ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES
UNIVERSIDAD POLITÉCNICA DE MADRID

TRABAJO DE FIN DE GRADO

ANÁLISIS Y DESARROLLO DE ALGORITMOS EFICIENTES BASADOS EN ÁRBOLES PARA RESOLVER EL PROBLEMA DE VECINDAD

JULIO 2017

Pablo Hiroshi Alonso Miyazaki

Director del trabajo fin de grado:

Santiago Tapia Fernández

Agradecimientos

Quiero expresar mi agradecimiento a mi tutor Don Santiago Tapia Fernández, por haberme invitado a colaborar en su proyecto y haber guiado mi aprendizaje estos últimos meses. Sus consejos, atención y entusiasmo han hecho de la realización de este trabajo, no sólo una experiencia enriquecedora académicamente, sino también personal, motivándome a seguir aprendiendo más allá de la tarea propuesta, implicándome de lleno en el trabajo y disfrutando con la realización del mismo.

También, quiero agradecer a familia y amigos no sólo su apoyo y cariño estos últimos meses de intenso trabajo, sino la ayuda que me han brindado durante los cuatro años de grado. Sin ellos no habría podido dar lo mejor de mí y encontrar el ánimo para seguir esforzándome en todo momento. Gracias de corazón.

Índice general

RESUMEN EJECUTIVO

El objetivo de este trabajo es proponer una solución al problema de la vecindad de puntos (*neighborhood problem*) para su aplicación en métodos numéricos sin mallado (*meshfree methods*), como el análisis de elementos finitos. Se determinan los rangos y situaciones óptimas de aplicación de la solución, así como la comparativa de la misma frente algoritmos existentes.

El problema de la vecindad de puntos o problema de proximidad (*nearest neighbor search NNS*), es un problema de optimización. Consiste en determinar cuántos y qué puntos pertenecientes a un conjunto se encuentran suficientemente próximos entre sí como para considerarse vecinos. Una tarea trivial como puede ser la búsqueda del punto más cercano a otro, se puede convertir en una tarea con un coste computacional inaceptable cuando el problema escala en búsquedas del orden de cientos de miles dentro de un conjunto de millones de puntos. Métodos de fuerza bruta que calculan las distancias de cada punto del conjunto al punto objetivo, pierden todo valor a gran escala debido a su lentitud, por lo que es necesario afrontar el problema mediante otros métodos.

La solución a este problema tiene numerosas aplicaciones en diversos campos de la ingeniería, así como en otras ciencias. Su generalización permite la aplicación de la misma en tareas tan dispares como el tratamiento de imágenes (donde los puntos son píxeles y la distancia es el parecido entre colores), la detección de plagios (donde los puntos son palabras y la distancia es la similitud entre estas), o métodos numéricos (donde los puntos son puntos en el espacio y la distancia es la distancia euclídea entre ellos).

Este trabajo se centra en la resolución del problema para esta última aplicación. Escalar problemas de elementos finitos permite calcular y diseñar con mayor precisión estructuras o máquinas. Por ejemplo, en el campo de la ingeniería, esto se podría traducir en la fabricación de aerogeneradores y turbinas más eficientes. Concretamente, la resolución del problema de la vecindad de puntos es sumamente útil en los métodos númericos sin mallado, pues a diferencia del método clásico de análisis con mallado, los vecinos de cada punto no son fijos y es necesario recalcularlos en cada iteración. La ausencia de mallado permite resolver simulaciones de dinámica de fluidos, o de grandes deformaciones en sólidos (para materiales con comportamiento plástico), donde la situación de cada punto puede variar significativamente de una iteración a la siguiente, y es necesario determinar sus puntos vecinos de forma dinámica.

Se propone primero una solución al problema, y posteriormente se comparará la misma respecto a otras soluciones existentes. En la solución propuesta, se cubren las búsquedas para d dimensiones, y se permitirán búsquedas para hallar los puntos dentro un radio r, así como los k puntos más próximos del punto objetivo. Para ello, se preprocesa dinámicamente el conjunto de datos, dividiendo los puntos del espacio en celdas y construyendo en ellas árboles de búsqueda binarios en caso de haber suficiente densidad de puntos, optimizando el empleo de métodos de fuerza bruta o búsqueda binaria según la celda.

La solución se compara respecto a las existentes para un conjunto de problemas en tres dimensiones y se determina en qué casos cuál es la mejor solución, la generalidad de los resultados y su extrapolación teórica a dimensiones superiores. Así mismo, se validarán las previsiones teóricas del coste computacional de la solución propuesta respecto de los resultados obtenidos en la práctica.

CÓDIGOS UNESCO

110213 FUNCIONES RECURSIVAS

120324 TEORIA DE LA PROGRAMACION

120406 GEOMETRIA EUCLIDEA

120601 CONSTRUCCION DE ALGORITMOS

EXECUTIVE SUMMARY

The objective of this project is to propose a solution to the neighborhood problem and its application to meshfree methods in numerical analysis, such as the finite element method. This is, to determine its range and optimal situations in which it can be applied while comparing the solution to already existing algorithms.

The nearest neighbor search NNS is an optimization problem.The problem is to find the point in a given set that is the closest (or most similar) to a given point. An easy task such searching the closest point in a set to a given point can turn into a computational complexity nightmare when the problem scales to multiple querys in big data sets. Brute force methods that evaluate the distances between the points in a set to the given point, are useless in large problems because of their poor performance. The problem must be solved with other methods.

The solution to these problems has multiple applications in the engineering field, aswell as in other sciences. Its generalization allows it to solve very different tasks such image pocessing (being the points pixels, and the distance the similarity of their colours), plagiarism detection (being the points words, and the distance the similarity between them), or numerical analysis (being the points, points in the space, and the distance the euclidean distance between them).

This project is centered around the solution of the problem for the numerical analysis. The scaling of finite element methods enables more accurate calculations and designs of structures and machines. For instance, in the engineering field, this could be translated to more efficient turbines. Specifically, solving the neighborhood problem is very useful in meshfree methods because unlike the classical numerycal analysis with meshes, the neighbors are not fixed in the space and they need to be recalculated in each iteration. The lack of a mesh enables the solving of fluid dynamics simulations, or great deformation in solids (for materials with plastic behaviour), where the place of each point may vary significantly from one iteration to the next, being a must to determine its neighbor points dynamically.

In the first place, a solution to the problem will be proposed, and then it will be compared against other existing solutions. The proposed solution allows querys for d dimensions, querying the neighbor points within a radius r, aswell as the k nearest points to the given point. Because of this, the data set will be preprocessed, dividing the space in cells and building binary search trees when there are enough points in a cell, optimizing the use of brute force methods or binary search according to the number of points in a cell.

The solution will be compared with existing ones for a set of three dimensional problems and the best solution, the generality of the results and its theoretical extrapolation will be determined for each case. Moreover, the theoretical computational complexity of the solution will be validated with the results obtained in practice.

1. Introducción

En este capítulo se explicará la motivación que ha llevado a la realización de este trabajo de fin de grado, así como los objetivos que se pretenden cumplir con la realización del mismo.

Motivación personal

La elección del tema y contenido del presente trabajo de fin de grado provienen de mi deseo de adquirir la capacidad de programar en lenguajes de alto de nivel y conocer la aplicación de los mismos a problemas de la industria real.

La preparación teórica necesaria para la realización de este trabajo la he podido ir adquiriendo durante la realización del mismo, pues el problema de la vecindad de puntos es principalmente un problema de implementación. Mi esfuerzo se ha concentrado en la adquisición de las habilidades y conocimientos necesarios para el uso de lenguajes de alto nivel, tal y como deseaba, sirviendo la realización del trabajo no sólo como una introducción a la investigación en este campo, sino también como un complemento a mi formación.

Con la guía y apoyo del tutor he podido acceder al problema de forma gradual, incrementando escalonadamente la dificultad de los problemas a resolver y adquiriendo poco a poco la capacidad de afrontar nuevos retos.

Creo en muchos sentidos que he escogido el mejor trabajo de fin grado posible para mi desarrollo académico en este campo pues el problema de la vecindad de puntos trata alguna de las áreas fundamentales de las ciencias de la computación. Estas son, complejidad de algoritmos y estructuras de datos. Este hecho me ha permitido introducirme de forma natural en esta rama de la ingeniería e ir aprendiendo sobre el tema mientras trabajaba. El tener un problema concreto en mente, para cuya resolución es necesaria el estudio de estas áreas, ha amenizado mi aprendizaje sobre las mismas. Aprendizaje que de otra forma sospecho que puede llegar a ser muy árido si se trata únicamente desde un punto de vista teórico.

En conclusión, me alegro de haber escogido un trabajo que me haya permitido aprender sobre las áreas en las que estaba interesado y no por ello dejar de estar contribuyendo a una aplicación tan útil para la ingeniería industrial como pueden ser los métodos numéricos sin mallado.

Objetivos

Los objetivos fundamentales del trabajo son:

- Mejora y desarrollo de una implementación existente que soluciona el problema de la vecindad de puntos.
- Comparativa de implementaciones existentes respecto a la elaborada.
- Análisis de los resultados obtenidos y determinación de casos de uso óptimos.

La implementación sobre la que se trabaja es una librería de cabeceras desarrollada por S. Tapia en colaboración con A. Beltrán y I. Romero [1]. En concreto, se aprovechará la idea de dividir el espacio en celdas en función de la codificación en coma flotante de los puntos del conjunto estudiado. La mejora significativa que se pretende añadir es dotar a estas celdas de un algoritmo de búsqueda a bajo nivel. De esta forma se posibilita el uso de diferentes métodos de búsqueda en función de las características de los puntos pertenecientes a la celda. Se pretende de esta forma acelerar la búsqueda de vecinos del conjunto, mejorando los resultados obtenidos. Se describe el algoritmo y su implementación en Capítulo 4 y Capítulo 5.

Se comparará con librerías existentes como [2], cuya elección se discute en sec. 3.3. Y se discutirán los resultados obtenidos en Capítulo 7 y Capítulo 8.

Para el cumplimiento de estos objetivos se ha seguido el siguiente procedimiento:

1. Adquisición de los conocimientos y habilidades necesarios para el uso de un lenguaje de alto nivel, así como de las herramientas necesarias para ello.
2. Estudio de implementaciones existentes y del trabajo en el que se basa la solución propuesta que se pretende mejorar.
3. Diseño del algoritmo y su implementación.
4. Prueba exhaustiva de los algoritmos con múltiples bancos de pruebas para asegurar su correcto funcionamiento.
5. Asegurar la viabilidad de la implementación para su paralelización en múltiples núcleos procesadores.
6. Comparación con otras implementaciones existentes y estudio de resultados.

Se muestra en Capítulo 9 la distribución temporal del trabajo y los recursos empleados para la realización del mismo.

2. Tecnología

En este capítulo se reseñarán las herramientas y tecnologías empleadas para la realización de este trabajo, así como los motivos que han llevado a su elección.

2.1. Lenguajes de programación: C y C++

Se ha escogido trabajar con estos lenguajes de programación debido a su flexibilidad a la hora de trabajar tanto a bajo nivel como a alto nivel. Los motivos que han llevado a la elección de estos lenguajes son:

- Se pretende estructurar el código con mentalidad orientada a objetos. La abstracción del problema de la vecindad de puntos hará su desarrollo más intuitivo y cercano al problema matemático que representa. Se requerirán por tanto lenguajes como C++, Java o Python que soporten la orientación a objetos.
- La aplicación a desarrollar es exigente respecto a rendimiento y velocidad. De querer optimizar a bajo nivel, se requerirá un lenguaje cercano a código máquina como es C. Lenguajes como Java, ejecutados en máquinas virtuales o Python, lenguaje interpretado, pueden presentar restricciones a la hora de acceder a características de bajo nivel.
- Se quiere elaborar una implementación que permita un uso generalizado para cualquier dimensión y tipo de búsqueda. Características de C++ como la sobrecarga de funciones o la existencia de *templates* o plantillas, permiten desarrollar un código más genérico que da soporte a una mayor gama de aplicaciones.
- Se posiblita trabajar en múltiples procesadores para acelerar las búsquedas, característica a la que Python no da sosporte.

Si bien la portabilidad de código escrito en C y C++ requiere de un mayor trabajo, no se considera esta una característica tan crítica como las anteriores debido a que el uso de esta implementación está orientada a personas con suficientes recursos como para poder emplear esta implementación en su entorno nativo.

2.2. Repositorio: Git

Git es un software de control de versiones para el mantenimiento de código de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

Se ha decidido emplear Git como software de control de versiones debido a la necesidad de trabajar en un proyecto colaborativo, el tener un control de versiones robusto y para un mejor seguimiento de los avances realizados en el trabajo. En concreto, se ha elegido el servicio de alojamiento basado en web "Bitbucket".

2.3. *Builder:* Cmake

Cmake es una familia de herramientas diseñada para construir, probar y empaquetar software. Cmake se utiliza para controlar el proceso de compilación del software usando ficheros de configuración sencillos e independientes de la plataforma que se use.

Se ha escogido Cmake ya que se van a emplear múltiples librerías y se necesita una forma de enlazarlas con el proyecto de forma robusta. Así mismo, Cmake permite desligar el código fuente del directorio de compilación, lo que ha facilitado trabajar en el código en varios entornos de desarrollo de forma rápida y segura. Por último, la flexibilidad de Cmake ha permitido una realización más cómoda de las comparativas de Capítulo 7 empleando la característica de *testing* de esta herramienta.

2.4. *Compiler:* GCC

El *compiler* o compilador empleado es la colección de compiladores GNU o GCC de sus siglas en inglés. Este es un sistema de compilación producido por el Proyecto GNU que da soporte a varios lenguajes de programación. Es el compilador estándar en la gran mayoría de Sistemas Operativos de la familia Unix, y uno de los más ampliamente conocidos.

Se ha elegido GCC debido a que se distribuye de forma gratuita y está especialmente optimizado para C++.

2.5. Sistema operativo: Linux

Durante el desarrollo de la implementación, se ha empleado el subsistema de Windows 10 para Linux que permite la ejecución de software de Linux directamente en Windows sin tener que recurrir a máquinas virtuales. A efectos prácticos para el desarrollo del trabajo ha sido como trabajar con Linux como sistema operativo.

Se ha escogido Linux, y en concreto, su distribución de Ubuntu debido a la facilidad de instalación de librerías respecto a otros sistemas operativos así como por la integración natural de Git. Finalmente, el hecho de ser una distribución libre y de código abierto, permiten un acceso gratuito a su uso y un fácil mantenimiento.

2.6. Entorno de desarrollo: Codeblocks

Para la compilación, *debugging*, y ejecución de los programas de prueba y de *benchmarking* generados durante el desarrollo de la implementación, se ha empleado Codeblocks. Codeblocks es un entorno de desarrollo integrado (*IDE*), gratuito, multiplataforma, que da soporte a mútiples compiladores como GCC, Clang y Visual C++. Está orientado hacia el desarrollo de aplicaciones en C y C++, lo que lo hace muy apropiado para este trabajo y se encuentra entre los entornos de desarrollo para los que Cmake puede generar archivos de proyecto.

Se han empleado tanto las versiones para Windows, en los primeros estadios del trabajo en los que no ha sido necesario emplear librerías de difícil portabilidad ya que se trabaja con programas piloto, como la de Linux cuando ya ha sido necesario compilar la implementación en sí.

2.7. Editor de texto: Notepad++

Notepad++ es un editor de texto y código fuente para Microsoft Windows. Da soporte a edición en pestañas, lo que permite trabajar con múltiples archivos abiertos de forma simultánea y en una única ventana. Notepad se distribuye como software gratuito.

Se ha elegido Notepad para escribir el código fuente debido a su flexibilidad a la hora de ser personalizado, ser más ligero que el editor de texto integrado de Codeblocks y la familiaridad con su uso.

2.8. Procesador de texto: Lyx

Para la elaboración de esta memoria se ha empleado el procesador de texto Lyx, de código abierto. Este es un procesador de texto que se basa en la filosofía: “lo que ves es lo que quieres decir”. Durante la edición de un documento, se muestra en pantalla una aproximación del documento que finalmente es generado, encargándose Lyx de su correcta maquetación e identación.

Se ha elegido esta herramienta debido al deseo de dotar a la memoria de un aspecto más formal y elegante, sin por ello tener que recurrir el uso de herramientas como LaTeX con una curva de aprendizaje inicial poco amigable para usuarios noveles, o recurrir a procesadores de texto de pago como Microsoft Word que no garantizan necesariamente un buen resultado.

3. Estado del arte

En este capítulo se explicará el estado actual de la investigación en este campo, así como las implementaciones existentes que dan solución al problema de la vecindad de puntos.

3.1. El problema de la vecindad de puntos

El problema de la vecindad de puntos o problema de proximidad (*nearest neighbour search NNS*), es el problema de optimizar la búsqueda del punto perteneciente a un conjunto dado, que sea el más cercano a un punto objetivo. Su aplicación consiste en determinar cuántos y qué puntos pertenecientes a un conjunto se encuentran suficientemente próximos entre sí como para considerarse vecinos [3].

La solución a este problema tiene numerosas implementaciones en diversos campos de la ingeniería y fuera de ella. Cabe citar su uso en minería de datos [4], reconocimiento de patrones y clasificación [5], inteligencia artificial[6] o compresión de datos [7].

Formalmente el problema de proximidad se formula de la siguiente forma:

> *Dado un conjunto S de puntos contenido en un espacio M y un punto de referencia $q \in M$, encuentre el punto en S más cercano a Q.*

En la mayoría de los casos, M es un espacio métrico[1], simétrico[2] y que satisface la desigualdad triangular. Todas ellas características propias de un espacio de vectorial de dimensión *d*, donde el radio de proximidad se puede medir mediante la distancia euclidiana, que es la particularización del problema que se estudia en este trabajo.

3.1.1. Terminología

De ahora en adelante se denotarán los siguientes conceptos con la terminología empleada en la literatura existente sobre el tema. Se pretende de esta forma una mayor

[1] Un espacio métrico es un conjunto que lleva asociada una función distancia, es decir, que esta función está definida sobre dicho conjunto, cumpliendo propiedades atribuidas a la distancia, de modo que para cualquier par de puntos del conjunto, estos están a una cierta distancia asignada por dicha función.

[2] Un espacio maximalmente simétrico (EMS) es un espacio métrico en el que puede definirse el concepto de dimensión y donde el grupo de simetría tiene la dimensión máxima posible.

consonancia con las investigaciones existentes y una mayor naturalidad a la hora de trasladar los resultados obtenidos en este trabajo respecto a otros.

Terminología que se va a emplear: *Searching point*, *Querying point*, *Cells, Trees, Hash-Table*:

- ***Searching point*:** O simplemente *searching*, punto perteneciente al conjunto de puntos sobre el que se realizarán las búsquedas.
- ***Querying point*:** O simplemente *querying*, punto de referencia o búsqueda respecto al cual se buscarán puntos vecinos en el conjunto objetivo.
- ***Cells:*** O celdas, partición del espacio, generalmente con límites ortogonales y paralelos a los ejes de referencia en los que se agrupan *searching points.*
- ***Trees:*** O árboles de búsqueda, son estructuras de datos que simulan la estructura jerárquica de un árbol, con un nodo base o raíz a partir del cual "crecen" subárboles o ramas [8]. En este trabajo se trabajará intensivamente con *kd-Trees*, consultar Apéndice B.
- ***Hash-Table:*** O *Hash-Maps* , o tablas de dispersión, son estructuras de datos que asocian llaves o claves con valores, y aseguran un tiempo de acceso constante a sus elementos [9].

3.1.2. Tipos de soluciones

Las soluciones a este problema se pueden clasificar según su precisión o el tipo de búsqueda que se realice, no siendo estas características excluyentes entre ellas.

3.1.2.1. Precisión

Métodos exactos Se determinan de forma exacta los puntos *searching* más cercanos al *querying* respecto al que se realiza la búsqueda.

Métodos aproximados Se obtienen aquellos puntos *searching* cuya distancia al *querying* es como mucho c veces la distanca del *querying* a sus verdaderos vecinos más cercanos, donde c es una constante. El atractivo de este método, es que en muchos cascos, los vecinos aproximados son tan buenos como los exactos [3], [10]. Formalmente:

Con S como conjunto de *searchings* y Q como conjunto de *queryings* :

$$Dado\, un\, \varepsilon > 0,\, decimos\, que\, un\, punto\, s \in S\, es\, un\, (1+\varepsilon) vecino\, aproximado\, de\, q \in Q\, si\, dist(s;q) \leq (1+\varepsilon) dist(s*;q)$$

donde $s*$ es el verdadero vecino de q. En otras palabras, s tiene un error relativo menor que ε respecto del verdadero vecino.

3.1.2.2. Búsquedas

Mejor vecino Se obtiene aquel punto *searching* **único** más cercano al *querying*. Formalmente:

Con S como conjunto de *searchings* y Q como conjunto de *queryings* :

$$Decimos\, que\, un\, punto\, s\ \in S\, es\, el\, mejor\, vecino\, de\, q \in Q\, si\, dist(s;q) = \min_{p \in S} dist(p;q)$$

Radio fijo Se obtienen todos aquellos puntos *searching* cuya distancia al *querying* es menor que una distancia euclídea a la cual se denomina r. Esto equivale a hallar todos los puntos dentro de la hiperesfera centrada en el *querying* y de radio r. Formalmente:

Con S como conjunto de *searchings* y Q como conjunto de *queryings* :

$$Dado\, un\, r > 0\, decimos\, que\, un\, punto\, s\ \in S\, es\, un\, vecino\, de\, q \in Q\, si\, dist(s;q) \leq r$$

Donde r es el radio de la hiperesfera centrada en q.

k-Vecinos Se obtienen los k puntos *seraching* más próximos al *querying*. Formalmente:

Con S como conjunto de *searchings* y Q como conjunto de *queryings* :

$$Decimos\, que\, un\, conjunto\, de\, puntos\, K\ \in S\, son\, los\, k\, vecinos\, de\, q \in Q\, si \max_{k \in K} dist(k;q) \leq \min_{s \in \overline{S \cap K}} dist(s;q)$$

Donde K es el conjunto de los k vecinos más próximos al querying.

3.2. Análisis formal, complejidad y coste computacional

3.2.1. Maldición de la dimensión

La maldición de la dimensión se refiere a varios fenómenos que se producen en el procesado de datos en espacios de un número de dimensiones elevadas. Estos fenómenos no se producen en espacios de pocas dimensiones como el espacio físico tridimensional de nuestro día a día [11].

El problema más común se produce cuando la dimensionalidad del espacio crece, y el volumen del espacio crece tan rápido que los datos disponibles se dispersan consecuentemente. Esta dispersión es problemática para cualquier método que requiera de significación estadística. Para obtener un resultado estadísticamente válido, la cantidad de datos necesaria para sostener este resultado crece exponencialmente con el orden dimensional del espacio. Así mismo, la capacidad para organizar y buscar datos se basa en la detección de áreas donde los objetos a estudiar presentan propiedades similares. Con datos de múltiples dimensiones, todos los objetos tienden a dispersarse y ser disimilares, lo que supone un problema a la hora de organizar datos por los métodos clásicos.

Este hecho en concreto afecta al objeto de estudio, pues si bien la mayoría de aplicaciones de la implementación que se desarrolla están dirigidas a espacios tridimensionales, se pretende dar una solución multidimensional. El uso de *kd-Trees* para espacios con un gran número de dimensiones degrada sus propiedades [10], provocando que sus resultados apenas mejoren, o incluso empeoren, respecto a aquellos obtenidos por fuerza bruta. Es por ello que en la implementación de este trabajo se tienen en cuenta estos casos, como se detalla en Capítulo 4, renunciando a la construcción de *kd-Trees* para espacios de gran número de dimensiones a favor de métodos de fuerza bruta combinados con *Hash maps.*

Cabe citar sin embargo que para conjuntos de datos multidimensionales reales, estos raramente están uniformemente distribuidos y es posible tratarlos según las relaciones entre sus atributos para conseguir mejores resultados [12].

3.2.2. Complejidad del algoritmo para la búsqueda del mejor vecino

La mayoría de trabajos no cubren el estudio formal de búsquedas de radio fijo o de k-vecinos debido a la dificultad de desarrollar expresiones para búsquedas de múltiples vecinos, pues los resultados son fuertemente dependientes del conjunto de *searchings* y de la búsqueda a realizar, influyendo su distribución en el espacio, su dispersión estadística, así como el radio de búsqueda empleado, o la cantidad de vecinos que se pretende obtener.

Por esta razón, se analiza en esta sección la complejidad para la búsqueda del mejor vecino únicamente, cuyas cotas formales fueron determinadas hace casi dos decadas

en [10]. A continuación se presenta una síntesis de los resultados obtenidos en este estudio.

Se denominará como n al número de puntos *searching* y d a la dimensión del espacio estudiado.

Distancia

Si bien la distancia en un espacio métrico se puede expresar de diversas formas[3], el cálculo de todas ellas tendrá una complejidad *O(d)*. Por tanto, extrapolando para la totalidad del conjunto, el problema de la búsqueda del mejor vecino se resolvería por fuerza bruta en *O(dn)*. Estos resultados son de demostración inmediata, pues los cálculos realizados a la hora de determinar una distancia son linealmente proporcionales al número de dimensiones en la que esta se evalua, y el cálculo total de distancias es linealmente proporcional al número de distancias a puntos calculadas. Para la búsqueda de los k mejores vecinos, habría que ordenar las distancias obtenidas para seleccionar aquellas más cercanas. Los algoritmos de ordenación más comúnmente empleados, como *quicksort* tienen una complejidad de *O(nlogn)* [13].

Preprocesamiento de datos y búsquedas

La información presentada a continuación se adapta para la solución a implementar a partir de las conclusiones a las que se llega en [10].

Caso promedio Para puntos uniformemente distribuidos se demuestra que es posible obtener por simple descomposición regular del espacio, soluciones al problema del mejor vecino en *O(n)* en memoria y *O(logn)* en tiempo de búsqueda empleando *kd-Trees.*

Estos resultados coinciden con una hipótesis inicial en el que se consideran tantos nodos en el árbol como puntos *searching* a estudiar, ocupando n espacio en memoria; y búsquedas binarias de *O(logn)* (con $log_2 n$), pues en cada nivel del árbol se descarta la mitad de los puntos a ese nivel.

Sin embargo, estos métodos se degradan a medida que la dimensionalidad del espacio crece. Las constantes ocultas por la notación en O, crecen para la búsqueda asintótica en al menos un factor 2^d.

[3] Las distancias más comúnmente empleadas son:

- Distancia Manhattan: $d_1(p,q) = \|p-q\|_1 = \sum_{i=1}^{n} |p_i - q_i|$
- Distancia Euclídea: $d_2(p,q) = \|p-q\|_2 = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$
- Distancia Máxima: $d_\infty(p,q) = \|p-q\|_\infty = \max_{i \in \{1,..,n\}} |p_i - q_i|$

Peor caso Si se estudia el coste computacional para el peor caso, se demuestra que una solución ideal sería preprocesar los puntos en *O(nlogn)*, en un estructura de datos que ocupe *O(n)* espacio en memoria y que permita realizar búsquedas en *O(logn)*.

Desafortunadamente, para dimensión mayores que dos, el espacio ocupado en memoria crece según $O(n^{[d/2]+\delta})$ para una $\delta > 0$, con búsquedas en tiempo *O(logn)*, que nuevamente oculta factores exponenciales en d [14].

Si bien se ha demostrado que es posible eliminar estos factores dependientes de la dimensionalidad, sólo es posible a costa de aumentar el espacio ocupado en memoria. No se conoce método alguno que simultáneamente consiga ocupación de espacio en memoria lineal y tiempo de búsqueda logarítmico [15].

Búsquedas aproximadas Al degradar las búsquedas a una tolerancia ε, es posible conseguir estructuras que ocupen *O(dn)*, generables en un tiempo *O(dnlogn)* y atender búsquedas en *O(logn)*, con factores exponenciales en *d*.

Se puede concluir por tanto que búsquedas aproximadas permiten construcción y ocupación de la estructura de datos lineal con *d*, al contrario que las estructuras exigidas por las búsquedas exactas que crecen exponencialmente con *d*.

Para búsquedas aproximadas, sí es posible determinar que la búsqueda de los k mejores vecinos es alcanzable en un tiempo de *O(kdlogn)*.

En consecuencia

En el presente trabaja se tratará de construir *Kd-trees* cuyo comportamiento sea acorde a los resultados de este estudio. Esto es, se construyan en *O(nlogn)* y permitan realizar búsquedas en *O(logn)* o de este orden, ya que se dará soporte no sólo a búsquedas del mejor vecino, si no a búsquedas de radio fijo y de k vecinos.

3.3. Implementaciones existentes

Se pretende comparar el rendimiento de la implementación desarrollada con respecto a otras implementaciones existentes, estudiando si se mejoran los resultados en algún caso y determinando para qué tipo de problemas cuál es la implementación más adecuada.

Si bien existen numerosas implementaciones de librerías que solucionan el problema de la vecindad de puntos, no todos están orientadas a la resolución de métodos numéricos. Se han escogido aquellas más significativas y relacionadas con este trabajo.

3.3.1. CGAL

CGAL es un proyecto software desarrollado por varios institutos de la Unión Europea e Israel que proporciona un fácil acceso a algoritmos geométricos de forma eficiente y estable en forma de una librería en C++. CGAL es empleada en áreas que requieren geometría computacional, como sistemas de información geográfica, diseño asistido por ordenador, biología molecular, imágenes en medicina, gráficos por ordenador y robótica [16].

3.3.2. ANN

ANN (Approximate Nearest Neighbors) es una librería escrita en C++, que incluye estructuras de datos y algoritmos tanto para búsquedas exactas como aproximadas de vecinos en espacios con un número de dimensiones aribitrario [2]. Es la librería desarrollada por los autores de [10]. Se elige esta librería como referencia respecto a la que comparar debido a su popularidad y número de usuarios.

3.3.3. FLANN

FLANN (Fast Library for Approximate Nearest Neighbors), es una librería escrita en C++, que permite obtener rápidamente búsquedas aproximadas de vecinos en espacios con un número de dimensiones arbitrario. Contiene una colección de algoritmos que son empleados de forma optimizada en función de la distribución de datos con la que se trabaja.

3.3.4. Muesli

MUSLI, Material UnivErSal LIbrary, es una colección de clases y funciones en C++ diseñadas para modelar el comportamiento de un material de forma continua. Desarrollada por el IMDEA, está disponible para las comunidades de las ciencias de materiales y mecánica computacional como una *suite* de modelos estándar y plataformas para el desarrollo de nuevos modelos [17].

3.3.5. Santy

Librería de cabeceras en C++ que implementa el Algoritmo de Vecindad Combinado [1] como se describe en el artículo [18]. La implementación descrita en este trabajo se basa en esta librería, de la cual se usa la idea de dividir en el espacio en celdas mapeadas en una *Hash-Table* como solución al problema de la vecindad de puntos.

4. Diseño de algoritmos

En este capítulo se explicará el funcionamiento del algoritmo empleado para solucionar el problema de la vecindad de puntos, así como las estructuras de datos empleadas y el fundamento en el que se basa la solución.

4.1. Antecedentes

Se pretende mejorar y replantear la implementación desarrollada por S. Tapia en colaboración con A. Beltrán y I. Romero [1]. En esta, se divide el espacio en celdas en función de la codificación en coma flotante (que se detalla en el Apéndice A) de los puntos del conjunto estudiado para posteriormente mapear estas celdas en una *Hash-Table* [18]. Para ello se enmascaran los bits de la codificación y operando con ellos a partir de un parámetro se obtiene la celda a la que corresponden. Posteriormente, se emplea esta celda para generar el *hash* o índice de la tabla.

Las *Hash-Table* son estructuras de datos que se caracterizan por un acceso *O(1)*, o constante, a todos sus elementos. Esto permite que incluso para grandes cantidades de datos el tiempo de acceso permanezca constante, siendo posible escalar la solución para cualquier número de elementos, y en este caso particular, puntos. Sin embargo, si bien este hecho es cierto para el caso promedio y para una *Hash-Table* adecuadamente diseñada, el caso peor presenta un acceso de *O(n)*, debido a colisiones de *hashing* al, dos o más claves distintas, generar un mismo índice en la tabla [9].

El algoritmo de búsqueda empleado en [1] sólo soporta búsquedas exactas de radio fijo en tres dimensiones. Esto es así porque se dimensiona el tamaño de las celdas en función del radio de búsqueda. Este dimensionamiento permite que, dado un *querying point*, sólo sea necesario buscar hasta en 2^3 celdas, como se aprecia en Fig. 4.1, ya que el radio de búsqueda r es justo la mitad del lado de las celdas. Este radio es el que se usará como parámetro a la hora de trabajar con la codificación en coma flotante y clasificar los puntos en las celdas, dividiendo consecuentemente el espacio en celdas del tamaño deseado.

Las celdas por tanto serán cubos que podrán contener más o menos *searching points* en función de la concentración de los mismos en esa zona particular del espacio. La obtención de los vecinos se obtendrá a partir de la aplicación de métodos de fuerza bruta para todos los *searching* pertenecientes a la celda.

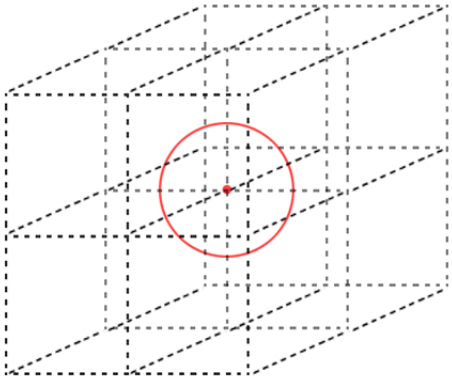


Figura 4.1.: Celdas objetivo para un espacio tridimensional

4.1.1. Ventajas

Las características a destacar de este algoritmo son su simplicidad y elegancia. Se emplean estructuras de datos clásicas como las *Hash-Tables* y se acelera el algoritmo de búsqueda mediante un dimensionamiento apropiado de las celdas.

El principal consumo de recursos (tiempo), se produce en el procesamiento inicial de los datos al clasificarlos dentro de celdas. Si la cantidad de puntos por celda es pequeña, la búsqueda de los mismos es de coste computacional despreciable, y de ahí el buen rendimiento de la solución para las búsquedas, pues estas sólo llegan a consultar hasta 2^3 celdas en el peor de los casos.

Para datos uniformemente distribuidos y radios de búsqueda adecuados, la celdas contendrán pocos puntos, pudiéndose despreciar el coste computacional del método de fuerza bruta dentro de las mismas y haciendo el tiempo de búsqueda constante *O(1)* en la práctica.

4.1.2. Inconvenientes

La implementación actual del algoritmo, si bien es escalable a un mayor número de dimensiones, no parece tener buenas perspectivas al ser el número de celdas a visitar exponencial con el número de dimensiones: 2^d . Siendo este algoritmo un claro ejemplo de víctima de la maldición de la dimensión.

Así mismo, en caso de existir un gran número de puntos en una celda en particular, el algoritmo de fuerza bruta ralentizaría significativamente el proceso pues este es *O(dn) + O(nlogn),* aunque esta n correspondería al número de *searchings* en cada celda, no al total del conjunto estudiado.

4.1.3. Conclusión

Del análisis de antecedentes se pueden extraer una serie de conclusiones:

1. **Falta de generalidad:** La implementación del algoritmo es muy específica y sólo es válida para un problema concreto, búsquedas de radio fijo en tres dimensiones. Posiblemente se pueda generalizar el funcionamiento de este algoritmo para un número de dimensiones pequeño, como puede ser de 1 a 4 dimensiones sin comprometer de forma significativa las propiedades del mismo. Por otro lado, es posible que también se pueda realizar una búsqueda de los k mejores vecinos empleando parte del mismo algoritmo.

2. **Escasa escalabilidad a espacios multidimensionales:** Se ha visto que las propiedades del algoritmo se verían degradadas para un gran número de dimensiones. Sería de interés determinar a partir de qué número de dimensiones y para qué tipo de distribuciones de datos se pierden estas propiedades e implementar un algoritmo alternativo, ya sea el empleo de algún tipo de árbol o directamente un método de fuerza bruta, que mejoren la solución al problema de la vecindad de puntos.

3. **Grandes diferencias entre caso promedio y peor:** Debido a la especialización del algoritmo, si bien este presenta muy buenos resultados para determinados conjuntos de datos, puede presentar resultados muy pobres en aquellas configuraciones a las que es susceptible.

    a) Superpoblación de celdas: Es posible que para determinadas muestras de datos, los puntos no estén uniformemente distribuidos y estos se encuentren muy concentrados en determinadas zonas del espacio. En estos casos, una celda contendrá un gran número de puntos, por lo que el algoritmo de fuerza bruta, que es el algoritmo de búsqueda empleado, afectará al rendimiento total de la aplicación. Se podría mejorar el rendimiento en estos casos preprocesando los datos de aquellas celdas con una cantidad de puntos crítica por medio de una estructura de datos tipo árbol.

b) Degradación con el radio: Al ser el tamaño de las celdas dependiente del radio de búsqueda, este afectará al rendimiento de la búsqueda, haciendo que el preprocesamiento de los datos haya sido en vano. Un radio demasiado pequeño generará demasiadas celdas, lo que equivaldría a recorrer prácticamente la totalidad de los puntos, haciendo cada búsqueda *O(n)* para la búsqueda del mejor vecino. Por el contrario, un radio grande generaría muy pocas celdas, degradando el algoritmo de búsqueda a un método de fuerza bruta nuevamente *O(dn) + O(nlogn)* para la búsqueda del mejor vecino.

4.2. Mejoras propuestas

Partiendo de las conclusiones obtenidas sobre la implementación que pretendemos mejorar, se encuentran varias líneas de trabajo a partir de las cuales definir la nueva solución.

1. Generalización del algoritmo. Posibilitar su aplicación para espacios k dimensionales. Habilitar nuevos tipos de búsqueda como la de mejor vecino y búsqueda de los k mejores vecinos además de la ya existente de radio fijo.
2. Estudiar el comportamiento del algoritmo para espacios con un gran número de dimensiones. Particularizar el algoritmo para estos espacios, empleando métodos alternativos que presenten rendimientos aceptables.
3. Implementar árboles de búsqueda binarios para aquellas celdas en las que haya suficientes puntos como para que se pueda justificar su construcción.
4. Determinar experimentalmente en qué tipos de distribuciones de datos las propiedades del algoritmo se degradan y elaborar una solución que gestione de forma dinámica el tamaño de las celdas, así como la construcción de árboles binarios de búsqueda para aquellas en las que sea necesario.
5. Determinar si la solución tras los cambios sigue siendo paralelizable, y en caso afirmativo, implementar la solución para que pueda resolverse en varios núcleos de forma paralela.

Se plantea por tanto un algoritmo que funcione en dos niveles. Un nivel alto en el que se divide el espacio en celdas, y un nivel bajo en el que se realicen las búsquedas por medio de algoritmos que se adapten dinámicamente a las características de cada celda.

- En primer lugar se preprocesa el conjunto de datos, distribuyendo los *searchings* en celdas según su distribución espacial. En una primera versión del algoritmo, el tamaño de las celdas podrá ser determinado por el usuario, aunque se pretende determinar los valores óptimos de forma experimental.

- A continuación, se construirán árboles de búsqueda binarios en aquellas celdas con suficientes puntos como para que la construcción de los mismos se vea amortizada para el número de búsquedas a realizar.
- Finalmente, se gestionará la búsqueda en celdas de forma paralela y se agruparán los resultados obtenidos, haciendo el proceso limpio a ojos del usuario final.

4.3. Recursos y alcance

4.3.1. Disponibilidad de recursos

El **tiempo** del que se dispone es el correspondiente a un Trabajo de Fin de Grado. Este equivale a 12 ECTS [1], esto es, 300~360 horas de trabajo. Se determinará el alcance de las tareas a realizar en función de este marco temporal, y se pretende que el número de horas dedicadas tras la finalización del trabajo coincida en un margen razonable con las horas estimadas.

El **presupuesto** del que se dispone es inexistente, por lo que en la medida de lo posible se empleará software de uso gratuito, y se trabajará empleando recursos privados (ordenadores) o aquellos proporcionados por la Universidad Politécnica de Madrid (UPM) (redes, licencias de software, espacio de trabajo).

4.3.2. Alcance

Se expone a continuación un análisis de cada una de las mejoras propuestas y se aceptan o descartan las mismas para ser acometidas o no en la realización

1. Generalización del algoritmo:
   a) Aplicación en espacios k dimensionales: Si bien la efectividad del algoritmo se degrade con toda seguridad a partir de espacios de 5 dimensiones, implementar el algoritmo para que trabaje en k dimensiones, independientemente de los rendimientos ofrecidos, será necesario para que pueda ser aplicado en espacios de 1 o 2 dimensiones, siendo implícita su extrapolación a espacios de dimensiones superiores. **Aceptada.**
   b) Nuevos tipos de búsqueda: Habilitar búsquedas de mejor vecino y de k mejores vecinos, entra en los propósitos generales de el problema de la vecindad de puntos y aporta una mayor flexibilidad de búsqueda de cara a un usuario final. Debido a que la búsqueda del mejor vecino es una particularización con $k = 1$ de la búsqueda de los mejores vecinos, se implementará esta última. **Aceptada.**

[1] *European Credit Transfer and Accumulation System*, estándar europeo para cuantificar el trabajo relativo al estudiante de educación superior.

c) Permitir la definición de la función distancia por el usuario: Si realmente se pretendiese realizar una implementación completamente generalista, sería necesario permitir al usuario definir su propio método para medir la distancia. Sin embargo, se ha considerado que para un implementación inicial de la solución, bastará con emplear la distancia euclídea como forma de medir la distancia entre dos puntos. **Rechazada.**

2. Tratamiento de la maldición de la dimensión:

   a) Estudio del comportamiento del algoritmo para un gran número de dimensiones: Aunque existe un gran interés por lo que puede suponer la comparación de algoritmos de fuerza bruta con respecto a métodos más refinados en espacios de un gran número de dimensiones, la realización de comparativas entre los mismos para determinar a partir de qué situaciones unos presentan mejores rendimientos que otros, se dejará para futuras investigaciones. **Rechazada.**

   b) Particularización del algoritmo para espacios de un gran número de dimensiones: Ya que el enfoque de la solución es su aplicación a métodos numéricos en tres dimensiones, la inversión de recursos en este área no está justificada. La solución a la maldición de la dimensión se encuentra fuera del propósito de la implementación. **Rechazada.**

3. Solución al *Hash - Collision* [2]: Empleando tablas diseñadas apropiadamente, la probabilidad de este suceso se minimiza como se explica en sec. 5.4. Además, en caso de ser ignorada una celda, sólo surgirán errores en la búsqueda si los puntos más cercanos al *querying* se encuentran en la misma, por lo que debido a la singularidad de este error se admite que no vale la pena su tratamiento. De todas formas, la implementación estándar de C++ para las *Hash-tables*, `std::unordered_map` [19] se encarga de realizar un *rehash* en caso de colisiones, gestionando este problema por el usuario. **Rechazada.**

4. Implementación de árboles de búsqueda binarios para aquellas celdas en las que haya suficientes puntos como para que se pueda justificar su construcción: Ya que la mayoría de implementaciones existentes se basan en el uso de árboles, la exploración de la implementación de los mismos y su particularización al problema, no es sólo de gran interés para mejorar la solución, si no también es útil para entender como han sido realizadas otras implementaciones existentes. **Aceptada.** Se implementarán dos versiones de árboles cuyas propiedades se detallan en sec. 4.5 y sec. 5.5.

[2] Todos aquellos números en la misma zona del espacio generan un mismo número a partir de su exponente, mantisa y parámetro de dimensionamiento empleado, con el que se clasifican en su celda correspondiente. Esta celda posteriormente se emplea para generar el *hash* o índice en la tabla. El conflicto se genera cuando varias celdas, correspondientes a distintas zonas del espacio generan un índice de la tabla igual. Esta situación podría provocar que se arrojasen resultados erróneos al estarse perdiendo la información de una celda .

   a) Árboles equilibrados [3]: Estos árboles requieren de un tiempo de construcción de *O(nlogn)* y garantizan búsquedas de un único vecino en *O(logn)*.

   b) Árboles no equilibrados: Estos árboles requieren de un tiempo de construcción de *O(nlogn)*, fuertemente dependiente de las características de los datos a tratar, y tiempos de búsqueda promedios en *O(logn)*.

5. Determinación de las propiedades del algoritmo:

   a) Influencia de tipos de distribuciones: Se determinará si las características de los datos a estudiar influyen en el rendimiento del algoritmo y de qué manera lo hacen. **Aceptada.**

   b) Gestión dinámica el tamaño de las celdas: Se determinará experimentalmente cómo maximizar el rendimiento del algoritmo en función del tamaño de celdas elegido. **Aceptada.** Sin embargo, se dejará para futuras implementaciones un algoritmo que gestione el tamaño de las mismas, limitándose este trabajo a determinar su influencia.

   c) Gestión dinámica de la construcción de árboles: Se determinará experimentalmente cómo maximizar el rendimiento del algoritmo en función de en qué casos construir árboles. **Aceptada.** Sin embargo, se dejará para futuras implementaciones un algoritmo que gestione la construcción de los mismos, limitándose este trabajo a determinar a partir de qué punto su construcción supone una mejora sobre el rendimiento.

6. Paralelización: Se estudiará la influencia en el rendimiento al paralelizar el algoritmo en varios núcleos procesadores. La implementación de la paralelización no se desarrollará por el alumno. **Aceptada.**

4.4. Alto nivel

4.4.1. Fundamento

El objetivo es agrupar los puntos pertenecientes al conjunto de *searchings* de forma que al realizar búsquedas no sea necesario analizar el total de puntos, sino en su lugar, sólo aquellos puntos que sean potenciales vecinos del *querying*.

Para ello se agrupan los puntos en celdas del espacio que tendrán forma de hipercubos. Para determinar a qué celda corresponde cada punto se trabajará con su codificación en coma flotante. De esta forma se evitará trabajar con operaciones como multiplicaciones o divisiones (computacionalmente más costosas), ya que se trabajará a nivel binario directamente (suma lógica, producto lógico, desplazamiento de bits), acelerando los cálculos. A cada punto se le asignará una celda, pudiendo

[3] Se dice que un árbol está equilibrado, si para cada nodo, las alturas de sus dos subárboles difieren como mucho en uno [20].

tener varios puntos en una misma celda. Todas aquellas celdas con al menos un punto estarán incluidas como una entrada dentro de una *Hash-Table*.

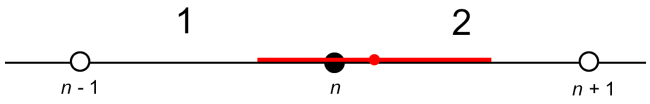
Las celdas se caracterizan en función de un punto en el espacio, en el caso de 2 dimensiones (un cuadrado) sería su esquina inferior izquierda. Es decir, el punto característico de las celdas, y a partir del cual se genera el hipercubo es aquel punto del mismo cuyas coordenadas estén más cercanas a $-\infty$. Es posible definir unívocamente el hipercubo a partir de un único punto ya que la longitud del lado del mismo es en función del radio de búsqueda. Este punto por el cual es posible determinar la celda, será el mismo que se obtiene al tratar la codificación de un punto. De esta forma, todos aquellos puntos que pertenezcan al hipercubo, al ser tratados, devolverán el punto que caracteriza a la celda (hipercubo) a la que pertenecen. La implementación de este tratamiento se explica detalladamente en sec. 5.3.

Se empleará este punto característico de la celda para obtener la clave de la *Hash-Table*. De esta forma, la codificación tratada de todos los puntos que podrían pertenecer a una celda, generarán la misma clave con la que entrar en la *Hash-Table*. Cabe destacar que el dominio de cada celda es inclusivo para sus frontera cercana a $-\infty$, y exclusivo para su frontera cercana a $+\infty$, de forma que aquellos puntos que se encuentren justo en la frontera entre dos celdas, no están compartidos por ambas, si no que pertenecen a la más cercana a $+\infty$.

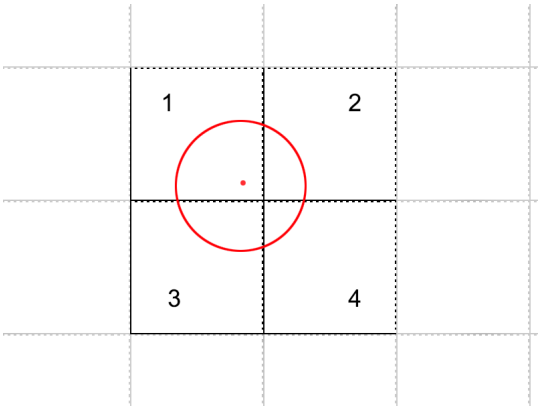
Una vez distribuidos todos los puntos en sus celdas correspondientes, cada vez que se realice una búsqueda, se aplicará a la codificación del punto *querying* el mismo tratamiento realizado a los *searching*, de esta forma se obtendrá la celda correspondiente al *querying*. Se comprobará entonces si existe esa celda o sus adyacentes dentro de la *Hash-Table*, y en caso afirmativo se procederá a la búsqueda de puntos mediante algoritmos de bajo de nivel.

Internamente, los puntos *searching* se almacenan en la celda de dos formas. Ya que un punto en el espacio puede tener una serie de propiedades escalares o vectoriales[4], será necesario separar esta información de las coordenadas espaciales del punto, ya que las primeras no tienen importancia alguna para solucionar el problema de la vecindad de puntos. Por ello, en las celdas se almacenará un vector de la librería estándar de C++ `std::vector` [21], con los puntos en forma de coordenadas espaciales, y en otro vector del mismo tamaño, las referencias a estos puntos. Mediante este método sólo será necesario duplicar la información correspondiente a las coordenadas, empleando *O(dn)* espacio en memoria y las referencias a los mismos *O(n)*, no siendo necesario el tener que almacenar datos adicionales que no van a ser tratados. En todas las operaciones en las que se intercambien de posición los elementos de un vector, será necesario intercambiar de la misma forma la posición de los elementos del vector de referencias. De esta forma se consigue que tanto las coordenadas de un punto, como la referencia al mismo, tengan el mismo índice en cada vector, pudiendo trabajar con estos últimos dentro de los métodos implementados en la celda.

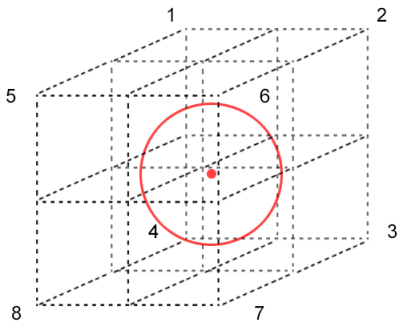
[4] Una partícula en una habitación puede tener una temperatura (escalar) y una velocidad (vectorial)



(a) 1 dimensión



(b) 2 dimensiones



(c) 3 dimensiones

Figura 4.2.: Celdas a visitar en función del número de dimensiones

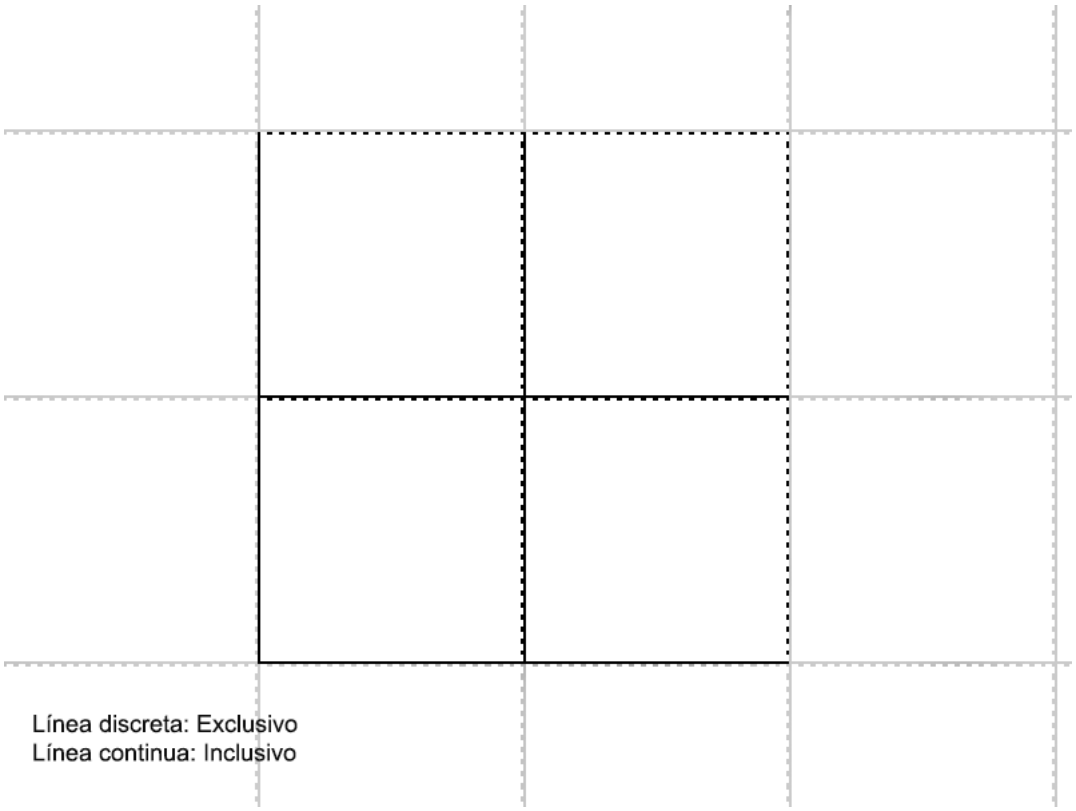


Figura 4.3.: Dominios de las celdas

4.4.2. Diagrama de funcionamiento

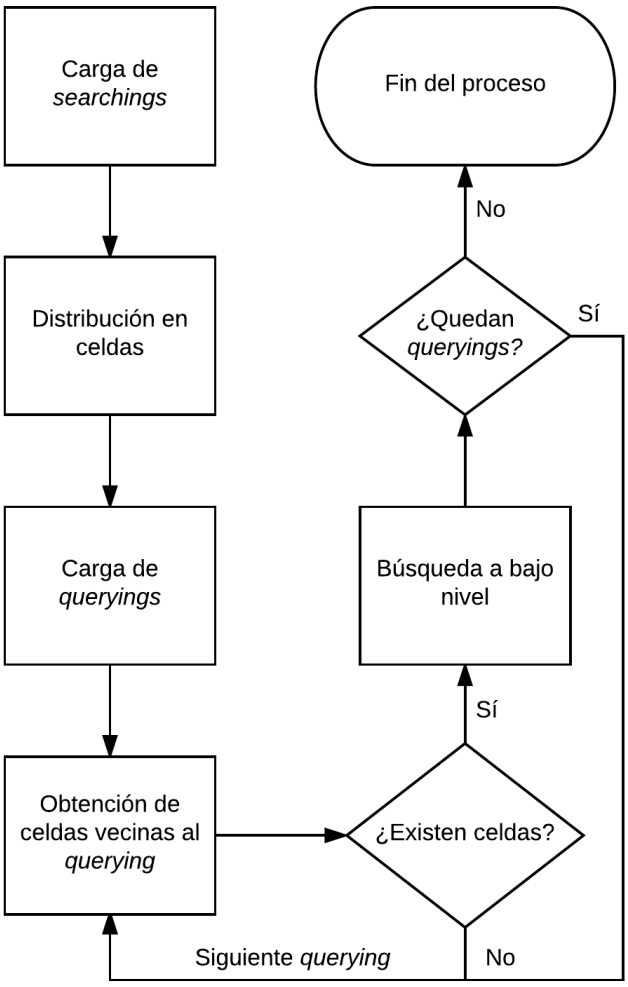


Figura 4.4.: Diagrama. Funcionamiento del algoritmo de alto nivel

4.4.3. Fuente

La idea de dividir el espacio en celdas agrupadas en una *Hash-Table*, ha sido heredada de la librería [1] sobre la que se basa este trabajo. De esta librería, se han adaptado las funciones para la carga de puntos, los algoritmos que gestionan la inserción de puntos en celdas, así como los algoritmos de búsqueda de celdas adyacentes para tres dimensiones. La generalización del algoritmo de búsqueda en celdas a k dimensiones ha sido elaborada en este trabajo. La implementación es recursiva ya que para visitar las celdas adyacentes según el patrón mostrado en Fig. 4.2, es necesario anidar un número de bucles igual a la dimensión de trabajo. Se presenta a continuación la implementación del algortimo de búsqueda de radio fijo para cualquier número de dimensiones.

Ejemplo 4.1: Búsqueda en varias celdas

```
1  void tessel_query(const std::array<double, DIM>& coord, const double
       sq_radius, const tessel<DIM>& c, tessel<DIM>& aux,
2                        std::vector<uintptr_t>& result, long x[DIM],
                             unsigned i, const long r ) const
3  {
4      if( i < DIM )
5      {
6          for ( x[i]= -r; x[i] <= r; ++x[i]) {
7              tessel_query( coord, sq_radius, c, aux, result, x, i + 1,
                   r );
8          }
9      }
10     else
11     {
12         unsigned j;
13         for( j = 0; j < DIM; ++j){
14             aux.coor[j] = c.coor[j] + x[j];
15         }
16
17         typename HashMap::const_iterator found = domain.find(aux);
18         if ( found != domain.end() )
19         {
20             query_result q_result = found->second->query(coord,
                   sq_radius);
21             result.reserve(result.size()+q_result.size());
22             result.insert(result.end(), q_result.begin(), q_result.end
                   ());
23         }
24     }
25 }
```

Las celdas a visitar son todas aquellas dentro de un hipercubo cuya celda central `c` es la celda a la que pertenece el punto de búsqueda *querying* de coordenadadas `coord`. El hipercubo tendrá $2*\texttt{r}+1$ celdas de lado, siendo `r` una constante obtenida a partir del radio de búsqueda [5].

[5]Se visitan por tanto más celdas de las necesarias. En el momento de redacción de esta memoria

4.5. Bajo nivel

4.5.1. Fundamento

El objetivo es acelerar las búsquedas dentro de aquellas celdas que tengan una cantidad suficiente de puntos *searching* como para que su preprocesamiento redunde en un mejor rendimiento. Este preprocesamiento se traduce en la construcción de celdas que internamente sean árboles de búsqueda binarios, *kd-Trees* (cuyo fundamento se explica en el Apéndice B), o celdas que se aprovechen de opciones de vectorización en el compilador.

Se tendrán por tanto, tantos subtipos de celdas como tratamientos a bajo nivel se den para los puntos *searching*. Estas celdas heredan de una clase *cell,* que será virtual y sus métodos públicos servirán de interfaz para los algoritmos de alto nivel. Distinguimos las siguientes celdas:

1. Métodos de fuerza bruta, recibirán el nombre de *brute_ cell.*
2. Métodos de fuerza bruta con vectorización, recibirán el nombre de *par_cell.*
3. Árboles de búsqueda equilibrados, recibirán el nombre de *utree_cell.*
4. Árboles de búsqueda no equilibrados o simples, recibirán el nombre *stree_cell.*

En las últimas, se construirá un árbol a partir de los puntos pertenecientes a la celda. Una vez construido, las características del mismo permiten búsquedas en *O(logn)* del punto más cercano. Las búsquedas de radio fijo y de los k mejores vecinos no cumplen esta complejidad pero se comprueba experimentalmente que siguen siendo significativamente más rápidas que empleando métodos de fuerza bruta como se verá en Capítulo 7. A partir de los resultados obtenidos experimentalmente se determinará la cota inferior de puntos *searching* en una celda a partir de la cual se construirán los árboles. La hipótesis sobre la que se basa la construcción de estos árboles es que se trabajará con puntos no repetidos. Es decir, no se insertará en el árbol dos o más veces el mismo punto exactamente igual coordenada a coordenada. También, se asume que la construcción de los árboles se produce una vez conocidos todos los puntos a construir, no admitiéndose su inserción o extracción posterior.

Para la construcción de los dos primeros árboles se ha empleado código inspirado en [22], debidamente adaptado y corregido como se explica en sec. 5.5, para que cubra todo tipo de distribuciones de puntos. El tercer tipo de árbol es de implementación completamente propia.

A continuación se detallarán las características de los subtipos de celdas implementadas.

se está trabajando en el desarrollo de un algoritmo que filtre las celdas a visitar dentro de este hipercubo, mejorando la eficacia del algoritmo.

4.5.2. Fuerza bruta. *Brute_cell*

Fundamento

Por fuerza bruta se entiende que se soluciona el problema de la vecindad de puntos calculando la distancia de todos los *searching* al *querying* y una vez obtenidas estas distancias se devuelve:

- El *searching* correspondiente a la menor si se realiza la búsqueda del mejor vecino
- Los *searching* cuyas distancias sean más pequeñas que r si se realiza la búsqueda de radio fijo.
- Los *searching* cuyas distancias sean las k más pequeñas si se realiza la búsqueda de los k vecinos.

Es importante destacar, que cuando se habla de distancias, se trabajará en todo momento con distancias al cuadrado, pues el cálculo de la raíz cuadrada del cuadrado de las componentes del vector distancia no aporta información adicional, siendo su cálculo innecesario y pudiéndo ser eliminado.

Es necesario prestar atención a la implementación de este método a pesar de su trivialidad, pues será a partir del cual se realizarán todas las verificaciones del correcto funcionamiento del resto de métodos. Como ya se explicó, sólo se implementará la búsquedas de radio fijo y la de los k vecinos, al ser la búsqueda del mejor vecino una particularización para k=1 de esta última.

Búsqueda de radio fijo

En la implementación de esta búsqueda, dado un *querying*, se calculan todas las distancias de los *searching* a este punto por medio un bucle simple, y añadiendo el *searching* a los resultados de la búsqueda si la distancia es menor que r.

Ejemplo 4.2: Búsqueda de radio fijo

```
1  query_result& query(const std::array<double, DIM>& querying, double
       sq_radius) override
2  {
3      // Clear previous results
4      this->result.clear();
5      size_t i = 0;
6      for ( const auto& p : this->point_vector )
7      {
8          if ( p.norm( querying ) < sq_radius )
9          {
10             this->result.push_back( this->ref_vector[i] );
11         }
12         ++i;
13     }
```

```
14      return this->result;
15  }
```

Búsqueda de los k vecinos

En la implementación de esta búsqueda, dado un *querying*, se calculan todas las distancias de los *searching* a este punto en una primera instancia. Una vez obtenidas todas las distancias, se emplea la implementación de *quick sort* de la librería estándar (`std::sort` [23]) para ordenar todas las distancias de menor a mayor. A continuación se añaden a los resultados de la búsqueda a aquellos *searching* cuyas distancia sean las k primeras. Si hay varios puntos *searching* a una misma distancia, y estos puntos ocupan posiciones k o mayores, la totalidad de estos puntos se incluirá en los resultados.

Ejemplo 4.3: Búsqueda de k vecinos

```
1  kquery_result& kquery(const std::array<double, DIM>& querying, size_t
       K, double target_sq_radius = -1.0 ) override
2  {
3      // Clear previous results
4      this->norm_and_refs.clear();
5      size_t i = 0;
6      for ( const auto& p : this->point_vector )
7      {
8          distance_and_ref dr;
9          dr.first = p.norm( querying );
10         dr.second = this->ref_vector[i];
11         if ( target_sq_radius < 0 || dr.first < target_sq_radius )
12         {
13             this->norm_and_refs.push_back( dr );
14         }
15         ++i;
16     }
17
18     std::sort( this->norm_and_refs.begin(), this->norm_and_refs.end()
           );
19
20     size_t _size = this->norm_and_refs.size();
21     size_t number_of = K <=_size ? K : _size; // number_of = min(K,
           _size)
22
23     // Add more points if they have the same distance:
24     while ( number_of < _size && this->norm_and_refs[number_of].first
           == this->norm_and_refs[number_of-1].first )
25     {
26         ++number_of;
27     }
28
29     this->norm_and_refs.resize(number_of);
30
31     return this->norm_and_refs;
```

```
32 }
```

4.5.3. Fuerza bruta por vectorización. *Par_cell*

4.5.3.1. Fundamento

La vectorización es un proceso mediante el cual, el compilador detecta qué instrucciones independientes pueden ser ejecutadas como una única instrucción SIMD [6]. Esto permite mejorar el rendimiento cuando se realizan tareas repetitivas bajo una serie de condiciones. Estas son:

- Los datos con los que se va a trabajar deben ocupar un espacio en memoria contiguo.
- En un bucle, las instrucciones ejecutadas en cada iteración no dependen de las instrucciones ejecutadas anteriormente.

En un procesador Intel ®, las instrucciones SIMD hacen uso de registros de 128 bits diseñados para operaciones en coma flotante. Cuando el compilador no realiza ninguna optimización, si se trabaja con números en coma flotante de 64 bits, sólo se aprovechará la parte baja de los registros, desperdiciándose la parte alta. Sin embargo, cuando el compilador optimiza el código se empleará el total de la memoria del registro para realizar dos operaciones de forma simultánea [24].

4.5.3.2. Implementación

La implementación del código adaptado para ser susceptible a ser vectorizado ha sido desarrollado por S. Tapia.

Ejemplo 4.4: Implementación de Vectorización

```
1  query_result& query(const std::array<double, DIM>& querying, double
       sq_radius) override
2  {
3      // Clear previous results
4      this->result.clear();
5
6      // Trying vectorization:
7      size_t i, _size = point_vector.size(), size_v = _size / DIM;
8      typedef double (*pointer2vect)[DIM];
9      pointer2vect auxv = (pointer2vect) point_aux_vector.data();
10
11     for ( i = 0; i < size_v; ++i ) // Preparing the vector
12     {
13         memcpy(auxv + i, querying.data(), DIM*sizeof(double));
```

[6] SIMD (del inglés, *Single instruction, multiple data*) es como indica su nombre una única instrucción en código máquina que realiza la misma operaciones en múltiples datos de forma simultánea.

```
14      }
15
16      double *p   = point_vector.data();
17      double *aux = point_aux_vector.data();
18      for ( i = 0; i < _size; ++i ) // Distance
19      {
20          aux[i] -= p[i];
21      }
22
23      for ( i = 0; i < _size; ++i ) // Quadratic distance
24      {
25          aux[i] *= aux[i];
26      }
27
28      for ( i = 0; i < _size / DIM; ++i )
29      {
30          double sum = 0; size_t j;
31          for (  j = 0; j < DIM; ++j ) { // Norm
32              sum += (*aux);
33              ++aux;
34          }
35          if ( sum < sq_radius ) {
36              this->result.push_back( this->ref_vector[i] );
37          }
38      }
39
40      return this->result;
41  }
```

Se puede apreciar cómo se duplica la inserción de puntos, de esta forma, se tienen dos vectores con la misma información. Así, se puede ir almacenando en el segundo el producto de cada uno de sus valores al cuadrado, aprovechando los registros de 128 bits, pudiendo realizarse tantos productos al mismo tiempo como número de registros se dispongan, acelerando consecuentemente el proceso.

Al final, sólo habría que recorrer el vector sumando sus *DIM* (dimensión del espacio) componentes para obtener la distancia al *querying* de cada punto *searching*. Sólo se ha implementado la búsqueda de radio fijo debido que las características de la búsqueda de los k vecinos la hacían poco susceptible a ser vectorizada.

4.5.4. Árbol equilibrado. *Utree_cell*

4.5.4.1. Construcción

Para que un árbol esté equilibrado, se debe, en cada nivel, hallar el punto situado en la mediana, dividiendo recursivamente la mitad menor o igual a la izquierda y la mitad mayor a la derecha. Se emplea el criterio de menor o igual a la izquierda, y mayor a la derecha ya que es el más ampliamente utilizado en las implementaciones

existentes, siendo su elección completamente arbitraria[7].

Ejemplo 4.5: Construcción de Utree

```
1  int build_recursive(int start, int len, unsigned axis) override
2  {
3      if( len <= 0 ) return -1;
4      int median;
5      int end = start + len - 1;
6      /* Find the median in order to build balanced trees*/
7      median = select_median(start, end, axis);
8      median = unbalance_median(start, end, axis, median);
9      if ( ++axis >= DIM ) axis = 0;
10     this->point_vector[median].left = build_recursive(start, median -
           start, axis);
11     this->point_vector[median].right = build_recursive(median + 1,
           start + len - (median + 1), axis);
12     return median;
13 }
```

En cada nivel de recursividad se trabaja en un único eje o dimensión, denotado por *axis*. Para un espacio de tres dimensiones, si nos encontramos en el primer nivel, empezaríamos trabajando en el eje X. En el segundo nivel en el Y, el tercero en el Z, y en el cuarto nivel volveríamos a trabajar con el eje X.

Como se aprecia en el código, en cada nivel de recursividad se haya en primer lugar la mediana de las coordenadas `axis` de todos los puntos de trabajo mediante la función `select_median`. A continuación se garantiza con la función `unbalance_median` que todos las coordenadas `axis` de los puntos a la izquierda sean menores o iguales al valor mediana, y mayores a la derecha. Teniendo entonces una mitad cuya coordenada `axis` es menor que la mediana y otra mitad que será mayor. Es fácil entender que el punto cuya coordenada es la mediana en ese nivel se verá desplazado de la posición central en caso de existir puntos con un valor de su coordenada `axis` igual al de la mediana, no teniendo entonces mitades exactas.

Finalmente, se subdividen los puntos a izquierda y derecha, teniendo cada nivel la mitad (o pseudo-mitad en caso de valores repetidos) de puntos que el inmediatamente anterior con los que trabajar, repitiéndose recursivamente el algoritmo, finalizando la construcción cuando un nivel sólo tiene un único punto.

Estos árboles sólo cumplirán la definición de equilibrados cuando se traten datos cuyas coordenadas no se repiten, caso poco probable en muestras reales, siendo por tanto árboles pseudo-equilibrados.

Es fácilmente demostrable que la construcción de estos árboles cuando no hay valores repetidos, se realiza en *O(nlogn)*. Las funciones de `select_median` y `unbalance_median`, se describen en sec. 5.5, y son ambas de *O(n)* para su caso promedio.

[7] Cabe decir que términos como izquierda o derecha carecen de sentido real, siendo simplemente una ayuda para poder visualizar la abstracción que supone el construir un árbol virtual que va dividiéndose en dos de forma recursiva.

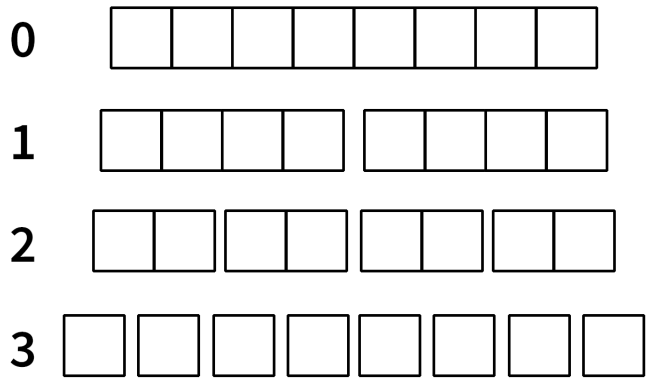
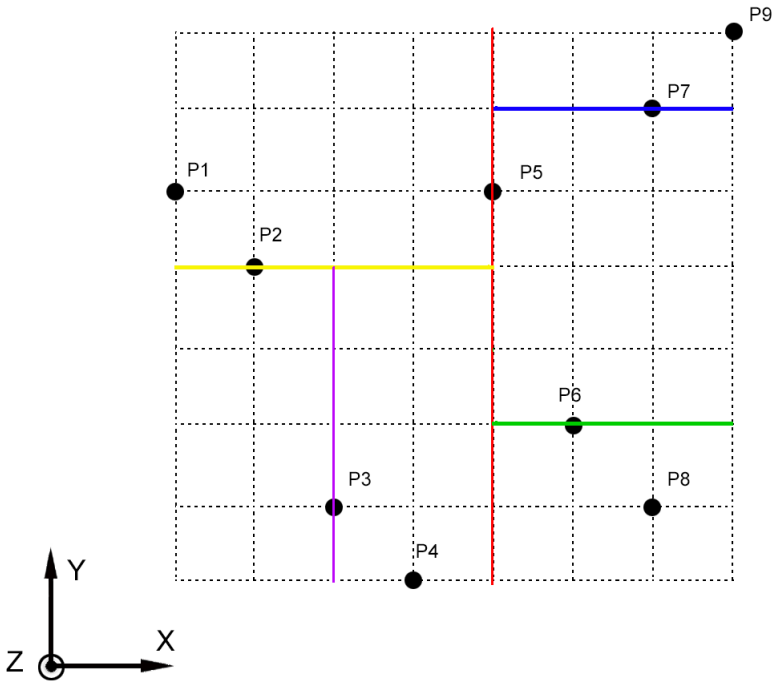


Figura 4.5.: Esquema de recursión al trabajar con un vector

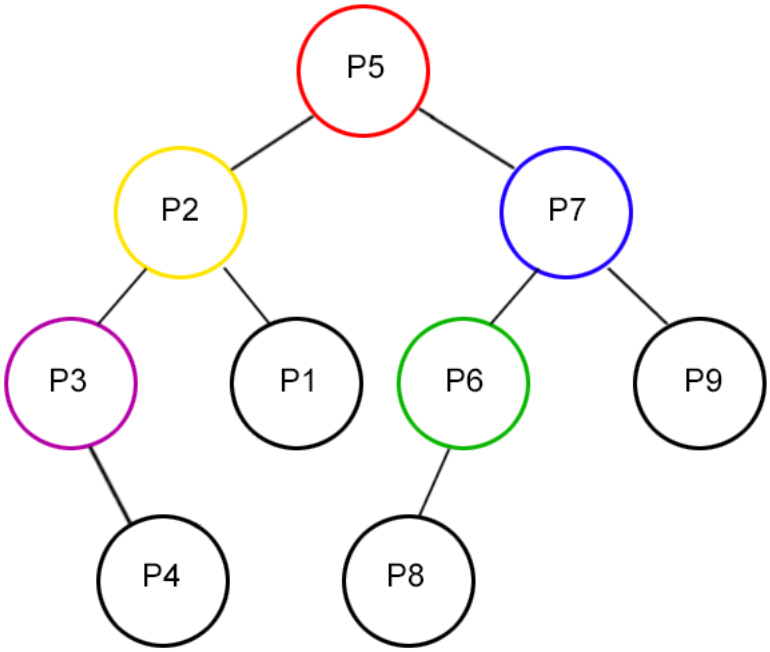
En el primer nivel se trabajará con n puntos, en el segundo con aproximadamente $n/2$ por cada rama (para n grande, la pérdida por nivel del punto seleccionado como mediana se puede despreciar), para el tercero se trabaja con $n/4$ por cada rama, etc, hasta llegar a un único punto. Por lo tanto en cada nivel se tratarán:

0	n
1	$n/2 + n/2$
2	$n/4 + n/4 + n/4 + n/4$
...	
$log_2 n$	$1 + 1 + 1 + ... + 1$, n veces

Si en cada nivel se tratan n puntos y hay un total de $log_2 n$ niveles, la construcción de los árboles se realizará en $O(nlogn)$, como se quería demostrar.



(a) Árbol en el espacio



(b) Árbol lógico

Figura 4.6.: Ejemplo de construcción en 2D

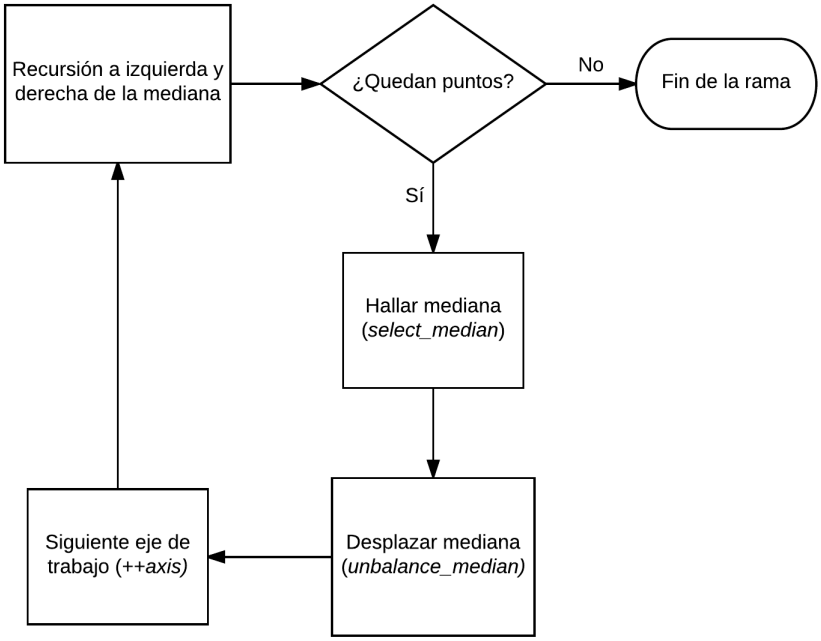


Figura 4.7.: Diagrama. Construcción de Utree_cell

4.5.4.2. Búsqueda de radio fijo[8]

Ejemplo 4.6: Búsqueda de radio fijo

```
1 void query_recursive (unsigned index, const std::array<double, DIM>&
      querying, const double sq_radius, unsigned axis)
2 {
3     if ( index == -1 ) return;
4
5     double d, dx, dx2;
6     d = point_vector[index].norm( querying );
7     dx = point_vector[index][axis] - querying[axis];
8     dx2 = dx*dx;
9
10    if ( d < sq_radius ) {
11    result.push_back(ref_vector[index]);
12    }
13
14    ++axis;
15    if ( axis >= DIM ) axis = 0;
16    query_recursive(dx > 0 ? point_vector[index].left : point_vector[
          index].right , querying, sq_radius, axis);
17    if ( dx2 > sq_radius ) return;
18    query_recursive(dx > 0 ? point_vector[index].right : point_vector[
          index].left, querying, sq_radius, axis);
19 }
```

Nuevamente, se denota como *axis* la coordenada en la que se trabaja. El término `index` se referirá al índice del punto *searching* correspondiente al nodo. El radio al cuadrado se denota como `sq_radius`. La distancia entre el *querying* y el *searching* se denota como `d`, la distancia entre las coordenadas `axis` entre *querying* y *searching* como `dx` y esta última al cuadrado como `dx2`.

Respetando las propiedades del árbol, se irá variando de `axis` en correspondencia con la variación del mismo durante su construcción. Si `d` es menor que el radio de búsqueda, el punto del nodo actual será uno de los vecinos resultado de la búsqueda. Se empleará el signo de `dx` para saber hacia qué rama del árbol se debe seguir buscando. Si `dx`>0, se sabe que el *searching* de ese nivel está a la derecha del *querying*, por lo tanto, si se quieren buscar puntos más cercanos al *querying*, se tendrá que visitar la rama izquierda del *searching*. Análogamente, si `dx`<=0, el *searching* de ese nivel se encontrará a la izquierda del querying, y si se quieren buscar puntos más cercanos al *querying,* habrá que visitar la rama derecha del *searching*.

La condición que permite acelerar las búsquedas es `dx2` > `sq_radius`. Cuando esta condición se cumple, se sabe que la distancia al cuadrado en un eje entre el *searching* y el *querying* supera al cuadrado del radio búsqueda, pudiendo descartar

[8] Se emplearán los términos radio y esfera en lugar de su generalización para k dimensiones, hiperradio e hiperesfera por una mayor comodidad y familiaridad con los primeros términos. La explicación de los métodos empleados es igualmente válida si se sustituyen estos términos por sus términos generales.

inmediatamente esa rama del árbol. La comparación es estrictamente mayor ya que se considera que un punto perteneciente justo en la superficie de la esfera definida por el radio de búsqueda está dentro de la misma.

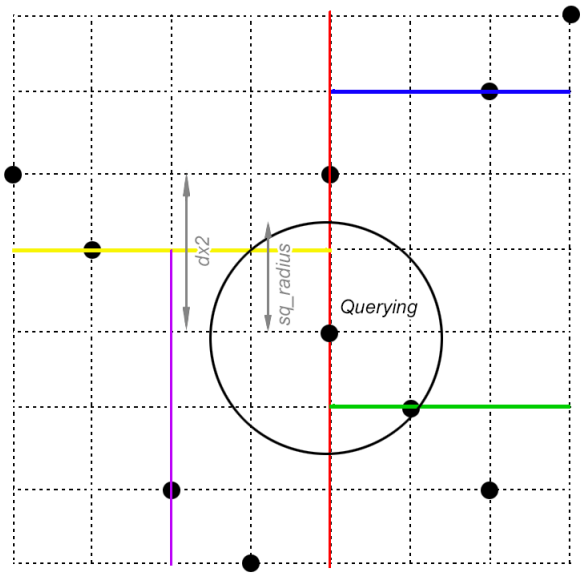


Figura 4.8.: Búsqueda de radio fijo en árboles

Esto es así ya que debido a la ordenación realizada durante la construcción del árbol, si la coordenada `axis` de un punto *searching* es menor o igual que la coordenada `axis` del punto *querying*, es decir, está a la izquierda, las coordenadas `axis` de todos los puntos pertenecientes a esa ramificación también serán menores o iguales. Esto es análogo para un punto a la derecha, cuya coordenada `axis` será mayor que la del *querying*, y también lo serán las coordenadas de todos los puntos pertenecientes a su ramificación.

La búsqueda termina cuando todos los nodos que restan por visitar están descartados por la condición `dx2` > `sq_radius`.

Debido a las características del árbol binario de búsqueda, las búsquedas se realizarán en $O(log_2 n)$, ya que en el caso promedio, se descartan la mitad de los puntos en cada nivel. Para radios pequeños, sólo se visitarán los nodos inferiores de una de las ramas del árbol, teniendo que trabajar en una cantidad reducida de niveles de recursividad, por lo que las búsquedas seguirán siendo razonablemente rápidas y no se degradará $O(log_2 n)$. Sin embargo, a medida que r crezca, la cantidad de nodos a visitar será mayor y las búsquedas empeorarán. Se comprueba este hecho experimentalmente en Capítulo 7.

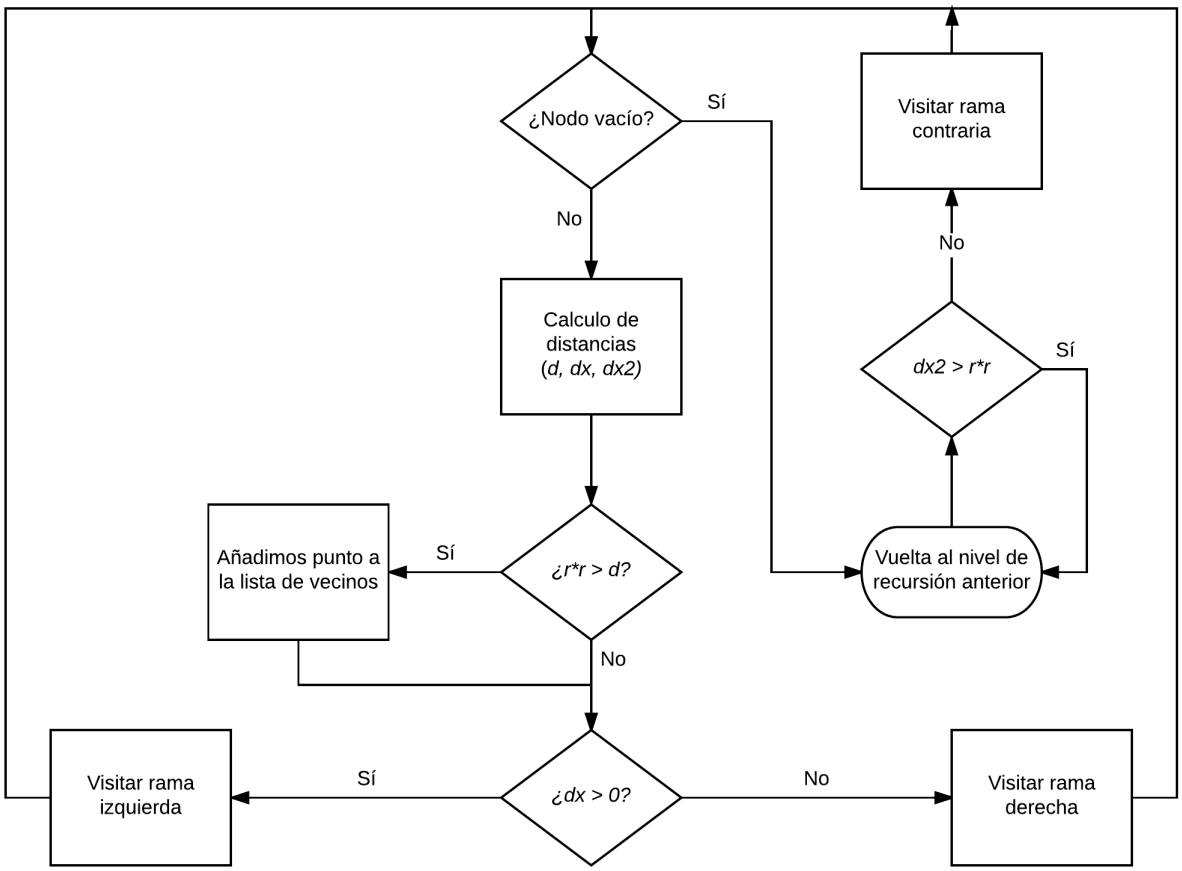


Figura 4.9.: Diagrama de búsqueda de radio fijo

4.5.4.3. Búsqueda de los k vecinos

Ejemplo 4.7: Búsqueda de k vecinos

```
1  void kquery_recursive (int index, const std::array<double, DIM>&
       querying, const int K,
2                         unsigned& max_idx, double& max_dist, double&
                              x_max_dist, unsigned axis)
3  {
4      if ( index == -1 ) return;
5
6      double d, dx, dx2;
7      d = point_vector[index].norm( querying );
8      distance_and_ref curr_point = { d, ref_vector[index] };
9
10     unsigned stored = norm_and_refs.size();
11     if( stored < K )
12     {
13         norm_and_refs.push_back( curr_point );
14         if( d > max_dist ){
15             max_dist = x_max_dist = d;
16             max_idx = stored;
17         }
18         stored++;
19     }
20     else if( d == max_dist )
21     {
22         norm_and_refs.push_back( curr_point );
23         x_max_dist = d;
24     }
25     else if ( d < max_dist )
26     {
27         if( max_dist == x_max_dist ) norm_and_refs.push_back(
               norm_and_refs[max_idx] );
28         norm_and_refs[max_idx] = curr_point;
29         max_dist = 0;
30         int i;
31         for( i = 0; i < K; ++i )
32         {
33             /* Tag the largest distance */
34             if( norm_and_refs[i].first > max_dist){
35                 max_dist = norm_and_refs[i].first;
36                 max_idx = i;
37             }
38         }
39         if( max_dist < x_max_dist )
40         {
41             norm_and_refs.resize ( K );
42             x_max_dist = max_dist;
43         }
44     }
45
46     dx = point_vector[index][axis] - querying[axis];
47     dx2 = dx * dx;
```

```
48      ++axis;
49      if ( axis >= DIM ) axis = 0;
50      kquery_recursive(dx > 0 ? point_vector[index].left : point_vector[
            index].right, querying, K, max_idx, max_dist, x_max_dist, axis
            );
51      if ( dx2 > max_dist && stored == K) return;
52      kquery_recursive(dx > 0 ? point_vector[index].right : point_vector
            [index].left, querying, K, max_idx, max_dist, x_max_dist, axis
            );
53  }
```

Manteniendo la misma nomenclatura, se denota como `axis` la coordenada en la que se trabaja y el término `index` se referirá al índice del punto *searching* correspondiente al nodo. El número de vecinos a encontrar será *K,* la mayor distancia entre los vecinos encontrados hasta el momento será `max_dist`, y el índice correspondiente al punto a esta distancia será `max_idx`. El vector donde se almacenan los puntos *searching* será `distance_and_ref`, donde se almacenarán las distancias y referecias de los puntos vecinos. Nuevamente, la distancia entre el *querying* y el *searching* se denota como `d`, la distancia entre las coordenadas `axis` entre *querying* y *searching* como `dx` y esta última al cuadrado como `dx2`.

Al tener que obtener los K mejores vecinos, será necesario llevar la cuenta de los vecinos almacenados en todo momento y modificar estos durante la búsqueda a medida que se mejore el resultado. Esto se consigue empleando un vector donde se almacenan los puntos *searching.*

Inicialmente, al recorrer el árbol, se rellenará este vector con cualquier punto encontrado hasta haber almacenado una cantidad K de puntos vecinos, almacenando en `max_idx` el índice del punto con la peor distancia, `max_dist`, que será la distancia más alejada del *querying.* Se consigue de esta forma tener marcado el punto más susceptible a ser mejorado.

A continuación, una vez que la cantidad de puntos almacenados haya alcanzado K, se recorrerá el árbol empleando el mismo fundamento que para la búsqueda de radio fijo. Esta vez, se rechazará una rama si la distancia al cuadrado en un eje, `dx2`, es mayor que la peor distancia almacenada, `max_dist`. Siendo entonces la condición: `dx2` $>$ `max_dist`. La comparación es estrictamente mayor ya que en caso de que la distancia sea la misma, podría haber en esa rama un vecino potencial que sería tan buen vecino como el peor vecino almacenado en ese momento.

Para resolver el conflicto que surge al encontrar dos o más vecinos que son peores que todos los demás vecinos almacenados, pero se encuentran estos a la misma distancia del *querying,* se ha decidido devolver la totalidad de estos vecinos, aunque se termine entregando un número de puntos superior a K. De esta forma se pretende dar la opción al usuario final de aceptar la solución completa, o rechazar aquellos puntos adicionales.

La búsqueda termina cuando se sabe por la condición que los nodos restantes a visitar no pueden mejorar el resultado.

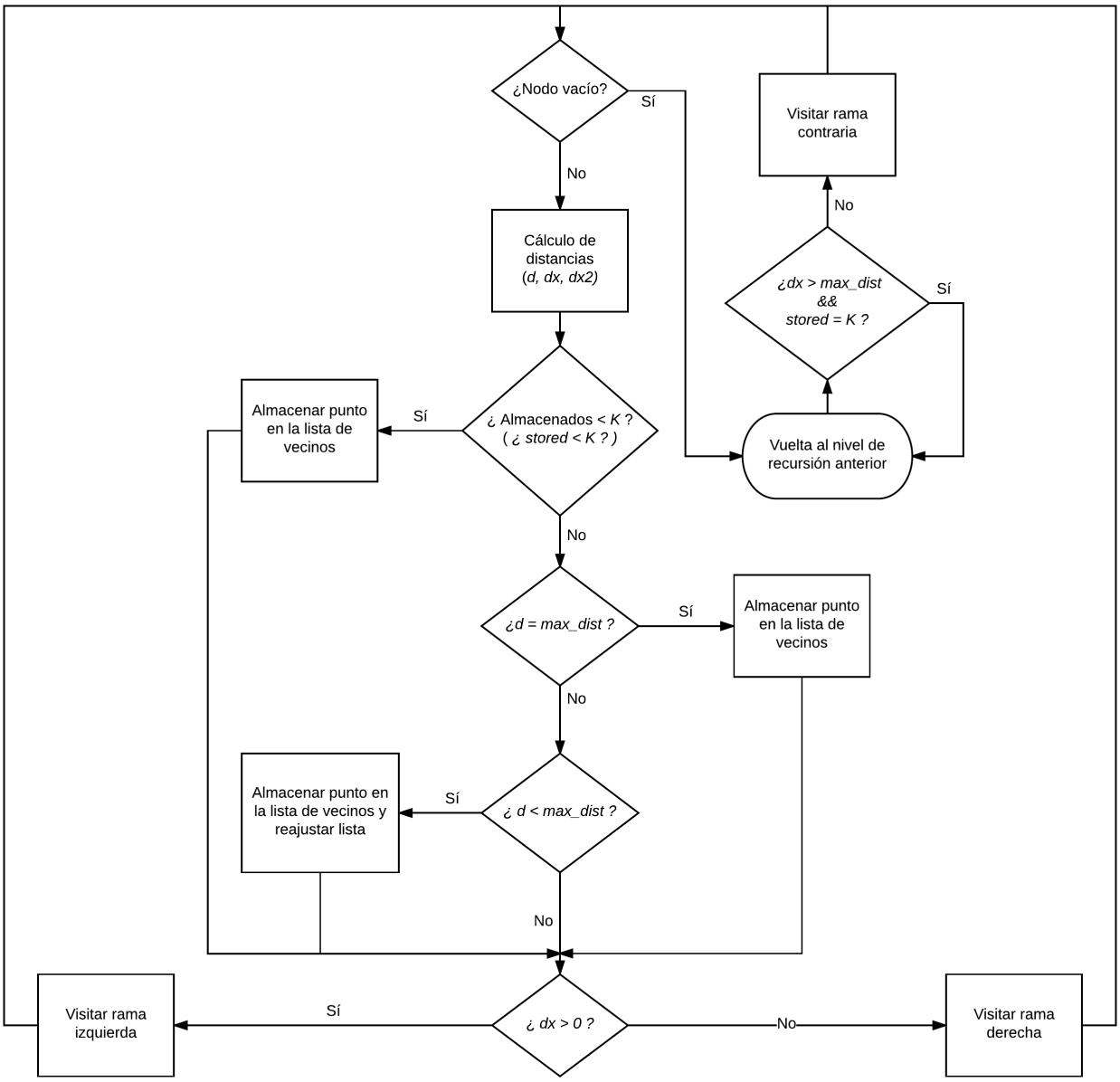


Figura 4.10.: Diagrama de búsqueda de K vecinos

4.5.5. Árbol no equilibrado. *Stree_cell*

4.5.5.1. Construcción

Se emplea nuevamente el criterio de menor o igual a la izquierda, y mayor a la derecha ya que es el más ampliamente utilizado en las implementaciones existentes, siendo su elección completamente arbitraria.

Ejemplo 4.8: Construcción de Stree

```
1  int build_recursive(int start, int len, unsigned axis) override
2  {
3      if( len <= 0 ) return -1;
4
5      int end = start + len - 1;
6      int pivot = start + ( end - start )/2;
7
8      pivot = partition(start, end, axis, this->point_vector[pivot][axis
           ]);
9
10     if ( ++axis >= DIM ) axis = 0;
11     this->point_vector[pivot].left = build_recursive(start, pivot -
           start, axis);
12     this->point_vector[pivot].right = build_recursive(pivot + 1, start
            + len - (pivot + 1), axis);
13     return pivot;
14 }
```

Al igual que en *Utree_cell*, en cada nivel de recursividad se trabaja en un único eje o dimensión, denotado por `axis`. Para un espacio de tres dimensiones, si nos encontramos en el primer nivel, empezaríamos trabajando en el eje X. En el segundo nivel en el Y, el tercero en el Z, y en el cuarto nivel volveríamos a trabajar con el eje X.

A diferencia del caso anterior, no se va a elegir la mediana sino que se va a elegir un punto arbitrario del conjunto de *searchings* de la celda. Con esto se pretende conseguir ahorrar tiempo de construcción del árbol, a costa de sacrificar rendimiento en la búsqueda. Esto es así porque en el peor caso, se escogería en cada nivel puntos en los extremos del conjunto, degenerando el árbol en una lista y teniendo tantas alturas como puntos *searching* en la celda.

Sin embargo, se ha implementado el algoritmo de forma que esto sólo ocurra para distribuciones de datos singulares que no se producen en conjuntos de datos empleados en la práctica. Para ello, en cada nivel de recursividad, se va a elegir como divisor o pivote, al punto que se encuentra justo en la mitad del conjunto. A continuación, se pseudo-ordenan los puntos de forma que para esa coordenada *axis,* todos los puntos menores o iguales se encuentren la izquierda del pivote y todos los puntos mayores a la derecha. Subdividiendo a izquierda y derecha los puntos menores o iguales que el pivote y los mayores respectivamente. Repitiendo recursivamente el algoritmo para

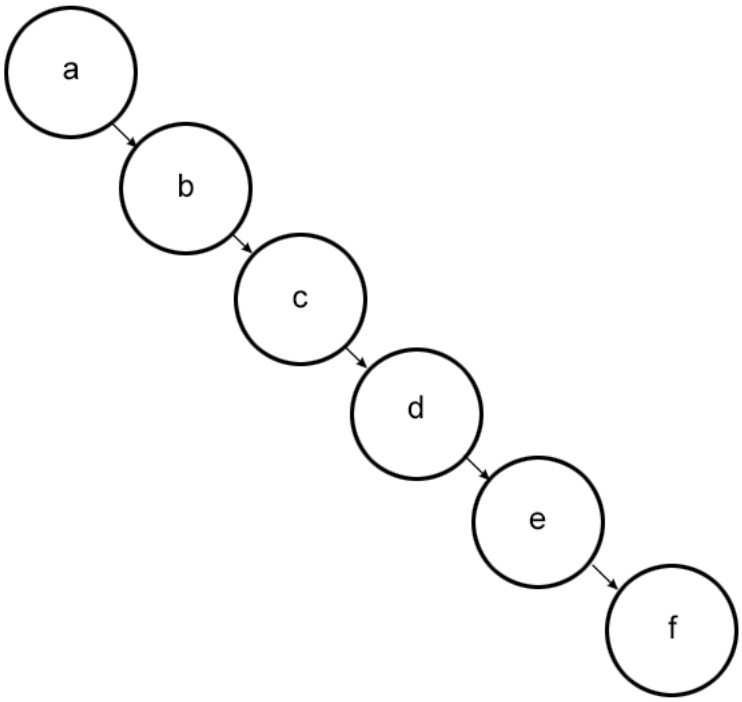


Figura 4.11.: Árbol degenerado en lista

ambas divisiones, finalizando la construcción cuando un nivel sólo tiene un único punto.

Al producirse una pseudo-ordenación en cada nivel, el riesgo de que el árbol degenere en una lista sólo se producirá para la primera vez que se ordenan los puntos para una coordenada, ya que en los siguientes subniveles, al estar la lista pseudo-ordenada, el riesgo de que un valor extremo se encuentre en la mitad de los puntos se minimiza.

El algoritmo empleado, aquí llamado `partition`, y llamado `unbalance_median` en el árbol equilibrado, es de complejidad *O(n)*, como se demuestra en sec. 5.5. En este árbol no equilibrado no es necesario emplear el algoritmo que devuelve la mediana. Por lo tanto, aunque la complejidad en cada nivel siga siendo *O(n),* la construcción del árbol será el doble rápida ya que:

Equilibrado Obtención de la mediana *O(n)* + Pseudo-ordenación *O(n)* = $2 \cdot O(n)$

No-equilibrado Obtención de la mediana = *O(n)*

La notación en *O(n)* esconde el factor de 2. Esta aproximación aparentemente ingenua, se cumple rigurosamente de forma experimental como se muestra en Capítulo 7, dónde el tiempo de construcción del árbol no equilibrado es la mitad que en el árbol equilibrado.

El tiempo de construcción de los árboles es un factor importante a tener en cuenta, ya que se deberán construir árboles en cada iteración de los métodos numéricos, pudiendo representar una parte significativa del tiempo empleado en solucionar el problema. Para que la construcción de un árbol esté justificada, el rendimiento ofrecido por sus búsquedas deberá amortizar el tiempo invertido en su construcción.

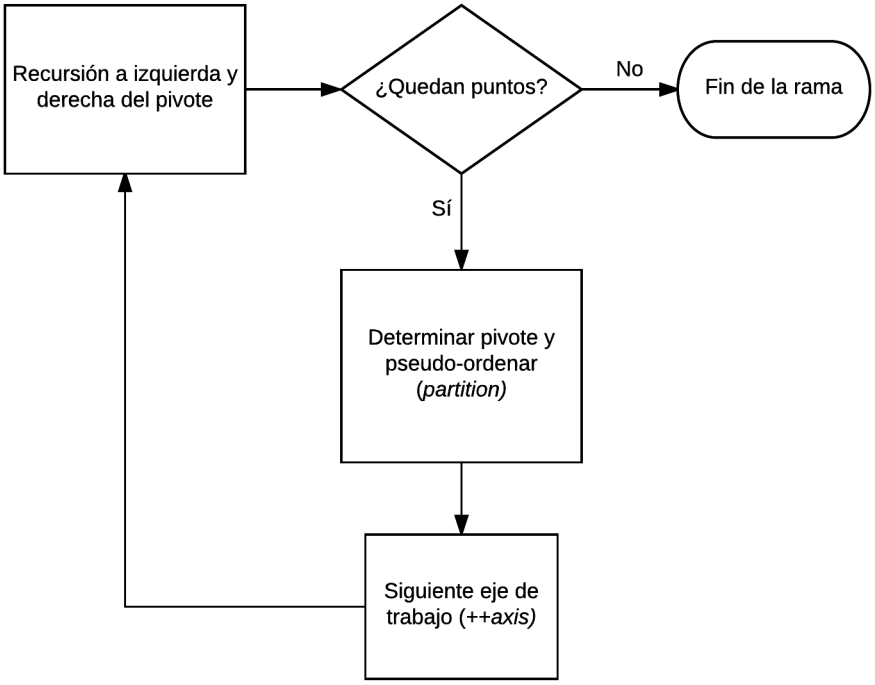


Figura 4.12.: Diagrama de construcción Stree_cell

4.5.5.2. Búsqueda de radio fijo

El código empleado y su fundamento son exactamente iguales que los empleados en *Utree_ cell* sec. 4.5.4.2.

4.5.5.3. Búsqueda de los k vecinos

El código empleado y su fundamento son exactamente iguales que los empleados en *Utree_ cell* sec. 4.5.4.3.

4.6. Paralelización

El objetivo de paralelizar el proceso para que trabaje en varios núcleos simultáneamente es acelerar el proceso del mismo. En una situación ideal, la carga de trabajo se repartiría uniformemente entre los procesadores usados, y el tiempo de ejecución T, se reduciría tantas veces como procesadores p usados. Siendo el nuevo tiempo de ejecución resultante: $T' \approx T/p$.

En la práctica la reducción del tiempo de ejecución no es tal, si no que generalmente uno de los hilos o *threads* de trabajo se convierte en un cuello de botella al no ser posible a priori determinar cuál es el trabajo que tendrá que realizar cada núcleo, impidiendo un reparto de trabajo uniforme entre los mismos. Sin embargo, es razonable esperar que el tiempo de ejecución se reduzca de forma significativa.

Otras librerías, como ANN [2], no son paralelizables. Esto se debe a que sus algoritmos de búsqueda emplean alguna variable común, que impide que el proceso pueda ser realizado en paralelo. Por ello, durante el desarrollo del algoritmo a bajo nivel se ha intentado respetar en todo momento la modularidad de cada celda, de forma que cada celda contase con la información total necesaria para realizar las búsquedas, aunque esto supusiese un consumo de memoria adicional, como se explica en sec. 4.4.1.

De esta forma, el tener un algoritmo a alto nivel nos permite gestionar la búsqueda en varias celdas a la vez, pudiendo trabajar con varias celdas simultáneamente para distintos puntos *querying*, incluso aunque cada celda en sí misma no pueda ejecutar *queries* en paralelo. Es decir, el algoritmo a alto nivel puede paralelizar las búsquedas incluso cuando los algoritmos de bajo nivel son estrictamente *serie*. Se comprobará experimentalmente la mejora del rendimiento respecto al número de núcleos empleados.

Se intuye que la mejora del rendimiento no será lineal con la cantidad de núcleos empleados. Es decir, no seguirá la relación ideal: $T' \approx T/p$. Para ver qué tipo de relación existe entre el rendimiento y la cantidad de núcleos empleados habrá que tener una serie de factores:

- Tiempo de inicialización de los núcleos. Puede darse el caso que la tarea que vaya a realizar un núcleo no justifique su inicialización, ya que se invierte más tiempo en la preparación de este para que ejecute la tarea, que en la ejecución de la tarea en sí.
- Capacidad para gestionar la paralelización. Dependiendo de la librería empleada, es posible que la gestión de los núcleos llegue a consumir más recursos que la ejecución de la tarea que se pretende paralelizar.
- Diseño del algoritmo. No todas las etapas del algoritmo son paralelizables. En general no se realizar más de una búsqueda a la vez en la misma celda, aunque si se pueden realizar varias búsquedas en distintas celdas para el mismo

o distintos puntos *querying,* pudiendo seguir encargando búsquedas a todas aquellas celdas que no estén trabajando en cada momento.

- Tipos de núcleos empleados. Dependiendo de la máquina en la que se ejecute el programa puede ser que no valga la pena paralelizar en función de las características de capacidad de cálculo que tengan los núcleos procesadores.

Teniendo esto en cuenta, y ya que la gran mayoría de procesadores en el mercado pertenecen a Intel®, se ha decidido emplear la librería TBB (Intel Threading Building Blocks) [25]. Esta librería se caracteriza por ser una librería en C++ desarrollada por Intel® que permite paralelizar un programa en procesadores con múltiples núcleos.

La implementación de referencia del algoritmo en paralelo ha sido realizada por S. Tapia, aunque se han intentado otras librerías y tecnologías, por ejemplo: hilos nativos de la librería estándar de C++ o ZeroMQ [26]. Con la librería TBB se han conseguido mejores resultados que con otras librerías que permiten *multi-threading*. En concreto, para la paralelización se ha usado el módulo *TBB flow graph*, utilizando una arquitectura denomina *"data flow"* que consiste en la creación de nodos que realizan tareas definidas a partir de la transmisión de mensajes con los datos del problema.

5. Detalles de implementación

En este capítulo se explicarán aquellos detalles de la implementación que se consideran de especial interés o que son demasiado específicos como para haber sido incluidos en otros capítulos.

5.1. Diagrama de clases y herencia

Los diagramas mostrados a continuación incluyen los atributos y funciones más significativos de cada clase. Se han omitido los parámetros de los métodos, así como los tipos de datos para dar énfasis a la organización de las clases y sus relaciones de dependencia y herencia.

5.1.1. Bajo nivel

Se puede observar en Fig. 5.1 como todas las celdas heredan de un prototipo de celda *cell*, que es la clase base de la que heredan el resto y es la cual impone la interfaz con la que va a trabajar el algoritmo de alto nivel. Los métodos de esta clase son todos ellos virtuales puros, realizándose su implementación en las clases *Base_ cell* y *Tree_ cell*, que agrupan respectivamente las implementaciones comunes a las clases que derivan de ellas. Estos métodos pretenden dar soporte a diferentes acciones de la siguiente forma:

`clone` Reservar espacio en memoria y crear una celda del mismo tipo.

`set_limits` Fijar los límites de la celda. Para los tipos de celdas empleados, se fijan los atributos *Origin_ point* y *Cell_ length*. [1]

`reserve` Reservar en memoria el espacio que la celda necesitará para generar las estructuras de datos que emplee.

`add` Añadir los puntos pertenecientes a la celda.

`build` Construir las estructuras de datos que la celda va a emplear para dar soporte a las búsquedas.

`query` Búsqueda de radio fijo.

[1] Para dos dimensiones, equivalen a la esquina inferior izquierda del cuadrado y al lado del cuadrado respectivamente.

`kquery` Búsqueda de los k vecinos.

En *Base_ cell*, los atributos protegidos son comunes a *Par_ cell* y *Brute_ cell*. Sin embargo, tanto `query` como `kquery`, tienen implementaciones distintas en *Par_ cell* y *Brute_ cell*. Empleándose de hecho un atributo auxiliar adicional en *Par_ cell*.

En cambio, las búsquedas tienen una implementación común a *Stree_ cell* y *Utree_ cell* en *Tree_ cell*. Son los métodos privados `query_recursive` y `kquery_recursive` que realizan una búsqueda de binaria exactamente igual para cada árbol. El método que cambia entonces, será el método `build`, que recurre a dos versiones distintas para construir el árbol dependiendo de la clase con la que se trabaje, teniendo `build_recursive` una implementación distinta en cada una.

La clase *cell_ factory* será una clase auxiliar empleada por el algoritmo de alto nivel para rellenar el espacio con celdas a medida que se procesen los puntos con los que se pretende trabajar.

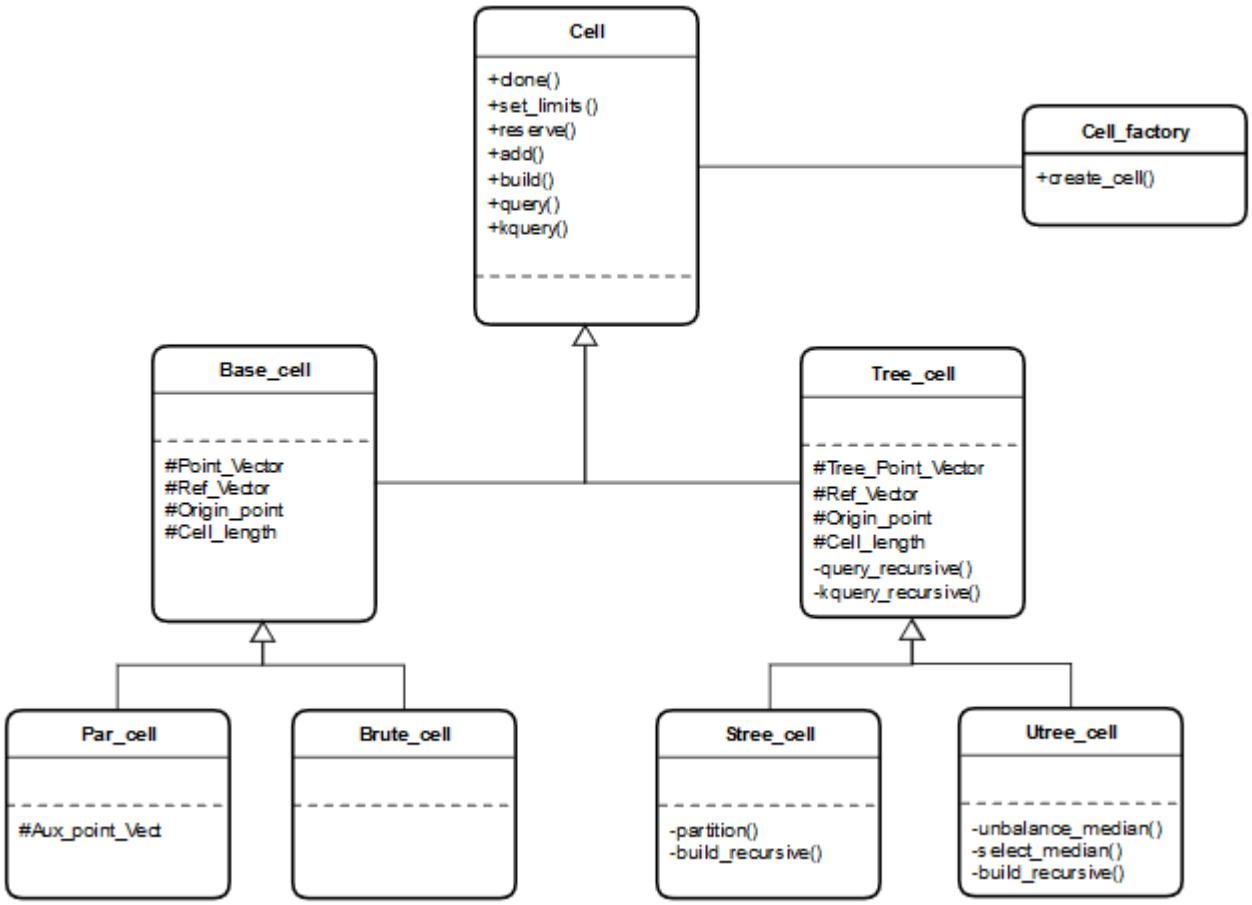


Figura 5.1.: Diagrama de clases de las celdas. Bajo Nivel

5.1.2. Alto nivel

En Fig. 5.2 se describen diferentes tipos de relaciones. Se muestra una relación de herencia que proviene de *searching_set* así como su dependencia respecto a *point_ adapter* y *tessel*. Esta clase, *searching_set*, es una clase abstracta, y todos sus métodos son virtuales.

Los métodos virtuales de esta clase pretenden dar soporte a las siguientes acciones:

`create_cells` Creación de celdas en función de los puntos del dominio de trabajo.

`build` Construcción de las estructuras de datos necesarias en cada celda creada.

`get_range` Obtención de `range`, variable que expresa el tamaño del lado de cada celda como $2^{\texttt{range}}$.

`count_point` Contabilización del número de puntos insertados en cada celda creada.

`insert` Inserción de los puntos en sus celdas correspondientes.

`query` Búsqueda de radio fijo.

`kquery` Búsqueda de k vecinos.

Esta clase abstracta, representa un conjunto de búsqueda en el cual se insertan los puntos con los que se trabaja y es posible obtener de forma eficiente los vecinos a un punto *querying*. La implementación de cada uno de estos métodos virtuales corre a cargo de los distintos subtipos de conjuntos. Esta clase tiene una relación de dependencia con *point_ adapter* y *tessel*. Esta última, representa la división del espacio en celdas a partir de un tamaño que introduce el usuario final. Está definida por su coordenada de origen. Por otro lado, *point_ adapter* es una adaptador para cualquier clase de punto que es capaz de obtener la dirección de memoria de un objeto donde se guarde un punto y las coordenadas del mismo a partir de una referencia o un puntero. Es decir, independientemente de la clase original del punto con la que trabaje el usuario, *point_ adapter* obtendrá sus coordenadas y una dirección de memoria. El único requisito es que la clase utilizada por el usuario tenga un método *get_ coordinates* que devuelva las coordenadas. Esta clase permite obtener los datos del punto a partir de iteradores de forma cómoda para el usuario, simplificando significativamente la interfaz. Además, encuentra la celda del punto mediante el cálculo del *tessel* correspondiente en el espacio en función del tamaño deseado.

Las clases derivadas de *point_ counter*, y por lo tanto de *searching_ set*, trabajan con *point_ adapter* y *tessel* e implementan los métodos definidos por estas clases base. Se distinguen dos tipos de conjuntos de búsqueda:

searching_set_rec Es la implementación más general del algoritmo de alto nivel. Se generaliza para cualquier número de dimensiones la búsqueda de radio fijo y de k vecinos y se gestionan los resultados obtenidos en las celdas para producir el resultado final. Cuenta con una serie de optimizaciones simples para dimensión 3.

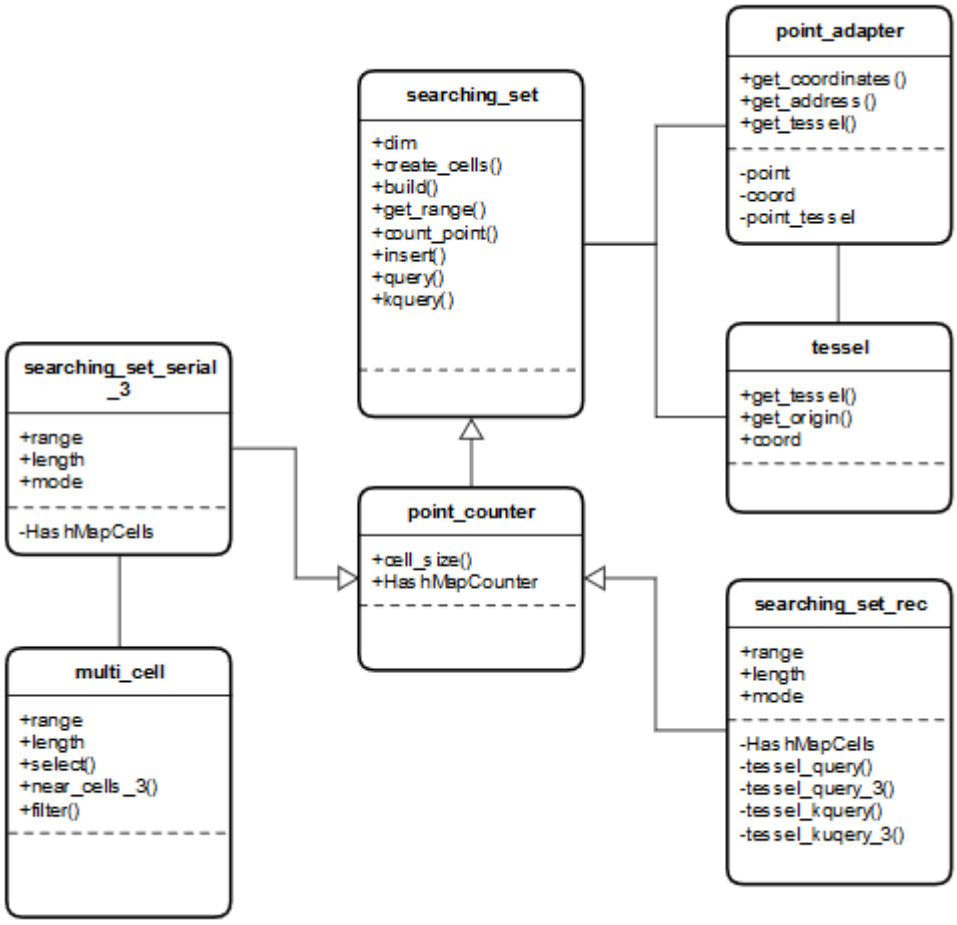


Figura 5.2.: Diagrama de clases de conjuntos de búsqueda. Alto Nivel.

searching_set_serial_3 Es la implementación específicamente diseñada y optimizada para trabajar en dimensión 3, permitiendo una búsqueda más eficiente de vecinos para las búsquedas. Su implementación ha sido realizada por S. Tapia.

La interfaz de alto nivel de la libraría utiliza un diseño parecido a los contenedores que se implementan en la STL, su diseño ha sido desarrollado por S. Tapia y consiste en el uso de los siguiente tipos y funciones globales sobrecargadas:

Ejemplo 5.1: Interfaz

```
/* first = norm, second = reference */
typedef std::pair< double, uintptr_t > distance_and_ref;

typedef std::unordered_map< uintptr_t, std::vector<uintptr_t> >
    multiquery_result_t;
typedef std::unordered_map< uintptr_t, std::vector<distance_and_ref> >
     multi_K_query_result_t;


template<typename Iterator, typename type_of_set>
void insert(type_of_set& the_set, Iterator first, Iterator last);

template<typename Iterator, typename type_of_set>
multiquery_result_t query(type_of_set& the_set, double radius,
    Iterator first, Iterator last);

template<typename Iterator, typename type_of_set>
multi_K_query_result_t kquery(type_of_set& the_set, size_t K, Iterator
     first, Iterator last);
```

Cada clase *searching_set* puede implementar dichas funciones mediante una plantilla o mediante una sobrecarga desde cero para un grado de especialización.

Se define una función, `insert`, a partir de la cual, dado un *searching_set* y dos iteradores que marquen el inicio y el final del conjunto de puntos *searching* a insertar, se prepara el *searching_set* para poder realizar búsquedas de radio fijo y de k vecinos. Además, se definen dos tipos de datos en forma de *Hash-maps* que serán los retornos de las búsquedas `query` y `kquery`, que generan los resultados a partir de un parámetro de búsqueda y de dos iteradores que marquen el inicio y el final del conjunto de puntos *querying*. Estos tipos de datos son:

`multiquery_result_t` Dada la dirección de memoria del punto de búsqueda *querying*, devuelve un vector de las direcciones de memoria de sus vecinos en la búsqueda de radio fijo.

`multi_K_query_result_t` Dada la dirección de memoria del punto de búsqueda *querying*, devuelve un vector de elementos en los que se almacenan las distancias al *querying* y la direcciones de memoria de sus vecinos en la búsqueda de k vecinos.

5.2. Cálculo de la distancia

Como ya se ha explicado, se entiende por distancia al cuadrado de la norma euclídea. Esto es así porque el cálculo de la raíz cuadrada del cuadrado de las componentes del vector distancia no aporta información adicional. Si un punto *searching* se encuentra más alejado que otro respecto a un *querying*, su distancia al cuadrado será obviamente mayor que la del más cercano.

Ejemplo 5.2: Cálculo de la distancia

```
1 inline double norm( const std::array<double, DIM>& p2 ) const
2 {
3     double sum = 0; unsigned i;
4     for ( i = 0; i < DIM; ++i ) {
5         sum += ((*this)[i]-p2[i])*((*this)[i]-p2[i]);
6     }
7     return sum;
8 }
```

5.3. Bit_lattice

Aprovechando la forma en que los números en coma flotante están codificados, como se explica en el Apéndice A, se trabajará con los bits de signo, exponente y mantisa para clasificarlos en el espacio. Se emplea un parámetro, `range`, mediante el cual se dimensiona el tamaño de la celda. Las celdas serán cubos (en el caso de tres dimensiones) de lado $2^{\texttt{range}}$. La función `bit_lattice` devuelve el índice correspondiente de la celda.

El procedimiento a seguir es el siguiente:

- Verificación de que el número no es cercano al origen, pudiéndose clasificar de inmediato como -1 ó 0. Con esta verificación se cubriría también el caso de los números denormalizados.
- Enmascaramiento de los distintos campos de bits de la codificación y almacenamiento de los mismos en forma de enteros sin signo.
- Desplazamiento de bits de la mantisa igual a la diferencia entre `range` y el exponente.

De esta forma, dada una coordenada comprendida entre $[n * 2^{\texttt{range}}, (n+1) * 2^{\texttt{range}})$, `bit_lattice` devolverá n.

Ejemplo:
Para un `range`=2, el índice de la celda correspondiente a cada una de estas coordenadas será:

- $-4{,}1 \longrightarrow -2$
- $-0{,}25 \longrightarrow -1$
- $0{,}25 \longrightarrow 0$
- $2{,}0 \longrightarrow 0$
- $3{,}9 \longrightarrow 0$
- $4{,}0 \longrightarrow 1$
- $7{,}9 \longrightarrow 1$
- $8{,}1 \longrightarrow 2$

Ejemplo 5.3: Bit lattice

```
1  inline long long bit_lattice(double x, int range)
2  {
3      if ( x > 0 && x < exp2(range) )
4      {
5          return 0;
6      }
7      else if ( x < 0 && -x < exp2(range) )
8      {
9          return -1;
10     }
11
12     const long long unsigned SIGN_BIT = 0x8000000000000000Lu;
13     const long long unsigned EXPO_BIT = 0x7FF0000000000000Lu;
14     const long long unsigned CAND_BIT = 0x000FFFFFFFFFFFFFLu;
15
16     union number {
17         double re;
18         long long unsigned i;
19     };
20     union number u;
21     u.re = x;
22
23     int sign = u.i & SIGN_BIT ? 1 : 0;
24     long long unsigned exponent = (u.i & EXPO_BIT) >> 52Lu;
25     long long unsigned significand = (u.i & CAND_BIT) + (1Lu << 52);
26
27      if(!exponent) // If exponent is all zeros, it is denormalized
28     {
29         return sign? -1 : 0;
30     }
31
```

```
32      /* significand is *2^52 and exponent exceed is 1023 */
33      long long unsigned u_result = significand >> ( 1023 - exponent +
            range + 52 );
34      if ( sign ) /* negative */
35      {
36          return -1L - (long long)u_result;
37      }
38      else
39      {
40          return (long long)u_result;
41      }
42  }
```

5.4. Hashing

Para garantizar que se minimizan las *Hash-collisions*, se emplea la siguiente operación adaptada de `boost::hash_combine` [27].

Ejemplo 5.4: Hashing

```
1  namespace std
2  {
3      template< unsigned DIM > struct hash< tessel<DIM> >
4      {
5          typedef tessel<DIM> argument_type;
6          typedef std::size_t result_type;
7          result_type operator()(argument_type const& k) const
8          {
9              std::hash<long long> hash_f;
10             size_t h = 0;
11
12             h =  hash_f(k.coor[0]);
13             // Magic Number and Expression adapted from boost::
                   hash_combine
14             unsigned i;
15             for ( i = 1; i < DIM; ++i ) {
16                 h ^= hash_f(k.coor[i]) + 0x9e3779b9 + (h<<6) + (h>>2);
17             }
18             return h;
19         }
20     };
21 }
```

Donde:

`hash_f` es la función de *hashing* implementada por defecto en la librería estándar de C++ `std::unordered_map` [19].

`h` es una clave de *hash* previamente generada por la función `hash_f` a partir de la primera coordenada del punto que caracteriza a la celda.

`k.coord[i]` es la coordenada `i` del punto que caracteriza a la celda.

`0x9e3779b9` es una constante generada a partir del número áureo, $\varphi = (1 + \sqrt{5})/2$, tal que `0x9e3779b9` $= 2^{32}/\varphi$.

Esta constante de 32 bits, se caracteriza por tener el mismo número de 0s que de 1s, sin una correlación simple entre los bits. La inclusión aleatoria de esta constante cambia cada uno de los bits de la clave de *hash*. Esto implica que valores consecutivos generarán claves separadas entre sí dentro de la tabla, y pequeñas diferencias en un sólo bit generarán números muy diferentes. De esta forma, incluso para muestras pequeñas y similares, las claves generadas se distribuyen uniformemente dentro de la tabla. Siendo la implementación de esta operación ampliamente reconocida y alabada por su buen funcionamiento en la práctica, se espera que mediante el uso de la misma la probabilidad de producirse *Hash-Collisions* sea mínima, garantizando la estabilidad de la implementación propuesta [28], ya que se producirían el número mínimo de operaciones de *rehashing*.

5.5. Algoritmos de ordenación

Los algoritmos de ordenación aquí presentados son aquellos que han sido implementados explícitamente durante el desarrollo del trabajo para ser empleados en la construcción de los árboles. Si bien se emplea también otro algoritmo de ordenación, *quicksort* en `std::sort` [23], su implementación difiere dependiendo de la implementación de la librería estándar empleada y no se ha estudiado en profundidad durante el desarrollo del trabajo.

5.5.1. *Quickselect*

El algoritmo empleado para implementar `select_median` en *utree* a partir del cual hayamos cuál es el valor de la mediana del conjunto de datos, es el comúnmente conocido como *quickselect*.

Quickselect es un algoritmo de selección para encontrar el k-ésimo elemento más pequeño de una lista desordenada de elementos. Está relacionado con el algoritmo de ordenación *quicksort*. Como este, es eficiente en la práctica y tiene un buen rendimiento para su caso promedio, aunque se puede degradar para su caso peor.

El algoritmo consiste en escoger un elemento como pivote, y dividir los datos en dos respecto a este pivote. Los elementos menores se situarían a su izquierda y los elementos mayores a su derecha. Sin embargo, a diferencia de *quicksort*, en vez de ejectuarse recursivamente a ambos lados, sólo sigue ejecutándose en el lado de interés, reduciéndose la complejidad del algoritmo a *O(n)*. Además, el algoritmo ordena parcialmente el conjunto de datos, garantizando que todos los elementos menores que el pivote se encuentren a la izquierda y todos los elementos mayores que el pivote a la derecha.

Algorithm 5.1 Algoritmo para selección de la mediana [29]

Require: A: lista de elementos, k := k-esimo elemento
  left := index inicial
  right := index final
  loop
    if left = right **then**
      return A[left]
    end if
    pivotIndex ← ...//*Seleccionar un pivote arbitrario*
    pivotIndex ← partition(list, left, right, pivotIndex)
    if k = pivotIndex **then**
      return A[k]
    else if $k <$ pivotIndex **then**
      right← pivotIndex−1
    else
      left← pivotIndex+1
    end if
  end loop

La elección del pivote es crucial para el buen rendimiento de este algoritmo. Una buena elección del pivote implica que las comparaciones a realizar en cada iteración se reducen exponencialmente. Para una elección de pivotes que redujese a la mitad la cantidad de elementos a examinar se tendría una progresión geométrica, cuyo límite está acotado y el tiempo consumido por el algoritmo en su conjunto sería constante para n elementos.

0	n
1	$n/2$
2	$n/4$
...	
logn	1

Sumando el conjunto de comparaciones para una n grande:

$$\lim_{n\to\infty} \sum_{r=0}^{logn} n \left(\frac{1}{2}\right)^r = \lim_{n\to\infty} n \sum_{r=0}^{logn} \left(\frac{1}{2}\right)^r = \lim_{n\to\infty} n \left(\frac{1}{1-\frac{1}{2}}\right) = \lim_{n\to\infty} 2n \to O(n)$$

La complejidad del algoritmo es *O(n)* para una elección apropiada de pivotes, como se quería demostrar. Por el contrario, si se escogen repetidamente malos pivotes, el número de comparaciones a realizar sólo disminuirá en un único elemento en cada iteración, siendo el rendimiento para el peor caso: $O(n^2)$. Esto ocurre, por ejemplo, en la búsqueda del elemento máximo de un conjunto, utilizando el primer elemento como pivote y con datos ordenados.

Se ha elegido este algoritmo porque permite conocer el valor de la mediana en un tiempo óptimo en el caso promedio, y a la facilidad de su implementación.

En la implementación realizada, se han incorporado una serie de optimizaciones. No se realizan intercambios en caso de tratarse del mismo índice y el subalgoritmo de partición de datos está implementado de forma iterativa (lo que ahorra saturar la pila con llamadas adicionales a funciones).Como pivote se ha escogido el valor del elemento en la posición correspondiente a la mitad del conjunto, ya que a la hora de tratar con datos reales, estos están parcialmente ordenados y se garantiza una menor probabilidad de degradar las propiedades del algoritmo.

Ejemplo 5.5: Implementación de quickselect

```
1  int select_median(int start, int end, const unsigned axis)
2  {
3      int i, pidx, median = start + (end-start)/2;
4      double pivot;
5      while(1){
6          if ( start == end ) return start;
7          pivot = this->point_vector[median][axis];
8
9          //Partition algorithm
10         this->swap(median, end); // Move pivot to end
11         for ( pidx = i = start; i < end - 1; ++i )
12         {
13             if (  this->point_vector[i][axis] <= pivot )
14             {
15                 if( i != pidx ) this->swap(pidx, i);
16                 ++pidx;
17             }
18         }
19         this->swap(end, pidx); // Move pivot to its final place
20         //End of partition
21         if ( pidx == end ) return pidx;
22         if ( end  <  pidx  ) end = pidx - 1;
23         else start = pidx + 1;
24     }
25 }
```

5.5.2. *National Dutch Flag Problem*

El algoritmo empleado para implementar `unbalance_median` y `partition`, para *utree* y *stree* respectivamente es el que soluciona el problema de la bandera de los Países Bajos (*National Dutch Flag Problem)* [30].

El enunciado del problema es el siguiente: La bandera de los Países Bajos está formada a partir de tres colores, rojo, blanco y azul. Si se tiene un conjunto de bolas de estos colores alineadas de forma aleatoria, ordene las bolas de forma que aquellas que tengan colores iguales se encuentren agrupadas, y que sus colores correspondientes estén en el orden correcto.

Algorithm 5.2 Algoritmo para *National Dutch Flag Problem*

Require: A: lista de elementos, mid := valor
  $i \leftarrow 0$
  $j \leftarrow 0$
  $n \leftarrow size(A) - 1$
  while $j \leq n$ **do**
    if A[j]$< mid$ **then**
      $Swap(A[i], A[j])$
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$
    else if $mid < A$[j] **then**
      $Swap(A[j], A[n])$
      $n \leftarrow n - 1$
    else
      $j \leftarrow j - 1$
    end if
  end while

El problema también se puede enfocar como un problema de reordenamiento de elementos en un vector. En el caso que se pretende resolver, se clasificarían los elementos en tres categorías (*left*, *mid*, *right*), menores que la mediana, iguales que la mediana y mayores que la mediana. El problema entonces trataría de agrupar los elementos del vector tal que todos los elementos menores que la mediana se encuentren antes que (tengan un índice menor que el índice de) todos los elementos iguales a la mediana, que a su vez se encuentran antes que todos los elementos mayores que la mediana.

Figura 5.3.: Bandera de los Países Bajos

El algoritmo consiste en hacer que el grupo de mayores crezca desde la parte derecha del vector, y el grupo de menores crezca desde la parte izquierda del vector, y mantener el grupo de iguales justo por encima de los menores. El algoritmo lleva la cuenta de tres índices, el del elemento más a la izquierda del grupo de los mayores, el del elemento más a la derecha del grupo de los menores, y el del elemento más

a la derecha del grupo de los iguales. Los elementos por ordenar se encontrarán entre el grupo de los iguales y de los mayores. En cada iteración, se examina el elemento que está justo a la derecha del grupo de los iguales. Si este pertenece al grupo de los mayores, se intercambia con el elemento que se encuentre justo a la izquierda del grupo de los mayores. Si pertenece al grupo de los menores, se intercambia este con el elemento que se encuentre justo a la derecha del grupo de los menores. Si pertenece el grupo de los iguales, no es necesario realizar ninguna acción. A continuación se actualizan los índices correspondientes y se ejecuta la siguiente iteración. El algoritmo termina cuando al actualizarse el índice que señala al elemento más a la derecha del grupo de los iguales, este pasa a señalar un elemento que pertence al grupo de los mayores. La complejidad del algoritmo es *O(n)* para intercambios y comparaciones.

Se ha elegido este algoritmo porque permite conocer en qué posición (índice) se encuentra el fin del grupo de los iguales, lo que equivale a obtener la posición más a la derecha en la que se encontraría la mediana en caso de haber valores repetidos.

Ejemplo:
Dado un vector formado por los siguientes siete elementos: { 7 5 1 6 5 2 5 }

- Tras aplicar el algoritmo: 2 1 5 5 5 7 6
- Mediana: 5
- Menores: 2 1
- Iguales: 5 5 5
- Mayores: 7 6

La mediana es el valor correspondiente al elemento en la cuarta posición. Sin embargo, hay valores repetidos. La posición que se obtiene es la quinta, a partir de la cual se hace la división del vector en izquierda (2 1 5 5 5) y derecha (7 6) para la construccion del árbol. Nótese que los elementos menores y mayores no tienen que estar necesariamente ordenados.

En la implementación realizada, no se realizan intercambios en caso de tratarse del mismo índice.

Ejemplo 5.6: Implementación del algoritmo de National Dutch Flag Problem

```
1 int partition (int start, int end, const unsigned axis, double
      pivot_value){
2     int left = start;
3     int mid = start;
4         int right = end;
5     while ( mid <= right )
6     {
7         if( this->point_vector[mid][axis] < pivot_value )
8         {
9             if( left != mid ) this->swap(left, mid);
```

```
10              ++left;
11              ++mid;
12          }
13          else if ( this->point_vector[mid][axis] > pivot_value )
14          {
15              if( mid != right ) this->swap(mid, right);
16              --right;
17          }
18          else ++mid;
19      }
20      if( right < start ) return start;
21      else return right;
22  }
```

6. Pruebas

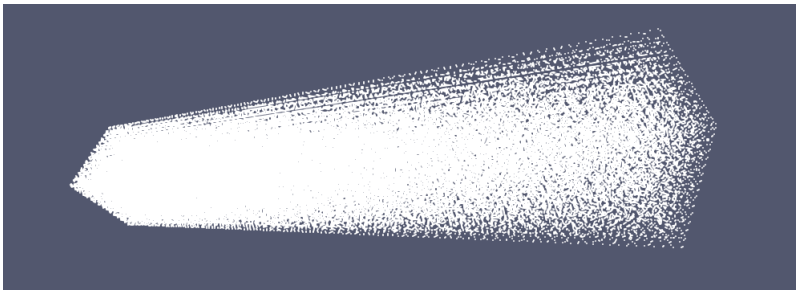
En este capítulo se explicarán las pruebas y metodología empleadas durante el desarrollo de la implementación.

6.1. Implementación de fuerza bruta

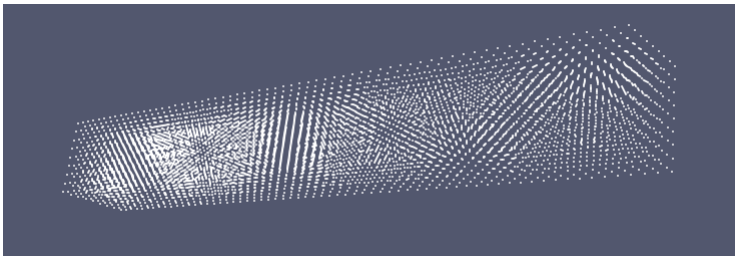
Para comprobar el correcto funcionamiento del método de fuerza bruta, se decide emplear una herramienta de procesamiento de datos existente, como puede ser OpenCalc. Debido a la imposibilidad de incluir una gran cantidad de datos debido a la falta de memoria, se escoge un subconjunto arbitrario de una muestra de datos real. Estos datos, que también serán los datos empleados para el benchmarking, son los siguientes:

Conjunto 1: Iris

Formado por un total de 98304 puntos *searching* y 5265 puntos *querying.*



(a) *Searching*



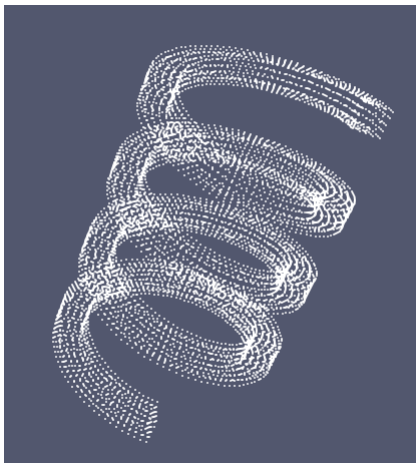
(b) *Querying*

Figura 6.1.: Iris

Conjunto 2: Muelle

Formado por un total de 51200 puntos *searching* y 10025 puntos *querying.*

(a) *Searching*

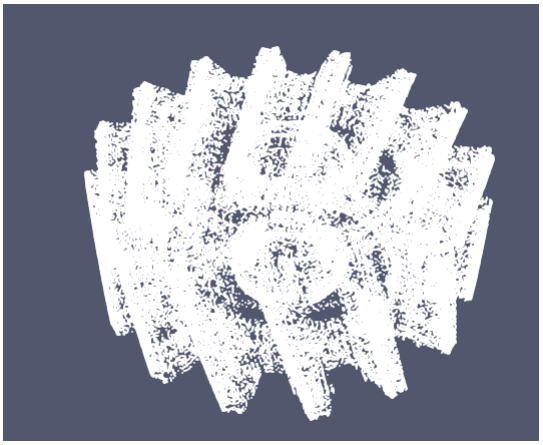


(b) *Querying*

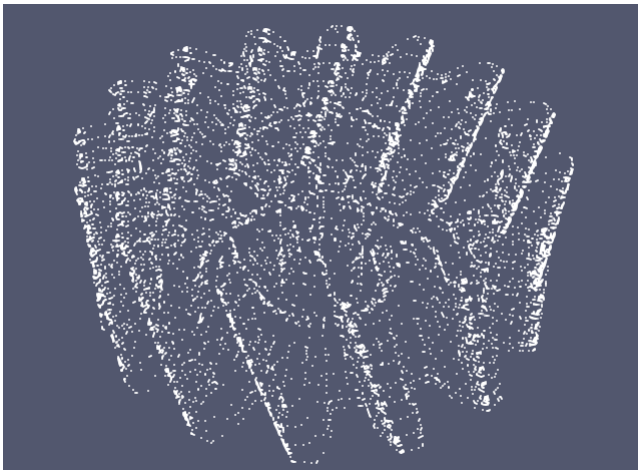
Figura 6.2.: Muelle

Conjunto 3: Rueda dentada

Formado por un total de 116528 puntos *searching* y 8896 puntos *querying.*



(a) *Searching*



(b) *Querying*

Figura 6.3.: Rueda dentada

Para comprobar el correcto funcionamiento, se cargan los 500 primeros *searching* del conjunto de datos **Iris**, y se realiza la búsqueda con diversos *querying* escogidos arbitrariamente. Se puede apreciar en la Tab. 6.1, que salvando los redondeos automáticamente realizados por OpenCalc, si se ordenan por distancia al *querying* los puntos *searching*, estos coinciden con aquellos obtenidos mediante la implementación de fuerza bruta.

Cuadro 6.1.: Verificación método de fuerza bruta para *querying* (-0.0125 0 0)

(a) Vecinos obtenidos con el algoritmo

label: 493, (-0.187500 0.008637 0.204775)

label: 492, (-0.215451 0.008637 0.204775)

label: 495, (-0.159549 0.036588 0.232725)

label: 494, (-0.187500 0.008637 0.232725)

label: 497, (-0.187500 0.008637 0.329775)

label: 496, (-0.215451 0.008637 0.329775)

label: 499, (-0.159549 0.036588 0.357725)

label: 498, (-0.187500 0.008638 0.357725)

label: 488, (-0.241363 0.034549 1.204775)

label: 490, (-0.241363 0.062500 1.204775)

(b) Vecinos obtenidos con OpenCalc

B	C	D	E	K
-1,88E-001	8,64E-003	2,05E-001		0,04591349
-2,15E-001	8,64E-003	2,05E-001		0,050188587
-1,60E-001	3,66E-002	2,33E-001		0,05669345
-1,88E-001	8,64E-003	2,33E-001		0,058141965
-1,88E-001	8,64E-003	3,30E-001		0,11273214
-2,15E-001	8,64E-003	3,30E-001		0,117007237
-1,60E-001	3,66E-002	3,58E-001		0,130499804
-1,88E-001	8,64E-003	3,58E-001		0,131948329
-2,41E-001	3,45E-002	1,20E+000		1,466216722
-2,41E-001	6,25E-002	1,20E+000		1,468929329

6.2. Toma de contacto: *kd-Trees*

Para la adquisición de los conocimientos y herramientas necesarios para poder desarrollar una implementación propia de árboles de búsqueda binarios k dimensionales, *kd-Trees,* se han empleado los siguientes materiales:

- Código libre de la comunidad Rosetta [22].

- Material de estudio de la Universidad de Stanford [31].
- Diversas implementaciones en GitHub, destacando [32].

Debido a la sencillez y a que el código está escrito en C, la implementación de los *kd-Trees* que se desarrolla se basa en el material perteneciente a la comunidad Rosetta. La implementación de la comunidad Rosetta sólo da soporte a búsquedas del mejor vecino, por lo que una vez adquirida una idea general del funcionamiento de los *kd-Trees,* se adapta como parte del aprendizaje la implementación de esta para la búsqueda de radio fijo, y de los k vecinos. Además, se reescribe en C++ parte del código.

A continuación se comprueba el correcto funcionamiento de la misma para una distribución de puntos aleatoria en tres dimensiones. Esta distribución ha sido generada por medio la función `rand()` de la librería estándar de C, constando de 100000 (cien mil) puntos en coma flotante *searching* entre 0 y 10, y 10000 (diez mil) puntos en coma flotante *querying* también entre 0 y 10. Para corroborar que los *kd-Trees* tienen el funcionamiento esperado, se comparan los resultados de las búsquedas obtenidas con las obtenidas por el método de fuerza bruta, verificando que la implementación de los *kd-Trees* para la búsqueda del mejor vecino, de radio fijo, y la de los k vecinos arrojan los mismos resultados.

Si bien parece que una primera implementación parece funcionar con la distribución de datos aleatoria, no parece hacerlo para ninguno de los conjuntos de datos de aplicación real. Los resultados obtenidos al comparar los métodos de búsqueda de fuerza bruta con el implementado mediante *kd-Trees*, presentan un porcentaje de errores variable.

Siendo en varios casos un porcentaje inadmisible de errores, se plantea entonces si el algoritmo de partición empleado por la comunidad Rosetta está mal implementado, o si el código propio que implementa las búsquedas de radio fijo y las de los k mejores vecinos funciona como se pretende.

6.3. Construcción del árbol

De la sección anterior se deduce que ni la parte de la implementación propia, ni la de Rosetta, tiene errores de base pues arroja resultados exactamente iguales a los generados por fuerza bruta para la distribución aleatoria de datos. El error por tanto debe tener relación con las características de las distribuciones de datos. Se descarta el tamaño del conjunto de datos con los que se trabaja, pues tanto el conjunto de puntos aleatorio como los conjuntos reales tienen tamaños del mismo orden de magnitud. La hipótesis es que el error debe estar relacionado con la repetición de los datos, ya que en la muestra completamente aleatoria, al tratarse de números reales, no se repite ningún valor para ninguna coordenada [1].

[1] Al tratarse de un continuo, la probabilidad de obtener aleatoriamente el mismo punto es igual a cero. Si bien los números en coma flotante están acotados superior e inferiormente y su precisión

Como se explica en Capítulo 2, aunque Codeblocks cuenta con un aplicación de *debugging*, debido a la naturaleza del programa, en el que las funciones tanto de construcción como de búsqueda son llamadas de forma recursiva, y se trabaja con grandes cantidades de datos, se va a descartar hacer un análisis instrucción a instrucción de código máquina generado por el compilador y en su lugar se intentará probar por medio de programas simples el funcionamiento de las partes del programa susceptibles a estar mal implementadas.

En primer lugar se crea una función que compruebe que el árbol esté bien construido. Es decir, que para el primer nivel, o raíz, el valor de la coordenada x para todos los puntos de la rama izquierda sea menor o igual que el valor de la coordenada x de la raíz, y que el valor de la coordenada x para todos los puntos de la rama derecha sea mayor. Se comprueba recursivamente esta propiedad del árbol en todas sus subramas, rotando en cada nivel el eje de trabajo.

Empleando esta función se descubre que el árbol no está debidamente construido. Tras un análisis exhaustivo, se descubre el error. En concreto, dado el valor de la coordenada correspondiente a ese nivel del nodo con el que se trabaja, es posible encontrar en sus subramas, arbitrariamente tanto a izquierda como a derecha, puntos con un valor igual, cuando estos deberían encontrarse exclusivamente en la subrama izquierda.

Se verifica entonces la hipótesis de que el error se produce debido a la repetición de valores para una misma coordenada. El error se debe producir por tanto en el algoritmo de *quickselect* empleado en la implementación de Rosetta. Este algoritmo, cuyo funcionamiento se explica en sec. 5.5, se caracteriza por devolver la mediana de un conjunto de datos y garantizar que todos los puntos a la izquierda de la mediana son menores que esta, y todos los puntos a la derecha son mayores. Por desgracia, si el dato de la mediana está repetido, estos dados son arbitrariamente situados a izquierda y derecha, inutilizando el el empleo de este algoritmo para la construcción del árbol cuando se trabaja con puntos cuyas coordenadas tienen valores repetidos.

Una vez se llega a esta conclusión surge la necesidad de arreglar este error. Se podría emplear directamente *quicksort* como se hace en el método de fuerza bruta, pero siendo este *O(nlogn),* se degradaría significativamente el tiempo de construcción del árbol, pues *quickselect* trabaja en *O(n).* Se decide por tanto buscar un algoritmo que resuelva este error con un tiempo de menor complejidad.

Consultando en la literatura se haya finalmente el algoritmo que soluciona el *Dutch national flag problem* [30]. Este algoritmo permite agrupar una lista de datos en tres bloques de diferentes características. En el caso que se trata, las características de cada bloque serán:

- Menores que la mediana

decimal es limitada, tanto las cotas como la precisión son suficientemente grandes como para que la probabilidad de que se repita un punto en una muestra aleatoria de cien mil puntos sea cero en la práctica.

- Iguales a la mediana
- Mayores a la mediana

Estando ordenados los bloques en ese orden. Por tanto, si se toma como nodo al último dato (punto) del bloque de "Iguales a la mediana", haciendo que aquellos puntos que estén a su izquierda sean tratados en la rama izquierda, y aquellos a su derecha en la rama derecha, repitiendo el proceso recursivamente, se conseguirá el objetivo deseado, construyendo un *kd-Tree* funcional.

Tras la implementación de este algoritmo en la construcción del árbol, se consigue que la función que comprueba que el árbol esté bien construido no arroje ningún error, cumpliendo el objetivo.

6.4. Creación de *utree* y *tree*

Tras conseguir el correcto fucionamiento del algoritmo de construcción del árbol gracias al empleo del algoritmo que soluciona el *Dutch national flag problem*, surge la idea de diseñar dos tipos de árbol en función del algoritmo empleado en su construcción. Es ahora cuando surge la idea de *utree_ cell* y *stree_ cell*.

El primero, *utree_ cell*, es el que directamente surge al solucionar el funcionamiento de la implementación de Rosetta para valores repetidos. La idea de este árbol es que en cada nivel se intenten distribuir en ambas ramas la mitad exacta de los puntos, lo que no es posible si el valor de la mediana está repetido. Las ramas se desequilibrarán en función del número de valores de la mediana repetidos.

Sin embargo, *stree_ cell*, se caracteriza por escoger un punto aleatorio para realizar la distribución de los puntos en ambas ramas, ya que sólo se emplea el algoritmo que soluciona el *Dutch national flag problem*, agrupando el conjunto de datos en menores, iguales o mayores al punto escogido aleatoriamente. De esta forma se emplea un único algoritmo de ordenación durante la construcción, acelerando el proceso.

Se comprueba la correcta construcción de este mediante la misma función y se verifica que no arroja ningún error.

6.5. Algoritmos de búsqueda

Una vez construidos apropiadamente los árboles, se procede a comparar los resultados de las búsquedas de radio fijo y de los k mejores vecinos con las de los métodos de fuerza bruta. Obtenemos los siguientes resultados:

- **Tamaño distinto:** Indica que el número de vecinos como resultado de la búsqueda difiere con el número de vecinos obtenido por el método de fuerza bruta.

- **No coincidentes:** Indica que si bien el número de vecinos es idéntico al obtenido por fuerza bruta, los vecinos encontrados no son los mismos.

Se entiende por errores que, dado un *querying*, la resultados de la búsqueda entre un método y el de fuerza bruta no coincidan.

Cuadro 6.2.: Errores obtenidos con **Iris**, $r = 0.005$, $k = 20$

	Radio fijo		k Vecinos	
	Tamaño distinto	No coincidentes	Tamaño distinto	No coincidentes
Stree_ cell	0	0	486	7
Utree_ cell	0	0	507	8

Cuadro 6.3.: Errores obtenidos con **Muelle**, $r = 0.0005$, $k = 20$

	Radio fijo		k Vecinos	
	Tamaño distinto	No coincidentes	Tamaño distinto	No coincidentes
Stree_ cell	0	0	0	1
Utree_ cell	0	0	0	0

Cuadro 6.4.: Errores obtenidos con **Rueda dentada**, $r = 0.0005$, $k = 20$

	Radio fijo		k Vecinos	
	Tamaño distinto	No coincidentes	Tamaño distinto	No coincidentes
Stree_ cell	0	0	2	0
Utree_ cell	0	0	3	0

Se comprueba que el algoritmo de búsqueda funciona para las búsquedas de radio fijo, coincidiendo los resultados, para ambos tipos de árbol con el método de fuerza bruta en un 100 % de los casos.

El error por tanto debe encontrarse en el algoritmo de búsqueda de los k vecinos, donde se aprecia claramente que el número de errores es muy similar en ambos árboles, por lo que en un principio se podría descartar como origen de los errores el algoritmo de construcción del árbol.

La implementación en este momento del algoritmo de búsqueda de los k vecinos, a diferencia de la implementación final sec. 4.5.4.3, emplea un vector auxiliar en el que se almacenan aquellos puntos que se encontraban a la misma distancia que el peor vecino. El uso de este vector auxiliar es innecesario y ralentiza innecesariamente el algoritmo al tener que pasar la referencia de un contenedor más a cada una de las llamadas recursivas de la función. Sin embargo, el error de este algoritmo, al margen de su peor rendimiento, se encuentra en que el vector auxiliar no llega a borrar su contenido cuando se mejora la distancia del peor vecino. Esto es así porque la lógica para el borrado está anidada dentro de un *if*. Este *if* tiene como condición que se encuentre un punto a la misma distancia que el peor vecino actual. Se puede

entonces dar el caso, de que una vez mejorada la distancia del peor vecino, no se vuelva a encontrar otro punto a la misma distancia, provocando que el contenido del vector auxiliar no se llegue a borrar. Al no borrarse, se añaden en los resultados de la búsqueda los puntos almacenados en este vector, que no corresponden a los vecinos reales.

Ejemplo 6.1: Implementación inicial de búsqueda de k vecinos

```
1  void kquery_recursive (int index, const double querying[DIM], std::
       vector<unsigned> &neighbors, std::vector<unsigned> &
       extra_neighbors, const int K, unsigned *max_idx, double *max_dist,
        double *extra_max_dist, unsigned axis)
2  {
3      if ( index== -1 ) return;
4      double d, dx, dx2;
5      d = norm(point_vector[index].p, querying);
6      unsigned stored = neighbors.size();
7
8      if( stored < K ){
9          neighbors.push_back(index);
10         if( d > *max_dist ){
11             *max_dist = d;
12             *max_idx = stored;
13         }
14         stored++;
15     }
16     else if ( d == *max_dist ){
17         if( *max_dist < *extra_max_dist ){
18             extra_neighbors.clear();
19         }
20         *extra_max_dist = *max_dist;
21         extra_neighbors.push_back( index );
22     }
23     else if( d < *max_dist ){
24         neighbors[*max_idx] = index;
25         *max_dist = 0;
26         int i;
27         for( i = 0; i < K; ++i )
28         {   /*Tag the biggest distance*/
29             double aux_dist = norm(point_vector[neighbors[i]],
                   querying);
30             if( aux_dist > *max_dist){
31                 *max_dist = aux_dist;
32                 *max_idx = i;
33             }
34         }
35     }
36     /* if chance of exact match is high */
37     if ( !*max_dist ) return;
38     dx = point_vector[index].p[axis] - querying[axis];
39     dx2 = dx * dx;
40
41     if ( ++axis >= DIM ) axis = 0;
42     kquery_recursive(dx > 0 ? point_vector[index].left : point_vector[
```

```
        index].right, querying, neighbors, extra_neighbors, K, max_idx
        , max_dist, extra_max_dist, axis);
43      if ( dx2 > *max_dist ) return;
44      kquery_recursive(dx > 0 ? point_vector[index].right : point_vector
        [index].left, querying, neighbors, extra_neighbors, K, max_idx
        , max_dist, extra_max_dist, axis);
45  }
```

Una vez solucionado este problema, llegando a la implementación final, se consigue reducir el error en todas los conjuntos de datos a 0, consiguiendo una búsqueda exacta.

6.6. Alto nivel

La parte del algoritmo de alto nivel desarrollada en este trabajo, ha sido la generalización del algoritmo de búsqueda a K dimensiones. Para ello, se ha trabajado sobre la implementación existente en [1] para adaptarla al entorno de bajo nivel y generalizar el algoritmo para una cantidad de dimensiones arbitraria. Se recuerda que la implementación existente sólo cuenta con búsqueda de radio fijo para tres dimensiones. La tarea a realizar por tanto ha constado de:

- Adaptación del algoritmo de búsqueda de radio fijo para K dimensiones.
- Implementación del algoritmo de búsqueda de k vecinos para tres dimensiones.
- Adaptación del algortimo de búsqueda de k vecinos para K dimensiones.

La adaptación del algoritmo de búsqueda de radio fijo, como ya se vió en sec. 4.4.3, se ha realizado de forma recursiva. Siendo el nivel de recursión el número de dimensiones del espacio de trabajo. La adaptación no presenta dificultades más allá de hacer que la ejecución del algoritmo sea recursiva y se comprueba su correcto funcionamiento verificando que arroja los mismos resultados que la implementación original para dimensión tres.

La implementación del algoritmo de búsqueda de k vecinos para tres dimensiones se inspira en el mismo algoritmo de visita a celdas adyacentes empleado en la búsqueda de radio fijo y en el algoritmo de selección de la peor distancia empleado en las búsquedas de k vecinos a en los árboles a bajo nivel. Tras una implementación inicial se encuentran errores.

Cuadro 6.5.: Errores obtenidos con **Muelle** $k = 10$

	Radio fijo		k Vecinos	
	Tamaño distinto	No coincidentes	Tamaño distinto	No coincidentes
Errores	0	0	25	623

Ya que la búsqueda de radio fijo no presenta errores, se deduce que los fallos provienen del algoritmo que gestiona los k vecinos, y que el algoritmo para visitar celdas

adyacentes funciona correctamente. Estudiando los errores obtenidos se deduce que no se visitan todas las celdas adyacentes que contienen potenciales vecinos, si no que el proceso se aborta prematuramente. Para explicar el origen de estos errores se va a presentar la implementación final sin errores[2] y se discutirán los detalles de la primera implementación respecto a esta.

Ejemplo 6.2: Búsqueda de alto nivel para K vecinos

```
1  void tessel_kquery_3( const std::array<double, DIM>& coord, const
       tessel<DIM>& c, std::vector<distance_and_ref>& result, const
       size_t K ) const
2  {
3
4  bool keep_going = true;
5  double worst_dist = INFINITY;
6  long x[DIM], r = 0;
7  tessel<DIM> aux;
8  while ( keep_going )
9  {
10     r++;
11     for ( x[0] = -r; x[0] <= r; ++x[0]  )
12     {
13         for ( x[1] = -r; x[1] <= r; ++x[1] )
14         {
15             for ( x[2] = -r; x[2] <= r; ++x[2] )
16             {
17                 if ( abs(x[0]) >= r || abs(x[1]) >= r || abs(x[2]) >=
                       r )
18                 {
19                     aux.coor[0] = c.coor[0] + x[0];
20                     aux.coor[1] = c.coor[1] + x[1];
21                     aux.coor[2] = c.coor[2] + x[2];
22                     typename HashMap::const_iterator found = domain.
                           find(aux);
23                     if ( found != domain.end() )
24                     {
25                         kquery_result q_result = found->second->kquery
                               (coord, K );
26                         result.insert(result.end(), q_result.begin(),
                               q_result.end());
27                         if ( K <= result.size() )
28                         {
29                             std::sort(result.begin(), result.end());
30                             worst_dist = result[K - 1].first;
31                         }
32                     }
33                 }
34             }
35         }
36     }
```

[2]La implementación final en el momento de la redacción de la memoria ya ha incorporado una serie de optimizaciones que no se reflejan en el código adjuntado y se proporciona una versión antigua de esta para que todo el código presente en la memoria sea consecuente entre sí.

```
37
38      keep_going = false;
39      if ( result.size() < K )
40      {
41          keep_going = true;
42      }
43      else // Copiado de Santiago:
44      {
45          for ( int i = 0; i < DIM; ++i )
46          {
47              keep_going |= coord[i] + sqrt(worst_dist) > (c.coor[i] + r
                    ) * length;
48              keep_going |= coord[i] - sqrt(worst_dist) < (c.coor[i] - r
                     + 1) * length;
49          }
50      }
51  }
52
53  }
```

Se observa que la búsquedas en las celdas adyacentes se seguirá realizando mientras que la variable `keep_going` sea cierta. Esta variable será cierta mientras las celdas a visitar en la siguiente iteración puedan potencialmente mejorar el resultado ya obtenido, de lo contrario se da por terminada por la búsqueda y se retorna con el resultado obtenido.

En la implementación inicial de funcionamiento erróneo se interrumpía el proceso cuando se llegaban a los k vecinos después de una iteración. Esto no siempre es cierto ya que es posible que el punto *querying* se encuentre en una de las fronteras de la celda, encontrándose más cerca de aquellos puntos *searching* a dos celdas de distancia que a los de la celda adyacente en la frontera contraria.

Tras la resolución de este error en el algoritmo, su adaptación a K dimensiones se realiza nuevamente de forma recursiva, verificándose el correcto funcionamiento de esta.

7. *Benchmarking* o comparativas

En este capítulo se medirán los rendimientos de la solución desarrollada frente a sí misma e implementaciones ya existentes.

7.1. Condiciones de *benchmarking*

7.1.1. Datos

Los conjuntos de datos empleados para el *benchmarking* son los descritos en Capítulo 6. A saber:

- Muestra **Iris** con 98304 puntos *searching* y 5265 puntos *querying*.
- Muestra **Muelle** con 51200 puntos *searching* y 10025 puntos *querying*.
- Muestra **Rueda dentada** con 116528 puntos *searching* y 8896 puntos *querying*.

Cuadro 7.1.: Características **Iris**

	Media	Varianza	Desviación típica
Coordenadas X	0.000000	0.020834	0.144338
Coordenadas Y	0.250000	0.020834	0.144338
Coordenadas Z	2.000000	1.333347	1.154706

Cuadro 7.2.: Características **Muelle**

	Media	Varianza	Desviación típica
Coordenadas X	-0.002970	0.002456	0.049563
Coordenadas Y	-0.002970	0.002456	0.049563
Coordenadas Z	0.127810	0.004660	0.068263

Así como un conjunto de 100000 (cien mil) puntos *searchings* y 10000 (diez mil) puntos *querying* en coma flotante que varían de -10.0 a 10.0. Estos puntos han sido generados aleatoriamente mediante la función `rand()` de la librería `cstdlib` de la librería estándar de C [33].

Cuadro 7.3.: Características **Rueda dentada**

	Media	Varianza	Desviación típica
Coordenadas X	0.501490	0.082228	0.286754
Coordenadas Y	0.499640	0.083448	0.288874
Coordenadas Z	0.224774	0.017844	0.133580

Cuadro 7.4.: Características **Distribución aleatoria**

	Media	Varianza	Desviación típica
Coordenadas X	0.032155	33.285087	5.769323
Coordenadas Y	-0.010747	33.331370	5.773333
Coordenadas Z	-0.004595	33.367454	5.776457

7.1.2. Entorno y método de medida

Todas las pruebas y mediciones de tiempos se han empleado con la librería `std::chrono` de la librería estándar de C++ [34]. Todas las mediciones se han realizando con una resolución de microsegundos. Todas las medidas de tiempo expresadas en este capítulo se expresan en **milisegundos.**

Las características del ordenador empleado para la medición de tiempos son:

- Procesador: Intel(R) Core(TM) i5-7200 CPU @ 2.5GHz
- Memoria RAM: 8058 MB
- Sistema operativo: Windows 10 Home 1703 64 bits

Todas los programas a partir de los cuales se ha medido el tiempo han sido compilados con GCC con las siguientes opciones de compilación:

```
-O3 -DNDEBUG -ftree-vectorizer-verbose=1
```

- `-O3` Es una opción que habilita un conjunto de opciones de compilación que tienen como objetivo mejorar el tiempo de ejecución de un programa, aunque sea a costa del espacio que ocupa el código máquina del programa.
- `-DNDEBUG` Es una opción que habilita la bandera `NDEBUG`. Esta bandera hace que el compilador ignore una serie de funcionalidades extra incluidas en el código durante el proceso de pruebas del mismo para comprobar su correcto funcionamiento. Estas funcionalidades no son necesarias para el usuario final, ni por tanto, para el *benchmarking*.
- `-ftree-vectorizer-verbose=1` Esta opción hace que el compilador indique qué vectores han sido optimizados.

Esto garantiza que los programas alcanzan la máxima velocidad de ejecución posible dentro de las posibilidades del compilador. Como se explicó en sec. 4.5.3, la

opción`-ftree-vectorizer-verbose=1`, permite indicar qué bucles han visto acelerado su funcionamiento, con lo que se pretende mejorar de forma significativa el funcionamiento de los métodos de fuerza bruta para que presenten rendimientos competitivos para pequeñas cantidades de datos.

Para el *benchmarking* de los tipos de celdas, es decir, la comparativa a **bajo nivel**, se trabajará con la **distribución de datos aleatoria**. Se añadirá el conjunto de puntos con el que se trabaja a una única celda, de forma que se tratará la totalidad de la muestra sin que haya influencia alguna de las optimizaciones del algoritmo a alto nivel. Se realizarán todas las operaciones para **10000** ***querying***s. Esto es así porque el tiempo consumido en la realización de una única búsqueda, no viene influido por la cantidad de búsquedas a realizar. Se mide el tiempo empleado en realizar la totalidad de las 10000 búsquedas y no el de la suma de cada una de ellas, ya que `std::chrono` carece de la precisión necesaria. Además al realizarse múltiples búsquedas se espera que el resultado sea más representativo.

Para el *benchmarking* del conjunto de la solución, es decir, la comparativa a **alto nivel**, se trabajará con la distribuciones de **datos reales**. Se añadirá el conjunto de puntos asignándolos en celdas. Todas las celdas serán del mismo tipo, pretendiéndose de esta forma ver en que casos unas presentan mejor rendimiento que otras. Con los resultados obtenidos a partir de estas comparativas se pretende sentar las bases para el desarrollo de un algoritmo que seleccione dinámicamente la celda a usar, maximizando el rendimiento de la solución.

No se empleará paralelización alguna para estas pruebas.

7.2. Punto de partida: *Brute force* vs *Par_ Cell*

Antes de empezar a comparar la optimización que supone el empleo de estructuras de datos tipo árbol, vamos a comprobar la mejora de rendimiento que supone el aprovechamiento de las capacidades de vectorización de los procesadores modernos. Esta comparativa sólo se realizará respecto a la búsqueda de radio fijo, ya que la búsqueda de los k vecinos no es susceptible a ser vectorizada.

7.2.1. Memoria

Se medirá el tiempo consumido en:

- **Creación:** Instanciación del objeto, se mide el tiempo empleado en invocar al constructor de la clase.
- **Adición:** Inserción de los puntos en la celda, se mide el tiempo empleado en insertar los puntos *searching* con los que se va a trabajar.
- **Destrucción**: Liberación del espacio de memoria reservado al crear el objeto, se mide el tiempo empleado en invocar al destructor de la clase.

Cuadro 7.5.: *Benchmarking*. Memoria. *Brute* vs *Par*

	Creación		Adición		Destrucción	
Puntos	*Brute*	*Par*	*Brute*	*Par*	*Brute*	*Par*
100	0.000	0.000	0.013	0.018	0.000	0.001
500	0.000	0.000	0.017	0.023	0.001	0.001
1000	0.001	0.000	0.029	0.029	0.001	0.001
5000	0.000	0.000	0.198	0.309	0.064	0.057
10000	0.000	0.000	0.447	0.673	0.042	0.094
50000	0.000	0.000	1.830	3.949	0.106	0.218
100000	0.000	0.000	4.002	9.130	0.626	0.581

Se puede apreciar en los resultados que la resolución de `std::chrono` no es suficiente como para medir el tiempo empleado en invocar al constructor de la clase, estando este por debajo de los microsegundos. Respecto a la inserción de puntos, se verifica que esta es proporcional a la cantidad de puntos insertados, siendo esta ligeramente superior en el caso de *Par_ cell*. Finalmente, a la hora de liberar memoria mediante la llamada del destructor de la clase, los resultados obtenidos no permiten obtener conclusiones que permitan determinar qué celda presenta un mejor rendimiento que la otra.

7.2.2. Búsqueda de radio fijo

Se medirá el tiempo consumido para diferentes radios de búsqueda. La elección del radio de búsqueda no ha sido completamente arbitraria. Se ha escogido un radio que devolviese una cantidad promedio de vecinos similar a la cantidad de vecinos que se pretenden encontrar posteriormente en la búsqueda de los k vecinos, del orden de decenas. La elección del número de vecinos se ha escogido de forma acorde al orden de magnitud del número de puntos que se pretenden desplazar en cada iteración de las simulaciones de métodos numéricos sin mallado.

Cuadro 7.6.: *Benchmarking*. Radio fijo. *Brute* vs *Par*

	Radio = 0.0010		Radio = 0.0015		Radio = 0.0020	
Puntos	*Brute*	*Par*	*Brute*	*Par*	*Brute*	*Par*
100	1.818	5.100	3.625	10.188	5.442	14.560
500	8.829	21.877	17.678	43.020	26.487	64.184
1000	28.866	49.280	57.377	98.685	89.803	147.162
5000	88.487	258.506	201.083	520.448	297.671	787.177
10000	214.161	548.665	425.352	1098.196	601.610	1642.719
50000	906.593	2871.791	1825.140	5751.373	2735.713	8619.524
100000	1950.163	7906.800	3896.896	18311.050	5819.188	31250.023

	Radio = 0.0025		Radio = 0.0030	
Puntos	*Brute*	*Par*	*Brute*	*Par*
100	7.256	18.915	9.108	23.283
500	35.280	84.858	44.086	106.034
1000	130.307	195.533	157.180	243.895
5000	399.700	1047.605	489.901	1306.888
10000	797.816	2187.334	975.667	2732.790
50000	3644.213	11510.782	4560.782	14359.579
100000	8270.974	42456.871	10383.912	52344.03

En contra de lo esperado, no sólo no se mejoran los tiempos de búsqueda empleando las celdas orientadas a ser vectorizadas, si no que se empeoran los resultados de forma significativa. Se comprueba en primer lugar que se ha producido vectorización de bucles en *Par_ cell*. Gracias a la opción `-ftree-vectorizer-verbose=1`, se puede comprobar que se han vectorizado varios bucles en esta celda.

Cuadro 7.7.: Mensajes de vectorización

```
../r2bt/cells/par_cell.hpp:55: note: LOOP VECTORIZED

../r2bt/cells/par_cell.hpp:60: note: LOOP VECTORIZED
```

Cabe entonces preguntarse por qué, a pesar de haber vectorizado el código, no se mejoran los resultados. Se plantea como hipótesis que el planteamiento del código adaptado para ser susceptible a ser vectorizado es tal que debido a su ineficiencia, aunque sea vectorizado, no va a arrojar mejores resultados que el código de fuerza bruta original, ya que otras optimizaciones de `-O3` mejoran más el rendimiento de fuerza bruta.

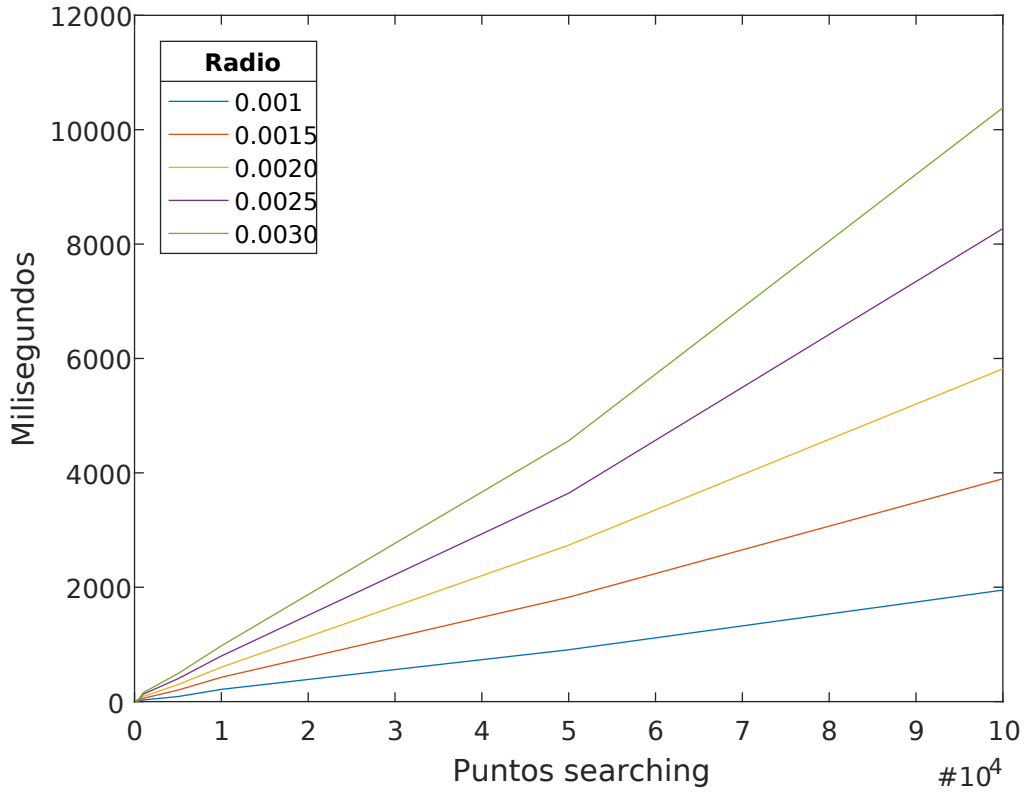


Figura 7.1.: *Benchmarking*. Radio fijo. *Brute*

Como resultados generales, se puede apreciar como el tiempo de búsqueda crece linealmente con la cantidad de *searchings*, y exponencialmente para radios de búsqueda grandes. Esto se explica ya que a mayor número de *searchings* mayor número de distancias se deben calcular ya que este cálculo es *O(n)*, con *n* el número de searchings. También, cuanto mayor sea el radio, mayor número de puntos se almacenarán como resultado. Esta acción de almacenamiento no es despreciable e influye de forma significativa en el tiempo empleado en las búsquedas.

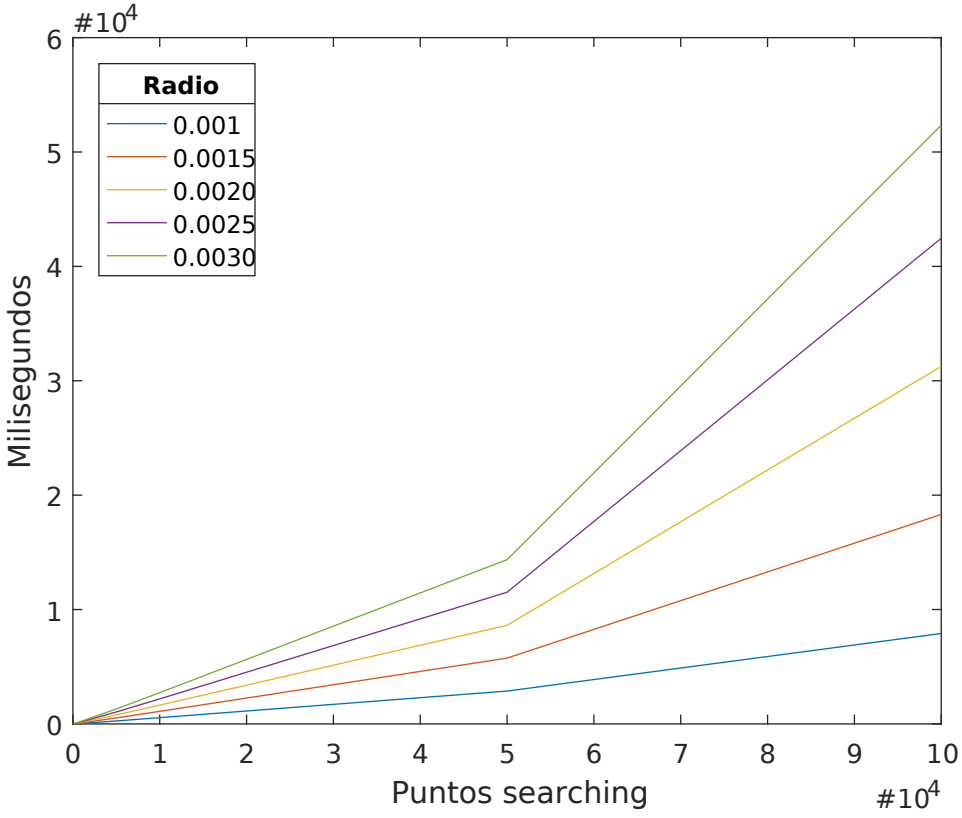


Figura 7.2.: *Benchmarking.* Radio fijo. *Par*

7.2.3. Búsqueda de *k* vecinos

Se medirá el tiempo consumido en búsquedas con diferentes números de vecinos. Como ya se ha dicho, la elección del número de vecinos se ha escogido de forma acorde al orden de magnitud del número de puntos que se pretenden desplazar en cada iteración de las simulaciones de métodos numéricos sin mallado.

De los resultados obtenidos se puede deducir claramente que el tiempo de búsqueda es claramente dependiente y proporcional al número de puntos *searching*. Además, se puede ver que la cantidad de vecinos a encontrar no parece influir en el tiempo de búsqueda, ya que por este método se computan las distancias a cada *querying* de todos los punto del conjunto de *searching*, independientemente de los vecinos deseados.

Cuadro 7.8.: *Benchmarking.* k vecinos. *Brute*

Puntos	$k = 10$	$k = 20$	$k = 30$	$k = 40$
100	34.572	58.926	52.458	37.360
500	238.001	235.450	243.481	305.961
1000	619.625	523.671	540.742	520.596
5000	3095.783	3129.200	3207.077	3381.283
10000	7026.879	7030.872	6664.297	6697.441
50000	39003.645	39055.149	39135.065	41258.491
100000	83537.971	83132.767	86030.147	83073.734

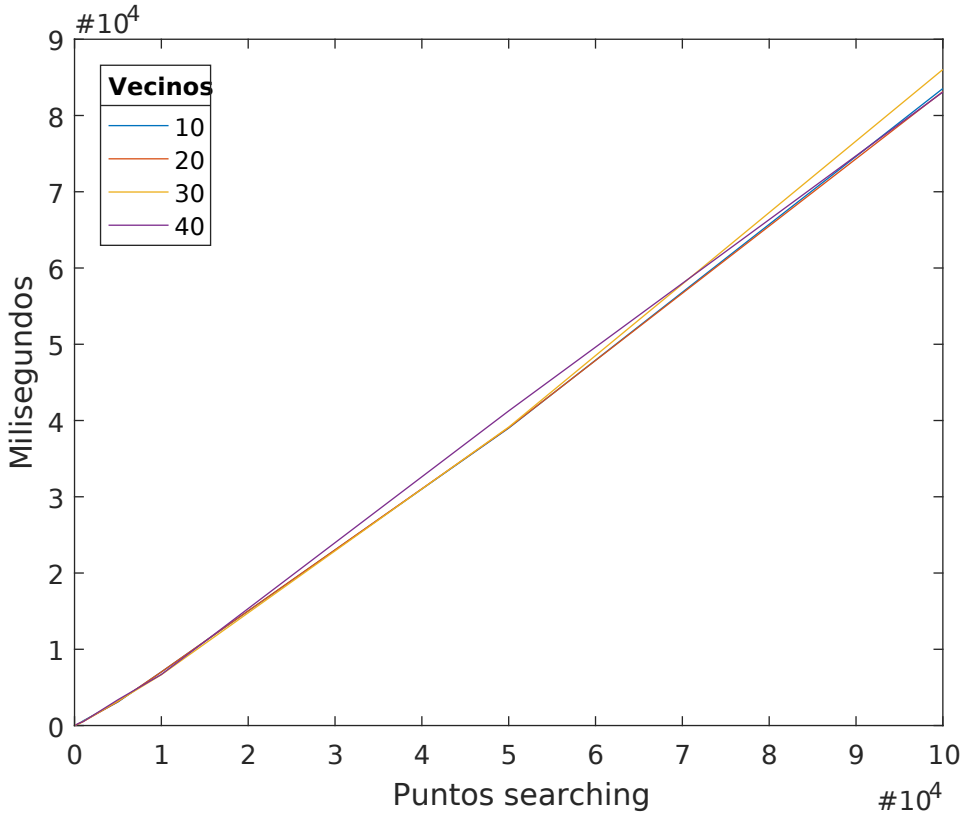


Figura 7.3.: *Benchmarking.* k vecinos. *Brute*

7.3. Uso de árboles

Debido a los resultados obtenidos en la sección anterior, se descartará el uso de *Par_ cell*, y se empleará *Brute_ cell*, es decir, el algoritmo de fuerza bruta como referencia respecto a la que comparar los rendimientos de los árboles.

7.3.1. Memoria

Nuevamente, se medirá el tiempo consumido en:

- **Creación:** Instanciación del objeto, se mide el tiempo empleado en invocar al constructor de la clase.
- **Adición:** Inserción de los puntos en la celda, se mide el tiempo empleado en insertar los puntos *searching* con los que se va a trabajar.
- **Destrucción**: Liberación del espacio de memoria reservado al crear el objeto, se mide el tiempo empleado en invocar al destructor de la clase.

Cuadro 7.9.: *Benchmarking.* Construcción. Árboles

	Creación			Adición			Destrucción		
Puntos	*Brute*	*Utree*	*Stree*	*Brute*	*Utree*	*Stree*	*Brute*	*Utree*	*Stree*
100	0.000	0.001	0.000	0.013	0.117	0.014	0.000	0.003	0.000
500	0.000	0.000	0.000	0.017	0.017	0.018	0.001	0.000	0.000
1000	0.001	0.000	0.000	0.029	0.024	0.023	0.001	0.001	0.000
5000	0.000	0.000	0.000	0.198	0.251	0.255	0.064	0.073	0.046
10000	0.000	0.000	0.000	0.447	0.540	0.550	0.042	0.098	0.092
50000	0.000	0.000	0.000	1.830	2.471	2.591	0.106	0.142	0.142
100000	0.000	0.001	0.000	4.002	6.028	5.240	0.626	0.307	0.276

Antes de comparar los resultados, se verifica que los tiempos de *Utree_ cell y Stree_ cell*, son prácticamente iguales, lo que coincide con lo que previsto, ya que ambos tipos de árboles comparten el mismo tipo de puntos para su funcionamiento interno, y tienen en común la implementación de la inserción de los puntos.

Al igual que en el caso de fuerza bruta, se puede apreciar en los resultados que la resolución de `std::chrono` no es suficiente como para medir el tiempo empleado en invocar al constructor de la clase, estando este por debajo de los microsegundos. Respecto a la inserción de puntos, se verifica que esta es proporcional a la cantidad de puntos insertados, siendo esta ligeramente superior en el caso de las celdas tipo árbol ya que el tamaño de los puntos internos es mayor en este tipo de celdas. Nuevamente, no se pueden extraer conclusiones del análisis del tiempo de destrucción de las celdas.

7.3.2. Construcción de árboles

Se mide el tiempo empleado en la construcción de los árboles. Se mide el tiempo de ejecución de la función `build()` de estas celdas.

Cuadro 7.10.: *Benchmarking.* Construcción de árboles

	Construcción	
Puntos	*Utree*	*Stree*
100	0.030	0.010
500	0.017	0.046
1000	0.209	0.097
5000	1.178	0.563
10000	2.615	1.155
50000	15.248	7.025
100000	33.530	14.403

Como ya se dedujo teóricamente en sec. 4.5.5.1, el tiempo empleado en construir árboles pseudo-equilibrados (*Utree*) es aproximádamente el doble que el que se emplea en la construcción de árboles no-equilibrados (*Stree*), verificando con este resultado experimental la hipótesis teórica. Además, siendo el tiempo de construcción promedio de los árboles $O(nlogn)$, cuya representación gráfica para n grandes es similar a una recta, coincide con la evolución que siguen los tiempos medidos.

7.3.3. Búsqueda de radio fijo

Se medirá el tiempo consumido para diferentes radios de búsqueda, siguiendo el mismo criterio que para el algoritmo de fuerza bruta.

De los resultados obtenidos cabe destacar, en primer lugar, que incluso para cantidades pequeñas de *searchings*, las estructuras tipo árbol presentan rendimientos claramente mejores, siendo las búsquedas de radio fijo en estas, de hasta dos órdenes de magnitud más rápidas. Es importante, sin embargo, tener en cuenta el tiempo de construcción de las mismas, estando este amortizado para búsquedas en más de 5000 puntos *searching*.

Por otro lado, los tiempos de búsqueda crecen asintóticamente con el número de puntos, verificando el desarrollo teórico que afirma que es posible realizar búsquedas en *O(logn)* en árboles binarios de búsqueda. De hecho, se aprecia claramente que influye mucho más el radio de búsqueda, que el tamaño n de la muestra. Esto insinúa que muy probablemente la notación en O esconde factores crecientes con el radio.

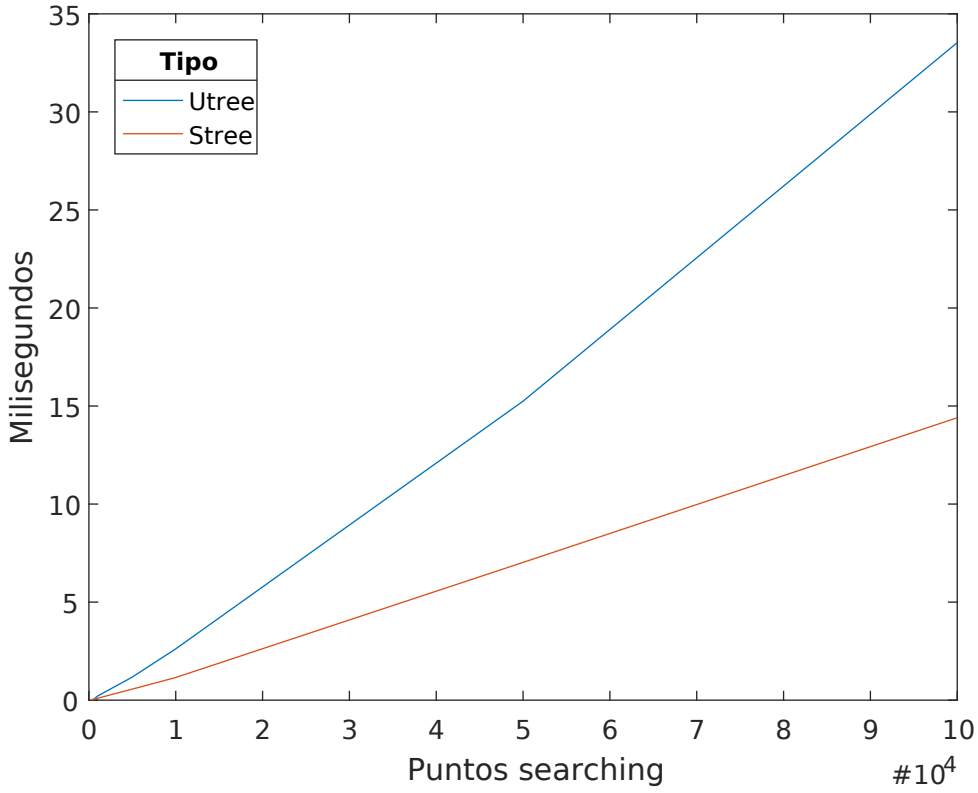


Figura 7.4.: *Benchmarking.* Construcción de árboles

Cuadro 7.11.: *Benchmarking*. Radio fijo. *Brute* vs Árboles

	Radio = 0.10			Radio = 0.15			Radio = 0.20		
Puntos	*Brute*	*Utree*	*Stree*	*Brute*	*Utree*	*Stree*	*Brute*	*Utree*	*Stree*
100	1.805	0.821	0.838	1.809	0.824	0.850	1.806	0.858	0.867
500	8.804	1.223	1.417	8.781	1.395	1.309	8.770	1.413	1.384
1000	17.550	1.734	1.529	17.503	1.642	1.609	17.633	1.829	1.783
5000	88.244	2.275	2.293	88.512	2.608	2.580	90.870	2.909	2.975
10000	180.390	2.781	3.436	176.394	3.232	3.386	177.321	5.080	4.103
50000	885.858	4.771	4.929	930.748	6.562	6.856	900.110	7.858	7.134
100000	1982.684	7.318	6.336	1942.883	9.312	8.902	1963.321	11.903	12.568

	Radio = 0.25			Radio = 0.30		
Puntos	*Brute*	*Utree*	*Stree*	*Brute*	*Utree*	*Stree*
100	1.806	0.880	0.904	1.806	0.916	0.923
500	8.810	1.458	1.482	8.772	1.534	1.576
1000	28.064	1.943	1.921	24.184	2.404	2.052
5000	87.923	3.285	3.287	87.987	3.698	3.798
10000	179.738	4.437	4.520	175.266	4.911	4.821
50000	915.149	9.017	9.210	935.037	10.628	10.457
100000	1961.418	14.536	12.731	2007.887	17.414	16.386

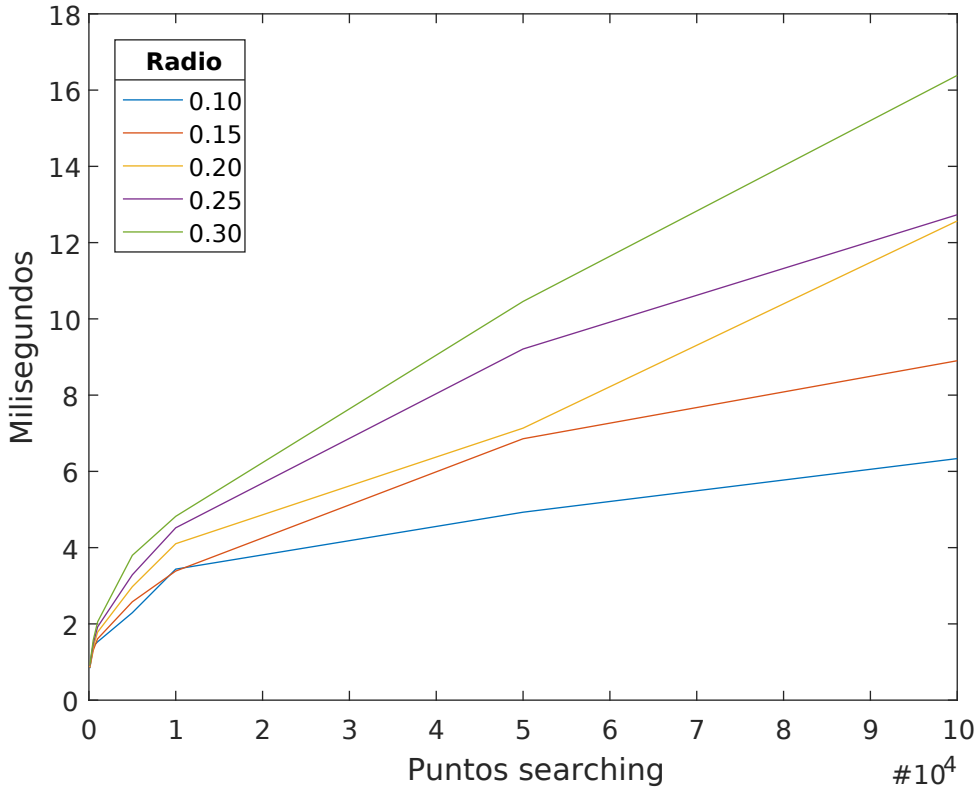


Figura 7.5.: *Benchmarking.* Radio fijo. *Stree*

7.3.4. Búsqueda de *k* vecinos

Cuadro 7.12.: *Benchmarking.* k vecinos. *Brute* vs Árboles

	$k = 10$			$k = 20$		
Puntos	*Brute*	*Utree*	*Stree*	*Brute*	*Utree*	*Stree*
100	40.555	19.918	20.896	34.809	31.033	31.578
500	226.653	40.871	38.428	226.556	56.926	57.534
1000	509.984	49.007	48.431	503.196	73.348	72.280
5000	3037.616	66.200	72.227	3041.082	114.766	122.563
10000	6589.198	86.348	80.911	6563.817	120.214	111.707
50000	38139.383	110.961	100.487	38203.901	219.311	207.505
100000	81444.139	142.968	125.936	81490.701	240.972	196.560

	$k = 30$			$k = 40$		
Puntos	*Brute*	*Utree*	*Stree*	*Brute*	*Utree*	*Stree*
100	34.77	40.895	40.140	35.146	50.289	47.608
500	227.599	76.912	76.423	230.846	95.273	95.419
1000	504.926	95.287	96.440	503.408	118.467	126.625
5000	3037.467	146.673	140.977	3097.765	166.162	172.558
10000	6581.569	153.986	142.063	6544.628	191.470	212.690
50000	38179.157	269.409	274.365	38190.879	252.355	256.495
100000	83654.322	296.444	267.175	87025.298	324.937	284.550

Observando los resultados, llama la atención que para 100 *searchings*, con k=30 y k=40, los resultados obtenidos por fuerza bruta son ligeramente mejores que los obtenidos mediante árboles. Habrá que tener en cuenta por tanto, que para una muestra pequeña n de *searchings* y k del mismo orden de magnitud que n, no vale la pena la construcción de árboles.

Exceptuando el citado caso, los resultados son los esperados, los tiempos de búsqueda por fuerza bruta no dependen del número de vecinos y crecen linealmente con el número de puntos, mientras que los tiempos de búsqueda de los árboles sí dependen del número de vecinos y crecen asintóticamente con el número de puntos. Debido a que estas búsquedas presentan rendimientos muy pobres por fuerza bruta, la construcción de los árboles binarios de búsqueda está amortizada en todos las demás situaciones.

Además, los datos llevan a una conclusión interesante. Las búsquedas en los árboles no-equilibrados (*Stree*) son ligeramente más rápidas que en los pseudo-equilibrados (*Utree*) para mayores cantidades de puntos, cuando en un principio esto debería suceder al contrario.

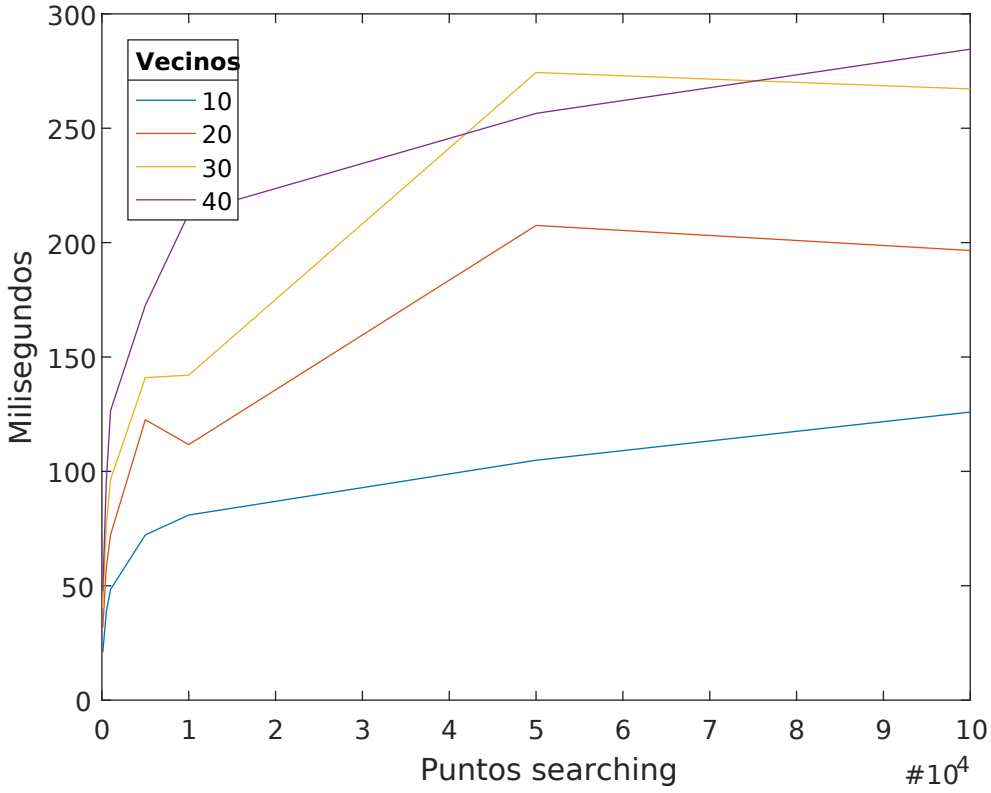


Figura 7.6.: *Benchmarking.* k vecinos. *Stree*

7.4. Algoritmo implementado frente a ANN

Para que las implementaciones trabajen en igualdad de condiciones, se ha **adaptado a bajo nivel** la implementación de ANN, de forma que el algoritmo a alto nivel siga trabajando con la misma interfaz y lo único que cambie sea la implementación del tipo de celda con el que trabaja.

7.4.1. Bajo Nivel

Se compará el rendimiento de una celda que internamente implemente la funcionalidad de ANN. Se comparán los mismos parámetros que en las secciones anteriores.

La medición de tiempos se realiza para una única celda, pero trabajando con la interfaz de alto nivel. Se pretende de esta forma obtener resultados equiparables a los que se obtendrán a alto nivel.

Cuadro 7.13.: *Benchmarking*. Celda ANN con **Iris**

	Stree	*Utree*	ANN
Llamada al constructor	0.019	0.018	0.018
Inserción de puntos	33.316	50.127	54.926
r =0.10	198.923	185.240	390.647
r =0.15	402.112	396.679	1584.599
r =0.20	725.055	731.019	5719.390
r =0.25	1184.059	1128.601	15742.502
r =0.30	1601.961	1633.128	35017.710
k=10	386.059	376.058	34.653
k=20	541.771	396.071	57.998
k=30	475.863	436.509	75.917
k=40	691.088	550.123	79.281
Llamada al destructor	0.009	0.009	7.288

Para la medición de estos tiempos, como se explicará posteriormente, se ha incluido dentro de la inserción de los puntos el tiempo empleado en construir el árbol de cada celda.

Respecto a *Utree* y *Stree*, los resulados arrojados por ambos son prácticamente equivalentes. Debido al tiempo de construcción adicional empleado por *Utree*, es prefirible el empleo de *Stree* si se va a realizar una única búsqueda ya que no siempre el tiempo de construcción del primero está amortizado.

También, se puede ver cómo ANN requiere de un tiempo de construcción y destrucción adicional para preparar las estructuras de datos internas con las que va a trabajar.

Cuadro 7.14.: *Benchmarking.* Celda ANN con **Muelle**

	Stree	*Utree*	ANN
Llamada al constructor	0.018	0.018	0.018
Inserción de puntos	13.920	22.042	22.739
r =0.01	141.833	138.075	332.900
r =0.15	268.333	260.178	928.446
r =0.20	378.082	368.166	2112.678
r =0.25	475.841	457.384	3401.925
r =0.30	546.964	531.503	4850.769
k=10	80.131	78.238	34.809
k=20	126.776	124.432	55.128
k=30	158.218	155.837	72.917
k=40	208.913	190.849	99.950
Llamada al destructor	0.006	0.004	3.580

Por otro lado se aprecia claramente cómo la búsqueda de radio fijo de ANN empeora de forma exponencial con el radio, lo que es un resultado esperado ya que ANN no está optimizada para este tipo de búsquedas, lo que hace inútil esta búsqueda respecto a la implementación desarrollada.

Sin embargo, para la búsqueda de k vecinos, ANN cuenta con unos rendimientos excelentes, pues ANN está diseñada específicamente para este tipo de búsqueda. Este resultado se acentúa especialmente en la distribución **Iris** lo que da a entender que ANN ofrece muy buenos rendimientos en distribuciones uniformes de puntos.

Cuadro 7.15.: *Benchmarking*. Celda ANN con **Rueda Dentada**

	Stree	*Utree*	ANN
Llamada al constructor	0.018	0.018	0.019
Inserción de puntos	35.588	55.115	58.181
r = 0.05	234.165	218.732	913.722
r =0.075	377.053	352.663	2075.773
r =0.10	573.533	543.321	4072.545
r =0.125	866.595	819.151	7948.015
r =0.15	1259.821	1202.639	15436.480
k=10	85.968	82.383	40.060
k=20	123.006	119.761	59.977
k=30	156.265	155.573	77.237
k=40	187.371	184.947	110.678
Llamada al destructor	0.009	0.010	8.385

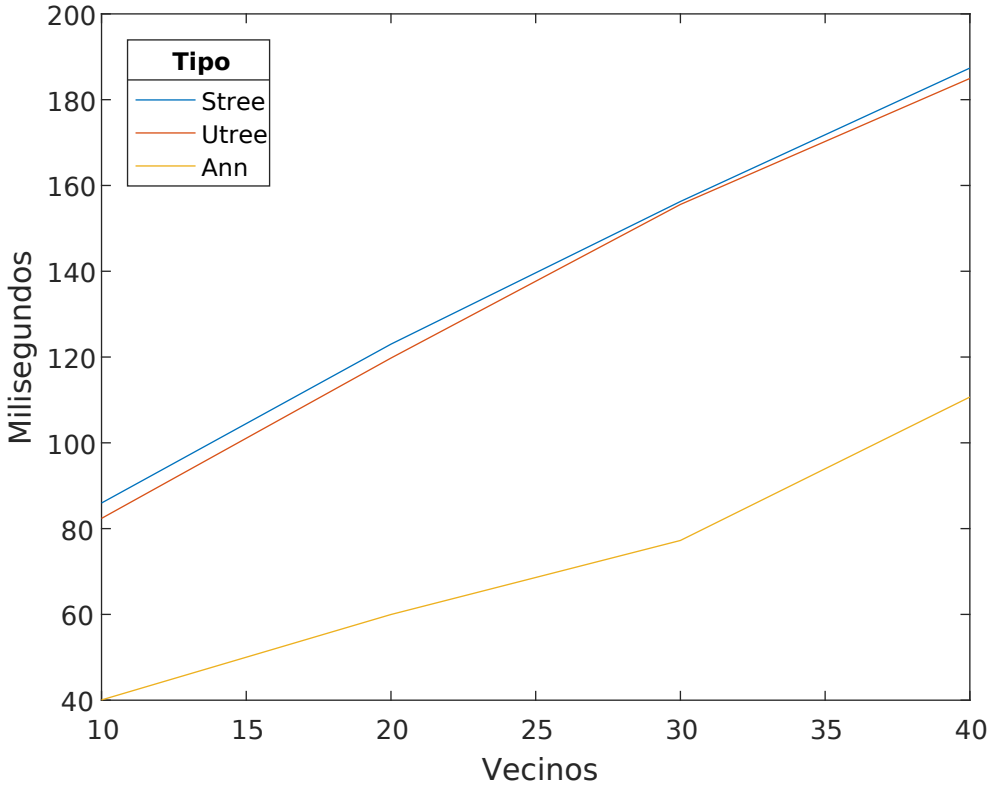


Figura 7.7.: *Benchmarking*. Celda ANN con **Rueda Dentada**

7.4.2. Alto Nivel

Se estudia el rendimiento de cada implementación para las distribuciones de datos Iris, Muelle y Rueda dentada. Para ello, se compara el tiempo empleado en la inserción de los puntos (que incluye la adición de los mismos y su procesamiento en estructuras de datos), el tiempo empleado en realizar búsquedas de radio fijo y de k vecinos.

El parámetro h indica el tamaño de la celda. Este número será redondeado por el algoritmo de alto nivel a la potencia de 2 más cercana.

El criterio a la hora de dimensionar el tamaño de los radios de búsqueda ha sido el mismo que en la sección anterior, pretendiéndose obtener en promedio resultados del mismo orden de magnitud que para la búsqueda de k vecinos.

7.4.2.1. Inserción

Por tiempo de inserción se entiende la suma del tiempo consumido por las siguientes acciones:

- Cálculo previo de número de celdas a construir.
- Inserción de los puntos en las celdas.
- Construcción de las estructuras de datos empleadas por cada celda.

Cuadro 7.16.: *Benchmarking.* Inserción. **Iris**

	$h = 0.1$	$h = 0.25$	$h = 0.5$	$h = 1.0$
STREE	28.759	29.569	30.428	30.875
UTREE	34.515	39.272	40.945	42.862
ANN	48.794	48.807	51.010	50.786
	512 celdas	64 celdas	16 celdas	8 celdas

Cuadro 7.17.: *Benchmarking.* Inserción. **Muelle**

	$h = 0.1$	$h = 0.25$	$h = 0.5$	$h = 1.0$
STREE	34.813	18.871	18.537	15.600
UTREE	21.629	21.428	21.512	21.524
ANN	23.993	23.699	23.726	22.832
	9 celdas	5 celdas	4 celdas	4 celdas

Se puede apreciar como los tiempos de inserción varián dependiendo de la celda a bajo nivel empleada. Acorde con los tiempos de construcción de los árboles, el tiempo empleado en *Utree* es mayor que en *Stree*.

Cuadro 7.18.: *Benchmarking.* Inserción. **Rueda dentada**

	$h = 0.1$	$h = 0.25$	$h = 0.5$	$h = 1.0$
STREE	36.373	33.373	33.606	35.084
UTREE	41.325	53.826	51.999	54.279
ANN	52.901	54.586	55.567	59.764
	227 celdas	32 celdas	4 celdas	1 celdas

En línea con los resultados obtenidos para una única celda, el tiempo de inserción, y por tanto de construcción de los árboles, es mayor que el del resto de las implementaciones.

También, cabe destacar que el tiempo de inserción en general parece no tener relación aparente con el tamaño de las celdas salvo para *Utree*, aunque esto probablemente sea debido a que el tiempo de construcción del árbol aumente cuanto mayor sea el número de puntos.

7.4.2.2. Búsqueda de radio fijo

Cuadro 7.19.: *Benchmarking.* Radio fijo. **Iris**

	$r = 0.10$	$r = 0.15$	$r = 0.20$	$r = 0.25$	$r = 0.30$
$h = 0.1$ | 512 celdas					
STREE	178.254	499.882	860.148	1417.351	2231.812
UTREE	178.171	470.640	788.024	1376.130	2037.106
ANN	338.452	916.627	1811.697	3173.151	4917.614
$h = 0.25$ | 64 celdas					
STREE	195.357	408.949	696.156	1204.721	1699.837
UTREE	196.441	384.131	688.783	1113.597	1585.492
ANN	366.724	1079.788	2382.400	4505.613	8803.482
$h = 0.5$ | 16 celdas					
STREE	200.113	399.102	847.857	1317.687	1714.408
UTREE	196.847	404.906	783.177	1117.014	1500.154
ANN	487.228	1497.531	3815.920	9010.827	14682.692
$h = 1.0$ | 8 celdas					
STREE	253.788	487.322	811.916	1189.822	1634.444
UTREE	212.209	433.532	790.925	1173.586	1619.187
ANN	371.263	1337.257	4015.865	9993.530	18988.029

Utree presenta, por lo general, rendimientos ligreramente mejores aunque no en todos los casos amortizan el tiempo invertido en su construcción.

Por otro lado, ANN, presenta en general un rendimiento muy pobre para la búsqueda de radio fijo, acorde a los resultados obtenidos para una celda, aunque cabe decir que

Cuadro 7.20.: *Benchmarking.* Radio fijo. **Muelle**

	$r = 0.01$	$r = 0.015$	$r = 0.020$	$r = 0.025$	$r = 0.03$
$h = 0.1$ | 9 celdas					
STREE	193.392	346.773	495.741	589.243	697.753
UTREE	193.695	332.503	475.756	562.976	666.144
ANN	386.235	1078.061	2260.161	3171.816	4326.030
$h = 0.25$ | 5 celdas					
STREE	190.623	353.296	504.852	619.726	737.369
UTREE	188.303	350.631	528.932	617.638	742.414
ANN	373.972	1010.044	2010.426	3172.270	4435.318
$h = 0.5$ | 4 celdas					
STREE	190.665	353.130	507.742	633.290	739.005
UTREE	189.405	353.558	513.322	635.375	746.812
ANN	356.721	965.596	1963.803	3155.265	4439.526
$h = 1.0$ | 4 celdas					
STREE	191.091	354.004	508.516	632.439	742.003
UTREE	188.739	352.268	513.296	636.220	748.106
ANN	362.667	973.131	2018.632	3168.839	4411.383

al integrarse en el algoritmo de alto nivel, los casos peores para radios de búsqueda muy grandes se ven atenuados, ofreciendo unos tiempos de búsqueda que ya no crecen exponencialmente con el número de radio.

Respecto a la mejora del rendimiento de los árboles desarrollados en múltiples celdas respecto del rendimiento proporcionado por una única celda, se concluye que no se puede generalizar el caso que proporciona el mejor rendimiento. Cabe observar que para las dos últimas distribuciones el mejor rendimiento se obtiene para cuanto menor es el número de celdas empleado, y menor es el radio de búsqueda, por lo que será necesario paralelizar para obtener un mejor rendimiento.

Sin embargo, para la primera distribución se observa cómo los mejores resultados se obtienen para el mayor número de celdas si se trabaja con radios pequeños, mientras que para radios más grandes es preferible trabajar con un menor número de celdas.

Aunque este comportamiento sea difícil de generalizar, sí es posible determinar que el tamaño de celda es un factor influyente, ya que cuanto menor sea el tamaño de celda respecto al radio de búsqueda, mayor número de celdas se tendrán que visitar. Habrá que tratar de buscar el equilibrio entre el tiempo empleado en la búsqueda dentro de estas celdas y la cantidad de celdas visitadas. Aunque cuanto más pequeñas sean las celdas menos puntos habrá en ellas, y por lo tanto más rápidas las búsquedas dentro de estas, mayor es el número de celdas a visitar. El rendimiento se maximizaría para un tamaño de celda en el que el tamaño de las mismas respecto a un radio de búsqueda permita cubrir el mayor número de puntos con el menor número de celdas posible. La división en celdas ofrece por tanto mejores rendimientos cuanto

Cuadro 7.21.: *Benchmarking.* Radio fijo. **Rueda dentada**

	$r = 0.05$	$r = 0.075$	$r = 0.10$	$r = 0.125$	$r = 0.15$
$h = 0.1$ | 227 celdas					
STREE	331.573	467.817	691.842	1290.603	1756.166
UTREE	292.271	436.608	642.541	1201.917	1697.117
ANN	887.682	1488.577	2332.956	3755.585	5047.959
$h = 0.25$ | 32 celdas					
STREE	304.335	470.253	745.670	1002.298	1441.625
UTREE	300.471	464.207	692.110	993.008	1430.335
ANN	912.970	1767.376	3133.639	5332.282	8411.882
$h = 0.5$ | 4 celdas					
STREE	286.817	490.241	715.805	1026.592	1529.869
UTREE	281.894	449.390	698.300	1019.010	1462.523
ANN	964.883	2047.174	3829.141	7305.367	13610.383
$h = 1.0$ | 1 celda					
STREE	277.932	464.595	702.898	1010.121	1465.043
UTREE	256.593	413.358	646.962	945.158	1380.479
ANN	943.607	2087.391	4089.876	8110.278	15831.247

más uniforme sea la distribución y más pequeño se el radio.

7.4.2.3. Búsqueda de *k* vecinos

Llama la atención, cómo el prometedor resultado ofrecido por ANN en su comparativa a bajo nivel se ve degradado cuando se integra su uso en la aplicación desarrollada a alto nivel cuando hay un gran número de celdas. Sin embargo, a medida que se limita el número de celdas empleadas, ANN mejora sustancialmente sus resultados, mejorando significativamente a los de la la implementación propia.

Por otro lado, en este caso, *Utree* y *Stree* son prácticamente equivalentes, y el tiempo adicional de construcción del primero no justifica su uso si sólo se va a realizar una única búsqueda con los *queryings*. Se aprecia nuevamente cómo los tiempos obtenidos varían en función de la distribución, del número de vecinos que se quieren obtener y del tamaño de las celdas.

Para determinar como el tamaño de la celda influye en el tiempo consumido para la búsqueda de k vecinos, se debe seguir un razonamiento similar al del apartado anterior. Se debe buscar un tamaño de celda que en promedio contenga alrededor de k vecinos, de forma que como mucho sea necesario buscar en las celdas adyacentes a la misma, acelerando el tiempo de búsqueda.

Respecto al rendimiento para una única celda, el algoritmo a alto nivel empeora el resultado. Habría que paralelizar para mejorar el rendimiento, ya no sólo para

Cuadro 7.22.: *Benchmarking.* k vecinos. **Iris**

	$k = 10$	$k = 20$	$k = 30$	$k = 40$
$h = 0.1$ | 512 celdas				
STREE	736.846	976.409	1221.503	1496.043
UTREE	662.987	918.043	1183.310	1438.995
ANN	13417.775	27943.786	44566.049	59807.198
$h = 0.25$ | 64 celdas				
STREE	140.035	209.829	255.964	304.141
UTREE	139.479	201.112	251.399	298.116
ANN	517.963	939.705	1071.479	1487.811
$h = 0.5$ | 16 celdas				
STREE	187.945	290.211	289.250	357.510
UTREE	179.080	241.177	274.817	332.203
ANN	158.524	274.081	356.659	460.021
$h = 1.0$ | 8 celdas				
STREE	226.396	281.234	310.063	368.095
UTREE	236.214	307.902	332.753	394.557
ANN	137.613	257.404	333.524	455.589

alcanzar los rendimientos conseguidos por una única celda de *Stree* o *Utree*, si no para potencialmente conseguir tiempos por debajo de una única celda de ANN, la cual no es susceptible de ser paralelizada.

Cuadro 7.23.: *Benchmarking.* k vecinos. **Muelle**

	$k = 10$	$k = 20$	$k = 30$	$k = 40$
$h = 0.1$ | 9 celdas				
STREE	422.415	514.081	586.316	669.139
UTREE	413.692	505.467	573.008	653.745
ANN	463.181	750.893	1036.214	1308.797
$h = 0.25$ | 5 celdas				
STREE	514.566	621.512	702.971	792.042
UTREE	518.420	644.597	709.517	799.079
ANN	231.363	374.754	519.291	643.125
$h = 0.5$ | 4 celdas				
STREE	493.955	598.314	674.272	757.095
UTREE	506.272	614.186	696.060	785.071
ANN	193.091	316.800	434.084	551.792
$h = 1.0$ | 4 celdas				
STREE	497.774	599.129	675.079	784.849
UTREE	514.331	615.920	697.858	810.584
ANN	204.376	315.333	424.117	579.228

Cuadro 7.24.: *Benchmarking.* k vecinos. **Rueda dentada**

	$k = 10$	$k = 20$	$k = 30$	$k = 40$
$h = 0.1$ | 227 celdas				
STREE	368.380	570.697	756.404	1317.782
UTREE	364.902	567.492	746.692	1325.324
ANN	8080.088	16979.364	27034.815	39607.604
$h = 0.25$ | 32 celdas				
STREE	231.222	330.626	410.894	499.692
UTREE	234.683	326.104	404.187	487.390
ANN	626.384	1114.067	1648.887	2343.088
$h = 0.5$ | 4 celdas				
STREE	381.613	468.945	536.108	609.622
UTREE	361.613	454.517	527.611	599.519
ANN	189.915	296.629	394.831	495.513
$h = 1.0$ | 1 celda				
STREE	82.929	123.457	158.914	189.632
UTREE	82.288	120.652	152.694	182.843
ANN	46.509	64.394	80.375	101.550

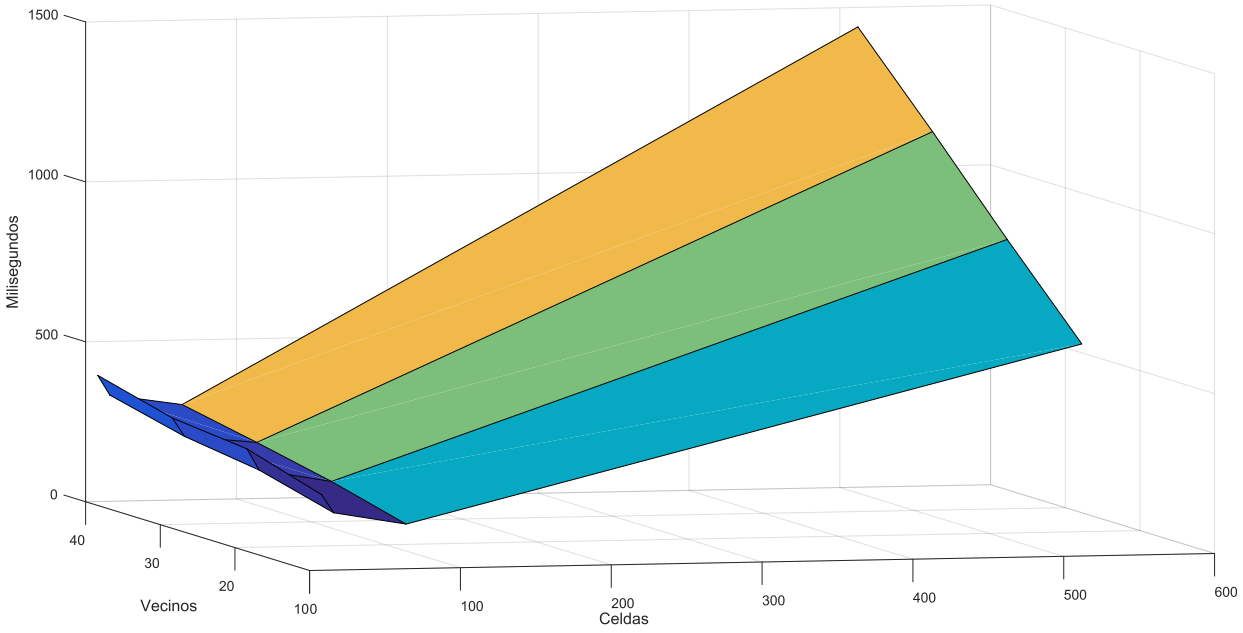


Figura 7.8.: *Benchmarking.* k vecinos. **Iris.** *Utree*

7.5. Paralelización

De los datos obtenidos en las secciones anteriores, se comparán los mejores tiempos para cada distribución y tipo de búsqueda para determinar el factor en el que el rendimiento mejora mediante paralelización. En el momento de la redacción de la memoria, la paralelización sólo está implementada para la búsqueda de radio fijo.

El procesador con el que se realizan las comparativas, Intel(R) Core(TM) i5-7200 CPU @ 2.5GHz, tiene capacidad de hasta 4 hilos de ejecución o *threads*.

7.5.1. Búsqueda de radio fijo

Cuadro 7.25.: Paralelización. Radio fijo**. Iris**

		$r = 0.10$	$r = 0.15$	$r = 0.20$	$r = 0.25$	$r = 0.30$
STREE	Tiempo	178.254	399.102	696.156	1189.822	1634.444
	Celdas	512 (0.1)	16 (0.5)	64 (0.25)	8 (1.0)	8 (1.0)
PARALL.	Tiempo	96.415	171.388	273.481	442.528	638.365
	Celdas	64 (0.25)	16 (0.5)	16 (0.5)	16 (0.5)	16 (0.5)
FACTOR		**1.849**	**2.329**	**2.546**	**2.689**	**2.560**
UTREE	Tiempo	178.171	384.131	688.783	1113.597	1500.154
	Celdas	512 (0.1)	64 (0.25)	64 (0.25)	64 (0.25)	16 (0.5)
PARALL.	Tiempo	92.286	163.898	269.170	424.698	605.245
	Celdas	64 (0.25)	64 (0.25)	16 (0.5)	16 (0.5)	16 (0.5)
FACTOR		**1.931**	**2.344**	**2.559**	**2.622**	**2.479**

Cuadro 7.26.: Paralelización. Radio fijo**. Muelle**

		$r = 0.01$	$r = 0.015$	$r = 0.020$	$r = 0.025$	$r = 0.03$
STREE	Tiempo	190.623	346.773	495.741	589.243	697.75
	Celdas	5 (0.25)	9 (0.1)	9 (0.1)	9 (0.1)	9 (0.1)
PARALL.	Tiempo	143.813	162.748	219.447	249.723	264.811
	Celdas	9 (0.1)	5 (0.25)	4 (1.0)	5 (0.25)	9 (0.1)
FACTOR		**1.325**	**2.131**	**2.259**	**2.359**	**2.635**
UTREE	Tiempo	188.303	332.503	475.756	562.976	666.144
	Celdas	5 (0.25)	9 (0.1)	9 (0.1)	9 (0.1)	9 (0.1)
PARALL.	Tiempo	137.252	128.133	225.574	255.331	276.457
	Celdas	9 (0.1)	9 (0.1)	4 (1.0)	4 (0.5)	9 (0.1)
FACTOR		**1.372**	**2.595**	**2.109**	**2.205**	**2.410**

Se aprecia claramente como el tiempo de búsqueda con paralelización se reduce significativamente. El efecto de paralelizar se hace más significativo cuanto mayor es el radio de búsqueda.

Cuadro 7.27.: Paralelización. Radio fijo**. Rueda Dentada**

		$r = 0.05$	$r = 0.075$	$r = 0.010$	$r = 0.125$	$r = 0.15$
STREE	Tiempo	277.932	464.595	691.842	1002.298	1441.625
	Celdas	1 (1.0)	1 (1.0)	227 (0.1)	32 (0.25)	32 (0.25)
PARALL.	Tiempo	183.661	205.538	256.490	436.817	539.067
	Celdas	32 (0.25)	4 (0.5)	32 (0.25)	4 (0.5)	32 (0.25)
FACTOR		**1.513**	**2.260**	**2.697**	**2.294**	**2.674**
UTREE	Tiempo	256.593	413.358	642.541	945.158	1380.479
	Celdas	1 (1.0)	1 (1.0)	227 (0.1)	1 (1.0)	1 (1.0)
PARALL.	Tiempo	174.240	258.798	287.530	446.056	557.527
	Celdas	32 (0.25)	227 (0.1)	32 (0.25)	4 (0.5)	32 (0.25)
FACTOR		**1.4726**	**1.597**	**2.235**	**2.119**	**2.476**

7.5.2. Búsqueda de *k* vecinos

Aunque en el momento de la redacción de esta memoria no se cuenta con la implementación en paralelización de k vecinos, sí se estudiará el factor mínimo en el que esta futura implementación tendría que mejorar el rendimiento de *Utree* y *Stree* para obtener resultados equivalentes a ANN. Las mediciones de tiempo de ANN reflejadas en este capítulo corresponden a una única celda de ANN pero integrada en la interfaz de alto nivel. Las mediciones de tiempo de *Utree* y *Stree* corresponden a los mejores resultados obtenidos entre los distintos tamaños de celda.

Cuadro 7.28.: Paralelización. k vecinos. **Iris**

	$k = 10$	$k = 20$	$k = 30$	$k = 40$
ANN	34.653	57.998	75.917	79.281
UTREE/STREE	139.479	201.112	251.399	298.116
FACTOR	**4.025**	**3.468**	**3.311**	**3.760**

Cuadro 7.29.: Paralelización. k vecinos. **Muelle**

	$k = 10$	$k = 20$	$k = 30$	$k = 40$
ANN	34.809	55.128	72.917	99.950
UTREE/STREE	78.238	124.432	155.837	190.849
FACTOR	**2.250**	**2.257**	**2.137**	**1.909**

Teniendo en cuanto los factores de mejora de la búsqueda de radio fijo y partiendo de la hipótesis de que se alcanzarán mejoras de rendimiento similares con la búsqueda de k vecinos, se puede afirmar que será posible igualar o mejorar el tiempo de búsqueda de ANN para distribuciones no uniformes, como reflejan los resultados en Muelle y Rueda dentada.

Cuadro 7.30.: Paralelización. k vecinos. **Rueda dentada**

	$k = 10$	$k = 20$	$k = 30$	$k = 40$
ANN	40.060	59.977	77.237	110.678
UTREE/STREE	82.288	119.761	152.694	182.843
FACTOR	**2.054**	**1.997**	**1.977**	**1.652**

Sin embargo, para distribuciones uniformes en el espacio como Iris, es muy probable que los tiempos obtenidos por ANN no puedan ser mejorados a menos que se emplee una cantidad de núcleos fuera de lo común.

8. Conclusión y futuros desarrollos

En este capítulo se presentan las conclusiones del estado actual del proyecto, así como sus futuros desarrollos y líneas de investigación a bajo nivel, a alto nivel, y en su conjunto.

8.1. Conclusión

Del alcance planteado para el trabajo a realizar se han llevado a cabo con éxito las siguientes mejoras propuestas:

- Generalización del algortimo para k dimensiones.
- Implementación de búsqueda de los k vecinos.
- Implementación de árboles de búsqueda binarios para la resolución del problema a bajo nivel.
- Determinación experimental de la influencia de los tipos de distribuciones, el tamaño de las celdas escogidas y el tipo de mecanismo de búsqueda empleado en el rendimiento del algoritmo.

Además, respecto de los resultados obtenidos en Santy[1] para la búsqueda de radio fijo, aunque las mediciones se han realizado en procesadores distintos, su capacidad de procesamiento es similar por lo se puede afirmar que se mejora la escalabilidad del problema como se aprecia en Tab. 8.1.

Cuadro 8.1.: Comparativa frente a Santy

Iris	$r = 0.10$	$r = 0.15$	$r = 0.20$
Santy	84.8	199.9	379.6
R2BT[1]	92.286	163.898	269.170

Gear	$r = 0.05$	$r = 0.10$	$r = 0.15$
Santy	101.3	372.8	1057.2
R2BT	174.240	205.538	539.067

Por otro lado, del Capítulo 7 de comparativas se pueden extraer una serie de conclusiones:

- El rendimiento de la implementación no se ve influido de forma significativa por diferentes tipos de distribuciones de datos reales, no presentando casos límite en los que el comportamiento de la implementación se degrade.
- Salvo para cantidades de puntos *searching* dos o más órdenes de magnitud menores que puntos *querying*, la construcción de una estructura tipo árbol siempre está amortizada.
- Para distribuciones de datos reales y especialmente, si se va realizar una única búsqueda de *queryings* con una distribución diferente en cada iteración, es preferible el empleo de árboles *Stree* respecto a *Utree* ya que los tiempos de construcción del primero son menores y ofrecen prácticamente el mismo rendimiento.
- Es preferible, a alto nivel, que el tamaño de las celdas, o lo que es lo mismo, el número de estas, se escoja de forma que cada celda agrupe en promedio una cantidad de puntos similar al tamaño de los resultados deseados, obteniéndose para este caso los mejores rendimientos.
- Es preferible usar, a bajo nivel, las celdas explicadas en este trabajo para la búsqueda de radio fijo.
- Es preferible usar, a bajo nivel, una única celda de ANN para la búsqueda de los k vecinos.

No se debe olvidar el propósito último del trabajo, que es proporcionar una librería funcional, fácilmente implementable y a la par que otras existentes para su aplicación en métodos numéricos. En ese sentido, se cumplen las siguientes características:

- No existen dependencias externas respecto a librerías no pertenecientes a la librería estándar de C++.
- El usuario no necesita gestionar la reserva y destrucción de memoria dinámica.
- La interfaz permite al usuario trabajar con su propia definición de puntos, y almacenados en cualquier contenedor de la librería estándar de C++.
- Se emplea un C++ moderno, ahorrando al usuario el empleo de sintaxis cercana a C más propensa a fugas de memoria.
- Se ofrecen rendimientos mejores o del mismo orden de magnitud que otras implementaciones existentes.

El desarrollo de aplicaciones orientadas a mejorar el rendimiento de soluciones existentes para problemas reales supone una mejora sustancial en cualquier proceso productivo. En este caso, dada la generalidad de los elementos finitos, cualquier mejora sustancial en las herramientas que permiten su empleo se traduce en la inmediata mejora de los cálculos en el mundo de la ingeniería. Esto incluye, pero no

se limita a numerosas áres del conocimiento como la transferencia de calor, la mecánica de fluidos, la resistencia de materiales o el electromagnetismo. Lo que permite la fabricación de componentes más baratos y eficientes, redundando en un ahorro energético y de materias primas.

8.2. Líneas de investigación

La solución propuesta puede encontrarse en el repositorio público `https://bitbucket.org/stapia/r2bt` . Aunque funcional, la solución aún es un prototipo, pues existen numerosas optimizaciones y líneas de desarrollo para mejorar su rendimiento.

8.2.1. Bajo nivel

Búsqueda del mejor vecino La primera mejora que se puede incluir al trabajo es dotar a las celdas de un método de búsqueda especializado para la búsqueda del mejor vecino, en vez de usar el método de búsqueda de los k vecinos particularizado con k=*1*. Esto es así porque para un único vecino sería posible ahorrar comparaciones y bucles usados en el algoritmo de búsqueda de los k vecinos, siendo esto cierto para todos los tipos de celda.

Inserción / Extracción de puntos en árboles Otra mejora posible es explorar la posibilidad de insertar o extraer puntos de los *kd-Trees.* De esta forma se iría construyendo el árbol a medida que se insertan puntos en la celda, no teniendo que esperar a que estén todos los puntos insertados para poder empezar a construir el árbol. Además, no sería necesario el uso de algoritmos de ordenación para su construcción. Habría que comprobar, sin embargo, que este método ofrece mejoras en el rendimiento respecto al empleo de algoritmos de ordenación. De realizarse la implementación de esta mejora con éxito, se podría hablar de *self-balancing-trees,* es decir, de árboles de búsqueda binarios que se mantienen equilibrados en todo momento.

Optimización de algoritmos de ordenación Las implementaciones de los algoritmos empleados para buscar la mediana (*quickselect*) y para realizar las particiones (*dutch national flag problem*), son implementaciones básicas en el sentido de que se aplican ciegamente a todos los casos y las optimizaciones con las que cuentan no son sofisticadas. Existen en la literatura numerosos algoritmos más óptimos que los empleados aunque de una implementación más laboriosa. Además, se intuye que el algoritmo de partición no es del todo óptimo ya que realiza una ordenación innecesaria. Esto es, no es necesario que todos los puntos con coordenada igual a la mediana se encuentren agrupados en bloque justo antes de la mediana desplazada y

a la derecha del bloque de puntos menores que la mediana. Es por ello que la búsqueda (o desarrollo) de un algoritmo de partición que gestione valores repetidos sin ordenarlos de forma especial podría suponer una mejora significativa en la velocidad de construcción. Destacamos los siguientes algoritmos:

- Algoritmo Floyd–Rivest, para buscar la mediana [35], sustituyendo a *quick_ select*.
- Algoritmo de partición de Hoare, para realizar la partición[2] [36].

Paso de recursividad a iteración Para árboles con un gran cantidad de puntos es posible que haya que sumergirse en muchos niveles de recursividad, llenando la pila de llamadas previas a funciones de niveles superiores. Para evitar esto, quizás sea interesante usar algún método iterativo empleando colas o algún otro tipo de estructura de datos que no altere la funcionalidad del algoritmo pero sí que haga un uso más eficiente de la memoria. Habría que comprobar que tal implementación es posible, y que esta produce mejores rendimientos que la implementación recursiva actual.

8.2.2. Alto nivel

Empleo de contenedores genéricos En las implementaciones existentes de celdas, y a la hora de trasladar resultados entre las distintas partes de la implementación se emplea el uso de vectores. Estos, que ocupan un espacio de memoria contiguo y no son la mejor estructura de datos a la hora de insertar elementos al principio o a mitad de los mismos, podrían ser sustituidos por otro tipo de estructuras de datos como listas o colas que puedan ofrecer mejores rendimientos.

Elección dinámica del algoritmo a bajo nivel En función de los resultados obtenidos en el *benchmarking*, y otros que puedan ser obtenidos a partir de nuevas distribuciones de datos, se puede inducir en qué casos cada tipo de celda de bajo nivel presenta mejores rendimientos. Se podría entonces optimizar el uso de los distintos tipos de celda dependiendo del tipo de distribución con el que se trabaje.

Machine learning En la misma línea que en el apartado anterior, en vez de ponderar y considerar manualmente los resultados obtenidos, podría ser posible elaborar un sistema de retroalimentación en el que se alimente al algoritmo de múltiples distribuciones de datos. De esta forma, se podría diseñar un sistema que solucione el problema de la vecindad de puntos para cada distribución empleando a bajo nivel, diversas combinaciones de subtipos de celdas. Así, el propio sistema podría modificar sus parámetros para favorecer aquellas celdas que presenten rendimientos mejores

[2] La implementación de este algoritmo no es especialmente compleja, pero a diferencia de *dutch national flag problem*, no hay forma trivial de obtener el índice de la mediana desplazada

sobre otras ante distintos tipos de distribuciones. Se tendría entonces un sistema de aprendizaje autónomo que optimizaría el algoritmo tras suficientes análisis de conjuntos de datos.

Distancia definida por el usuario Se podría por otro lado, fomentar la generalidad del algoritmo. En vez de limitarlo al trabajo en el espacio con distintas euclídeas, se podría, emulando las funciones de *templates* de la librería estándar de C++, crear una librería "plantilla" que cada usuario final podría adaptar a sus necesidades.

Gestión de espacios multidimensionales Si bien se ha realizado un estudio teórico de la viabilidad del algoritmo para espacios multidimensionales, afirmando que sería posible adaptar el algoritmo para que en este tipo de espacios se limite a emplear métodos de fuerza bruta, o construir *kd-Trees* sin emplear el algoritmo de alto nivel, la implementación no se llega a realizar. Se podría determinar experimentalmente a partir de qué dimensión las búsquedas se empiezan a degradar, y gestionar las búsquedas por los métodos alternativos mencionados anteriormente. Cabe decir que esta tarea podría ser también asimilable dentro de un sistema de aprendizaje autónomo.

Paralelización empleando GPU Se sabe que los fabricantes de unidades de procesamiento gráfico (GPU en inglés) han avazando en los últimos años en la mejora de librerías que permiten la ejecución de procesos paralela en estas unidades. Este avance es muy prometedor respecto a la posibilidad de paralelizar el proceso no ya en las decenas de núcleos CPU, si no en los cientos de núcleos de la GPU, lo que supondría una gran mejora en el rendimiento.

9. Planificación temporal y presupuesto

En este capítulo se detallará la gestión de los recursos, temporales y materiales empleados en la realización del trabajo de fin de grado.

9.1. Planificación temporal

Se considera como fecha de comienzo del presente trabajo fin de grado, el 27 de Septiembre de 2016, día en el que se realiza la primera reunión sobre el mismo, finalizando el 13 de Julio de 2017, día en el que se termina de redactar esta memoria.

Si bien las fechas de comienzo del calendario Fig. 9.1 son las reales, las fechas de finalización mostradas son aquellas que se habrían podido alcanzar de haberse dedicado la totalidad de un horario laboral normal al desarrollo de este. Debido que su desarrollo se ha compaginado con los estudios, el trabajo se ha realizado en varias etapas, con las correspondientes pausas durante época de exámenes:

- Primer semestre (Septiembre de 2016 a Noviembre de 2016): Introducción al proyecto, familiarización con LYX, los *Kd-Trees*, Git y Ubuntu (Linux).
- Segundo semestre (Febrero de 2017 a Marzo de 2017): Aprendizaje y transición de C a C++ y desarrollo de los algoritmos de búsqueda de radio fijo y k vecinos para cualquier dimensión mediante *Kd-Trees*.
- Segundo semestre (Abril de 2017 a Julio de 2017): Integración de los árboles a bajo nivel y desarrollo de los algoritmos de alto nivel. Comparativas y redacción de la memoria.

Haciendo un total de **316** horas. Pudiéndose apreciar el desglose en tareas de estas etapas en el Apéndice C.

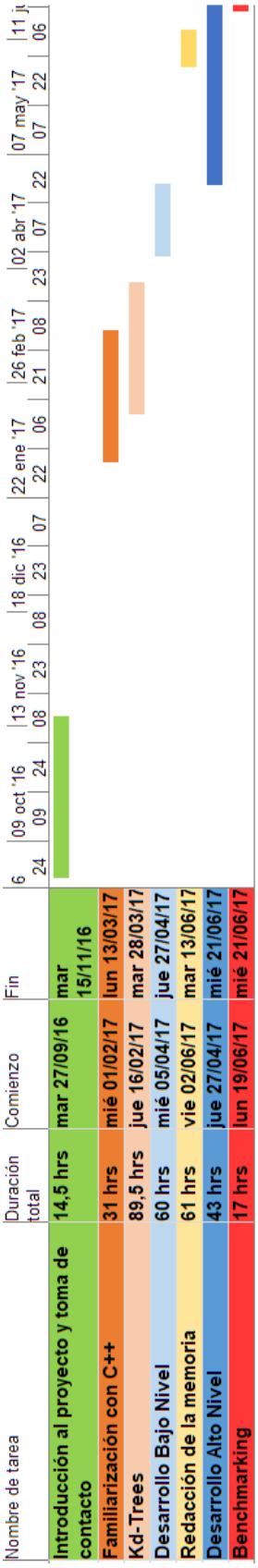


Figura 9.1.: Diagrama de Gantt

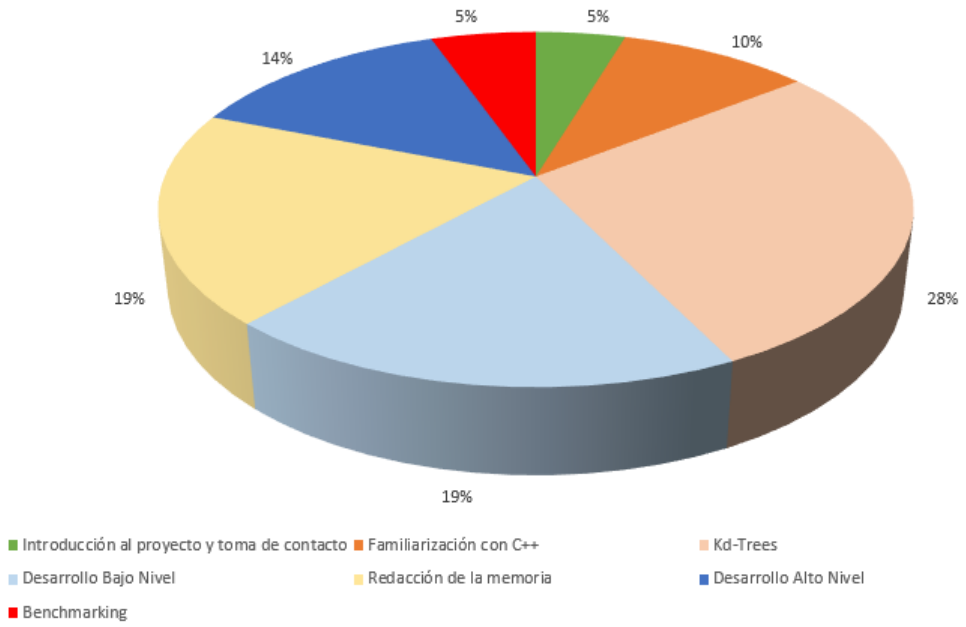


Figura 9.2.: Distribución temporal

9.2. Presupuesto

En este apartado se procede a detallar la valoración económica del trabajo de fin de grado realizado. Al tratarse de un proyecto de naturaleza teórica, el presupuesto de ejecución material coincide con el presupuesto total, siendo las partidas consideradas:

- Los gastos de personal precisado en este proyecto.
- El coste de las licencias del software empleado.
- El coste de los equipos materiales empleados.

Los costes de personal han sido más difíciles de cuantificar con precisión. Para ello, se ha decidido asignar un sueldo en bruto acorde a un ingeniero sénior para el tutor y acorde a un empleado en prácticas para el estudiante, siguiendo los estándares propios del sector. Sólo se han considerado la horas empleadas en tutorías o trabajo en común a la hora contabilizar el tiempo dedicado por el tutor al trabajo. No se han contabilizado las horas empleadas por el mismo en el desarrollo de diferentes partes de la implementación.

Para el cálculo de la amortización se ha supuesto una depreciación lineal a 4 años. Siendo el material de trabajo empleado un ordenador portátil valorado en 500€, y habiendo durado el proyecto 10 meses, resulta de un valor de 125 €.

Ascendiendo el presupuesto general a la cantidad de SEIS MIL OCHOCIENTOS CINCUENTA Y DOS EUROS CON OCHENTA Y DOS CÉNTIMOS.

CONCEPTO	UNIDAD	COSTE UNITARIO	CONSUMO	IMPORTE
PERSONAL				**5.417,00 €**
Tutor	Hora	50 € / hora	32,5	1.625,00 €
Estudiante	Hora	12 € / hora	316	3.792,00 €
SOFTWARE				**1.310,82 €**
Bitbucket uso académico	Licencia anual	8,92 € / Licencia anual	1	8,92 €
Cmake para Linux	Licencia	Sofware libre	2	0,00 €
Windows 10 home	Licencia	107 € / Licencia	1	107,00 €
Linux Ubuntu	Licencia	Sofware libre	1	0,00 €
Codeblocks para Linux	Licencia	Sofware libre	2	0,00 €
Codeblocks para Windows	Licencia	Sofware libre	1	0,00 €
Notepad++	Licencia	Sofware libre	1	0,00 €
Lyx	Licencia	Sofware libre	2	0,00 €
Open Office	Licencia	Sofware libre	1	0,00 €
Matlab Student	Licencia	35€ / Licencia	1	35,00 €
Microsoft Project	Licencia	1159,9 € / Licencia	1	1.159,90 €
OTROS				**125,00 €**
Amortización	No aplica	125,00 €	No aplica	125,00 €
				Total
				6.852,82 €

Figura 9.3.: Presupuesto

A. Estándar IEEE para números en coma flotante

A.1. Introducción

El estándar IEEE 754[37] para números en coma flotante (IEEE standard 754 floating point numbers) es el estándar más comúnmente utilizado hoy en día para la representación de números reales en ordenadores, incluyendo prácticamente todos los PCs basados en Intel, Mac y la gran mayoría de plataformas basadas en UNIX.

A.2. Almacenamiento en memoria

Los números en coma flotante tienen tres componentes básicos, el signo S, el exponente E y la mantisa o fracción F. La Fig. A.1 muestra la distribución en memoria de números con precisión *single* (32-bit) y *double* (64-bit).

S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
S EEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

S= Signo, E=Exponente, F=Fracción

Figura A.1.: Precision *single* y *double*

Por tanto el número de bits que conforman cada componente, así como sus índices en memoria son los indicados en Fig. A.2.

	Signo	Exponente	Fracción
Precision *single*	1 [31]	8 [30-23]	23 [22-00]
Precision *double*	1 [63]	11 [62-52]	52 [51-00]

Figura A.2.: Descomposición de números en coma flotante

Bit de Signo

Este bit impone el signo del número real. Un 0 denota un número positivo, mientras que un 1 denota un número negativo.

Exponente

Este campo debe representar tanto los exponentes positivos como los negativos. Este exponente se representa mediante un número en binario natural al que se le suma un sesgo de 127 en caso de precisión *single*, constando de 8 bits. Por tanto un exponente compuesto por ocho ceros equivaldrá a un 127 almacenado en su espacio de memoria. Análogamente un valor de 200 almacenado en memoria equivaldrá a un exponente que en binario natural representará (200-127) 73. Los exponentes -127 (todos 0s) y 128 (todo 1s) están reservados para números especiales como se verá más adelante sec. A.4.

Mantisa

La mantisa, fracción, o *significand* en inglés, representa la precisión en bits del número. Está compuesta por un bit implícito a la izquierda de la coma y una fracción en bits a la derecha de la coma. Para hallar el valor de la mantisa de un número, debemos considerar su expresión en notación científica. Para maximizar la cantidad de números representables, los números en coma flotante se encuentran almacenados según mantisas normalizadas. Esto quiere decir que la coma se sitúa tras el primer dígito diferente de cero. Sin embargo, en base dos es posible optimizar un poco este método, pues el único dígito distinto de cero es el 1 y no será necesario almacenarlo en memoria. Se podrá entonces asumir un bit implícito igual a 1, y un número en coma flotante de 32-bit tendrá este bit más 23 bits de mantisa explícitos, teniendo en total una mantisa de 24 bits.

A.3. Rangos de los números en coma flotante

Particularizando para los números de precisión *single*, estaríamos tomando un número de 32-bit y reinterpretando sus diferentes campos para cubrir un rango mucho más amplio de representación. Esta representación por tanto estará limitada, recibiendo esta limitación el nombre de precisión. Por ejemplo, un entero de 32-bit en binario natural podrá almacenar números enteros comprendidos entre 0 y 2^{32}, sin embargo será incapaz de representar números con decimales. Por otro lado, un número en coma flotante será incapaz de alcanzar esta resolución con sus 24 bits, pero será capaz de aproximar el valor truncando sus decimales menos significativos y redondeando hacia arriba. Podemos apreciar esta pérdida de precisión en el ejemplo de Fig. A.3, que aunque aproxima el valor, no alcanza una representación exacta.

Por otra parte, al margen de la habilidad de los números en coma flotante para representar números con decimales, estos pueden alcanzar rangos de hasta 2^{127}, mientras que en binario natural sólo llegaríamos hasta 2^{32}. El rango de los números en coma flotante puede dividirse en números normalizados (que conservan la preicisón

Entero de 32-bit	11110000 11001100 10101010 10101111
Coma flotante de 32-bit	1.1110000 11001100 10101011 2^{31}
Valor correspondiente	11110000 11001100 10101011 00000000

Figura A.3.: Resolución en 32-bit de un entero frente a un número en coma flotante

completa de la mantisa) y los números denormalizados (que veremos en sec. A.4), que sólo usan parte de la precisión de la mantisa.

	Denormalizados	Normalizados
Precisión *single*	$\pm\ 2^{-149}$ to $(1-2^{-23})\times 2^{126}$	$\pm\ 2^{-126}$ to $(2-2^{-23})\times 2^{127}$
Precisión *double*	$\pm\ 2^{-1074}$ to $(1-2^{-52})\times 2^{1022}$	$\pm\ 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$

Figura A.4.: Rangos de representación de los números en coma flotante

Como cada número en coma flotante posee, gracias al bit de signo, su correspondiente valor negativo, los rangos que se puede apreciar en Fig. A.4 deberían ser simétricos respecto a cero. Deducimos por tanto que existen rangos de números que un número en coma flotante no es capaz de representar, estos son:

1. Números negativos menores que $-(2-2^{-23}) \times 2^{127}$ (overflow negativo)
2. Número negativos mayores que -2^{-149}(underflow negativo)
3. El cero
4. Números positivos menores que 2^{-149}(underflow positivo)
5. Números positivos mayores que $(2-2^{-23}) \times 2^{127}$ (overflow positivo)

Overflow significa que el valor a representar está por encima del rango de representación. Underflow sin embargo, implica una pérdida de precisión cercana a cero y en principio su error no debe propagarse. El rango efectivo de representación será el representado en Fig. A.5.

	Binario	Decimal
Precisión *single*	$\pm\ (2-2^{-23}) \times 2^{127}$	$\approx \pm\ 10^{38.53}$
Precisión *double*	$\pm\ (2-2^{-52}) \times 2^{1023}$	$\approx \pm\ 10^{308.25}$

Figura A.5.: Rango efectivo de representación de los números en coma flotante

A.4. Valores especiales

El IEEE reserva para exponentes compuestos por todo 0s y todo 1s, unos valores especiales.

Zero

Como ya se ha dicho, el cero no es representable de forma directa. Esto es debido al 1 implícito de la mantisa, sería necesario especificar una mantisa que representase específicamente al cero. En su lugar, el cero es un valor especial que se denota con un exponente con todos sus bits a ceros y una mantisa con todos sus bits a ceros. Cabe destacar que gracias al bit del signo, -0 y +0 se consideran distintos valores, aunque al ser comparados se consideran iguales.

Denormalizados

Si el exponente está compuesto por todo 0s, pero su mantisa es diferente de cero, entonces el valor que representa será un número normalizado, que ahora tiene un bit implícito de cero delante de la coma. Los números denormalizados equivaldrán entonces a:

$$(-1)s\times 0.f\times 2^{-126} \text{ para } \textit{single}$$

$$(-1)s\times 0.f\times 2^{-1022} \text{ para } \textit{double}$$

Donde s es el bit de signo y f es la mantisa. Se podría decir por tanto que el cero es un tipo especial de número denormalizado.

A medida que un número denormalizado se va haciendo cada vez más pequeño, irá perdiendo precisión, ya que los bits del lado izquierdo de la mantisa se convertirán en ceros. Para el valor denormalizado más pequeño (sólo el bit más a la derecha de la mantisa es 1), un número en coma flotante de 32-bit tendrá un único bit de precisión, frente a los 24-bits de los valores normalizados.

Infinity

Los valores $+\infty$ y $-\infty$ se denotan mediante exponente compuestos por todo 1s y mantisa de todo 0s. El bit de signo permite distinguir entre infinitos negativos y positivos. La capacidad de representar infinito mediante un valor específico es útil porque permite continuar con una operación aunque se haya producido un overflow, pues estas operaciones se encuentran definidas por el IEEE.

Not A Number

El valor NaN (en inglés, Not a Number), se usa para representar un valor que no representa un número real. Los NaN se representan por un exponente de todo 1s y una mantisa distinta de 0. Existen dos categorías de NaN:

QNaN (Quiet NaN)

Un QNaN es un NaN con el bit más significativo de la mantisa en 1. Un QNaN puede propagarse a través de casi todas las operaciones aritméticas. Estos valores son generados en operaciones en las que el resultado no está matemáticamente definido.

SNaN (Signalling NaN)

Un SNaN es un NaN con el bit más significativo de la mantisa en 0. Se usa para señalizar una excepción en caso de ser usado en una operación. Un SNaN puede ser útil al ser asignado a variables sin inicializar para detectar un uso prematuro de las mismas.

En resumen, los QNaN denotan operaciones indeterminadas mientras que los SNaN denotan operaciones no válidas.

A.5. Operaciones Especiales

Operaciones con valores especiales están bien definidas en el IEEE. En el más sencillo de los casos, cualquier operación con un NaN generará otro NaN como resultado. El resto de las operaciones definidas por el estándar se pueden encontrar en la Fig. A.6.

Operación	Resultado
$n \div \pm\infty$	0
$\pm\infty \times \pm\infty$	$\pm\infty$
$\pm cte \div \pm 0$	$\pm\infty$
$\pm cte \times \pm\infty$	$\pm\infty$
$\infty + \infty$	$+\infty$
$-\infty - \infty$	$-\infty$
$\infty - \infty$	NaN
$\pm 0 \div \pm 0$	NaN
$\pm\infty \div \pm\infty$	NaN
$\pm\infty \times 0$	NaN

Figura A.6.: Resultados aritméticos especiales

Bit de Signo

Este bit impone el signo del número real. Un 0 denota un número positivo, mientras que un 1 denota un número negativo.

Exponente

Este campo debe representar tanto los exponentes positivos como los negativos. Este exponente se representa mediante un número en binario natural al que se le suma un sesgo de 127 en caso de precisión *single*, constando de 8 bits. Por tanto un exponente compuesto por ocho ceros equivaldrá a un 127 almacenado en su espacio de memoria. Análogamente un valor de 200 almacenado en memoria equivaldrá a un exponente que en binario natural representará (200-127) 73. Los exponentes -127 (todos 0s) y 128 (todo 1s) están reservados para números especiales como se verá más adelante sec. A.7.

Mantisa

La mantisa, fracción, o *significand* en inglés, representa la precisión en bits del número. Está compuesta por un bit implícito a la izquierda de la coma y una fracción en bits a la derecha de la coma. Para hallar el valor de la mantisa de un número, debemos considerar su expresión en notación científica. Para maximizar la cantidad de números representables, los números en coma flotante se encuentran almacenados según mantisas normalizadas. Esto quiere decir que la coma se sitúa tras el primer dígito diferente de cero. Sin embargo, en base dos es posible optimizar un poco este método, pues el único dígito distinto de cero es el 1 y no será necesario almacenarlo en memoria. Se podrá entonces asumir un bit implícito igual a 1, y un número en coma flotante de 32-bit tendrá este bit más 23 bits de mantisa explícitos, teniendo en total una mantisa de 24 bits.

A.6. Rangos de los números en coma flotante

Particularizando para los números de precisión *single*, estaríamos tomando un número de 32-bit y reinterpretando sus diferentes campos para cubrir un rango mucho más amplio de representación. Esta representación por tanto estará limitada, recibiendo esta limitación el nombre de precisión. Por ejemplo, un entero de 32-bit en binario natural podrá almacenar números enteros comprendidos entre 0 y 2^{32}, sin embargo será incapaz de representar números con decimales. Por otro lado, un número en coma flotante será incapaz de alcanzar esta resolución con sus 24 bits, pero será capaz de aproximar el valor truncando sus decimales menos significativos y redondeando hacia arriba. Podemos apreciar esta pérdida de precisión en el ejemplo de Fig. A.7, que aunque aproxima el valor, no alcanza una representación exacta.

Por otra parte, al margen de la habilidad de los números en coma flotante para representar números con decimales, estos pueden alcanzar rangos de hasta 2^{127}, mientras que en binario natural sólo llegaríamos hasta 2^{32}. El rango de los números en coma flotante puede dividirse en números normalizados (que conservan la preicisón

Entero de 32-bit	11110000 11001100 10101010 10101111
Coma flotante de 32-bit	1.1110000 11001100 10101011 2^{31}
Valor correspondiente	11110000 11001100 10101011 00000000

Figura A.7.: Resolución en 32-bit de un entero frente a un número en coma flotante

completa de la mantisa) y los números denormalizados (que veremos en sec. A.7), que sólo usan parte de la precisión de la mantisa.

	Denormalizados	Normalizados
Precisión *single*	$\pm\ 2^{-149}$ a $(1-2^{-23})\times 2^{126}$	$\pm\ 2^{-126}$ a $(2-2^{-23})\times 2^{127}$
Precisión *double*	$\pm\ 2^{-1074}$ a $(1-2^{-52})\times 2^{1022}$	$\pm\ 2^{-1022}$ a $(2-2^{-52})\times 2^{1023}$

Figura A.8.: Rangos de representación de los números en coma flotante

Como cada número en coma flotante posee, gracias al bit de signo, su correspondiente valor negativo, los rangos que se puede apreciar en Fig. A.8 deberían ser simétricos respecto a cero. Deducimos por tanto que existen rangos de números que un número en coma flotante no es capaz de representar, estos son:

1. Números negativos menores que $-(2-2^{-23}) \times 2^{127}$ (overflow negativo)
2. Número negativos mayores que -2^{-149}(underflow negativo)
3. El cero
4. Números positivos menores que 2^{-149}(underflow positivo)
5. Números positivos mayores que $(2-2^{-23}) \times 2^{127}$ (overflow positivo)

Overflow significa que el valor a representar está por encima del rango de representación. Underflow sin embargo, implica una pérdida de precisión cercana a cero y en principio su error no debe propagarse. El rango efectivo de representación será el representado en Fig. A.9.

	Binario	Decimal
Precisión *single*	$\pm\ (2-2^{-23}) \times 2^{127}$	$\approx \pm\ 10^{38.53}$
Precisión *double*	$\pm\ (2-2^{-52}) \times 2^{1023}$	$\approx \pm\ 10^{308.25}$

Figura A.9.: Rango efectivo de representación de los números en coma flotante

A.7. Valores especiales

El IEEE reserva para exponentes compuestos por todo 0s y todo 1s, unos valores especiales.

Zero

Como ya se ha dicho, el cero no es representable de forma directa. Esto es debido al 1 implícito de la mantisa, sería necesario especificar una mantisa que representase específicamente al cero. En su lugar, el cero es un valor especial que se denota con un exponente con todos sus bits a ceros y una mantisa con todos sus bits a ceros. Cabe destacar que gracias al bit del signo, -0 y +0 se consideran distintos valores, aunque al ser comparados se consideran iguales.

Denormalizados

Si el exponente está compuesto por todo 0s, pero su mantisa es diferente de cero, entonces el valor que representa será un número denormalizado, que ahora tiene un bit implícito de cero delante de la coma. Los números denormalizados equivaldrán entonces a:

$$(-1)s\times 0.f\times 2^{-126} \text{ para } \textit{single}$$

$$(-1)s\times 0.f\times 2^{-1022} \text{ para } \textit{double}$$

Donde s es el bit de signo y f es la mantisa. Se podría decir por tanto que el cero es un tipo especial de número denormalizado.

A medida que un número denormalizado se va haciendo cada vez más pequeño, irá perdiendo precisión, ya que los bits del lado izquierdo de la mantisa se convertirán en ceros. Para el valor denormalizado más pequeño (sólo el bit más a la derecha de la mantisa es 1), un número en coma flotante de 32-bit tendrá un único bit de precisión, frente a los 24-bits de los valores normalizados.

Infinity

Los valores $+\infty$ y $-\infty$ se denotan mediante exponente compuestos por todo 1s y mantisa de todo 0s. El bit de signo permite distinguir entre infinitos negativos y positivos. La capacidad de representar infinito mediante un valor específico es útil porque permite continuar con una operación aunque se haya producido un overflow, pues estas operaciones se encuentran definidas por el IEEE.

Not A Number

El valor NaN (en inglés, Not a Number), se usa para representar un valor que no representa un número real. Los NaN se representan por un exponente de todo 1s y una mantisa distinta de 0. Existen dos categorías de NaN:

QNaN (Quiet NaN)

Un QNaN es un NaN con el bit más significativo de la mantisa en 1. Un QNaN puede propagarse a través de casi todas las operaciones aritméticas. Estos valores son generados en operaciones en las que el resultado no está matemáticamente definido.

SNaN (Signalling NaN)

Un SNaN es un NaN con el bit más significativo de la mantisa en 0. Se usa para señalizar una excepción en caso de ser usado en una operación. Un SNaN puede ser útil al ser asignado a variables sin inicializar para detectar un uso prematuro de las mismas.

En resumen, los QNaN denotan operaciones indeterminadas mientras que los SNaN denotan operaciones no válidas.

A.8. Operaciones Especiales

Operaciones con valores especiales están bien definidas en el IEEE. En el más sencillo de los casos, cualquier operación con un NaN generará otro NaN como resultado. El resto de las operaciones definidas por el estándar se pueden encontrar en la Fig. A.10.

Operación	Resultado
$n \div \pm\infty$	0
$\pm\infty \times \pm\infty$	$\pm\infty$
$\pm cte \div \pm 0$	$\pm\infty$
$\pm cte \times \pm\infty$	$\pm\infty$
$\infty + \infty$	$+\infty$
$-\infty - \infty$	$-\infty$
$\infty - \infty$	NaN
$\pm 0 \div \pm 0$	NaN
$\pm\infty \div \pm\infty$	NaN
$\pm\infty \times 0$	NaN

Figura A.10.: Resultados aritméticos especiales

B. *Kd-Trees*

B.1. Definición

Los *kd-Trees* (árboles de dimensión k) son una herramienta matemática que sirve para dividir o particionar el espacio, organizando los puntos que se encuentran en este en una construcción geométrica similar a un árbol, dividiendo inicialmente el espacio en dos con el 'tronco' principal y volviendo a dividir en dos recursivamente los siguientes subespacios con cada rama hasta que cada rama "termina" en un punto del conjunto [38]. Fueron desarrollados por Jean Louis Bentley en 1975 [39].

B.2. Construcción de un *kd-Tree*

En Fig. B.1 podemos ver un *kd-Tree* que almacena varios puntos pertenecientes a un espacio tridimensional.

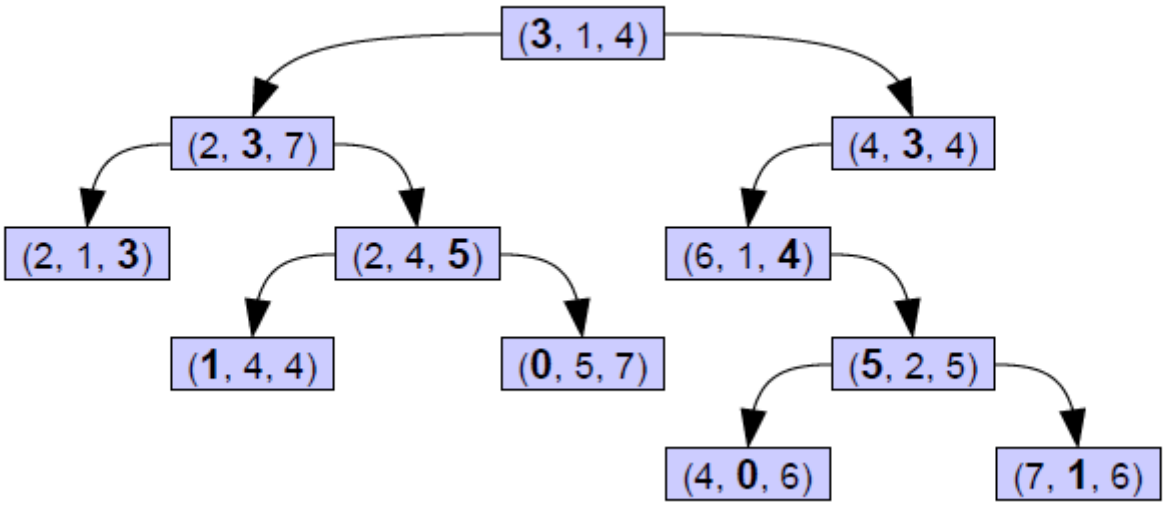


Figura B.1.: *Kd-tree.* [31]

En cada nivel del *kd-Tree* se ha destacado en negrita uno de los componentes de cada punto. Si se numerasen las componentes de cada punto, se ve como el destacado en

negrita es aquel que coincide con el nivel del árbol en el que se encuentra, volviendo a empezar una vez alcanzado el tercer nivel. Esto es así porque cada componente sirve para que el árbol realice una búsqueda binaria. Por ejemplo, el primer componente de cada punto en la rama izquierda es menor que el primer componente del punto raíz. Así, considerando el punto (2, 3, 7), se ve como todos los puntos en su rama izquierda tienen su segunda componente menor que este punto, y los puntos en su rama derecha tienen una segunda componente mayor o igual que este punto. Este patrón se repite en la totalidad del árbol.

B.3. Búsqueda en un *kd-Tree*

Por la forma en la que los *kd-Trees* almacenan los datos, se pueden realizar búsquedas de forma eficiente para saber si un punto está almacenado en el árbol. Dado un punto Q, se empieza en la raíz del arból. Si la raíz es Q, se finaliza la búsqueda. Si la primer componente de Q es menor que la primera componente de la raíz, se seguirá la búsqueda por la rama izquierda, comparando esta vez la segunda componente de Q. En caso contrario, la primera componente de Q, será mayor que la primera componente de la raíz y la búsqueda se seguirá por la rama derecha, comparando esta vez la segunda componente de Q. Si se continua el proceso, incrementando la componente a considerar en cada paso, se encontrará eventualment el punto buscado, o se llegará a las *hojas* del árbol.

C. Diagrama de Gantt

Se expone en este apéndice el diagrama de Gantt detallado de cada etapa del proyecto.

Introducción al proyecto y toma de contacto	14,5 hrs	27/09/16	15/11/16
Codificación coma flotante y operadores de bit	2,5 hrs	27/09/16	27/09/16
LYX	3 hrs	27/09/16	27/09/16
Kd-Trees	2,5 hrs	13/10/16	13/10/16
Git	4 hrs	15/11/16	15/11/16
Ubuntu	2,5 hrs	15/11/16	15/11/16

Figura C.1.: Introducción al proyecto y toma de contacto

Kd-Trees	89,5 hrs	16/02/17	28/03/17
Estudio de implementaciones existentes	30,5 hrs	16/02/17	21/02/17
Adaptación de Rosetta a radio fijo y k vecinos	16 hrs	10/03/17	13/03/17
Algoritmos de ordenación	12 hrs	14/03/17	15/03/17
Utree	10 hrs	15/03/17	16/03/17
Stree	8 hrs	16/03/17	17/03/17
Libxml	4 hrs	14/03/17	14/03/17
Debugging	9 hrs	27/03/17	28/03/17

Figura C.2.: Kd-Trees

En el desarrollo de bajo nivel se produce un intento de desarrollar un nuevo tipo de árbol. Este árbol, inspirado en los árboles implícitos, pretende que cada nodo, en vez de ser un punto, sea una división del plano. No se llega a completar debido a que no se consigue hacer que funcione correctamente para algunas distribuciones de datos.

Desarrollo Bajo Nivel	60 hrs	05/04/17	27/04/17
Fuerza Bruta	2 hrs	05/04/17	05/04/17
Utree	4 hrs	05/04/17	05/04/17
Stree	4 hrs	05/04/17	05/04/17
Optimizaciones del compilador (Par)	4 hrs	05/04/17	05/04/17
Debugging	28 hrs	05/04/17	10/04/17
Intento de ltree	16 hrs	18/04/17	19/04/17
Mejora de bit lattice	2 hrs	27/04/17	27/04/17

Figura C.3.: Desarrollo bajo nivel

Redacción de la memoria	61 hrs	02/06/17	13/06/17

Figura C.4.: Redacción de la memoria

Desarrollo Alto Nivel	43 hrs	27/04/17	21/06/17
Tessels	2 hrs	27/04/17	27/04/17
Preparación de benchmarking	7 hrs	27/04/17	27/04/17
Gestión de compilación y tests con Cmake	3 hrs	05/05/17	05/05/17
Clean-up de código	1,5 hrs	19/06/17	19/06/17
Integración de ANN en celda	5,5 hrs	19/06/17	19/06/17
Reimplementación kquery	6 hrs	19/06/17	19/06/17
Diseño de interfaz y esquema de herencia	6 hrs	19/06/17	20/06/17
Debugging	12 hrs	19/06/17	21/06/17

Figura C.5.: Desarrollo alto nivel

Benchmarking	17 hrs	19/06/17	21/06/17

Figura C.6.: Benchmarking

Bibliografía

[1] S. Tapia, I. Romero, and A. Beltran, "Combine algorithm for neighborhood," 2016. [Online]. Available: https://bitbucket.org/stapia/santy

[2] D. M. Mount and S. Arya, "Ann: A library for approximate nearest neighbor searching." [Online]. Available: https://www.cs.umd.edu/~mount/ANN/

[3] t. f. e. Wikipedia, "Nearest neighbor search," *Wikipedia, the free encyclopedia.* [Online]. Available: https://en.wikipedia.org/wiki/Nearest_neighbor_search

[4] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, "Advances in knowledge discovery and data mining," *AAAI Press/Mit Press*, 1996.

[5] T. M. Cover and P. E. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inform. Theory 13, 57-67*, 1967.

[6] S. Cost and S. Salzberg, "A weighted nearest neighbor algorithm for learning with symbolic features." *Machine Learning 10, 57-78*, 1993.

[7] A. Gersho and R. M. Gray, "Vector quantization and signal compression," *Kluwer Academic, Boston, MA.*, 1991.

[8] Wikipedia, "Tree (data structure)," *Wikipedia, the free encyclopedia.* [Online]. Available: https://en.wikipedia.org/wiki/Tree_(data_structure)

[9] t. f. e. Wikipedia, "Hash table," *Wikipedia, the free encyclopedia.* [Online]. Available: https://en.wikipedia.org/wiki/Hash_table

[10] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," *Journal of the ACM*, 1998. [Online]. Available: www.cse.ust.hk/faculty/arya/pub/JACM.pdf

[11] t. f. e. Wikipedia, "Curse of dimensionality," *Wikipedia, the free encyclopedia.* [Online]. Available: https://en.wikipedia.org/wiki/Curse_of_dimensionality

[12] F. Korn, B.-U. Pagel, and C. Faloutsos, "On the dimensionality curse and the self-similarity blessing," *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 2001.

[13] t. f. e. Wikipedia, "Quicksort," *Wikipedia, the free encyclopedia.* [Online]. Available: https://en.wikipedia.org/wiki/Quicksort

[14] K. L. Clarkson, "A randomized algorithm for closest-point queries." *SIAM Journal on Computing 17*, 1988.

[15] P. K. Agarwal and J. Matousek, "Ray shooting and parametric search," *SIAM J. Comput. 22*, 1993.

[16] C. O. S. Project, "The computational geometry algorithms library." [Online]. Available: http://www.cgal.org/

[17] M. E. department of the ETSI Industriales, "Muesli." [Online]. Available: http://materials.imdea.org/research/simulation-tools/muesli/

[18] S. Tapia, I. Romero, and A. Beltran, "A new approach for the solution of the neighborhood problem in meshfree methods." [Online]. Available: http://www.readcube.com/articles/10.1007/s00366-016-0468-8

[19] "Stl unordered map." [Online]. Available: http://www.cplusplus.com/reference/unordered_map/unordered_map/

[20] "Balancedtrees," stackoverflow.com. [Online]. Available: https://stackoverflow.com/questions/8015630/definition-of-a-balanced-tree

[21] "Stl vector." [Online]. Available: http://www.cplusplus.com/reference/vector/vector/

[22] "Rosetta tree." [Online]. Available: https://rosettacode.org/wiki/K-d_tree

[23] "Stl algorithm." [Online]. Available: http://www.cplusplus.com/reference/algorithm/

[24] "Intel (auto) vectorization tutorial." [Online]. Available: https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf

[25] "Intel threading building blocks." [Online]. Available: https://www.threadingbuildingblocks.org/

[26] "Zeromq." [Online]. Available: http://zeromq.org/

[27] "Boost libraries." [Online]. Available: http://www.boost.org/

[28] "Stackoverflow. magic number." [Online]. Available: https://stackoverflow.com/questions/4948780/magic-number-in-boosthash-combine

[29] "Quickselect." [Online]. Available: https://en.wikipedia.org/wiki/Quickselect

[30] "Dutch national flag problem." [Online]. Available: https://en.wikipedia.org/wiki/Dutch_national_flag_problem

[31] "Assignment 3: Kdtree." [Online]. Available: http://web.stanford.edu/class/cs106l/assignments.html

[32] "a c++ kd-tree implementation http://juliansimioni.com." [Online]. Available: https://github.com/orangejulius/kdtree

[33] "C standard general utilities library." [Online]. Available: http://www.cplusplus.com/reference/cstdlib/

[34] “Stl chrono.” [Online]. Available: http://www.cplusplus.com/reference/chrono/

[35] “Floyd rivest algorithm.” [Online]. Available: https://en.wikipedia.org/wiki/Floyd%E2%80%93Rivest_algorithm

[36] “Hoare partition algorithm.”

[37] *754-2008 - IEEE Standard for Floating-Point Arithmetic*, Std. [Online]. Available: http://ieeexplore.ieee.org/document/4610935/

[38] t. f. e. Wikipedia, “k-d trees,” *Wikipedia, the free encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/K-d_tree

[39] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, 1975. [Online]. Available: http://dl.acm.org/citation.cfm?id=361007