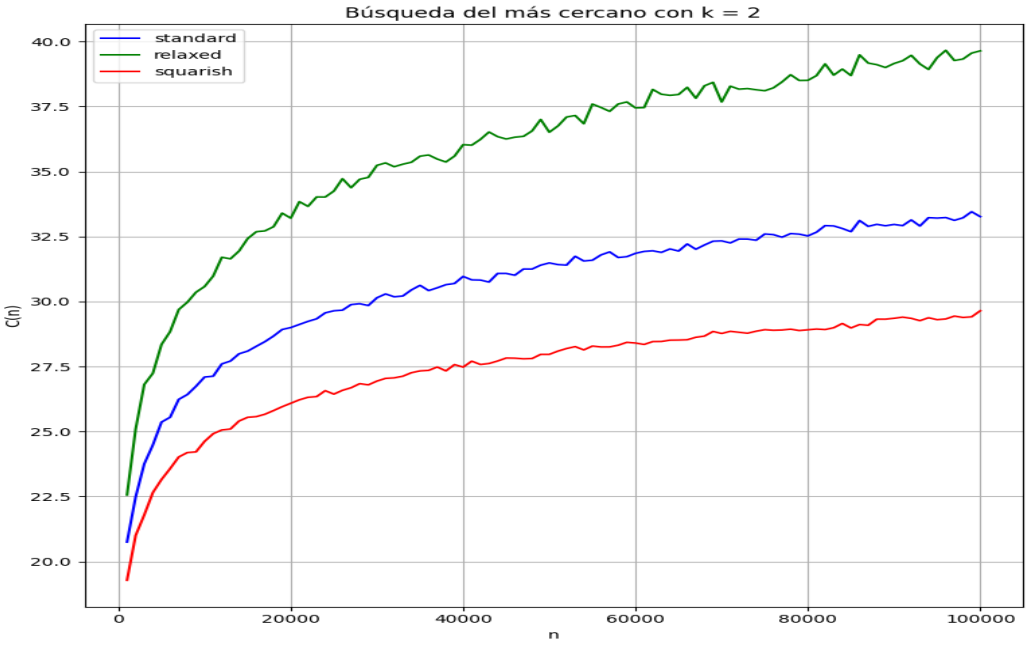

Figura 2: Gráfica para $k = 2$

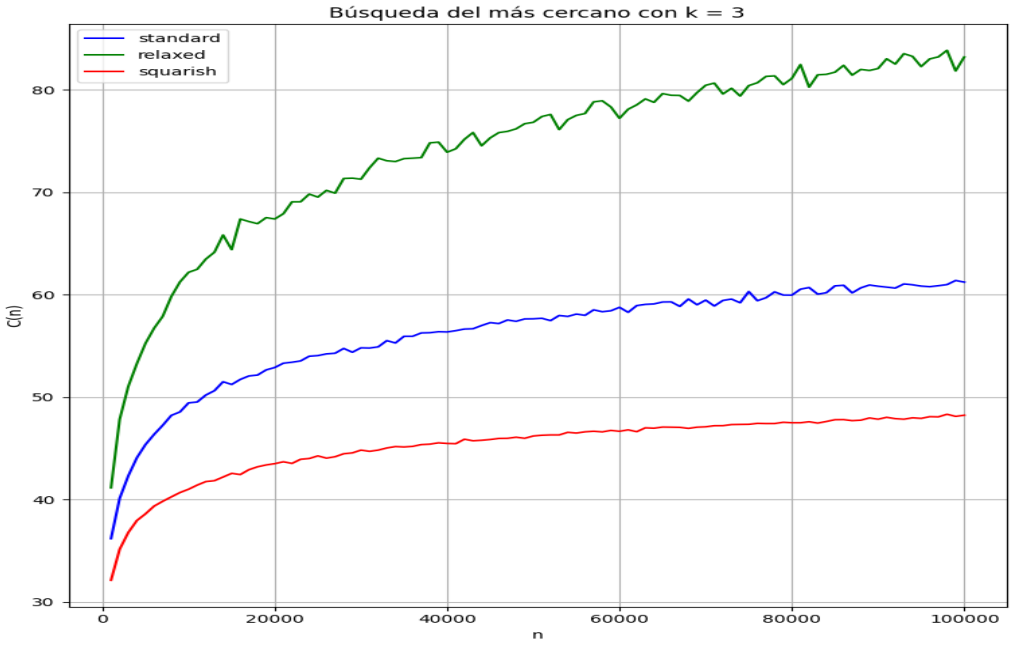

Figura 3: Gráfica para $k = 3$

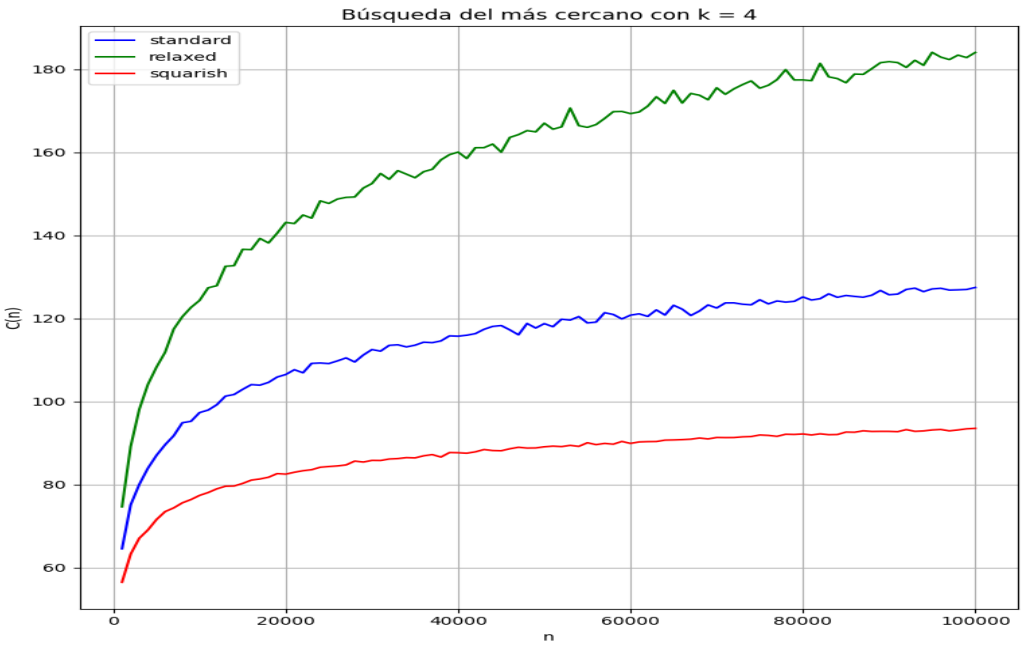

Figura 4: Gráfica para $k = 4$

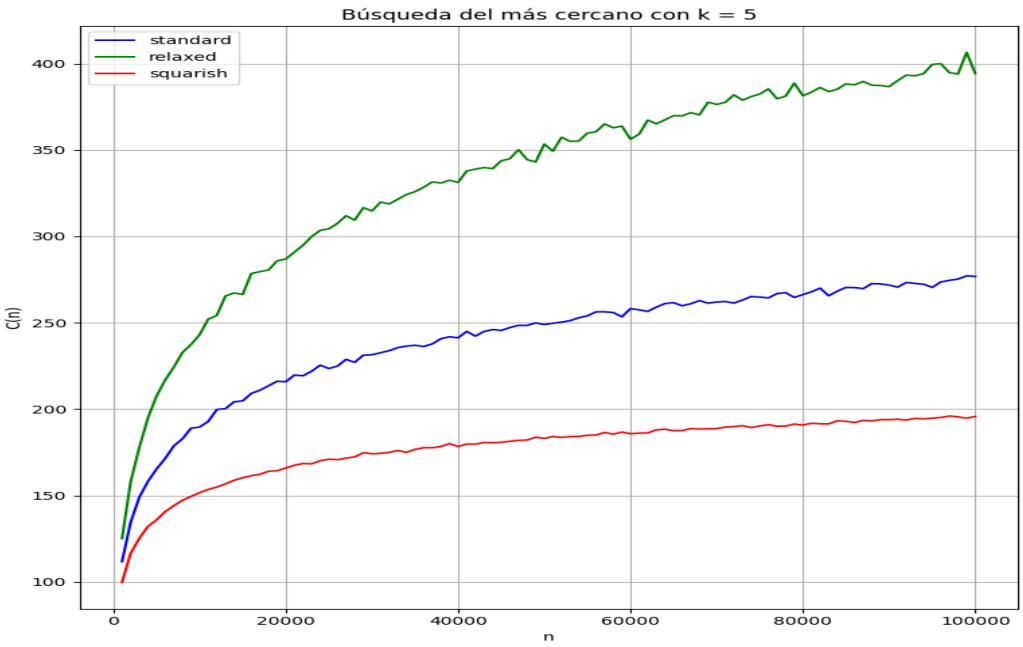

Figura 5: Gráfica para $k = 5$

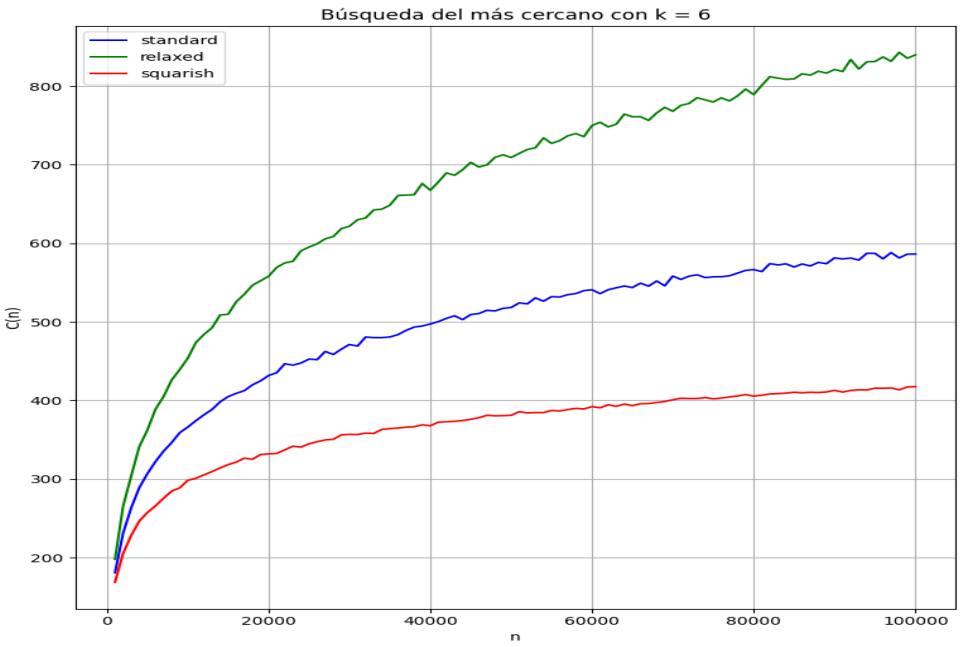

Figura 6: Gráfica para $k = 6$

En cuanto a la variabilidad de los datos, se ha decidido mostrar barras verticales de longitud 2σ para cada $n \in [1000, 2000, ..., 10^5]$, donde σ es la desviación estándar. Es decir, se espera que aproximadamente el 95 % de los datos caigan dentro de estas barras de error, incluyendo valores atípicos. También se ha optado por mostrar la variabilidad para solo una dimensión, $k = 3$, ya que mostrarlo todo haría demasiado extenso este documento. De todos modos, en el anexo se tendrá acceso a un repositorio con las gráficas para cada k-dimensión. Lo que cabe destacar es que a medida que aumenta el valor de k la desviación típica también aumenta. Además, puede parecer que tenemos mucha variabilidad, pero la cantidad de nodos visitados varia considerablemente entre diferentes árboles, debido a la aleatoriedad en la generación de los árboles y consultas. Por este mismo motivo se han realizado Q consultas en cada uno de los N árboles de n nodos, ya que interesa el coste en caso medio, haciendo muchas ejecuciones y quedándonos con la media. Se ha separado la información en tres gráficas, una para cada variante de *k-d tree*. Se puede ver en la Figura 8 como en los *relaxed k-d trees* hay bastante más variabilidad, debido a la aleatoriedad al escoger el discriminante en la creación de los árboles.

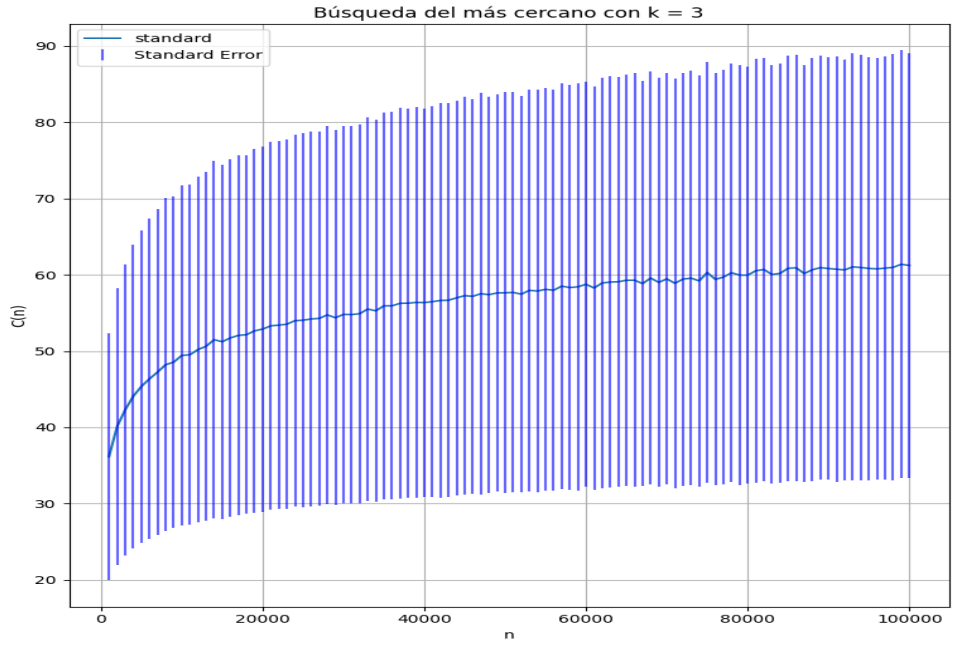

Figura 7: Variabilidad en *standard k-d trees*

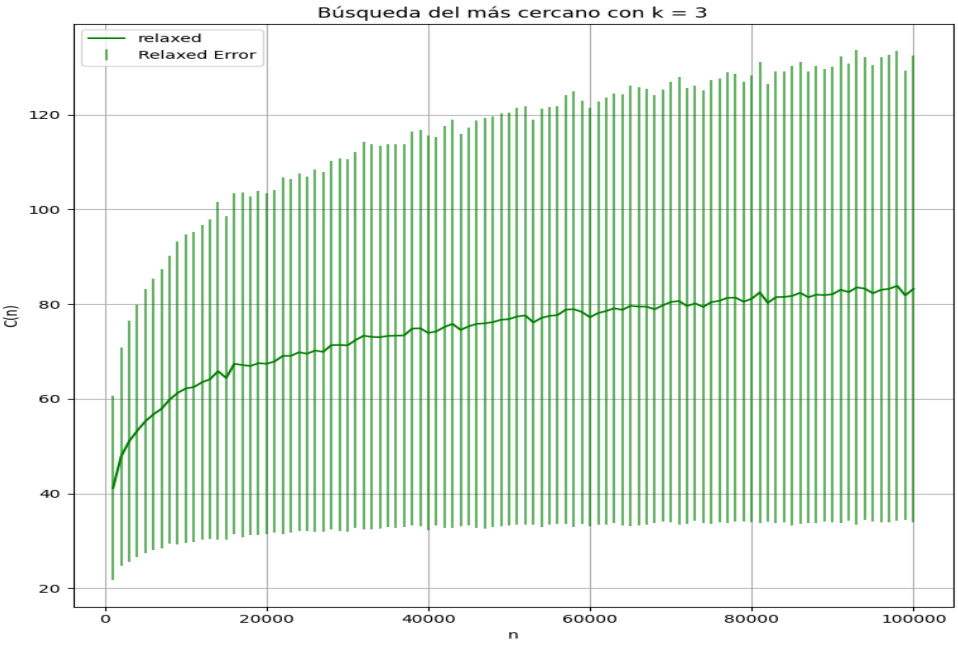

Figura 8: Variabilidad en *relaxed k-d trees*

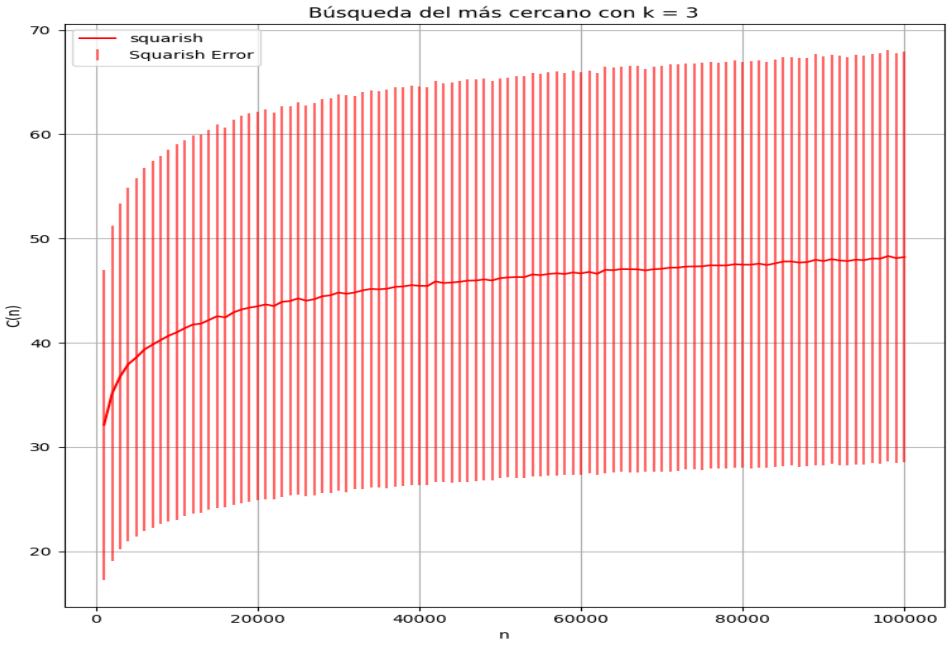

Figura 9: Variabilidad en *squarish k-d trees*

6. Deducción del coste medio de las búsquedas

Por su dificultad técnica, el análisis en caso medio de las búsquedas del vecino más cercano en árboles binarios k-dimensionales no se ha hecho formalmente (es un problema abierto) pero existen conjeturas serias que dicen que este coste es de la forma $\theta(n^{\zeta} + \log n)$, siendo n el número de nodos del árbol y donde $\zeta >= 0$ es un valor bastante cercano a 0 que depende de la dimensión k. El objetivo de esta parte del proyecto es encontrar experimentalmente (en base a los resultados obtenidos) el valor del exponente ζ para la distancia L_2 en función de la dimensión k y del tipo de *k-d tree*.

Para poder estimar el parámetro ζ primeramente se ha aplicado una transformación logarítmica a los datos, es decir, se pasa a tener $\log(C(n))$ en función de $\log n$. Al realizar esta transformación, se obtiene una relación casi lineal, ya que $\log(C(n)) = \zeta \cdot \log n + O(\log \log n)$. De este modo, se puede interpretar que el valor de ζ coincide con el valor de la pendiente de la recta que se ajusta a dicha función. Para poder encontrar esta recta que se ajusta de la mejor forma posible, maximizando el valor del coeficiente de determinación, se ha usado una librería de *Python* (*SciPy*) que nos permite realizar una regresión lineal usando el método de mínimos cuadrados. Una vez se tiene la ecuación de esa recta $y = m \cdot x + b$, se sabe que ζ es aproximadamente el pendiente m.

A continuación se muestra, para cada dimensión k, una gráfica donde se ve la transformación logarítmica de los datos y la recta que se ajusta (en cada variante de *k-d tree*), junto a las ecuaciones de dichas rectas, el valor R^2 o coeficiente de

determinación y la aproximación de ζ.

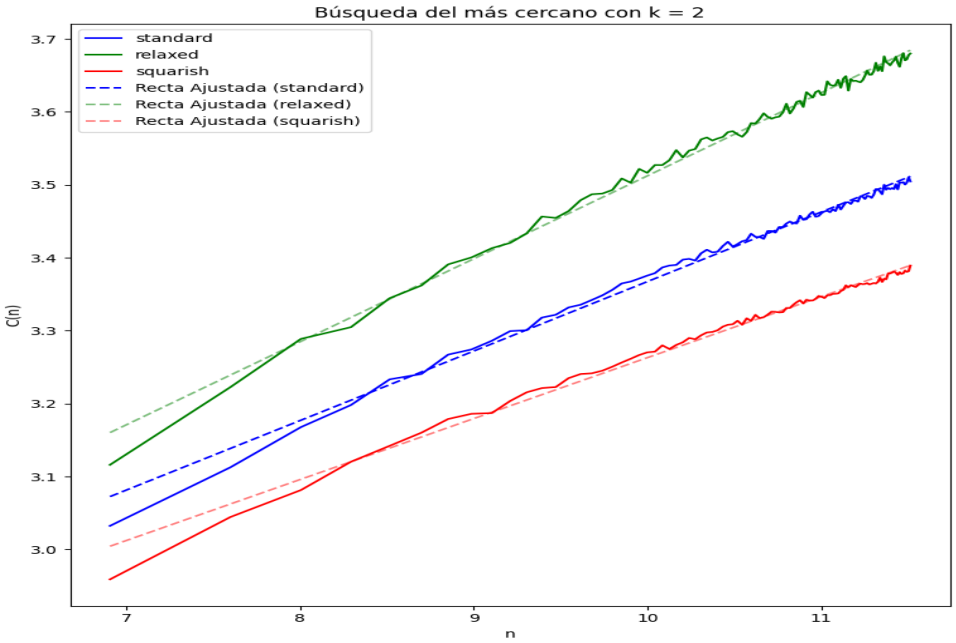

Figura 10: Gráfica para $k = 2$

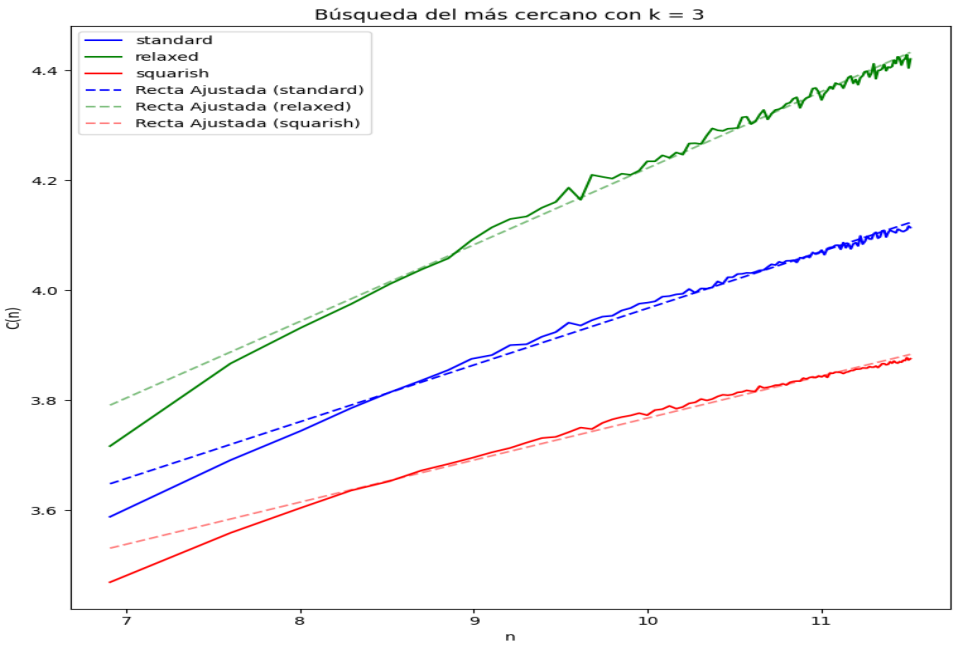

Figura 11: Gráfica para $k = 3$

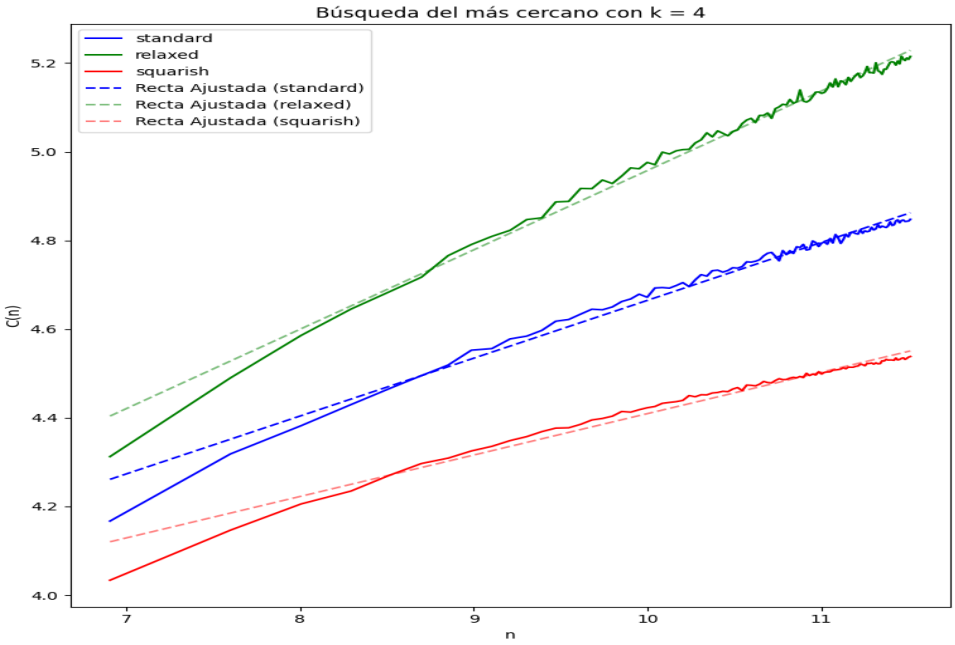

Figura 12: Gráfica para $k = 4$

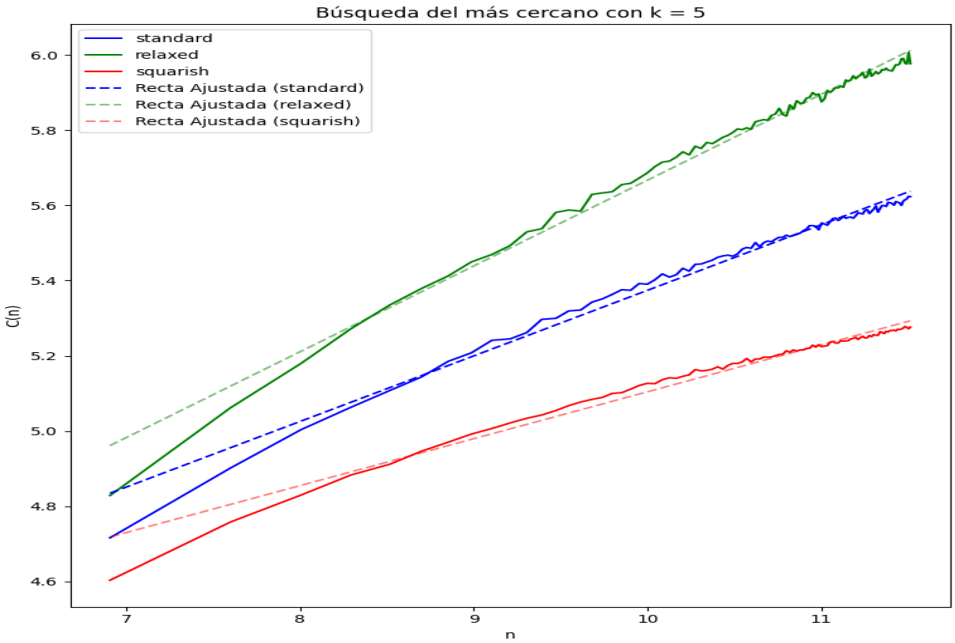

Figura 13: Gráfica para $k = 5$

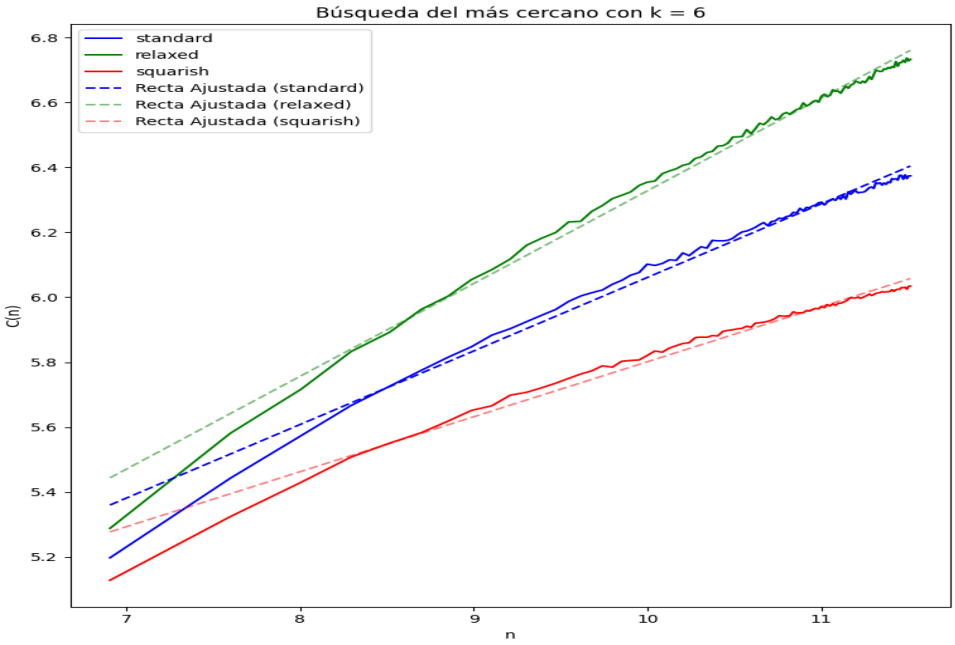

Figura 14: Gráfica para $k = 6$

Deducción del valor ζ para *k-d trees standard*			
k-dimensión	Ecuación de la recta	Valor de R^2	Valor de ζ
2	$y = 0{,}0953x + 2{,}4140$	0,9964	$\approx 0{,}0953$
3	$y = 0{,}1032x + 2{,}9354$	0,9946	$\approx 0{,}1032$
4	$y = 0{,}1306x + 3{,}3593$	0,9922	$\approx 0{,}1306$
5	$y = 0{,}1745x + 3{,}6288$	0,9929	$\approx 0{,}1745$
6	$y = 0{,}2267x + 3{,}4681$	0,9928	$\approx 0{,}2267$

Cuadro 1: Valor de ζ en los *standard k-d trees*

Deducción del valor ζ para *relaxed k-d trees*			
k-dimensión	Ecuación de la recta	Valor de R^2	Valor de ζ
2	$y = 0{,}1138x + 2{,}3738$	0,9967	$\approx 0{,}1138$
3	$y = 0{,}1394x + 2{,}8287$	0,9951	$\approx 0{,}1394$
4	$y = 0{,}1791x + 3{,}1666$	0,9956	$\approx 0{,}1791$
5	$y = 0{,}2284x + 3{,}6288$	0,9947	$\approx 0{,}2284$
6	$y = 0{,}2860x + 3{,}4681$	0,9956	$\approx 0{,}2860$

Cuadro 2: Valor de ζ en los *k-d trees relaxed*

Deducción del valor ζ para *squarish k-d trees*			
k-dimensión	Ecuación de la recta	Valor de R^2	Valor de ζ
2	$y = 0{,}0836x + 2{,}4268$	0,9958	$\approx 0{,}0836$
3	$y = 0{,}0766x + 3{,}0024$	0,9914	$\approx 0{,}0766$
4	$y = 0{,}0935x + 3{,}4746$	0,9880	$\approx 0{,}0935$
5	$y = 0{,}1249x + 3{,}8555$	0,9884	$\approx 0{,}1249$
6	$y = 0{,}1695x + 4{,}1065$	0,9894	$\approx 0{,}1695$

Cuadro 3: Valor de ζ en los *k-d trees squarish*

El coeficiente de determinación es prácticamente 1, por lo que el modelo de regresión se ajusta muy bien a los datos. Se puede ver también como, a medida que la dimensión crece, el valor de ζ también crece en las tres variantes. Además, también se ve como ζ es menor en los *squarish k-d trees* y peor en los *relaxed*.

7. Conclusiones

Los resultados experimentales han permitido comprobar las diferencias entre tres variantes de *k-d tree* así como aproximar el coste asintótico en caso medio al realizar búsquedas del vecino más cercano. Las principales conclusiones a las que se ha llegado son:

- La variante de *k-d tree* con mejor rendimiento para las búsquedas del vecino más cercano dentro de las que se han visto se trata de la *squarish*, seguida de la *standard* y finalmente la *relaxed*. Pero esto no quiere decir que las últimas variantes no sirvan, ya que seguro que deben tener otras ventajas en según qué operaciones.

- Los *standard* y *squarish k-d trees* ofrecen una varianza similar, mientras que los *relaxed* se han mostrado mucho más inestables. Esto puede ser debido a la aleatoriedas a la hora de escoger los discriminantes.

- El coste de hacer búsquedas del vecino más cercano varía de forma notable en función de la dimensión k del árbol.

- El parámetro exponencial de la función asintótica que describe el coste de las búsquedas en caso medio se puede aproximar fielmente usando una recta.

- Almacenar los puntos k-dimensionales de la forma en la que lo hacen los *k-d trees* es crucial para poder realizar consultas asociativas de forma eficiente.

8. Descripción y valoración del proceso de aprendizaje

8.1. Metodología

Nuestro enfoque de trabajo en equipo se ha basado en una comunicación constante, manteniendo canales abiertos para compartir regularmente nuestros avances y problemas. En cuanto a la programación, optamos por dividir el código entre todos, permitiendo que cada miembro se encargara de una parte específica del proyecto. Además, nos aseguramos de revisar y entender el código de los demás, lo que nos ayudó a tener una visión global del proyecto. La redacción del informe la llevamos a cabo de manera colaborativa, con cada uno de nosotros centrados en la sección en la que habíamos trabajado con mayor profundidad. Esta estrategia de trabajo en equipo nos ha permitido garantizar la cohesión y calidad del proyecto en su conjunto.

8.2. Valoración del proceso de auto aprendizaje

Este proyecto nos ha permitido valorar como de importante es personalizar las estructuras de datos que usamos pensando en el uso que se le va a dar después a los datos. En este caso, hemos aprendido a la perfección como crear y manipular los *k-d trees*, centrándonos en tres de sus variantes. Durante el proceso de aprendizaje, también valoramos muy positivamente el conocimiento adquirido en diferentes tecnologias que nos han ayudado a hacer este informe. Entre ellas, podemos destacar:

- LaTeX: Nunca antes habíamos usado este procesador de textos. Estamos muy satisfechos con haber decidido utilizarlo, ya que somos conscientes de que en un futuro nos va a ser de gran ayuda.
- *Multithreading*: Hemos usado *multithreading* en c++ con la librería *pthread*, cosa que nos ha permitido reducir de forma considerable el tiempo de espera para recoger nuestros datos.
- Análisis estadístico con *Python*: Hemos decidido usar este lenguaje para realizar las diferentes regresiones, gráficas y cálculos estadísticos que habéis podido ver con anterioridad. No lo habíamos usado nunca para este propósito y nos ha parecido muy sencillo e útil.

Referencias

[1] Conrado Martínez Parra, V. Estivill-Castro y Amalia Duch Brown. *Randomized K-dimensional binary search trees*. Universitat Politècnica de Catalunya. (1998-10-02)

[2] Maria Mercè Pons Crespo. *Design, Analysis and Implementation of New Variants of Kd-trees. Master Thesis*, Universitat Politècnica de Catalunya. (2010-09-08)

[3] Luc Devroye, Jean Jabbour y Carlos Zamora-Cura. *Squarish k-d trees. SIAM Journal on Computing.* (2000)

[4] Amalia Duch Brown. *Design and analysis of multidimensional datastructures. PhD thesis*, Universitat Politècnica de Catalunya. (2004)

[5] *Overleaf LaTeXdocumentation*: https://www.overleaf.com/learn

[6] *Python. SciPy documentation*: https://docs.scipy.org/doc/scipy/

[7] *Python. Matplotlib documentation*: https://matplotlib.org/stable/index.html

[8] *Pthread. Pthread documentation*: https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

ANEXO

A través del siguiente enlace se puede acceder a un repositorio de *GitHub* donde se puede ver todo el código del proyecto, gráficas e información relacionada: https://github.com/bielaltes/Practica-A-FIB