Universidad del Bío-Bío, Chile

Facultad de Ciencias Empresariales

Departamento de Sistemas de Información

Procesamiento de consultas de proximidad espacial sobre datos almacenados en la estructura de datos compacta k^2-tree

Tesis presentada por Fernando Santolaya Franco
para obtener el grado de Magíster en Ciencias de la Computación
Dirigida por Dra. Mónica Caniupán Marileo
Mg. Luis Gajardo Díaz

2018

Agradecimientos

Esta tesis va dedicada a mis hijos Laura y José Raúl.

Agradecer a mi esposa por su apoyo emocional y su gran fortaleza para soportar mi cambiante humor.

A mi profesora guía, Dra. Mónica Caniupán por su eterna paciencia en todo el proceso de tesis, su dedicación y exhaustiva revisión en los documentos generados en este proceso. Debo recalcar su compromiso por ayudar a mejorar mi terrible forma de redactar y la escueta forma en que expreso mis ideas.

Al profesor Luis Gajardo, por su guía técnica en las labores de programación y su apoyo como Director de Departamento para que tuviese los tiempos para lograr los objetivos encomendados en esta tesis.

A Miguel Romero, Carlos Quijada y el profesor Gilberto Gutiérrez por su buena voluntad de contestar mis innumerables preguntas.

A mi colega de trabajo, Juan Carlos Figueroa, por su eterno consejo: "Termina luego la tesis...".

Por último, agradecer que esta tesis de magíster fue parcialmente financiada por el proyecto de investigación exploratorio 2030 titulado "*Estructuras de Datos Compactas para procesar eficientemente datos espaciales y espacio-temporales en el contexto del Big Data*", código 1638 y el Grupo de Investigación ALBA, código GI 160119/EF.

Abstract

Big Data is one of the current concepts with more interest in the computer science area. Basically, big data deals with the problem of management, analyzing and processing huge volumes of data, which cannot be treated in a conventional manner, since they exceed the processing limits of existing software tools. As an illustration, the storage and query of spatio-temporal data, such as: points of interest on a map, has grown considerably over the years, basically motivated by the inclusion of GPS technologies to most of today's devices. This trend has opened new approaches to handle massive amount of data, which involve taking advantage of the increase capacity of the main memory (RAM), the use of data compression techniques, distributed processing, among others. This, in order to answer queries more quickly and use the smallest possible memory space. In this direction the concept of *compact data structures* becomes relevant. Compact data structures are data structures that allow to reduce the data size without losing the capability of querying the data over the compact form, use less main space and offer efficient processing of queries over the compact representation of the data. They have been used in diverse scenarios, such as, to represent graphs of the World Wide Web (WEB), in information retrieval, in document databases, in bioinformatics, and recently in geographical information systems, among others.

In this thesis we focalize in the implementation of algorithms over compact data structures to answer efficiently two useful spatial data queries: (a) KNN: given a point p search for the K-nearest neighbors to p, and (b) KCPQ: given two sets of points R and S search for the K-closest pairs of neighbors over the set of points. To achieve our goals, we use the k^2-tree compact data structure, which has been used efficiently to store and query adjacency matrices that represent directed graphs of the WEB. We represent points of interest on a map and execute the two spatial queries KNN and KCPQ over the compact representation of the data. Through experimentation on synthetic and real data sets, we show that by using the compact data structure k^2-tree we can work with a huge amount of points in main memory, and also, that the KNN and KCPQ spatial data queries can be efficiency computed over the compact data structures.

Finally, we implement a JAVA library, that is left available to the academic and industrial community. It implements the k^2-tree compact data structure, with basic operations to operate it, and implements the two spatial queries defined previously.

Keywords — Compact data structures, Spatial Queries, Big Data, k^2-tree.

Resumen

Big Data es un concepto que ha tomado bastante atención en el área de ciencias de la computación en los últimos años, y trata el problema de gestionar, analizar y procesar enormes volúmenes de datos, los cuales no pueden ser tratados de manera convencional, debido a que superan los límites de procesamiento de las actuales herramientas de software. Por ejemplo, el almacenamiento y procesamiento de consultas de datos espacio-temporales, como puntos de interés en un mapa, ha crecido considerablemente, básicamente por la inclusión de tecnologías GPS en la mayoría de los dispositivos actuales. Esta tendencia ha permitido que se desarrollen nuevos enfoques para procesar grandes volúmenes de datos, lo cual implica aprovechar al máximo la memoria principal (RAM), el uso de técnicas de compresión de datos, procesamiento paralelo, entre otros. Esto, con el fin de responder eficientemente consultas sobre datos y utilizar el menor espacio posible. Desde este punto de vista, la utilización de *estructuras de datos compactas* (EDCs) se ha vuelto interesante. Las EDCs son estructuras de datos modificadas para representar datos en poco espacio manteniendo las propiedades de acceso, sin la necesidad de descomprimir los datos. Las EDCs permiten usar menos espacio y procesar eficientemente consultas sobre datos compactados. Las EDCs han sido utilizadas en diversos escenarios tales como: representación de grafos de la World Wide Web (WEB), en recuperación de información, bases de datos de documentos, en bioinformática, y recientemente en sistemas de información geográfica, entre otros.

En esta tesis nos focalizamos en implementar algoritmos sobre EDCs para responder de manera eficiente dos consultas espaciales: (a) KNN: dado un punto p encontrar los K-vecinos (puntos) más cercanos a p, y (b) KCPQ: dado dos conjuntos de puntos R y S encontrar los K-pares de vecinos (puntos) más cercanos sobre R y S. Los puntos se almacenan en la EDC k^2-tree y luego se ejecutan las consultas espaciales KNN y KCPQ sobre la EDC. Mediante la ejecución de experimentos sobre datos ficticios y reales, se demuestra que es posible trabajar sobre grandes volúmenes de datos espaciales, en memoria principal, de manera compacta y además, que es posible computar de manera eficiente las consultas KNN y KCPQ. Finalmente, se implementa una librería JAVA que se deja disponible a la comunidad académica e industrial, la cual implementa la estructura de datos compacta k^2-tree con las operaciones básicas para operar sobre ella y las consultas KNN y KCPQ.

Palabras Clave — Estructuras de Datos Compactas, Consultas Espaciales, Big Data, k^2-tree.

Índice General

1. Introducción **1**
- 1.1. Contribuciones . . . 4
- 1.2. Organización de la Tesis . . . 4

2. Propuesta de Tesis **5**
- 2.1. Hipótesis . . . 5
- 2.2. Objetivos . . . 5
  - 2.2.1. Objetivo General . . . 5
  - 2.2.2. Objetivos Específicos . . . 5
- 2.3. Alcance de la Investigación . . . 6
- 2.4. Metodología . . . 7

3. Estado del Arte **8**
- 3.1. Algoritmos Tradicionales para Computar Consultas de Proximidad Espacial 8
- 3.2. Estructuras de Datos Compactas . . . 10
- 3.3. Algoritmos para Computar Consultas de Proximidad Espacial sobre Estructuras de Datos Compactas . . . 11

4. Preliminares **12**
- 4.1. Consultas Espaciales . . . 12
- 4.2. Estructura de Datos Compacta k2-tree . . . 16
  - 4.2.1. Construcción de un k2-tree . . . 17
  - 4.2.2. Navegación sobre la Estructura de Datos Compacta k2-tree . . . 18

5. Algoritmos para Computar Consultas de Proximidad Espacial **20**
- 5.1. Los K-vecinos más Cercanos (KNN) . . . 20
- 5.2. Algoritmo K-pares de vecinos más Cercanos (KCPQ) . . . 26
- 5.3. Análisis de Complejidad Temporal de los Algoritmos KNN y KCPQ . . . 31
- 5.4. Algoritmo para Calcular KNN sin EDCs . . . 32
- 5.5. Algoritmo para Calcular KCPQ sin EDCs . . . 34
- 5.6. Librerías Externas . . . 36
  - 5.6.1. Librerías que Implementan Operaciones Sobre Estructuras Compactas 36
  - 5.6.2. Librería ALBA-K2-TREE . . . 38

6. Experimentación **44**
6.1. Escenario de Experimentación 44
6.2. Diseño de Experimentos 45
6.3. Resultados de Ahorro de Espacio al Utilizar k^2-trees 47
6.4. Resultados en Tiempos de Ejecución para KNN 48
6.4.1. Tiempos de Ejecución para KNN sobre Datos Sintéticos 48
6.4.2. Cálculos de Distancia para KNN sobre Datos Sintéticos 49
6.4.3. Tiempos de Ejecución para KNN sobre Datos Reales 51
6.5. Resultados para KCPQ 55
6.5.1. Tiempos de Ejecución para KCPQ Sobre Datos Sintéticos 55
6.5.2. Cálculos de Distancia para KCPQ sobre Datos Sintéticos 56
6.5.3. Tiempos de Ejecución para KCPQ Sobre Datos Reales 59

7. Conclusiones y Trabajo Futuro **61**

Referencias **63**

Índice de Figuras

4.1. Tipos de puntos espaciales sobre un plano espacial de Google Maps 13
4.2. Los $K = 5$ hospitales/centros de salud más cercanos respecto del punto amarillo sobre un mapa de Google Maps 14
4.3. Los $K = 4$ pares de hoteles/restaurantes sobre un mapa de Google Maps . . 15
4.4. Traspaso de puntos de interés a matriz de adyacencia 16
4.5. Construcción de árbol k^2-ario a partir de una matriz de adyacencia M . . . 17
4.6. k^2-tree para la matriz de adyacencia de la Figura 4.5 y bitmaps generados . 18
4.7. k^2-tree para el Ejemplo 4.3 19

5.1. k^2-tree para la matriz de adyacencia de la Tabla 5.1 23
5.2. Heap Mínimo *pQueue* para el Ejemplo 5.1 24
5.3. Puntos de interés en un mapa y sus respectivas matrices de adyacencia R y S 29
5.4. k^2-trees para las matrices de adyacencia ilustradas en la Figura 5.3 29
5.5. Heap Mínimo *pQueue* para el Ejemplo 5.2 31
5.6. Estructura interna de la librería ALBA-k^2-tree 39
5.7. Interface WEB para consultar los K-vecinos más cercanos 41
5.8. Interface WEB para consultar los K-pares de vecinos más cercanos 42
5.9. Interfaces Android para consultas KNN y KCPQ 42
5.10. Representación del funcionamiento de los servicios WEB para KNN y KCPQ 43

6.1. Gráficas de distribución de puntos 46
6.2. Uso de memoria de la estructura compacta k^2-tree v/s puntos en RAM . . 48
6.3. Tiempos de ejecución de los algoritmos KNN con $N = 100{,}000$ 49
6.4. Tiempos de ejecución de los algoritmos KNN con $N = 1{,}000{,}000$ 50
6.5. Tiempos de ejecución de los algoritmos KNN con $N = 10{,}000{,}000$ 50
6.6. Tiempos de ejecución de los algoritmos KNN con un K fijo de $K = 25$ . . . 51
6.7. Comportamiento en cálculos de distancia de KNN para distintos N y un $K = 5$ 52
6.8. Comportamiento en cálculos de distancia de KNN para distintos N y un $K = 25$ 53
6.9. Tiempo de ejecución en Nano Segundos para conjuntos de datos reales . . . 53
6.10. Tiempo de ejecución en Nano Segundos para conjuntos de datos reales . . . 54
6.11. Tiempo de ejecución en Nano Segundos para conjuntos de datos reales . . . 54

6.12. Tiempos de ejecución de los algoritmos KCPQ con $N = 100{,}000$ 56
6.13. Tiempos de ejecución de los algoritmos KCPQ con $N = 1{,}000{,}000$ 56
6.14. Tiempos de ejecución de los algoritmos KCPQ con $N = 10{,}000{,}000$ 57
6.15. Comportamiento en cálculos de distancia de KCPQ para distintos N y un $K = 5$ 57
6.16. Comportamiento en cálculos de distancia de KCPQ para distintos N y un $K = 25$ 58
6.17. Tiempo de ejecución en Nano Segundos para conjuntos de datos reales . . . 59
6.18. Tiempo de ejecución en Nano Segundos para conjuntos de datos reales . . . 60

Índice de Tablas

5.1. Puntos en una matriz de tamaño 16×16 y punto $q = (8, 18)$ 23
5.2. Comparación de tiempos entre librerías en segundos 38

6.1. Especificaciones hardware servidor experimentos 45
6.2. Memoria utilizada por la estructura de datos compacta k^2-tree frente a memoria en RAM en distribución aleatoria 47
6.3. Memoria utilizada por la estructura de datos compacta k^2-tree frente a memoria en RAM en distribución Gaussiana 47
6.4. Tabla de cálculos de distancia para consulta KNN en conjuntos de datos con distribución aleatoria 51
6.5. Tabla de cálculos de distancia para consulta KNN en conjuntos de datos con distribución Gaussiana 52
6.6. Tabla de número de cálculos de distancia para consulta KCPQ en conjuntos de datos con distribución aleatoria 58
6.7. Tabla de número de cálculos de distancia para consulta KCPQ en conjuntos de datos con distribución Gaussiana 58

Índice de Algoritmos

1. Obtener los K-vecinos más cercanos al punto q 22
2. Obtener los K-pares de vecinos más cercanos 27
3. Obtener los K-vecinos más cercanos al punto q sin EDCs 33
4. Obtener los K-pares de vecinos más cercanos sin EDCs 36
5. Función bestDistance 36
6. Función closest 37

Capítulo 1

Introducción

El término Big Data ha cobrado relevancia y se ha convertido en una disciplina ubicua en los últimos años. Su definición no está del todo consensuada, debido a que es un área que ha evolucionado de forma veloz, por lo tanto, no existe una definición formal, universalmente aceptada que denote su significado (De Mauro et al., 2015). Sin embargo, (De Mauro et al., 2015) propone una definición amplia para el concepto: "Big Data representa activos de información caracterizados por ser altos en volumen, velocidad y variedad y requerir tecnologías y métodos analíticos específicos para su transformación en valor".

El concepto de Big Data es de gran relevancia en distintos ámbitos de la sociedad, aunque puede estar oculto, en el sentido que podemos estar haciendo uso de tecnología Big Data pero no estar al tanto de ello. Por ejemplo, se encuentra presente en distintas disciplinas tales como la biología, la astronomía, las redes sociales, la geografía, la economía, entre otras. Por lo tanto, estamos ante escenarios donde se generan enormes volúmenes datos de distinta índole y son consumidos ávidamente.

El análisis de grandes conjuntos de datos no es algo nuevo, aún así es uno de los actuales retos de la minería de datos y otras disciplinas afines. Según algunos hechos expuestos por la empresa nodegraph[1] en junio del 2017 casi 4 billones de usuarios de Internet fueron registrados. Esto equivale aproximadamente a que el 51 % de la población mundial está en línea. Por otra parte, se hacen 3,9 millones de búsquedas en el motor de Google cada minuto. Para el 2020 se estima que cada persona genere 1,7 Megabytes de información por segundo y que $\frac{1}{3}$ de todos los datos se procesará en la nube. Las posibilidades son infinitas, y se hace interesante evaluar que las actuales estructuras de datos y dispositivos hardware poseen límites para procesar grandes volúmenes de datos. Esto debido a que mientras la velocidad de procesamiento de la CPU se ha duplicado casi anualmente, no pasa lo mismo con la capacidad de almacenar cantidades masivas de datos.

En consecuencia, surgen requerimientos tecnológicos para dar respuesta a los retos que propone Big Data. Una de ellas, es la necesidad de aprovechar los recursos de memoria principal (RAM) al máximo, permitiendo almacenar más datos en ella (generalmente más pequeña respecto de la memoria en disco), acceder a los datos de forma eficiente y por

[1] https://www.nodegraph.se/big-data-facts/

sobre todo, lograr que los sistemas (software) sean capaces de administrar estos grandes volúmenes de información en memoria principal (Navarro, 2016).

Hoy en día los dispositivos móviles han cobrado gran relevancia en Big Data, ya que, no sólo sirven como medio de comunicación y de cómputo, sino que también como un conjunto de sensores embebidos, por ejemplo, el sensor de posicionamiento global (GPS), que permite tomar información geoespacial en tiempo real (Lane et al., 2010). La asociación mundial de operadores móviles GSMA[2] indica que sólo en América Latina y el caribe hay 414 millones de suscriptores en telefonía móvil, en Chile la tasa de penetración es del 92 % con 16,7 millones de suscriptores. Las estadísticas ilustradas indican un potencial crecimiento en el segmento de dispositivos móviles, y por consiguiente, la producción de datos por parte de estos dispositivos móviles, en especial de tipo geoespacial o de geolocalización no deja de ser menor. Es por este motivo que la comunidad de base de datos ha potenciado las bases de datos espaciales, para así dar mejor soporte a la administración de este tipo de datos. Para ello, se han definido nuevos tipos de datos, tales como polígonos, líneas, puntos, entre otros, junto con dar soporte a consultas sobre estos tipos de datos utilizando operadores espaciales (Shekhar y Chawla, 2003).

En función de la posición espacial de un objeto, es de utilidad que las aplicaciones de tipo geográficas, como por ejemplo: Foursquare, Waze, Google Maps, entre otras, permitan etiquetar un lugar de interés y hacer diversas consultas respecto a los datos capturados. Con respecto a las consultas que se pueden aplicar sobre los datos espaciales, es relevante, que se puedan responder consultas de proximidad espacial, como por ejemplo: obtener los objetos que se encuentran en un radio determinado (denominadas consultas de rango), los objetos más cercanos a un objeto particular, entre otras. En esta tesis nos concentramos en dos consultas espaciales relevantes: (a) KNN: obtener los K-vecinos más cercanos a un objeto q, y (b) KCPQ: obtener los K-pares de vecinos más cercanos sobre dos conjuntos de puntos R y S sobre un mismo plano espacial. Un ejemplo de KNN es dada una ubicación (punto) q en una ciudad (plano espacial) encontrar los $K = 3$ hoteles más cercanos a q. Un ejemplo de KCPQ puede ser, encontrar los $K = 4$ pares de colegios (conjunto de puntos R) y centros de comida rápida (conjunto de puntos S) de una ciudad (plano espacial).

Por consiguiente, al analizar estos escenarios de Big Data, surgen interrogantes de cómo manejar grandes volúmenes de datos (puntos de interés o POI) y más aún, cómo hacerlo de forma eficiente. El problema en definitiva no recae en el tamaño de los datos, sino, en las estructuras de datos necesarias que se deben diseñar para almacenar y consultar grandes volúmenes de datos (Navarro, 2016). Buscando una solución al tema antes planteado, surge el concepto de estructuras de datos compactas (EDCs), las cuales son estructuras de datos que permiten compactar datos de cierta naturaleza sin perder la capacidad de consultar los datos en su forma compacta. Las EDCs usan una cantidad de espacio cercana al límite inferior teórico, es decir el número óptimo de bits para almacenar datos (Fariña et al., 2009; Navarro, 2016).

El concepto de EDCs es introducido inicialmente por Jacobson (Jacobson, 1989) bajo el nombre de *estructuras sucintas*, cuyo fin es optimizar datos de tipo estático, es decir,

[2]`http://www.gsma.com/latinamerica/wp-content/uploads/2016/09/ME_LATAM_2016-Spanish-Report-FINAL-Web-Singles-1.pdf`

estructuras que no se modifican en tiempo real (manejadas de manera off-line), enfocándose principalmente en estructuras de tipo árbol. Las estructuras de tipo árbol, por ejemplo un árbol de n nodos, son eficientes para hacer búsquedas transversales e inserciones dinámicas. Sin embargo, en ocasiones son ineficientes en términos del espacio utilizado, producto de que la mayoría de las veces son representadas con punteros, de tal manera que para representar n diferentes ubicaciones necesitan $\lceil \lg n \rceil$ bits y una estructura no compacta con $O(n)$ nodos necesita $O(n \log n)$ bits en memoria principal. No obstante, una representación sucinta de la misma estructura puede requerir $2n + o(n)$ bits, más la implementación de operaciones para acceder a los datos que requieren tiempo constante (Arroyuelo et al., 2010).

En particular, la estructura de datos compacta k^2-tree se propuso en (Brisaboa et al., 2009) y corresponde a una estructura de datos de tipo árbol, creada inicialmente para representar y explorar grafos de la WEB, por ejemplo, los de las redes sociales, que pueden contener millones de usuarios lo que supone no solo problemas de eficiencia, sino que también de complejidad de análisis (Brisaboa et al., 2014b). La idea detrás de la estructura es utilizar resúmenes de grafos, construidos con técnicas de reducción mediante la agrupación de aristas y nodos, que permiten obtener representaciones simplificadas de los grafos. Al utilizar la estructura de datos compacta k^2-tree se evita acceder al disco para leer datos, trabajando directamente sobre la estructura de datos compacta en memoria principal.

En esta tesis se propone utilizar la estructura de datos compacta k^2-tree para representar puntos en un mapa y ejecutar sobre los datos compactos las consultas espaciales KNN y KCPQ. A través de experimentos sobre datos sintéticos y reales demostramos que es posible trabajar con grandes volúmenes de puntos espaciales en memoria RAM y computar de manera eficiente las consultas KNN y KCPQ sobre los datos compactos. Para comparar nuestros algoritmos y demostrar su eficiencia, se implementan algoritmos que no utilizan EDCs para ambas consultas, los cuales operan sin indexación de datos, y tienen una complejidad en tiempo de $O(n \log n)$. Se realizaron comparaciones utilizando tres métricas: tiempo de consulta, espacio utilizado y cantidad de cálculos de distancia respecto de las implementaciones sobre la estructura de datos compacta k^2-tree.

Regularmente las implementaciones de algoritmos de esta naturaleza son desarrollados en lenguaje C, dada sus características de flexibilidad, trabajo a bajo nivel, manejo detallado de memoria y la disponibilidad de librerías (Brisaboa et al., 2009). Sin embargo, en esta tesis las implementaciones generadas, es decir, la implementación de la estructura de datos compacta k^2-tree, las operaciones de navegación sobre el k^2-tree, las consultas de proximidad espacial KNN y KCPQ, se desarrollaron bajo el lenguaje JAVA, decisión respaldada por la idea de generar una librería que pueda ser utilizada en la industria, donde JAVA es ampliamente utilizado. Por otro lado, esto permite que la librería que se deja a disposición de la comunidad industrial y académica pueda ser utilizada en máquinas con Apache Spark (ambientes distribuidos).

1.1. Contribuciones

Las contribuciones de la tesis son las siguientes:

- Implementación de la estructura de datos compacta k^2-tree junto con toda su funcionalidad en lenguaje JAVA.
- Implementación de algoritmos para responder la consulta espacial de los K-vecinos más cercanos (KNN) sobre puntos representados en un k^2-tree.
- Implementación de algoritmos para responder la consulta espacial de los K-pares de vecinos más cercanos (KCPQ) sobre dos conjuntos de puntos representados en un k^2-tree, respectivamente.
- Generación de un escenario para experimentación con datos sintéticos y reales.
- Experimentación sobre datos sintéticos y reales.
- Implementación de algoritmos que no utilizan EDCs, de orden temporal $O(n \log n)$ para resolver las consultas espaciales KNN y KCPQ.
- Generación de una librería JAVA con la implementación de la estructura de datos compacta k^2-tree junto con toda su funcionalidad y las consultas KNN y KCPQ.

1.2. Organización de la Tesis

El resto del documento se organiza de la siguiente manera. En el Capítulo 2 se presenta la hipótesis de la tesis, el objetivo general, los objetivos específicos, los alcances y limitaciones y la metodología de trabajo. En el Capítulo 3 se presentan trabajos relacionados respecto a los algoritmos tradicionales para computar consultas de proximidad espacial, el uso de las EDCs en diferentes escenarios, incluyendo el uso de EDCs para el cómputo de consultas de proximidad espacial. En el Capítulo 4 se presenta la estructura de datos compacta k^2-tree y la definición de las consultas espaciales KNN y KCPQ. En el Capítulo 5 se presenta los algoritmos para responder a las consultas espaciales KNN y KCPQ sobre la estructura de datos compacta k^2-tree, los algoritmos que no utilizan EDCs para KNN y KCPQ, además de la librería JAVA. En el Capítulo 6 se presentan los resultados de la experimentación realizada sobre datos reales y datos sintéticos. Finalmente, en el Capítulo 7 se presentan las conclusiones de esta tesis y posibles trabajos futuros.

Capítulo 2

Propuesta de Tesis

En este capítulo se presenta la hipótesis de la tesis, junto con el objetivo general, los objetivos específicos, los alcances y límites de la tesis y la metodología de trabajo.

2.1. Hipótesis

La hipótesis de esta tesis es la siguiente:

- Es posible computar eficientemente las consultas espaciales de los K-vecinos más cercanos (KNN) y los K-pares de vecinos más cercanos (KCPQ) sobre grandes volúmenes de datos, representados en la estructura de datos compacta k^2-tree en memoria principal.

2.2. Objetivos

En esta sección se presenta el objetivo general de la tesis en conjunto con los objetivos específicos a realizar.

2.2.1. Objetivo General

Implementar la estructura de datos compacta k^2-tree en lenguaje JAVA para representar grandes conjuntos de puntos espaciales y algoritmos para responder de manera eficiente las consultas KNN y KCPQ, sobre conjuntos de puntos espaciales representados en la estructura de datos compacta k^2-tree en memoria principal.

2.2.2. Objetivos Específicos

Los objetivos específicos de la tesis son:

- Analizar algoritmos de proximidad espacial existentes en la literatura.

- Implementar o adaptar un algoritmo para generar la estructura de datos compacta k^2-tree en lenguaje JAVA.
- Diseñar e implementar un algoritmo para responder eficientemente la consulta KNN sobre grandes conjuntos de puntos espaciales representados en un k^2-tree.
- Diseñar e implementar un algoritmo para responder eficientemente la consulta KCPQ sobre dos grandes conjuntos de puntos representados en estructuras de datos compacta k^2-tree.
- Implementar algoritmos para responder eficientemente, sin uso de EDCs, la consulta KNN y la consulta KCPQ.
- Diseñar conjuntos de datos ficticios y recolectar datos reales para experimentación.
- Evaluar experimentalmente la eficiencia de los algoritmos de proximidad espacial en tiempo, espacio y cálculo de distancias, comparando con las implementaciones ingenuas.
- Generar una librería JAVA con las implementaciones anteriores para dejarla disponible a la comunidad académica e industrial.

2.3. Alcance de la Investigación

Esta tesis se enmarca en el proyecto de investigación exploratorio 2030 titulado "*Estructuras de Datos Compactas para procesar eficientemente datos espaciales y espacio-temporales en el contexto del Big Data*", código 1638, implementado por los miembros del Grupo de investigación ALBA, código GI 160119/EF, pertenecientes a la Universidad del Bío-Bío.

Este trabajo se enfoca en la implementación eficiente de dos consultas espaciales comunes sobre grandes conjuntos de puntos espaciales, representados en la estructura de datos compacta k^2-tree implementada en lenguaje JAVA. Se contempla además la implementación de una librería para la comunidad académica e industrial. En esta última comunidad JAVA es ampliamente utilizado, por lo tanto, nuestro aporte puede motivar el desarrollo de aplicaciones industriales o prototipos de las consultas implementadas u otras consultas de proximidad espacial.

Por simplicidad, en esta investigación, los puntos de interés (POI) sobre un mapa se representan con el número 1. Por tanto, la estructura de datos k^2-tree que se utiliza compacta ceros, opción por defecto (Brisaboa et al., 2009), esto debido a que se desea compactar las zonas de mapas donde no existe nada interesante de evaluar y por lo tanto pueden ser descartadas para el cómputo de las consultas. Esta tesis solo se restringe a las consultas KNN y KCPQ.

Para la experimentación se construyen dos algoritmos que no utilizan EDCs para computar la consulta espacial KNN y KCPQ. El primer algoritmo se basa en los algoritmos

presentados en (Aha et al., 1991; Dasarathy, 1991; Sedgewick y Wayne, 2011; Wettschereck et al., 1997). El segundo, que computa la consulta KCPQ se basa en el algoritmo presentado en (Weiss, 2007).

Para la experimentación se utilizan datos ficticios (grandes volúmenes) y datos reales obtenidos de la WEB que consideran datos de hospitales, centros de evacuación, estaciones de metros, museos, universidades, entre otros, de la ciudad de New York (US). Los algoritmos propuestas para responder a las consultas KNN y KCPQ fueron comparados con los algoritmos que no utilizan EDCs considerando las métricas de espacio utilizado, tiempo de respuesta y la cantidad de cálculos de distancia.

2.4. Metodología

La metodología de trabajo de esta investigación está basada en función de los objetivos específicos planteados anteriormente y considera las siguientes etapas:

1. Revisión bibliográfica: revisión de la literatura respecto de estructuras de datos compactas y consultas de proximidad espacial.

2. Diseño e implementación de los algoritmos para responder consultas espaciales: analizar e implementar los algoritmos de K-vecinos más cercanos (KNN) y K-pares de vecinos más cercanos (KCPQ) sobre la estructura de datos compacta k^2-tree en lenguaje JAVA.

3. Implementación de los algoritmos que no utilizan EDCs, para las consultas KNN y KCPQ.

4. Experimentación: diseño e implementación de experimentos en base a datos sintéticos y datos reales, de manera de evaluar el rendimiento en tiempo, cálculos de distancia y espacio de los algoritmos implementados.

5. Evaluación de resultados: interpretar resultados de los experimentos, generar gráficas sobre el rendimiento de los algoritmos implementados.

6. Generación de librería JAVA: implementar una librería con todos los algoritmos desarrollados en esta tesis y habilitarla para la comunidad industrial y académica.

Capítulo 3

Estado del Arte

Este capítulo se divide en tres secciones. En la Sección 3.1 se realiza una revisión de los principales algoritmos que computan consultas de proximidad espacial. En la Sección 3.2 se presenta el concepto de EDCs y los escenarios donde se han utilizado. Finalmente, en la Sección 3.3 se revisan algoritmos que computan consultas de proximidad espacial sobre EDCs.

3.1. Algoritmos Tradicionales para Computar Consultas de Proximidad Espacial

Las bases de datos espaciales han cobrado relevancia en estos tiempos. Estas se han extendido de manera de representar y consultar objetos espaciales, tales como: puntos, líneas, segmentos, regiones, polígonos, volúmenes y/o otras figuras en 2D/3D, de manera más natural, contemplando la utilización de nuevas tecnologías y el uso de avanzadas investigaciones en estructuras de datos para dar solución a los múltiples escenarios tecnológicos del mundo actual, en específico, sobre geolocalización[1]. Por consiguiente, una de las características clave de una base de datos espacial es que se pueda manipular la información espacial (Corral, 2002). En relación a lo anterior, existen consultas espaciales típicas, tales como: buscar objetos espaciales que se encuentran dentro de un rango espacial dado (conocida como window query), buscar objetos espaciales que están contenidos en otros objetos espaciales, unir varios conjuntos de datos de acuerdo a algún criterio espacial (tales como: overlap, intersects, disjoint, entre otros), buscar objetos espaciales por proximidad o en su defecto consultas basadas en distancias, entre otras (Shekhar y Chawla, 2003).

Los actuales sistemas de bases de datos, tienen la limitación de que solo pueden trabajar con una cantidad limitada de datos en memoria principal, dadas las limitantes de memoria RAM. Usualmente, la forma de procesar consultas espaciales es por medio del uso de índices espaciales sobre los datos en memoria secundaria, de tal forma de traer de manera más rápida los datos a memoria principal. Actualmente, existen bastantes estructuras de

[1] Por ejemplo, el administrador de bases de datos PostgreSQL ha implementado una extensión espacial llamada POSTGIS https://postgis.net/

datos para indexar datos multidimensionales tales como: R^*-tree (Beckmann et al., 1990), *Quadtree* (Samet, 1984), *Tv-tree* (Lin et al., 1994), *X-tree* (Berchtold et al., 1996), *B-tree*, *K-d-tree*, *Bd-tree* (Gaede y Günther, 1998), *Pyramid-technique* (Berchtold et al., 1998). Otros indices son reportados en (Manolopoulos et al., 2010; Samet, 2006; Shekhar y Chawla, 2003).

Las consultas KNN y KCPQ no se encuentran implementadas directamente en motores de bases de datos. Sin embargo, existen algunos algoritmos que permiten computarlas en orden $O(n\ log\ n)$, tales como (Aha et al., 1991; Dasarathy, 1991; Papadopoulos y Manolopoulos, 1997; Roussopoulos et al., 1995; Sedgewick y Wayne, 2011; Wettschereck et al., 1997) para la consulta KNN y (Weiss, 2007), (Corral et al., 2000) y (Corral, 2002) para la consulta KCPQ. Estos algoritmos usan diferentes estructuras de datos, especialmente estructuras de datos de tipo árbol específicas para almacenar datos espaciales, como la familia de R-trees (Guttman, 1984) y *QuadTrees* (Samet, 1984). En específico, (Papadopoulos y Manolopoulos, 1997; Roussopoulos et al., 1995) ofrecen una solución para la consulta KNN sobre R-trees, introduciendo métricas como $MINDIST$, $MINMAXDIST$ y el uso de Heaps (Cormen et al., 2009) para podar y mejorar la eficiencia de los algoritmos para ordenar y podar la búsqueda del vecino más cercano.

En (Yao et al., 2010) se presentan algoritmos relacionales que pueden ser implementados usando operadores primitivos `SQL` para computar la consulta KNN y *KNN-Join* (de un conjunto de puntos de consulta), de forma exacta y aproximada en grandes bases de datos relacionales, sin la dependencia de utilizar funciones definidas por el usuario, de manera de que el optimizador de consultas del motor de base de datos pueda entender y optimizar. Lo anterior en función de utilizar las funciones legadas de los motores de bases de datos relacionales y no depender de la estructura de datos R-tree, la cual no ha sido totalmente implementada en muchos motores de bases de datos, tiene pobre rendimiento cuando los datos tienen más de seis dimensiones y que además, no provee ninguna garantía teórica en los costos de la consulta KNN.

En (Kim y Patel, 2010) se realiza una comparación de dos índices espaciales populares, el R^*-tree (Beckmann et al., 1990) y el *Quadtree* (Samet, 1984), para consultas KNN y consultas basadas en distancia, como por ejemplo KCPQ. Una de las conclusiones interesantes de este trabajo es que el método de partición regular y disjunta utilizado por el *Quadtree* (base de la estructura compacta k^2-tree) tiene una ventaja estructural inherente sobre el R^*-tree en la ejecución de las consultas.

En (Chen y Patel, 2007) se presenta la consulta *ANN* (All Nearest Neighbor), que corresponde a encontrar todos los vecinos más cercanos a un punto sobre la estructura R^*-tree, para lo cual usa una nueva métrica llamada $NXDIST$ que mejora a la existente $nitMAXMAXDIST$ para podar nodos en el índice que deben ser visitados durante el cómputo de la consulta. Alternativamente ofrece una estructura modificada de un *Quadtree* llamado $MBRQT$, la cual mejora los tiempos de cómputo de la consulta *ANN*. Un trabajo similar se presenta en (Zhang et al., 2016), donde se presentan algoritmos para computar la consulta $AKNN$ (all-K-nearest-neighbor), que obtiene los K vecinos más cercanos para cada objeto p en un conjunto de puntos Q en espacios métricos, considerando un conjunto de datos sin indexación. También, en (Miranda et al., 2013) se proponen algoritmos para

computar consultas de proximidad espacial como obtener todos los K-vecinos más cercanos (ANN) sin indexación, si no que a través del uso de hardware paralelo, como GPGPU (General Purpose Graphics Processing Unit).

Por otra parte, también existen trabajos que implementan soluciones para las consultas de proximidad KNN y KCPQ, pero difieren de los trabajos nombrados previamente, en que no utilizan los índices más comunes del área como lo son el R-tree o el *Quadtree*. Por ejemplo, en (Mavrommatis et al., 2017) se ilustra el cómputo de la consulta de proximidad KCPQ, sobre grandes conjuntos de puntos espaciales, utilizando para ello técnicas como RDD (Resilient Distributed Dataset) y MapReduce, por lo tanto, se proponen algoritmos paralelos para procesar esta consulta. Técnicamente el método utilizado corta el plano en porciones y utiliza muestras de estas porciones para limitar el espacio de la solución. Este límite es usado como un criterio de poda en varias fases de cálculo. Por tanto, el algoritmo, iterativamente e incrementalmente, aproxima la solución en tres fases principales, hasta obtener la solución exacta en la cuarta fase.

En (Katayama y Satoh, 1998) se presenta una estructura de indexación rápida llamada *SR-tree* (Sphere/Rectangle tree) la cual se deriva de la estructura de datos *SS-tree* (White y Jain, 1996), para resolver eficientemente consultas de vecinos más cercanos para datos multidimensionales, tales como bibliotecas de imágenes digitales, entre otros. La estructura de datos *SR-tree* usa elementos del R^*-tree como los MBRs, combinados con otros elementos como esferas para particionar el espacio, ofreciendo mejoras en tiempo de CPU, accesos a disco y un particionado más eficiente del espacio sobre el R^*-tree y *SS-tree*. En la misma dirección, en (Nene y Nayar, 1997) se presenta un algoritmo para encontrar el vecino más cercano en conjuntos de datos multidimensionales. Dado que las estructuras de datos, tales como *K-D-tree* y R-tree crecen exponencialmente mientras más dimensiones se considera, en (Nene y Nayar, 1997) se presenta una técnica de búsqueda por proyección, propuesta inicialmente por (Friedman et al., 1975), lo que se denomina particionado dinámico de espacio sobre espacios vectoriales.

3.2. Estructuras de Datos Compactas

Las EDCs son estructuras de datos que permiten compactar datos sin perder la capacidad de consultar los datos en su forma compacta. Las EDCs usan una cantidad de espacio cercana al límite inferior teórico, es decir el número óptimo de bits para almacenar datos (Fariña et al., 2009; Navarro, 2016). Las EDCs han sido utilizadas en diversos escenarios. Por ejemplo, en (Brisaboa et al., 2009; Claude y Navarro, 2007) se presenta la estructura compacta k^2-tree que se utiliza para representar grafos de la WEB, que como ejemplo, en el año 2004 contenía unos 11.5 millones de nodos y 150 mil millones de links. Entonces este grafo de la WEB necesita unos 600 GB de RAM para ser almacenado. Sin embargo, utilizando estructuras de datos compactas, se puede almacenar en alrededor de 100 GB, permitiendo realizar consultas sobre el grafo de manera eficiente. En (Brisaboa et al., 2013a; Claude y Navarro, 2008; Navarro, 2014; Navarro y Sadakane, 2014) se reportan estructuras compactas para representar documentos en el contexto de recuperación de la información. En (Brisaboa et al., 2014a) se presenta la estructura de datos compacta

k^2-treap para el cómputo de consultas *top-k* sobre matrices de dos dimensiones con datos numéricos, la cual es mejorada en (Brisaboa et al., 2016) para n dimensiones. En (Vallejos et al., 2017) se presenta la estructura de datos compacta k^2-treap para computar consultas de agregación sobre data warehouses con dimensiones lineales.

3.3. Algoritmos para Computar Consultas de Proximidad Espacial sobre Estructuras de Datos Compactas

El uso de EDCs en el ámbito de bases de datos espaciales es reciente. En (Gagie et al., 2015) y (Venkat y Mount, 2014) se proponen *Quadtrees* sucintos (estructuras de datos que son asintóticamente óptimas en espacio, cercanas al límite inferior teórico) para codificar datos de tipo punto y soportar consultas sobre estos. Los *Quadtrees* son dinámicos y espacio-eficientes para responder consultas de proximidad con conjuntos de datos en 2D. En (Ishiyama et al., 2017) se proponen *Quadtrees* sucintos para almacenar segmentos de líneas y computar consultas de manera eficiente. Los autores se enfocan en el problema de "map-matching", que consiste en que dado un punto de consulta en un plano bi-dimensional y un conjunto de segmentos de línea, se desea encontrar la K-línea o segmento más cercano al punto consultado. Estos algoritmos tienen bastante aplicabilidad, como por ejemplo, los segmentos de línea pueden representar carreteras y el punto puede ser un vehículo, y la consulta puede ser encontrar las 3-carreteras más cercanas al vehículo.

En (Brisaboa et al., 2013b) se presentan EDCs para mejorar el procesamiento de consultas de rango (o window query) en sistemas GIS (Geographical Information Systems). En (De Bernardo et al., 2013) se presentan estructuras de datos compactas para representar y consultar datos espaciales de tipo ráster (por ejemplo, datos de temperaturas), las consultas que se implementan son consultas básicas, tales como obtener el valor de una celda (temperatura en una región), etc.

Recientemente, en la tesis doctoral (Romero, 2017) se desarrolla una EDC basada en la estructura de datos compacta k^2-tree para computar de manera eficiente en memoria principal, la consulta KNN sobre conjuntos de puntos considerando espacio y tiempo (bases de datos espacio-temporales). Los algoritmos presentados en esta tesis para el computo de la consulta KNN se basan en los algoritmos presentados en (Romero, 2017), pero considerando puntos sin considerar la dimensión tiempo. Sin embargo, hasta donde hemos podido estudiar, no existen soluciones para la consulta KCPQ utilizando estructuras de datos compactas.

Capítulo 4

Preliminares

En este capítulo se presentan las consultas espaciales más comunes y las implementadas en esta tesis. Por otra parte, también se presenta la estructura de datos k^2-tree, considerando su construcción y navegación.

4.1. Consultas Espaciales

Un sistema de base de datos espacial, es un sistema administrador de bases de datos, encargado principalmente de manejar datos de tipo espacial. Los datos de tipo espacial, tienen dos posibilidades de representación, los objetos espaciales y el espacio en sí mismo. Por consiguiente, mientras la primera permite modelar ciudades, ríos, entre otras, la segunda permite describir mapas, como por ejemplo mostrar una división política (Güting, 1994). Los datos espaciales se utilizan para representar de forma simplificada objetos y espacios físicos del mundo real, puesto que estos no pueden ser representados con tipos de datos convencionales. Algunos de los tipos de datos espaciales más comunes son: puntos, líneas y polígonos (Güting, 1994; Shekhar y Chawla, 2003). En la Figura 4.1 se ilustran los tipos de datos espaciales mencionados sobre un mapa de Google Maps.

Un punto es una coordenada (x, y) de un plano cartesiano aplicado a un plano espacial (latitud, longitud) que describe la forma de un objeto de dimensión cero, por ejemplo, un punto de interés dentro de un mapa que representa la ubicación de un hotel (marcador rojo Figura 4.1). Una línea corresponde a un conjunto de puntos, que al unirlos, forman una figura no cerrada de una dimensión, por ejemplo, las calles en el mapa de una ciudad (línea de color negro Figura 4.1). Un polígono corresponde a un conjunto de puntos, que a diferencia de una línea, al unirlos, forman una figura cerrada (área), como por ejemplo, la laguna que se muestra en la Figura 4.1.

Existen muchas consultas espaciales que se aplican a estos tipos de datos, siendo las más comunes (Shekhar y Chawla, 2003):

1. Consulta de rango espacial (Range Query o Window Query): Esta consulta retorna todos los objetos O que se encuentran dentro de un rango espacial determinado

Figura 4.1: Tipos de puntos espaciales sobre un plano espacial de Google Maps

P. Por ejemplo: "Encontrar todas las gasolineras dentro de la ciudad de Santiago". Formalmente se define como:

$$RQ(P) = \{O | O.G \cap P.G \neq \varnothing\}$$

Donde G es la geometría del objeto.

2. Consulta de unión espacial (Spatial Join): Esta consulta combina dos conjuntos de objetos espaciales R y S para formar un nuevo conjunto, con los objetos que satisfacen un predicado θ dado. Por ejemplo, "obtener las parcelas que intersectan a un lago". Formalmente se define como:

$$R \bowtie_\theta S = \{(o, o\prime) | o \in R, o\prime \in S, \theta(o.G, o\prime.G)\}$$

Donde G es la geometría del objeto.

3. Consulta de los K-vecinos más cercanos (KNN): Esta consulta retorna los K puntos (vecinos) más cercanos a un punto q dado. Por ejemplo: "Encontrar los $K = 3$ hospitales más cercanos al lugar de un accidente q". Formalmente se define como:

$$KNN(P, q, K) = \begin{cases} \{p_1, p_2, \ldots, p_K\} \in P, p_i \neq p_j, i \neq j, 1 \leqslant i, j \\ \leqslant K, \forall p_i \in P - \{p_1, p_2, \ldots, p_K\}, d_p(p_1, q) \\ \leqslant d_p(p_2, q) \leqslant \cdots \leqslant d_p(p_K, q) \leqslant d_p(p_l, q), \\ con\ l > K \end{cases}$$

Donde, P es un conjunto de puntos ($P \neq \emptyset$), q es un punto de consulta, K es un número positivo real y $d_p(\cdot, \cdot)$ es una función de distancia.

4. Consulta de los K-pares de vecinos más cercanos sobre dos conjuntos de datos de puntos (KCPQ): La consulta de los K pares de vecinos más cercanos consiste en, dado dos conjuntos de puntos R y S, encontrar los K pares más cercanos, de la

forma $(r,\ s)$ tal que, $r \in R$ y $s \in S$. Por ejemplo: "Encontrar los hoteles (R) más cercanos a aeropuertos (S)". Formalmente, se define como:

$$KCPQ(R,S,K) = \begin{cases} \{(r_1,s_1),(r_2,s_2),\ldots,(r_K,s_K)\} \in R \times S, \\ \{r_1,r_2,\ldots,r_k\} \in R, \{s_1,s_2,\ldots,s_k\} \in S, \\ (r_i,s_i) \neq (r_j,s_j), i \neq j, 1 \leqslant i,j \leqslant K, \\ \forall (r_i,s_i) \in R \times S - \{(r_1,s_1),(r_2,s_2),\ldots, \\ (r_K,s_K)\}, d_p(r_1,s_1) \leqslant d_p(r_2,s_2) \leqslant \ldots \\ \leqslant d_p(r_K,s_K) \leqslant d_p(r_l,s_v), con\ l,v > K \end{cases}$$

Donde P y Q son dos conjuntos de puntos, tal que $P \neq \emptyset$ y $Q \neq \emptyset$, K es un número positivo real y $d_p(\cdot,\cdot)$ es una función de distancia.

El Ejemplo 4.1 muestra la consulta KNN y la consulta KCPQ sobre un mapa real de la ciudad de Concepción.

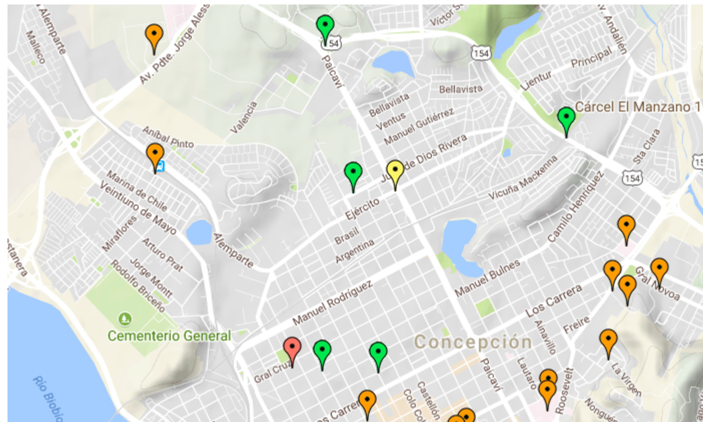


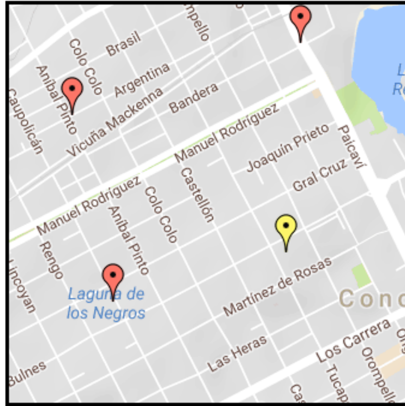
Figura 4.2: Los $K = 5$ hospitales/centros de salud más cercanos respecto del punto amarillo sobre un mapa de Google Maps

Ejemplo 4.1 La Figura 4.2 ilustra el resultado de la consulta de proximidad espacial KNN sobre puntos de interés ubicados en un mapa de la ciudad de Concepción, donde los puntos de color naranjos corresponden a hospitales/centros de salud y los de color verde corresponde al resultado de la consulta, indicando los $K = 5$ hospitales/centros de salud más cercanos respecto del punto amarillo que corresponde a la ubicación actual (punto de consulta). La Figura 4.3 ilustra el resultado de la consulta de proximidad espacial KCPQ sobre puntos de interés ubicados en un mapa de la ciudad de Concepción, donde los puntos de color rojo corresponden a hoteles y los de color azul corresponden a restaurantes. Los puntos unidos con líneas verdes representan a los $K = 4$ pares de puntos más cercanos. □

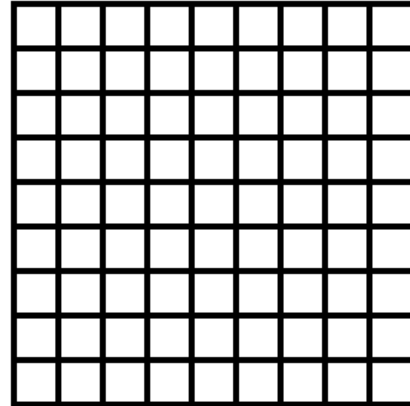
Figura 4.3: Los $K = 4$ pares de hoteles/restaurantes sobre un mapa de Google Maps

En esta tesis, se consideran puntos que representan elementos de interés en un mapa. Por ejemplo, hospitales, farmacias, estaciones de servicio, etc. Se utiliza la relación métrica por sobre las de dirección y de topología, dado que básicamente se necesita saber la distancia Euclidiana de un punto a otro para realizar los cálculos de distancia. Actualmente, en sistemas de bases de datos espaciales, estos objetos geográficos son representados con estructuras especiales de índices, lo cual permite responder adecuadamente consultas espaciales limitadas. Para lo anterior existen diversas estructuras para representar objetos espaciales, por ejemplo, usando una aproximación geométrica de los objetos reales o MBR (Minimun Bounding Rectangle) que es capaz de contener completamente la geometría del objeto. Por consiguiente, estos objetos son agrupados por jerarquías, usando relaciones de proximidad con otros MBRs, y estos a su vez almacenados sobre diversas estructuras de datos, tales como *K-D-B-Tree*, *R-Tree*, *R^*-Tree*, *Árboles B*, entre otras (Papadias y Theodoridis, 1997).

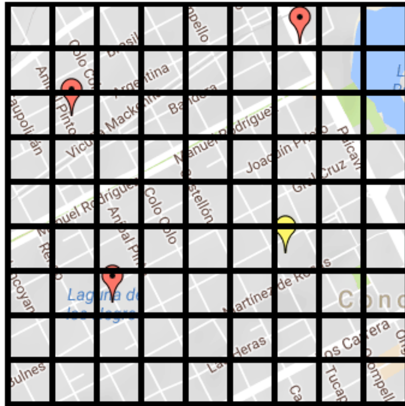
Dado que en esta tesis nos centramos en puntos y no en polígonos o líneas, utilizamos un método sencillo de traspaso de puntos en un mapa a una matriz de adyacencia. La Figura 4.4 ilustra conceptualmente esta conversión, el mapa de la Figura 4.4(a) contiene puntos de interés, la Figura 4.4(b) muestra la estructura de la grilla que representa la matriz de adyacencia donde se almacenan los puntos de interés del mapa. La Figura 4.4(c) muestra la superposición de la grilla sobre el mapa de la Figura 4.4(a) y finalmente, la Figura 4.4(d) muestra la matriz de adyacencia, que es la base para construir la estructura de datos compacta k^2-tree (ver Sección 4.2).



(a) Plano con puntos de interés



(b) Estructura de matriz de adyacencia



(c) Estructura de la matriz(b) sobre plano(a)

0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

(d) Matriz de adyacencia resultante

Figura 4.4: Traspaso de puntos de interés a matriz de adyacencia

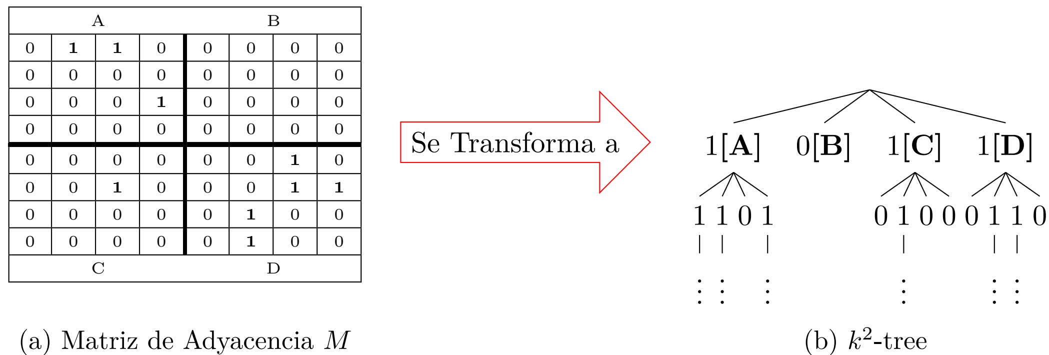
4.2. Estructura de Datos Compacta k^2-tree

Esta estructura de datos compacta fue presentada en (Brisaboa et al., 2009) para materializar grafos de la WEB, donde los nodos del grafo representan páginas WEB y las aristas representan enlaces entre las páginas. En este escenario, la estructura compacta k^2-tree permite representar la relación binaria de enlace entre nodos del grafo, mediante una matriz de adyacencia. De esta forma, una celda (i, j) de la matriz de adyacencia contiene un 1 si existe una arista entre el nodo i y el nodo j en el grafo WEB, en caso contrario, contiene un 0.

En esta tesis se utiliza la matriz de adyacencia para representar coordenadas geográficas (puntos de interés) de un mapa, en donde un 1 indica la presencia de un punto de interés y un 0 representa la ausencia del mismo (vacío), como se ilustra en la Figura 4.4(d).

4.2.1. Construcción de un k^2-tree

A partir de una matriz de adyacencia se puede construir un k^2-tree. La altura del árbol k^2-tree es $h = [log_k n]$, donde n es la cantidad de elementos en la matriz, y k es la constante que indica la cantidad de particiones que se realizan en la matriz, siendo esta última siempre potencia de 2. Para la Figura 4.5 se considera un valor para k de 2. Bajo los parámetros antes descritos se crea un árbol k^2-ario que se obtiene mediante un proceso de divisiones recursivas de la matriz de adyacencia. Para obtener el primer nivel del árbol se subdivide la matriz en k^2-submatrices cuadradas del mismo tamaño. Cada una de estas submatrices se representan en el árbol con un 1, sí y solo sí, la partición correspondiente, contiene al menos un elemento y con un 0 en caso contrario. Para los siguientes niveles, cada submatriz que contiene al menos un elemento (las que están representadas con un 1) son representadas con k^2-hijos en el siguiente nivel, los que se obtienen subdividiendo las submatrices en k^2 submatrices nuevamente utilizando el mismo procedimiento. Este proceso recursivo continúa hasta que cada elemento del último nivel del árbol corresponde a una celda de la matriz de adyacencia. El Ejemplo 4.2 muestra la creación de la EDC a partir de una matriz de adyacencia.



(a) Matriz de Adyacencia M

(b) k^2-tree

Figura 4.5: Construcción de árbol k^2-ario a partir de una matriz de adyacencia M

Ejemplo 4.2 En la Figura 4.5 se presenta el k^2-tree para la matriz de adyacencia M de la Figura 4.5(a), considerando un valor de $k = 2$. Dado que $k = 2$ se deben realizar $2^2 = 4$ particiones de la matriz, lo que genera cuatro hijos para la raíz del árbol de la Figura 4.5(b). Cada división o cuadrante de la raíz contiene un 1 si existe por lo menos un 1 en el cuadrante, en caso contrario se registra un 0. El proceso de división continua recursivamente para cada cuadrante hasta llegar al nivel de una celda, que corresponden a las hojas del k^2-tree y representa un valor de la matriz de adyacencia original.

A partir del k^2-tree se obtienen 2 bitmaps, el primero T para representar los nodos internos y el segundo L para representar las hojas. En la Figura 4.6 se visualizan los bitmaps T y L generados a partir del k^2-tree de la Figura 4.5(b). □

El árbol k^2-tree se utiliza solo como una representación visual, siendo lo relevante dos bitmaps denominados T y L (de Árbol (Tree) y Hojas (Leafs), los cuales representan a los

nodos internos y a las hojas del k^2-tree, respectivamente. Esta representación está diseñada para comprimir grandes matrices, en donde existen muchas áreas de 0's (Brisaboa et al., 2009). Debemos destacar que en la implementación, T y L se unen como un solo bitmap.

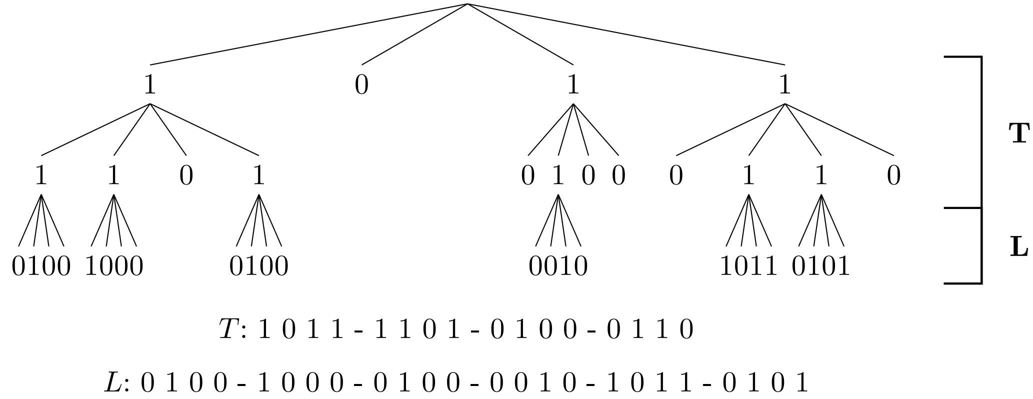


Figura 4.6: k^2-tree para la matriz de adyacencia de la Figura 4.5 y bitmaps generados

4.2.2. Navegación sobre la Estructura de Datos Compacta k^2-tree

Para navegar sobre el árbol k^2-tree, hay que recorrer los bitmaps T y L, que parten desde la posición 1. Para esto es necesario utilizar las operaciones *Rank* y *Select* (Fariña et al., 2009; González et al., 2005). Dado un bitmap T y una posición p.

- La operación $Rank_1(T,p)$ cuenta la cantidad de 1s (o 0s), hasta una posición p en T.
- La operación $Select_1(T,j)$ retorna la posición de la j-ésima ocurrencia de 1 (o 0) en T.

Para obtener el hijo de un nodo, se utiliza la función $child_i(x) = Rank(T,x) \times k^2 + i$ que permite obtener el i-ésimo hijo del nodo x en el k^2-tree. En el Ejemplo 4.3 muestra la aplicación de las operaciones *Rank*, *Select* y función *child* sobre los bitmaps de la Figura 4.6.

Ejemplo 4.3 Considere el k^2-tree de la Figura 4.6 y sus respectivos bitmaps T y L. La operación $Rank(T,6) = 5$ nos indica que existen cinco 1s hasta la posición 6 del bitmap (desde la posición 1). La operación $Select(T,12) = 26$ nos indica que el doceavo 1 está en la posición 26 del bitmap T, pero dado que T tiene 16 posiciones, se continúa la búsqueda en el bitmap L.

Supongamos que deseamos obtener el tercer hijo (las posiciones parten desde 1, por lo tanto $i = 3$) del cuarto hijo de la raíz ($x = 4$) del k^2-tree, se debe aplicar $child_3(4) = Rank(T,4) \times k^2 + 3 = 3 \times 2^2 + 3 = 15$, lo que indica que el tercer hijo del cuarto hijo de la raíz se encuentra en la posición 15 del bitmap T (ver Figura 4.7).

Si se desea obtener todos los hijos del tercer hijo del cuarto hijo de la raíz, es decir, los hijos del nodo 15, debemos obtener la posición del primer hijo del nodo 15, por lo tanto, se obtiene $child_1(15) = Rank(T, 15) \times k^2 + 1 = 9 \times 2^2 + 1 = 37$. Esta posición sobrepasa el tamaño del bitmap T ($|T|$) correspondiente a los nodos intermedios del árbol, entonces, el primer hijo del nodo 15 se encuentra en la posición $37 - |T| = 37 - 16 = 21$, y los tres restantes bits corresponden a los siguientes hijos del nodo 15, hasta la posición 24 del bitmap L (ver Figura 4.7). □

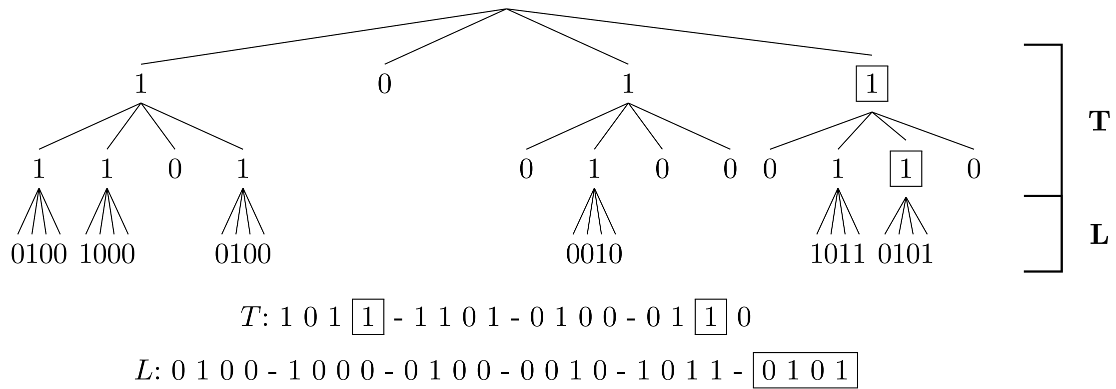


Figura 4.7: k^2-tree para el Ejemplo 4.3

Como se mencionó anteriormente, el k^2-tree fue diseñado para realizar análisis sobre grafos WEB. Por consiguiente, éste ofrece operaciones para consultar sobre la estructura, como los vecinos directos a un nodo en particular o los reversos, como también comprobar la existencia de un enlace. Existen otras operaciones como las *operaciones de rango* descritas en (Brisaboa et al., 2015), que permiten recuperar celdas activas (celdas con puntos) para una región o cuadrante determinado de la matriz de adyacencia. Estas operaciones se realizan mediante un recorrido descendente por el árbol, desde la raíz (primer nivel) hasta las hojas. Específicamente, para obtener los vecinos de un nodo en la matriz de adyacencia, corresponde verificar todas las celdas de la fila que le corresponde al nodo en la matriz de adyacencia, y si lo extrapolamos al k^2-tree, corresponde a descender por las ramas del árbol que contengan alguna celda de dicha fila.

Capítulo 5

Algoritmos para Computar Consultas de Proximidad Espacial

Este capítulo presenta la implementación de los algoritmos, introducidos en las Secciones 5.1 y 5.2, que permiten computar las consultas KNN y KCPQ sobre la estructura de datos compacta k^2-tree, respectivamente. Por otra parte, en las Secciones 5.4 y 5.5 se presentan implementaciones para los algoritmos KNN y KCPQ, las cuales no utilizan estructuras de datos compactas para responder las consultas antes descritas. Estos algoritmos que no utilizan EDCs son utilizados para generar escenarios de comparación y experimentación (descritos en Capítulo 6) respecto de los algoritmos desarrollados en la tesis, que trabajan sobre la estructura de datos compacta k^2-tree.

5.1. Los K-vecinos más Cercanos (KNN)

La consulta para obtener los K-vecinos más cercanos a un punto q fue definida en el Capítulo 4, Sección 4.1. A continuación, se presenta el Algoritmo 1 que permite responder la consulta KNN considerando que los puntos se encuentran representados en un k^2-tree.

El Algoritmo 1 necesita tres parámetros de entrada: una constante K que indica el número de vecinos que se desea obtener; un punto q que corresponde al punto de consulta y que puede estar dentro del conjunto inicial de puntos o fuera de él y un k^2-tree T que representa la estructura compacta que contiene los puntos, y que se crea antes de invocar al Algoritmo 1. La salida del algoritmo es el conjunto de los K vecinos (puntos) más cercanos al punto q. Para procesar los puntos en el k^2-tree, el Algoritmo 1 implementa dos heaps, uno llamado *pQueue* que corresponde a un heap de valores mínimos, que almacena las distancias de los puntos extremos[1] de cada sub-árbol del k^2-tree T, que se encuentran más cercanos al punto q, ordenadas de menor a mayor. De esta forma, el primer elemento de *pQueue* (elemento en el tope del heap) corresponde al sub-árbol o cuadrante de la matriz de adyacencia que está más cercano al punto q. La idea, por lo tanto, es ordenar los sub-árboles del k^2-tree en *pQueue*, respecto a la distancia mínima de cada uno de los puntos

[1] Un punto extremo de un cuadrante es un punto que se encuentra en el borde del cuadrante.

extremos al punto q, de manera de no tener que buscar en aquellos sub-árboles que están más lejos del punto q. El otro heap se denomina *Cand* y corresponde a un heap de valores máximos, que almacena los K potenciales puntos más cercanos a q (puntos candidatos). De esta forma, el elemento en el tope de *Cand* corresponde al punto que está más lejos del punto q. El algoritmo se beneficia de este heap ya que termina, si se han encontrado los K vecinos más cercanos, y si la distancia que existe entre el elemento visitado es mayor o igual a la distancia del elemento en el tope de *Cand* (Línea 8 del Algoritmo 1). Esto último indica que el nodo visitado contiene un punto que no está dentro de los más cercanos.

Para calcular la distancia entre un punto q y un sub-árbol o cuadrante de la estructura de datos compacta k^2-tree, se utiliza una métrica llamada *MINDIST* (distancia mínima) (Roussopoulos et al., 1995). Esta métrica es una variación de la distancia Euclidiana, y mide la distancia desde un punto a un cuadrante. De esta manera, si el punto está dentro del cuadrante, entonces la distancia es 0. Si el punto está fuera del cuadrante, se usa el cuadrado de la distancia Euclidiana entre el punto y el borde más cercano del cuadrante, ya que esto involucra hacer menos cálculos matemáticos a nivel de procesador, dado que se evita evaluar la raíz cuadrada de la fórmula original de la distancia Euclidiana. De esta manera, *MINDIST* se define como, la distancia de un punto $q(x, y)$ a un cuadrante R, definido por dos puntos extremos $s(x, y)$ y $t(x, y)$ en el mismo espacio Euclidiano. Formalmente, se denota por $\mathit{MINDIST}(q, R)$, y se define como:

$$\mathit{MINDIST}(q,R) = |q_x - r_x|^2 + |q_y - r_y|^2, \quad \text{donde:}$$
$$r_x = \begin{cases} s_x, & \text{si } q_x < s_x \\ t_x, & \text{si } q_x > t_x \\ q_x, & \text{en otro caso} \end{cases} \qquad r_y = \begin{cases} s_y, & \text{si } q_y < s_y \\ t_y, & \text{si } q_y > t_y \\ q_y, & \text{en otro caso} \end{cases} \tag{5.1}$$

En las (Líneas 1 y 2 del Algoritmo 1) se crean los heap *Cand* y *pQueue*, respectivamente. De esta manera, el primer elemento a insertar en el heap *pQueue*, denominado e (Línea 3), creado por la función *CreateElementQueue*, tiene una *estructura* con cuatro atributos: (i) *pos*: que corresponde a la posición del sub-árbol en el bitmap T, (ii) *quad*: que corresponde al cuadrante en evaluación (en primera instancia corresponde al k^2-tree completo), (iii) *minDist*: que corresponde a la distancia mínima del punto extremo del sub-árbol al punto q, y (iv) *nivel*: que corresponde al nivel del sub-árbol del k^2-tree T. Este campo es muy útil, ya que permite al algoritmo identificar hojas de nodos intermedios. En cada paso *pQueue* se actualiza considerando las distancias mínimas de los sub-árboles al punto q.

El algoritmo busca los K puntos más cercanos a q mientas *pQueue* no esté vacío (Líneas 5-21). En cada paso, el algoritmo obtiene un elemento e desde el heap *pQueue* existiendo tres posibilidades:

1. El número de candidatos K se ha alcanzado y el elemento e obtenido del tope de *pQueue* tiene una distancia mayor que el elemento en el tope de *Cand* (Línea 7). En este caso, el algoritmo termina retornando *Cand*.

2. Se evalúa si el elemento e es una hoja con la función $isLeaf(e)$ (Línea 9). Esta función

Algoritmo 1: OBTENER LOS K-VECINOS MÁS CERCANOS AL PUNTO q

```
   input  : k^2-tree T, a point q, a constant K
   output: A set of K points nearest to q
   /*Initially the heaps are empty                                */
1  Cand ← CreateMaxHeap();
2  pQueue ← CreateMinHeap();
   /*Creates the first element to pQueue                          */
3  e ← CreateElementQueue(0, T.quad(), minDist(q, T.quad()), 1);
4  pQueue.insert(e);
5  while pQueue ≠ ∅ do
6      e ← pQueue.delete();
7      if Cand.size() = K AND e[3] ≥ Cand.Max() then
8          return Cand
9      if isLeaf(e) then
10         e2 ← CreateEntry(e.getPoint(), e[3]);
11         if Cand.size() < K then
12             Cand.insert(e2);
13         else if e[3] < Cand.Max() then
14             Cand.delete();
15             Cand.insert(e2);
16     else if hasChildren(e) then
17         forall child h do
18             minD ← MinDist(q, h.quad());
19             if (Cand.size() < K) OR (minD < Cand.Max()) then
20                 e2 ← CreateElementQueue(h, h.quad(), minD, e[4] + 1);
21                 pQueue.insert(e2);
22 return Cand
```

determina si el elemento e es una hoja del k^2-tree, para ello, revisa el cuarto campo (nivel) del elemento e. Si el nivel es mayor a la altura de T, entonces el elemento e corresponde a una hoja, por lo tanto, el elemento e es un posible candidato y se puede insertar en *Cand*. La función *createEntry*() crea un elemento candidato e_2, para ser insertado en *Cand* (Línea 10). Para ello utiliza la función *GetPoint*(), que retorna uno de los puntos extremos del cuadrante en e. El elemento e_2 tiene dos campos: (i) punto: que corresponde al punto obtenido de uno de los extremos del cuadrante de e y (ii) dist: que corresponde a la distancia mínima entre e y el punto q. Entonces, si el número de candidatos es menor a K (no se han completado todos las respuestas) (Línea 11), e_2 se inserta en *Cand*. En caso contrario, si la distancia del elemento e es menor que la distancia del elemento en el tope de *Cand* (Línea 13), es decir, e es

Tabla 5.1: Puntos en una matriz de tamaño 16×16 y punto $q = (8, 18)$

A								B							
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C								D							

mejor candidato que el elemento que está en el topo de *Cand*, entonces el elemento en el tope se elimina y se inserta el elemento e_2.

3. El elemento e es un nodo con hijos (Línea 16), evaluado por la función $hasChildren(e)$ que verifica si el elemento e tiene hijos, mediante la observación del cuarto campo de e (nivel). Si el nivel de e es menor a la altura del k^2-tree T, entonces e es un nodo intermedio y por lo tanto, posee hijos. Entonces, por cada hijo h del elemento e, es decir, por cada sub-cuadrante del cuadrante padre de e, y solo aquellos sub-cuadrantes en la cual una operación de tipo *Rank* entregue como resultado un 1, el algoritmo obtiene la distancia mínima entre h y el punto q (Líneas 17 y 18). Luego, si el número de candidatos es menor que K o la distancia mínima del hijo actual es menor que la distancia del elemento en el tope de *Cand*, es posible que se haya encontrado un nuevo candidato en el sub-árbol h, y por lo tanto, el nuevo elemento e_2 se agrega a *pQueue* (Líneas 20-21).

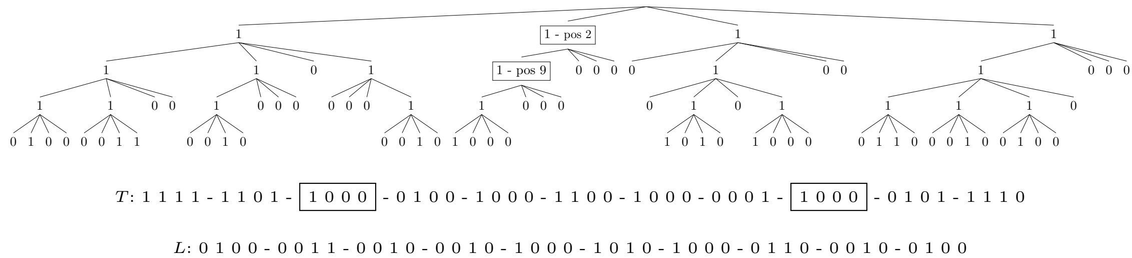


Figura 5.1: k^2-tree para la matriz de adyacencia de la Tabla 5.1

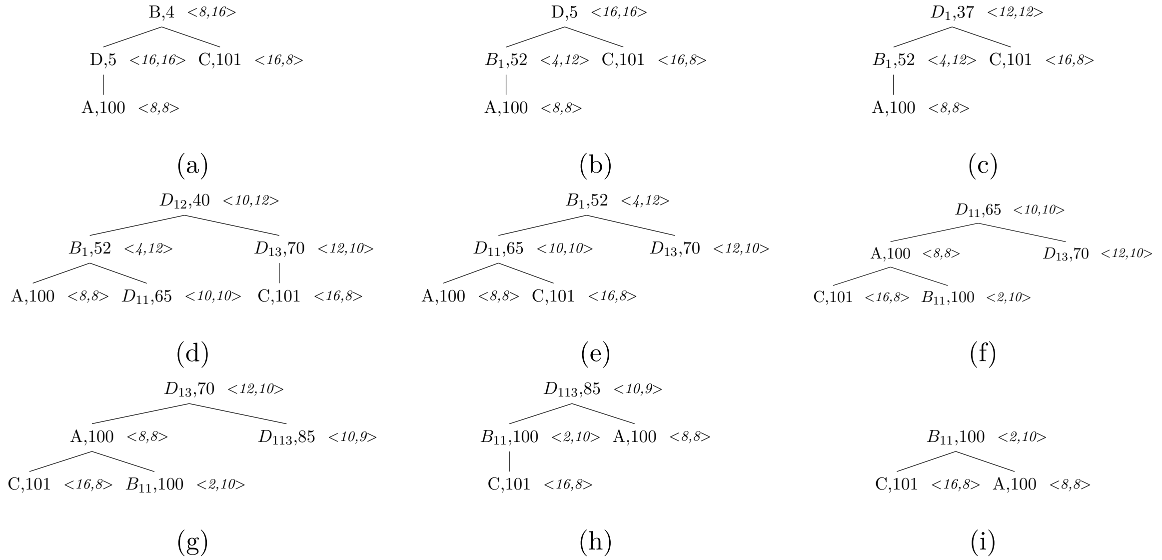


Figura 5.2: Heap Mínimo *pQueue* para el Ejemplo 5.1

El Ejemplo 5.1 ilustra el funcionamiento del Algoritmo 1.

Ejemplo 5.1 Considere la matriz de adyacencia de la Tabla 5.1, con los cuadrantes A, B, C y D, el k^2-tree de la Figura 5.1, el punto $q = (8, 18)$, y $K = 4$. Inicialmente *pQueue* contiene a todo el k^2-tree T, esto es, la matriz completa, la cual tiene como puntos extremos a los puntos $(1, 1)$ y $(16, 16)$. Así, de acuerdo a la fórmula para calcular la mínima distancia *MINDIST*, $s_x = 1$, $s_y = 1$, $t_x = 16$, $t_y = 16$, $q_x = 8$, $q_y = 18$. Entonces r_x es 8 dado que q_x no es menor a s_x y tampoco es mayor a t_x, y r_y es 16 dado que q_y es mayor a t_y de manera que, la distancia mínima a q es: $|q_x - r_x|^2 + |q_y - r_y|^2 = |8 - 8|^2 + |18 - 16|^2 = 4$, por tanto, el heap *pQueue* queda como $pQueue = T(4)$. Los $K = 4$ puntos candidatos más cercanos a q serán almacenados en el heap *Cand* que inicialmente está vacío. El proceso para obtener estos puntos es el siguiente:

1. El ciclo iterativo es inicializado (Línea 5), entonces el algoritmo elimina el elemento que está en el tope del heap *pQueue* (Línea 6) que corresponde a todo el k^2-tree T. Dado que este cuadrante tiene cuatro hijos con valor 1 (Línea 16) y no existen elementos en *Cand* (Línea 19), todos los cuadrantes son insertados en *pQueue* (Línea 21). De esta manera, *pQueue* es actualizado con nuevos valores, y ahora queda de la siguiente manera $pQueue = \{B(4), D(5), C(101), A(100)\}$. Dado que el cuadrante B es el cuadrante más cercano al punto q, éste queda en el tope del heap (ver Figura 5.2(a)). Las coordenadas de cada uno de los puntos extremos más cercanos a q de cada cuadrante son $pQueueCoord = \{(8, 16), (16, 16), (16, 8), (8, 8)\}$. Por lo tanto, los puntos candidatos más cercanos a q deben ser buscados en el primer cuadrante B.

2. El algoritmo retorna (Línea 6), entonces se elimina del tope del heap *pQueue* (Línea 6) el elemento que corresponde al cuadrante B. Dado que este cuadrante tiene un solo hijo con 1's (Línea 16), y aún no existen elementos en el heap *Cand*, el cuadrante B_1 con distancia mínima de 52 se inserta en *pQueue* (Línea 21)(ver Figura 5.2(b)).

3. El algoritmo retorna (Línea 6) y procesa el cuadrante D. Dado que D también posee sólo un hijo (Línea 16), el cuadrante D_1, con distancia mínima de 37, es insertado en *pQueue* (Línea 21), dado que el número de candidatos aún no ha sido alcanzado. La Figura 5.2(c) muestra el heap *pQueue*.

4. Luego el algoritmo retorna a la (Línea 6) y toma el elemento D_1, el cual tiene 3 hijos, que corresponden a los cuadrantes D_{11}, D_{12}, y D_{13}, con distancias mínimas de 65, 40, y 70, respectivamente. Todos estos nodos se insertan en *pQueue*, porque el número de candidatos sigue siendo menor que $K = 4$ (ver Figura 5.2(d)).

5. El algoritmo retorna a la (Línea 6) y toma el elemento D_{12}. Este cuadrante, tiene un único hijo D_{123} representado por el punto $(10, 11)$ con distancia mínima de 53 a q. Dado que no hay elementos en *Cand* (Líneas 9-15), este punto corresponde al primer candidato y se inserta en $Cand = \{D_{123}(53)\}$. La Figura 5.2(e) muestra el estado de *pQueue*.

6. El algoritmo retorna a la (Línea 6) y toma el elemento B_1 que tiene como único hijo a B_{11} con distancia mínima de 100. Este elemento se inserta en *pQueue* dado que aún no se logran los $K = 4$ puntos más cercanos a q. La Figura 5.2(f) muestra el estado de *pQueue*.

7. El algoritmo retorna a la (Línea 6), toma el elemento D_{11} que tiene dos hijos, D_{112} y D_{113} con distancias mínimas a q de 65 y 85, respectivamente. Estos hijos son hojas representando los puntos con coordenadas $(9, 10)$ y $(10, 9)$ y por tanto se insertan en el heap *pQueue*. Entonces, en la siguiente iteración (Línea 6), se toma el elemento $D_{112}(65)$ que representa una hoja, el cual está actualmente en el tope del heap *pQueue*, y dado que el heap *Cand* tiene un solo elemento (Línea 9), este último se inserta como nuevo candidato, quedando el heap *Cand* de la siguiente manera $Cand = \{D_{112}(65), D_{123}(53)\}$ (Línea 12). La Figura 5.2(g) muestra el heap *pQueue* luego de estas operaciones.

8. El algoritmo retorna a la (Línea 6), toma elemento D_{13} que tiene un hijo D_{132} con distancia mínima de 73 a q. Este elemento se inserta en *Cand* dado que actualmente hay solo dos candidatos ($K = 4$). Por lo tanto, el heap *Cand* queda de la siguiente manera $Cand = \{D_{132}(73), D_{112}(65), D_{123}(53)\}$. La Figura 5.2(h) muestra el nuevo estado del heap *pQueue*.

9. El algoritmo retorna a la (Línea 6), toma el elemento D_{113} del tope de *pQueue*, el cual es una hoja. Como aún no se han alcanzado los $K = 4$ candidatos, el elemento D_{113} se inserta en *Cand*. De esta manera, el heap *Cand* queda como

$Cand = \{D_{113}(85), D_{132}(73), D_{112}(65), D_{123}(53)\}$. La Figura 5.2(i) muestra el nuevo estado del heap *pQueue*.

10. Nuevamente se repite el proceso con el elemento B_{11}, y dado que este elemento tiene hijos, el algoritmo obtiene la distancia del cuadrante al punto q, la cual es 100. Como ya se tienen cuatro candidatos, y la distancia de $B_{11} = 100$ a q es mayor que el peor candidato $D_{113}(85)$, entonces el elemento B_{11} se descarta y el algoritmo finaliza. □

5.2. Algoritmo K-pares de Vecinos más Cercanos (KCPQ)

La consulta para obtener los K-pares de vecinos más cercanos fue definida en el Capítulo 4 Sección 4.1 y corresponde a encontrar los K-pares de puntos más cercanos de la forma (r, s) tal que $r \in R$ y $s \in S$, con R y S conjuntos de puntos distintos. A continuación, se presenta el Algoritmo 2 que permite responder la consulta KCPQ considerando que los conjuntos de puntos se encuentran representados, cada uno, por un k^2-tree.

El Algoritmo 2 sigue la misma lógica que el Algoritmo 1. Sin embargo, los heaps *pQueue* y *Cand* ahora almacenan pares de sub-árboles correspondientes a los k^2-trees de R y S, lo que es equivalente a almacenar los cuadrantes de las matrices de adyacencia de cada conjunto. Los elementos en el heap *pQueue* son ordenados considerando la mínima de las mínimas distancias entre cuadrantes (R y S), llamada *MINMINDIST* (la mínima de las mínimas distancias(*MINDIST*)), y el heap *Cand* (heap de máximos) almacena los K potenciales candidatos y es completado cuando los sub-árboles de R y S se recorren completamente, es decir, se alcanzan los nodos hojas.

La métrica *MINMINDIST* definida en (Corral, 2002), es una variación de la distancia Euclidiana, y mide la distancia desde un cuadrante a otro. De esta manera, si los cuadrantes de R y S se intersectan (un cuadrante sobre otro), entonces la distancia es 0. Caso contrario, se obtiene el mínimo de las mínimas distancias obtenidas desde cada punto extremo de un cuadrante de R a un cuadrante de S. De esta manera, *MINMINDIST* se define como, la distancia de un cuadrante en R definido por dos puntos extremos $p(x, y)$ y $q(x, y)$ a un cuadrante en S definido también por dos puntos extremos $s(x, y)$ y $t(x, y)$. Formalmente, se denota *MINMINDIST*(R, S), y se define como:

$$MINMINDIST(R, S) = min\{MINDIST(p, S), MINDIST(q, S)\} \quad (5.2)$$

donde *MINDIST* corresponde a la distancia mínima entre un punto y un cuadrante definida en la Ecuación 5.1.

En las (Líneas 1 y 2 del Algoritmo 2) se crean los heap *Cand* y *pQueue*, respectivamente. De esta manera, el primer elemento a insertar en el heap *pQueue*, denominado e (Línea 4), creado por la función *CreateElementQueue*(), tiene una *estructura* con seis atributos: (i) *pos1*: que corresponde a la posición del sub-árbol en el bitmap R, (ii) *quad1*: que corresponde al cuadrante en evaluación (en primera instancia corresponde al k^2-tree completo) de R, (iii) *pos2*: que corresponde a la posición del sub-árbol en el bitmap S,(iv)

quad2: que corresponde al cuadrante en evaluación (en primera instancia corresponde al k^2-tree completo) de S,(v) *dist*: que corresponde a la mínima de las mínimas distancias de los puntos extremos del sub-árbol de R al cuadrante de S, y (vi) *nivel*: que corresponde al nivel de los sub-árboles de R y S. Este campo es muy útil, ya que permite al algoritmo identificar hojas de nodos intermedios. En cada paso *pQueue* se actualiza considerando las distancias mínimas (*MINMINDIST*) de los sub-árboles de R y S.

Algoritmo 2: OBTENER LOS K-PARES DE VECINOS MÁS CERCANOS

```
   input  : Two k^2-tree R, S, a constant K
   output: A set of K-pairs of points of the form (r, s)
   /*Initially the heaps Cand and pQueue are empty                    */
 1 Cand ← CreateMaxHeap();
 2 pQueue ← CreateMinHeap();
   /*Creates the first element to pQueue                              */
 3 distance ← MINMINDIST(R.quad(), S.quad());
 4 e ← CreateElementQueue(0, R.quad(), 0, S.quad(), distance, 1);
 5 pQueue.insert(e);
 6 while pQueue ≠ ∅ do
 7     e ← pQueue.delete();
 8     if Cand.size() = K AND e[5] ≥ Cand.Max() then
 9         return Cand
10     if areLeaves(e) then
11         e_2 ← CreateEntry(e[2].getPoint(), e[4].getPoint(), e[5]);
12         if Cand.size() < K then
13             Cand.insert(e_2);
14         else if e[5] < Cand.Max() then
15             Cand.delete();
16             Cand.insert(e_2);
17     else if hasChildren(e) then
18         forall pair of (child h_1 ∈ e[2] and h_2 ∈ e[4] ) do
19             distance ← MINMINDIST(h_1.quad(), h_2.quad());
20             if (Cand.size() < K) OR (distance < Cand.Max()) then
21                 e_2 ←
                    CreateElementQueue(h_1, h_1.quad(), h_2, h_2.quad(), distance, e[6]+1);
22                 pQueue.insert(e_2);
23 return Cand
```

El algoritmo busca los K-pares de puntos más cercanos entre R y S mientras el heap *pQueue* no esté vacío (Líneas 6-22). Cada vez que obtiene un nuevo elemento e desde *pQueue* existen tres posibilidades:

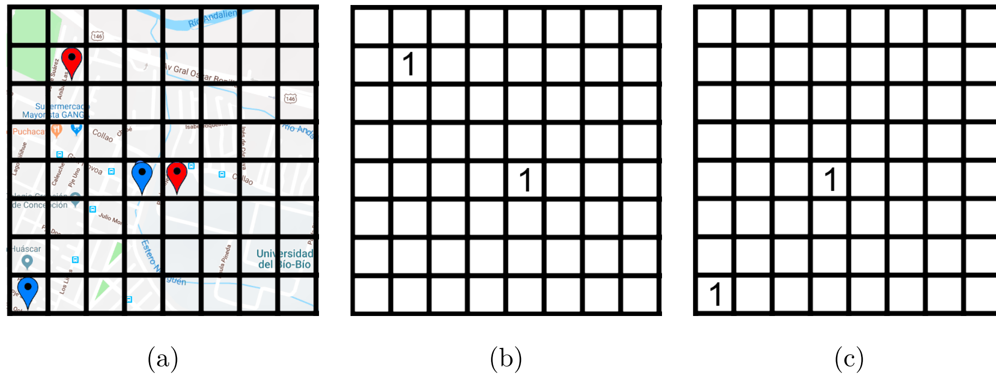
1. El número de candidatos K se ha alcanzado y el elemento e obtenido del tope de *pQueue* tiene una distancia mayor que el elemento en el tope de *Cand* (Línea 8). En este caso, el algoritmo termina retornando *Cand*.

2. Se evalúa si el elemento e es un par de hojas con la función $areLeaves(e)$ (Línea 10). Esta función determina si el elemento e es un par de hojas en los k^2-tree de R y S, para ello, revisa el sexto campo (nivel) del elemento e. Si el nivel es mayor a la altura de R y S, entonces el elemento e corresponde a un par de hojas, por lo tanto, el elemento e es un posible candidato y se puede insertar en *Cand*. La función *createEntry* crea un elemento candidato e_2, para ser insertado en *Cand* (Línea 11). Para ello utiliza la función *GetPoint*, que retorna uno de los puntos extremos del cuadrante de R o S en e, según corresponda. El elemento e_2 tiene tres campos: (i) punto1: que corresponde al punto obtenido de uno de los extremos del cuadrante de R en e, (ii) punto2: que corresponde al punto obtenido de uno de los extremos del cuadrante de S en e y (iii) dist: que corresponde a la distancia mínima (*MINMINDIST*) entre los cuadrantes de e. Entonces, si el número de candidatos es menor a K (no se han completado todos las respuestas) (Línea 12), e_2 se inserta en *Cand*. En caso contrario, si la distancia del elemento e es menor que la distancia del elemento en el tope de *Cand* (Línea 14), es decir, e es mejor candidato que el elemento que está en el tope de *Cand*, entonces el elemento en el tope se elimina y se inserta el elemento e_2.

3. El elemento e es un nodo con hijos (Línea 17), evaluado por la función *hasChildren*(e) que verifica si el elemento e tiene hijos, mediante la observación del sexto campo de e (nivel). Si el nivel de e es menor a la altura del k^2-tree de R y S, entonces e es un nodo intermedio (en ambos k^2-trees) y por lo tanto, posee hijos. Entonces para cada hijo $h_1 \in e[2]$ y $h_2 \in e[4]$ del elemento e, es decir, por cada sub-cuadrante del cuadrante padre de $e[2]$ y $e[4]$, y solo aquellos sub-cuadrantes en la cual una operación de tipo *Rank* entregue como resultado un 1, el algoritmo obtiene la mínima de las mínimas distancias entre h_1 y h_2 (Líneas 18 y 19). Luego, si el número de candidatos es menor que K o la distancia mínima calculada del par de hijos de R y S en e es menor que la distancia del elemento en el tope de *Cand*, es posible que se haya encontrado un nuevo candidato en el par de sub-árboles, y por lo tanto, el nuevo elemento e_2 se agrega a *pQueue* (Líneas 21-22).

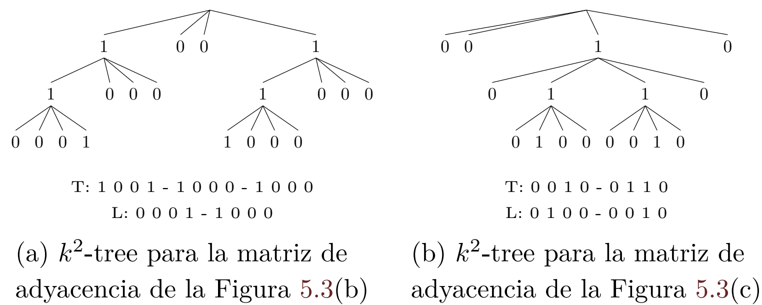
El Ejemplo 5.2 ilustra el funcionamiento del Algoritmo 2, la clave (R,S,$dist$) presente en el ejemplo 5.2, representa al objeto que almacena los cuadrantes de R y S, respectivamente, y *Dist* corresponde a la mínima de las distancias mínimas (*MINMINDIST*) entre ambos cuadrantes.

Ejemplo 5.2 Considere la Figura 5.3(a) que muestra un mapa con 4 puntos de interés pertenecientes a dos conjuntos de puntos identificados por color (rojos y azules). La Figura 5.3(b) muestra la matriz de adyacencia que representa los puntos rojos (conjunto de puntos R) y la Figura 5.3(c) muestra la matriz de adyacencia que representa los puntos azules (conjunto de puntos S). Con estas matrices de adyacencia se construyen dos k^2-trees (Figura 5.4) que son utilizados por el Algoritmo 2 para obtener los $K = 2$ pares de



(a) (b) (c)

Figura 5.3: Puntos de interés en un mapa y sus respectivas matrices de adyacencia R y S



(a) k^2-tree para la matriz de adyacencia de la Figura 5.3(b)

(b) k^2-tree para la matriz de adyacencia de la Figura 5.3(c)

Figura 5.4: k^2-trees para las matrices de adyacencia ilustradas en la Figura 5.3

puntos (R, S) entre ambos conjuntos. Inicialmente *pQueue* contiene a los dos k^2-trees R y S, esto es, dos matrices, las cuales ambas por igual tienen como puntos extremos a los puntos (1,1) y (8,8). Así, de acuerdo a la fórmula para calcular la mínima de las mínimas distancias (*MINMINDIST*) entre R y S, da como resultado 0 dado que ambas se solapan o intersectan totalmente (mismas coordenadas), por lo tanto, el heap *pQueue* queda como $pQueue = R, S(0)$. Los K=2 pares de puntos candidatos más cercanos serán almacenados en el heap *Cand* que inicialmente está vacío. El proceso para obtener esos pares de puntos es el siguiente:

1. El ciclo iterativo es inicializado (Línea 6), entonces el algoritmo elimina el elemento que está en el tope del heap *pQueue* (Línea 7) que corresponde a los k^2-trees R y S. Dado que ambos k^2-trees están recién siendo explorados, es decir, los elementos $e[2]$ y $e[4]$ no son hojas (Línea 17), R tiene dos hijos con valor 1 y S tiene un solo hijo con valor 1, y no existen elementos en *Cand* (Línea 20), entonces todos los cuadrantes son insertados *pQueue* (Línea 22). De esta manera, *pQueue* es actualizado con nuevos valores, y ahora queda de la siguiente manera $pQueue = \{(R_A, S_C, 1), (R_D, S_C, 1)\}$ (ver Figura 5.5(a)). Inicialmente, el par de cuadrantes $(R_A, S_C, 1)$ están el el tope de *pQueue* dado que fueron los primeros en ser insertados en el heap *pQueue*. Por

lo tanto, los pares de puntos candidatos deben ser buscados en el primer par de cuadrantes $(R_A, S_C, 1)$.

2. El algoritmo retorna (Línea 7) y procesa el par de cuadrantes $(R_A, S_C, 1)$. Como el par de cuadrantes $(R_A, S_C, 1)$ posee hijos (Línea 17), y no se ha alcanzado la cantidad de candidatos K, entonces todos los hijos se insertan en *pQueue* (Línea 22) (ver Figura 5.5(b)).

3. El algoritmo retorna (Línea 7) y toma el elemento en el tope del heap *pQueue*, que contiene los pares de cuadrantes $(R_D, S_C, 1)$. Como estos pares de cuadrantes poseen hijos (Línea 17), y aún no se ha cumplido la cantidad de candidatos K, entonces también se agregan sus hijos a *pQueue* (ver Figura 5.5(c)).

4. El algoritmo retorna (Línea 7) y elimina del tope del heap *pQueue* al elemento $(R_{D1}, S_{C2}, 1)$. Puesto que aún no se obtienen los K-pares de puntos candidatos (elementos en *Cand*) y tampoco el elemento en el tope de *pQueue* es un par de hojas, entonces se agregan a *pQueue* los hijos del par de cuadrantes $(R_{D1}, S_{C2}, 1)$ (ver Figura 5.5(d)). Luego *pQueue* es actualizado ubicando al elemento con la menor distancia entre pares de cuadrantes, correspondiente a las hojas $(R_{D11}, S_{C22}, 1)$ con la menor distancia igual a 1.

5. El algoritmo retorna (Línea 7), se obtiene el elemento en el tope de *pQueue* que corresponde a un par de hojas, y como el heap *Cand* aún se encuentra vacío (Línea 12), entonces el par de hojas $(R_{D11}, S_{C22}, 1)$ se convierten en posibles candidatas, y por tanto, se inserta un nuevo elemento e_2 con las coordenadas del primer y segundo elemento $e[2]$ y $e[4]$ en el heap $Cand = \{(R_{D11}, S_{C22}, 1)\}$, junto con la distancia entre ellos (en este caso la distancia es 1), obteniendo así el primer par de puntos más cercanos entre los conjuntos R y S (ver Figura 5.5(e)).

6. El algoritmo retorna (Línea 7), se obtiene el elemento en el tope de *pQueue*, que corresponde al par de cuadrantes $(R_{D1}, S_{C3}, 3{,}16)$, los cuales poseen hijos, correspondientes al par de hojas $(R_{D11}, S_{C33}, 5)$ que también son insertados en *pQueue* dado que aún no se encuentran los K candidatos (ver Figura 5.5(f)).

7. El algoritmo retorna (Línea 7), toma el elemento en el tope de *pQueue* que contiene los pares de cuadrantes $(R_{A1}, S_{C2}, 3{,}16)$ que también poseen hijos, el par de hojas $(R_{A14}, S_{C22}, 3{,}605)$, que son insertadas en *pQueue* y ubicadas en el tope del heap *pQueue*, ya que poseen la menor distancia respecto del resto de los pares y no se han encontrado aún los K candidatos (ver Figura 5.5(g)).

8. El algoritmo retorna (Línea 7), toma el elemento en el tope de *pQueue*, el cual corresponde al par de hojas $(R_{A14}, S_{C22}, 3{,}605)$ y dado que aún no se cumplen los K candidatos, entonces se crea un nuevo elemento e_2, con las coordenadas del primer y segundo elemento $e[2]$ y $e[4]$, y este último se inserta en el heap $Cand = \{(R_{A14}, S_{C22}, 3{,}605), (R_{D11}, S_{C22}, 1)\}$, obteniendo así, el segundo par de puntos más cercanos (ver Figura 5.5(h)).

9. El algoritmo retorna (Línea 7), toma el elemento en el tope de *pQueue*. Como ya existen $K = 2$ pares de puntos más cercanos, y la distancia del elemento en el tope de *pQueue*, la cual es 5, es mayor a la distancia del elemento en el tope del heap *Cand*, actualmente 3.605, entonces el algoritmo finaliza y retorna *Cand*. □

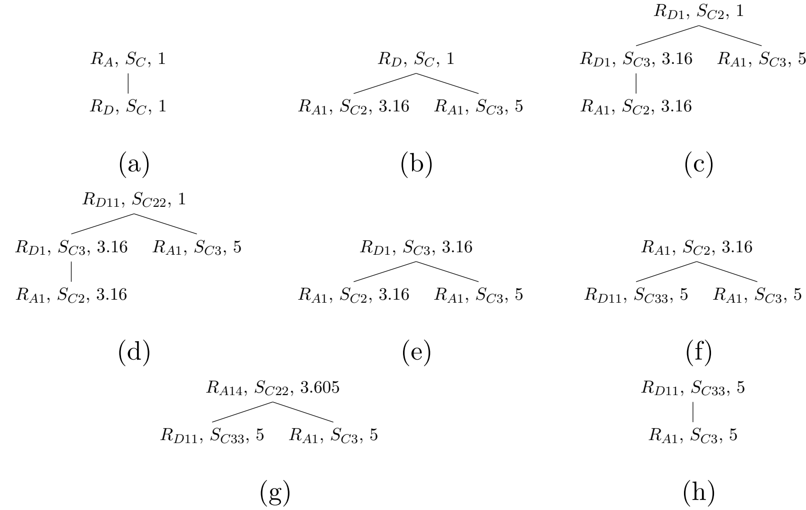


Figura 5.5: Heap Mínimo *pQueue* para el Ejemplo 5.2

5.3. Análisis de Complejidad Temporal de los Algoritmos KNN y KCPQ

En esta sección se explica la complejidad temporal en términos de tiempo en el peor de los casos, para los algoritmos KNN y KCPQ que operan sobre la estructura compacta k^2-tree.

Inicialmente, para ambos algoritmos, la cantidad de nodos internos de un k^2-tree usados para representar m puntos en una matriz de adyacencia M de $n \times n$ es la siguiente:

$$\sum_{i=0}^{\log_k n-1} (k^2)^i \tag{5.3}$$

En el peor de los casos, es decir, la matriz M está completa con 1's, se necesitan $\frac{(n^2-1)}{(k^2-1)}$ nodos, lo que corresponde finalmente a una complejidad en espacio de $O(m)$, es decir, necesitamos n nodos para representar m puntos. Por lo tanto, la complejidad temporal para el algoritmo KNN se determina por:

$$\overbrace{m(\log_2 m + k^2 \log_2 m)}^{\text{Procesamiento de nodos}} + \overbrace{m(\log_2 m + 2\log_2 K)}^{\text{Procesamiento de Puntos}} \tag{5.4}$$

Donde m corresponde al número de puntos que están en la matriz de adyacencia M y se considera un valor de k=2 (particiones sobre la matriz de adyacencia). La primera parte de la ecuación, corresponde al recorrido interno sobre los nodos de la estructura compacta k^2-tree. Específicamente cuando se saca un elemento del heap *pQueue* (Algoritmo 1, Línea 6) equivale a $\log_2 m$, más las inserciones de los k^2-hijos de los nodos internos analizados sobre *pQueue*, lo que equivale a $k^2 \log_2 m$. Por otra parte, está también el procesamiento de los puntos (cuando se llega a los nodos hojas), los que también se deben sacar del tope del heap *pQueue*, lo que equivale a $\log_2 m$ más las inserciones y eliminaciones del heap *Cand*, lo que equivale a $2\log_2 K$. Finalmente, el algoritmo KNN tiene una complejidad temporal de $O(m \log_2 m)$.

Para el caso del algoritmo KCPQ, la complejidad temporal se determina por:

$$\overbrace{(m_1 \times m_2)(\log_2(m_1 \times m_2) + k^4 \log_2(m_1 \times m_2))}^{\text{Procesamiento de nodos}} + \overbrace{(m_1 \times m_2)(\log_2(m_1 \times m_2) + 2\log_2 K)}^{\text{Procesamiento de Puntos}} \tag{5.5}$$

Donde m_1 equivale a los puntos del primer conjunto R y m_2 equivale a los puntos del segundo conjunto S. Semejante a la explicación de la complejidad en tiempo del algoritmo KNN, la complejidad en tiempo del algoritmo KCPQ agrega a la complejidad la operación sobre dos matrices. De esta manera $(m_1 \times m_2)$ explica la operación de encontrar los puntos que se encuentran en la primera matriz contra los puntos almacenados en la segunda matriz. El procesamiento de los nodos internos es similar a KNN, se deben sacar del heap *pQueue* $\log_2(m_1 \times m_2)$ nodos más $k^4 \log_2(m_1 \times m_2)$ inserciones sobre el mismo heap, donde k^4 corresponde a los k^2 hijos de ambas matrices. Por otra parte, también esta el procesamiento de los nodos hojas. Nuevamente, de forma similar a la complejidad del algoritmo KNN, se deben quitar $\log_2(m_1 \times m_2)$ pares de puntos del heap *pQueue* más $2\log_2 K$ inserciones en el heap *Cand*. Finalmente, el algoritmo KCPQ tiene una complejidad temporal de $O((m_1 \times m_2)\log_2(m_1 \times m_2))$.

5.4. Algoritmo para Calcular KNN sin EDCs

En esta sección se explica el algoritmo para calcular los K vecinos más cercanos sin el uso de estructuras de datos compactas, el cual corresponde al Algoritmo 3. Este algoritmo opera sin indexación, es decir, se le debe entregar una lista de puntos y se trabaja con ellos directamente en memoria RAM. De esta manera, el algoritmo recibe como entrada una lista de puntos P, un punto de consulta q y una constante K, que indica la cantidad de vecinos a encontrar. Para realizar el cálculo, utiliza un heap de máximos llamado *maxHeap*, el cual almacena los K puntos con menor distancia respecto a q. La operación de este

algoritmo es simple y directa. Se llena el heap *maxHeap* con valores iniciales, y luego se recorre la lista completa de puntos buscando las K menores distancias. Para esto último, el heap *maxHeap*, juega un rol fundamental, dado que permite ordenar eficientemente los K potenciales candidatos. El algoritmo realiza los siguientes pasos:

1. Inicia un primer ciclo iterativo (Línea 2) el cual permite llenar el heap *maxHeap* con los K primeros elementos de la lista de puntos P.

2. Luego, el segundo ciclo iterativo (Línea 7) permite encontrar los K puntos más cercanos a q. Para esto, se recorren los restantes puntos de la lista P. Por cada iteración, se obtiene un punto p de P (Línea 8), se calcula la distancia Euclidiana de p a q (Línea 9) y se evalúa si la distancia de p a q es menor a la distancia que se encuentra en el tope de *maxHeap* (Línea 11). En caso de ser afirmativo, se remueve el tope de *maxHeap* y se inserta p en *maxHeap* (Líneas 12-13).

3. El algoritmo finaliza (Línea 14) cuando se evalúan todos los puntos en P, y retorna los K elementos que están en *maxHeap*.

Algoritmo 3: Obtener los K-vecinos más cercanos al punto q sin EDCs

input : List of points P, a constant K, a point q
output: A set of K points nearest to q

```
   /*Initially a maximum heap is created                          */
1  maxHeap ← CreateMaxHeap() ;           // Store points Point(x, y, dist)
2  for i = 0; i < K; i++ do
3  |   p ← P.get(i);
4  |   dist ← EuclideanDistance(p, q);
5  |   p.setDistance(dist);
6  |   maxHeap.add(p);
7  for i = K; i < P.length; i++ do
8  |   p ← P.get(i);
9  |   dist ← EuclideanDistance(p, q);
10 |   p.setDistance(dist);
   |   /*peek() retrieves but not remove top elements of the heap   */
11 |   if maxHeap.peek() > dist then
12 |   |   maxHeap.delete();
13 |   |   maxHeap.add(p);
14 return first_K(maxHeap, K)
```

La solución planteada presenta una complejidad de $O(K \log_2 K + (n - K) \log_2 K)$, en donde $K \log_2 K$ equivale al cálculo del primer ciclo del algoritmo, y $(n - K) \log_2 K$ al segundo ciclo iterativo, siendo $\log_2 K$ el costo de insertar en el heap. Finalmente, la complejidad temporal del algoritmo es $O(n \log_2 K)$, donde n es la cantidad de puntos.

5.5. Algoritmo para Calcular KCPQ sin EDCs

En esta sección se explica el algoritmo para calcular los K-pares de vecinos más cercanos sin el uso de estructuras de datos compactas. Para ello se utilizó una implementación propuesta en (Sedgewick y Wayne, 2011)[2] que calcula el par de vecinos más cercanos en un conjunto de puntos (CPQ), y se modificó para que operara sobre dos conjuntos de puntos en una cota superior de $O(n \log n)$. La implementación original ocupa la técnica divide y vencerás para buscar el par de vecinos más cercanos en un conjunto de puntos de la siguiente manera:

1. Se ordenan los puntos de acuerdo a la coordenada x.
2. Se divide el conjunto de puntos en dos subconjuntos de igual tamaño por una línea vertical $x = x_{mid}$.
3. Se resuelve el problema recursivamente tanto para el conjunto de puntos izquierdo como para el conjunto de puntos derecho. Esto produce las distancias mínimas D_{Lmin} para el conjunto de puntos izquierdo y D_{Rmin} para el conjunto de puntos derecho.
4. Luego se busca la mínima distancia D_{LRmin} entre el conjunto de pares de puntos, en los que un punto se encuentra a la izquierda de la división vertical y el otro punto a la derecha.
5. La respuesta final para el par de vecinos más cercanos es el mínimo entre: D_{Lmin}, D_{Rmin}, D_{LRmin}.

Para responder a la consulta KCPQ se procedió a crear un algoritmo que utiliza en parte la solución planteada en el algoritmo CPQ. Este nuevo algoritmo funciona con la técnica divide y vencerás, por tanto su funcionamiento es recursivo. De esta manera, el algoritmo recibe como entrada dos listas de puntos R y S, y una constante K que indica la cantidad de pares de puntos a encontrar. Para realizar el cálculo, el algoritmo ordena por el eje x a los puntos en R y S. En la sección recursiva el algoritmo define como caso base que una de las listas posea un solo elemento. En este caso, se calculan las distancias entre ese único punto y todos los puntos de la otra lista para guardar en el heap los candidatos. Se considera candidato aquel par de puntos cuya distancia es menor a la mayor distancia en el tope del heap. El heap almacena como máximo K-pares de puntos. En el caso general (ambas listas tienen más de un punto) se dividen ambas listas en 2 sub-listas cada una (R_1, R_2, S_1 y S_2). Luego se realizan llamadas recursivas para cada combinación posible:

- R_1 contra S_1
- R_1 contra S_2
- R_2 contra S_1
- R_2 contra S_2

Con lo anterior el algoritmo calcula todas las distancias posibles entre cada elemento de R y S. Sin embargo, se incluye un criterio de poda el cual calcula la menor distancia

[2] Código disponible en `https://algs4.cs.princeton.edu/99hull/`

posible (solo en el eje x) entre los elementos de la dos listas. Si esta distancia mínima entre los conjuntos es mayor que la peor distancia almacenada en el heap, entonces se descarta la llamada recursiva. En detalle, el nuevo Algoritmo 4 realiza los siguientes pasos:

1. Se ordenan los puntos de R y S de acuerdo a la coordenada x (ordenamiento fuera de la recursión) (Líneas 2-3).

2. Se invoca a la función recursiva *closest*() que permite encontrar los K-pares de vecinos más cercanos. Esta función recibe como parámetro, un arreglo con los puntos de R y S, el tamaño del arreglo de R y S, K y un entero desde donde se inicia la búsqueda, además de los indices de comienzo y corte para cada conjunto (Línea 6).

3. Se evalúa el caso base de la función recursiva, la cual corresponde a que uno de los dos sub-arreglos posee sólo un elemento. Entonces, se calculan las distancias entre el sub-arreglo menor contra todos los puntos que posee el otro sub-arreglo (Líneas 6 a 21).

4. Luego se realiza una poda en la función *closest*():

   a) Todos elementos de R son mayores que S en el eje x y la menor distancia entre los sub-arreglos es mayor que la mejor distancia encontrada a ese momento (Línea 2).

   b) Todos los elementos de R son menores que S en el eje x y la menor distancia entre los sub-arreglos es mayor que la mejor distancia encontrada a ese momento (Línea 4).

5. Se calculan las mitades de los arreglos R y S, correspondientes a *midA* y *midB*, sobre los cuales se ejecutan los siguientes pasos (Líneas 22-23).

6. Se hace la llamada recursiva para (Línea 23):

   a) La mitad izquierda del arreglo R contra la mitad izquierda de S.

   b) La mitad izquierda de R contra la mitad derecha de S.

   c) La mitad derecha de R contra la mitad izquierda de S.

   d) La mitad derecha de R contra la mitad derecha de S.

7. Finalmente, una vez terminado el procesamiento se retorna los primeros K elementos del heap *maxHeap* (Línea 7).

La solución planteada presenta una complejidad de $O(n_1 \times n_2) + (n_1 \log_2 n_1 + n_2 \log_2 n_2)$, con n_1 el tamaño (cantidad de puntos) del primer conjunto R y n_2 el tamaño (cantidad de puntos) del segundo conjunto S, donde $(n_1 \times n_2)$ es el costo de calcular la distancia de un punto $l \in n_1$ con todos los puntos de n_2, y $(n_1 \log_2 n_1 + n_2 \log_2 n_2)$ que corresponde a las inserciones en el heap para cada conjunto (siendo $\log_2 n$ el costo de insertar n puntos en un heap).

Algoritmo 4: OBTENER LOS K-PARES DE VECINOS MÁS CERCANOS SIN EDCs

```
input  : Two lists of points R and S
output: A set of K pair of points of the form (r,s)
1 maxHeap ← CreateMaxHeap() ;             /*Initially max heap is created*/
2 pointsA ← sort(R) ;                      /*R are sorted by x-coordinate*/
3 pointsB ← sort(S) ;                      /*S are sorted by x-coordinate*/
4 nA ← pointsA.length;
5 nB ← pointsB.length;
6 closest(pointsA, 0, nA-1, pointsB, 0, nB-1, K) ;   /*Call a recursive function*/
7 return first_K(maxHeap, K)
```

Algoritmo 5: FUNCIÓN BESTDISTANCE

```
1 Function bestDistance(K):
2     if maxHeap.size() < K then
3         return ∞
4     point ← maxHeap.peek() ;     /*peek() retrieves the head element from
       maxHeap*/
5     dist ← point.getDistance();
6     return dist
```

5.6. Librerías Externas

En la presente sección se explican las librerías investigadas y utilizadas para generar una librería en JAVA, que contiene las consultas implementadas en esta tesis, para que pueda ser utilizada por la comunidad académica o la industria. En particular, la Sección 5.6.1 ilustra una revisión sobre librerías que implementan ciertas funcionalidades requeridas para dar respuesta a la consultas KNN y KCPQ, además de ser necesarias para generar y utilizar la estructura compacta k^2-tree. La Sección 5.6.2 explica en detalle la librería generada en esta tesis, los algoritmos implementados, las estructuras de datos utilizadas, ejemplos de uso, entre otros.

5.6.1. Librerías que Implementan Operaciones Sobre Estructuras Compactas

Actualmente, el conjunto de librerías oficiales de JAVA no posee una manera eficiente de almacenar una secuencia de bits (bitmaps) y hacer consultas de tipo *Rank* y *Select* sobre ellos. Sin embargo, existen algunas, tales como la denomina Sux[3], que implementa en lenguaje JAVA bitmaps y consultas *Rank* y *Select* (Vigna, 2008). Se realizaron pruebas sobre esta librería, sin embargo, los resultados no fueron satisfactorios, ya que se encontraron errores en algunas consultas de tipo *Rank*. Además los tiempos de ejecución obtenidos en

[3] http://sux.di.unimi.it/

Algoritmo 6: FUNCIÓN CLOSEST

```
1  Function closest(pointsA, loA, hiA, pointsB, loB, hiB, K):
       /*Pruning                                                          */
2      if pointsA[loA].x() > pointsB[hiB].x() and
        pointsA[loA].x() − pointsB[hiB].x() > bestDistance(K) then
3          return;
       /*Pruning                                                          */
4      if pointsA[hiA].x() < pointsB[loB].x() and
        pointsB[loB].x() − pointsA[hiA].x() > bestDistance(K) then
5          return;
       /*Base case                                                        */
6      if loA == hiA then
7          for i = loB; i <= hiB; i++ do
8              distance ← pointsA[loA].distanceTo(pointsB[i]);
9              if distance < bestDistance(K) then
10                 maxHeap.add(PointPair(pointsA[loA], pointsB[i], distance));
11                 if maxHeap.size() > k then
12                     maxHeap.poll() ;   /*poll() retrieves and remove the head
                        from maxHeap*/
13         return;
       /*Base case                                                        */
14     if loB == hiB then
15         for i = loA; i <= hiA; i++ do
16             distance ← pointsB[loB].distanceTo(pointsA[i]);
17             if distance < bestDistance(K) then
18                 maxHeap.add(PointPair(pointsB[loB], pointsA[i], distance));
19                 if maxHeap.size() > k then
20                     maxHeap.poll();
21         return;
22     midA ← loA + (hiA − loA)/2 ;                     /*Half calculated for A*/
23     midB ← loB + (hiB − loB)/2 ;                     /*Half calculated for B*/
       /*Recursion                                                        */
24     closest(pointsA, loA, midA, pointsB, loB, midB, k);
25     closest(pointsA, loA, midA, pointsB, midB + 1, hiB, k);
26     closest(pointsA, midA + 1, hiA, pointsB, loB, midB, k);
27     closest(pointsA, midA + 1, hiA, pointsB, midB + 1, hiB, k);
```

consultas sobre conjuntos de datos sintéticos de prueba no eran constantes, lo cual contradice las investigaciones sobre tiempos de ejecución contantes para las operaciones de *Rank* y *Select*, como por ejemplo las presentadas en (Fariña et al., 2009) y (Golynski et al.,

2007).

La Tabla 5.2 muestra una comparación entre las librerías SUX y RG-C++[4], esta última presentada en (Claude y Navarro, 2008) e implementada mediante el lenguaje C++, la cual entrega valores de tiempos de ejecución constantes en sus respuestas. Para este experimento, se ejecutaron operaciones *Rank* y *Select* sobre bitmaps (N) de distintos tamaños $1.000, 10.000, 100.000$ y $1.000.000$. Se utilizó un computador con sistema operativo Linux, procesador Intel Core I5, con 4GB de RAM y 256 GB de disco duro SSD.

Tabla 5.2: Comparación de tiempos entre librerías en segundos

N	RG-C++	SUX
1.000	0,07	0,05
10.000	0,06	0,16
100.000	0,06	0,49
1.000.000	0,07	3,15

Dado el mal rendimiento de la librería SUX, se decidió implementar una nueva librería en JAVA basada en las implementaciones de la librería RG-C++ que se explica en detalle en la Sección 5.6.2.

5.6.2. Librería ALBA-k^2-tree

La librería *ALBA*-k^2-tree representa uno de los productos de esta tesis. La decisión de implementar una librería en lenguaje JAVA obedece a la necesidad de dejar un producto útil a la comunidad, tanto para aquellas personas que se dedican a la investigación, y para aquellos que trabajan en la industria de desarrollo de software. Por otra parte, el lenguaje elegido permite la posibilidad de ampliar las características implementadas a otras plataformas, como por ejemplo: dispositivos con sistema operativo Android, plataformas WEB desarrolladas en JAVA (utilizadas ampliamente en la industria) y el Framework APACHE SPARK[5], de manera que con este último se pueda migrar los algoritmos implementados a ambientes de trabajo distribuido. El uso de APACHE SPARK es una interesante alternativa para mejorar la construcción de la estructura compacta k^2-tree. Actualmente, se utiliza el algoritmo para la construcción propuesto en (Brisaboa et al., 2009), el cual utiliza un tiempo lineal en base al tamaño de la matriz de adyacencia utilizada, es decir, de $O(n^2)$. Por lo tanto, el tiempo de construcción de la estructura compacta puede ser costoso si existen demasiados datos en ella.

De acuerdo a experimentos preliminares, en nuestra librería las operaciones de *Rank* y *Select* se ejecutan en tiempos cercanos a los entregados por la librería RG-C++.

La Figura 5.6 describe la estructura interna de la librería ALBA-k^2-tree, que se compone de los siguientes paquetes:

[4] https://github.com/fclaude/libcds/blob/master/src/static/bitsequence/BitSequenceRG.cpp
[5] https://spark.apache.org/

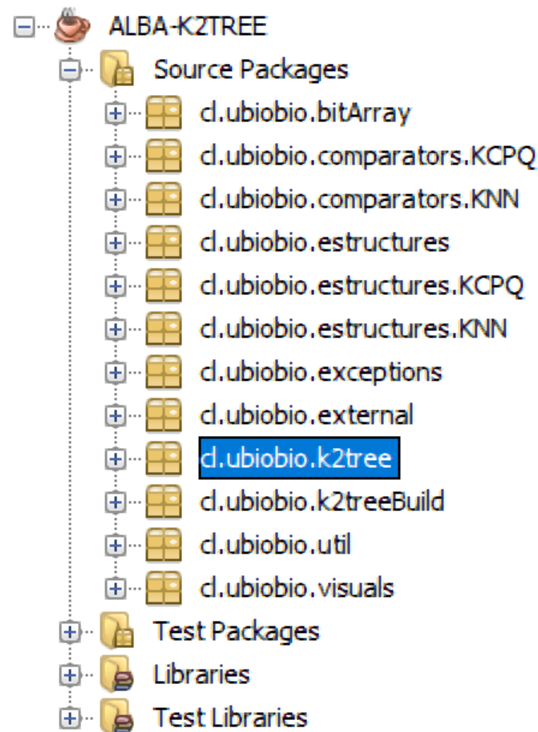


Figura 5.6: Estructura interna de la librería ALBA-k^2-tree

- Paquete bitArray: Está compuesto por dos clases que implementan los BitArrays utilizados por la estructura de datos compacta k^2-tree.
    - La clase BitArray: Implementa un bitmap básico, es decir, un arreglo que permite almacenar bits. Además provee operaciones de inserción, recuperación y seteo de los bits específicos dentro del arreglo de bits.
    - La clase BitArrayRS: Permite crear un array de bits estáticos con soporte para las operaciones de *Rank* y *Select*. La operación de *Rank* añade un 5 % más de espacio al tamaño del bitmap. Esta operación de *Rank* utiliza solamente un nivel para hacer los pre-cálculos para dar respuestas en tiempos constantes.
- Paquete comparators.KCPQ y comparators.KNN: Clases en JAVA que implementan la clase genérica *Comparator*. Estas clases se utilizan para configurar el heap *pQueue* y *Cand*, explicados en las Secciones 5.1 y 5.2 como heap de mínimos y máximos.
- Paquete estructures: Contiene la clase JAVA que modela un cuadrante. Este cuadrante está compuesto por 2 objetos de tipo Punto.
- Paquete estructures.KCPQ: Contiene las clases que permiten modelar las estructuras utilizadas para dar respuesta a la consulta KCPQ.

- Paquete estructures.KNN: Contiene las clases que permiten modelar las estructuras utilizadas para dar respuesta a la consulta KNN.

- Paquete exceptions: Contiene clases que permiten manejar y lanzar excepciones propias a situaciones anormales de funcionamiento en el código.

- Paquete external: Este paquete contiene las clases que hacen de interface para que externos puedan utilizar la librería. En específico contiene una clase llamada *CompactEstructure* que permite crear un k^2-tree entregando como parámetro un archivo de texto con los puntos, llamar a las consulta de proximidad espacial KNN y KCPQ, obtener todos los puntos de la estructura compacta, tamaño en bytes de la estructura compacta, entre otras.

- Paquete k2tree: Contiene la implementación de la estructura de datos k^2-tree usando los bitmaps expuestos en el paquete bitArray. Además tiene implementaciones para la consulta por rango, consultar por un nodo específico en la estructura k^2-tree, acceder a un hijo, obtener un hijo, entre otras. El paquete además incluye de las clases que implementan las consultas KCPQ y KNN, y la clase *Punto*.

- Paquete k2treeBuild: Contiene las clases que permiten construir un k^2-tree a partir de una lista de puntos o desde un archivo de texto con puntos. La implementación utilizada para construir el k^2-tree está basada en el pseudo-código presentado en (Brisaboa et al., 2009).

- Paquete util y visuals: Contiene clases de ayuda para hacer pruebas, medición de rendimiento en tiempo, y utilidades para visualizar matrices de adyacencia.

La librería puede ser descargada desde el sitio WEB de la Universdad del Bío-Bío `http://arrau.chillan.ubiobio.cl/fernando/`, en donde se encuentran los siguientes documentos:

- Un archivo llamado ALBA-K2TREE.jar, que corresponde a la librería lista para ser utilizada en algún ambiente que utilice el lenguaje JAVA.

- Un archivo llamado ALBA-K2TREE-24-8-2017.zip en el cual se encuentran los fuentes para ser revisados y utilizados por la comunidad.

Actualmente, la librería ha sido utilizada en dos proyectos de título de estudiantes de la Universidad del Bío-Bío. El primer proyecto[6] corresponde a la implementación visual de la consulta KNN a través de un sitio WEB (ver Figura 5.7) y una aplicación móvil (ver Figura 5.9(a)) que permiten consultar los K-vecinos más cercanos sobre puntos que corresponden a centros de salud de la provincia de Concepción. La Figura 5.7 muestra el punto de consulta en color amarillo y el resultado de la consulta KNN con puntos de color verde.

[6] `https://dsi.face.ubiobio.cl/sistemaknn/`

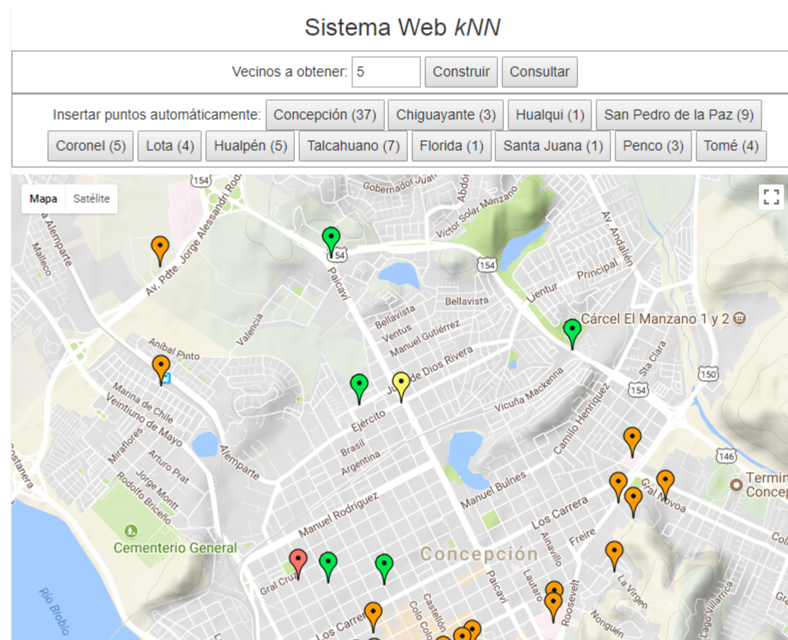


Figura 5.7: Interface WEB para consultar los K-vecinos más cercanos

El segundo proyecto[7] corresponde a la implementación visual de la consulta KCPQ, a través de un sitio WEB (ver Figura 5.8) y una aplicación móvil (ver Figura 5.9(b)) que permiten consultar los K-pares de vecinos más cercanos sobre conjuntos de puntos elegidos por el usuario, como hoteles, restaurantes, colegios entre otros. En la Figura 5.8 se puede apreciar puntos de interés en color rojo, correspondientes a gimnasios y puntos de interés en color azul que identifican a estaciones de servicio. De esta manera, aparece dibujado con líneas en color verde el resultado de la consulta KCPQ para los K=4 pares de vecinos más cercanos. La Figura 5.9 (b) ilustra la interfaz del software desarrollado en Android para consultar los K-pares de vecinos más cercanos.

Para dar solución a ambas implementaciones, se desarrollaron dos servicios WEB en lenguaje JAVA para que cada uno de estos proyectos hicieran uso de las consultas KNN y KCPQ respectivamente. La Figura 5.10 ilustra como una aplicación WEB o una aplicación móvil hace uso de la librería (el archivo .jar) que está montada en un servidor WEB GlassFish[8]. De esta manera se consultan y se da respuesta a las peticiones del lado del cliente a través de la tecnología Javascript Object Notation (JSON[9]).

[7] `https://dsi.face.ubiobio.cl/kcpqsystem/`
[8] `http://www.oracle.com/technetwork/es/middleware/glassfish/overview/index.html`
[9] `https://www.json.org/`

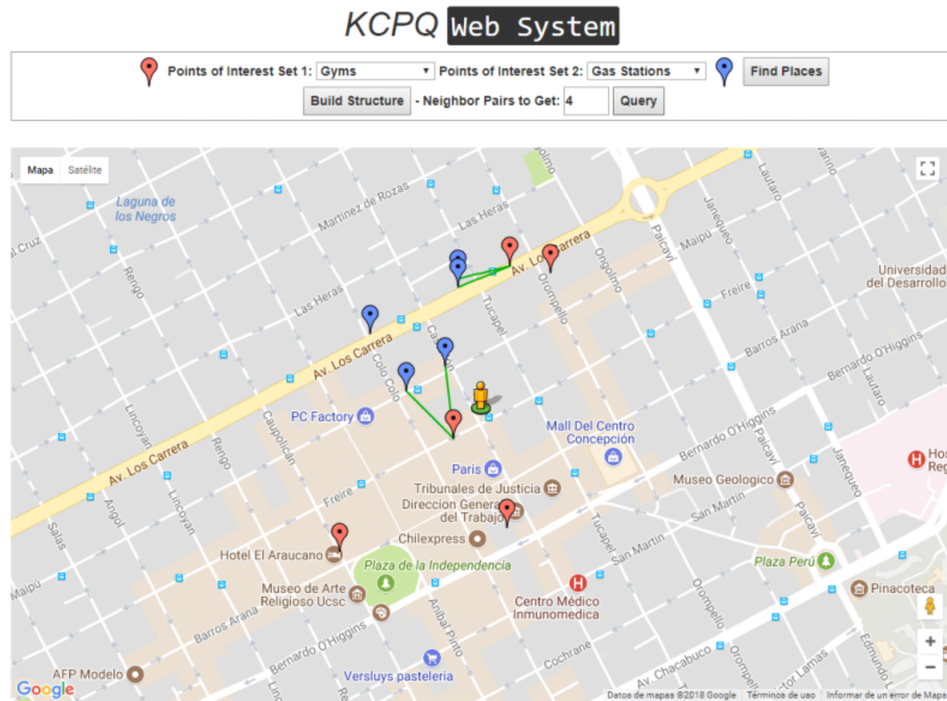
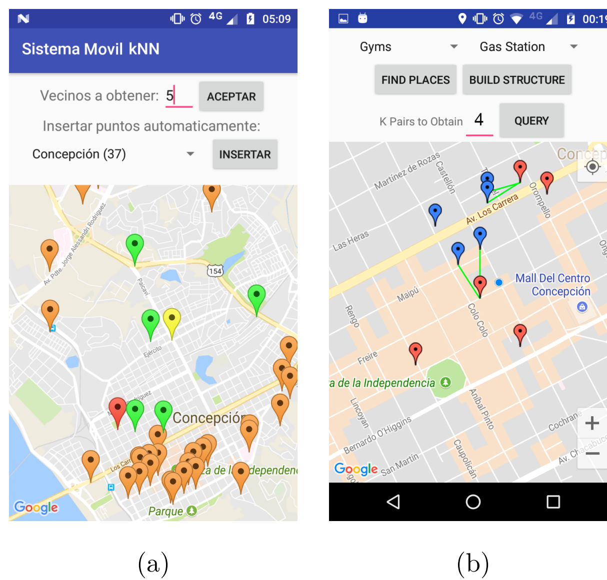


Figura 5.8: Interface WEB para consultar los K-pares de vecinos más cercanos



(a) (b)

Figura 5.9: Interfaces Android para consultas KNN y KCPQ

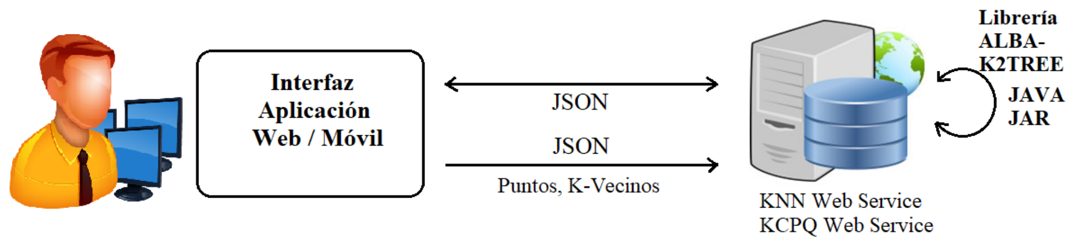


Figura 5.10: Representación del funcionamiento de los servicios WEB para KNN y KCPQ

Capítulo 6

Experimentación

En este capítulo se presentan los experimentos realizados para probar la eficiencia de las consultas de proximidad KNN y KCPQ. En la Sección 6.1 se detalla el escenario en donde se ejecutaron los experimentos. La Sección 6.2 presenta el diseño de los conjuntos de datos para ser utilizados en los experimentos.

La Sección 6.3 muestra los resultados de compactación para todos los experimentos. Las Secciones 6.4 y 6.5 muestran los resultados de los experimentos sobre las consultas KNN y KCPQ, respectivamente, usando los algoritmos vistos en el Capítulo 5 tanto para conjuntos de datos sintéticos como reales.

6.1. Escenario de Experimentación

Los algoritmos para calcular los K-vecinos más cercanos a un punto q, desde ahora denominado *ALBA-KNN*, sobre un k^2-tree y los K-pares de vecinos más cercanos, desde ahora denominado *ALBA-KCPQ*, sobre k^2-trees, más las implementaciones que no usan EDCs de las mismas (denominadas *NAIVE-KNN* y *NAIVE-KCPQ*) fueron implementadas y ejecutadas según un diseño de experimentos que emula diferentes escenarios, con el fin de evaluar estos algoritmos bajo tres parámetros: tiempo de ejecución, cantidad de consultas de distancias ejecutadas y espacio utilizado por la estructura compacta para los diversos conjuntos de puntos.

Cada resultado de la ejecución de los algoritmos *ALBA-KNN* y *ALBA-KCPQ* fue contrastada para el mismo conjunto de datos con los algoritmos que no utilizan EDCs *NAIVE-KNN* y *NAIVE-KCPQ*, respectivamente. Por otra parte, se considera que los puntos están almacenados sobre la estructura de datos compacta k^2-tree, por lo tanto, los algoritmos *NAIVE-KNN* y *NAIVE-KCPQ* deben recuperar los puntos desde la estructura para poder realizar los cálculos respectivos. Esto añade tiempo de ejecución a las consultas respectivas.

Para la ejecución de los experimentos se utilizó el lenguaje de programación JAVA versión 8. Por otra parte, la ejecución de los experimentos se realizó sobre un servidor con las características que se presentan en la Tabla 6.1.

Tabla 6.1: Especificaciones hardware servidor experimentos

Procesador:	Intel(R) Xeon(R) CPU E3-1220 V2 @ 3.10 GHz)
Cantidad de Procesadores:	4 Físicos
Memoria Ram:	24 Gb
Sistema Operativo:	Linux Debian 3.2.0-4-AMD64
Disco Duro:	1 Tb, SAS

6.2. Diseño de Experimentos

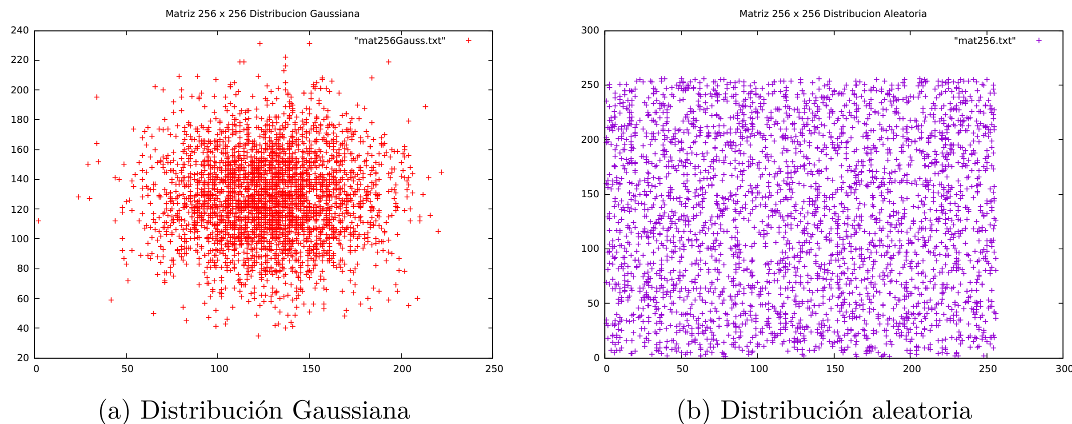
Para la experimentación de los algoritmos, se utilizaron tanto datos sintéticos como reales. En cuanto a los conjuntos de datos sintéticos, se crearon 2 grupos. El primer grupo corresponde al conjunto de datos que permiten evaluar la consulta KNN, y el segundo grupo permite evaluar la consulta KCPQ. Específicamente, para el primer grupo (*ALBA-KNN* y *NAIVE-KNN*) se utilizaron 6 conjuntos de datos, 3 conjuntos con distribución Gaussiana (ver Figura 6.1a) y 3 conjuntos con distribución aleatoria (ver Figura 6.1b).

Para la creación de los conjuntos de datos, tanto para los sintéticos como reales, se utilizó un único tamaño de matriz de 65,536 $\times$ 65,536. Esta medida se utilizó pensando en un plano que tiene una longitud de 65.536 metros, es decir, 1 metro por cada celda de la matriz de adyacencia. El tamaño está en función de la potencia de 2, de manera que la librería sea capaz de construir la estructura compacta k^2-tree. Por otra parte, bajo el tamaño de la matriz descrita, se insertaron puntos en base a escala logarítmica de base 10, consiguiendo obtener conjuntos con un N de 100.000, 1.000.000 y 10.000.000 de puntos para los conjuntos de datos sintéticos.

Por otra parte, se creó un conjunto de datos con distribución aleatoria para los puntos de consulta. Este conjunto de puntos se utiliza para hacer las consultas de los K-vecinos más cercanos sobre los conjuntos de puntos con anterioridad. Este conjunto de datos, tiene 10.000 puntos. De esta manera, por cada punto del conjunto se hace una consulta KNN sobre las matrices antes descritas. En resumen, se hacen 10.000 consultas KNN sobre las matrices con los distintos N, y se obtiene un promedio del tiempo utilizado por cada consulta individual ejecutada, además de contar la cantidad de cálculo de distancias hechos en cada algoritmo y el espacio de compactación utilizado.

Para la consulta de los K-pares de vecinos más cercanos, se utilizó un esquema semejante al anterior. Para este caso en particular ya no existen los puntos de consulta, dado que sólo se necesita como entrada dos matrices correspondiente a los dos conjuntos de puntos R y S y la constante K que determina la cantidad de pares a encontrar. De esta manera, para la consulta KCPQ se posee un total de 24 conjuntos de datos con ambas distribuciones.

Los puntos se generaron utilizando un programa en JAVA, que en parte emplea las clases para generar distribuciones gaussianas y aleatorias obtenidos desde el programa GSTD (Generating Spatio-Temporal Datasets) propuesto por (Theodoridis et al., 1999).



(a) Distribución Gaussiana (b) Distribución aleatoria

Figura 6.1: Gráficas de distribución de puntos

Los datos reales fueron obtenidos desde el portal NYC OPEN DATA[1]. Específicamente se obtuvo 6 conjuntos de datos que corresponden a la ciudad de New York, EEUU. Estos conjuntos de datos se encuentran en formato de coordenadas geográficas (latitud,longitud) y sus detalles se describen a continuación:

1. Centros de evacuación para huracanes: Corresponde a puntos dentro de la ciudad de New York en donde las personas pueden acudir en caso de ocurrir un huracán. En total este conjunto contiene 64 puntos.
2. Centros hospitalarios: Corresponde a puntos donde están ubicados centros hospitalarios de todo tipo dentro de la ciudad de New York. En total este conjunto contiene 78 puntos.
3. Accesos al Metro: Corresponde a puntos donde están ubicados los accesos al metro de la ciudad de New York. En total este conjunto contiene 1928 puntos.
4. Museos: Corresponde a puntos donde están ubicados los museos de la ciudad de New York. En total este conjunto contiene 128 puntos.
5. Universidades o centros de estudio: Corresponde a puntos dentro de la ciudad de New York, en donde están ubicados centros de estudios y universidades. En total este conjunto contiene 77 puntos.
6. Ubicaciones HotSpot: Corresponde a puntos de acceso a WIFI público dentro de la ciudad de New York. En total este conjunto contiene 2566 puntos.

Para lo anterior, se traspasaron los puntos a formato UTM[2] y luego se extrapolaron a un tamaño admisible para el algoritmo de construcción de la estructura de datos k^2-

[1] `https://data.cityofnewyork.us/browse?category=`
[2] `http://desktop.arcgis.com/es/arcmap/10.3/guide-books/map-projections/universal-transverse-mercator.htm`

tree. Se utilizó una matriz de tamaño 65.536 para todos los conjuntos de datos. Por otra parte, se crearon puntos de consulta para hacer consultas de los K-vecinos más cercanos sobre los datos reales. Todos los conjuntos de datos se probaron con los mismos puntos de consulta. El conjunto de puntos de consulta tiene 100 puntos distribuidos aleatoriamente, por lo tanto, se realizaron 100 consultas KNN sobre los datos reales y se obtuvo el tiempo promedio.

6.3. Resultados de Ahorro de Espacio al Utilizar k^2-trees

En esta sección se describe el espacio utilizado por la estructura de datos compacta k^2-tree en comparación a almacenar los puntos en memoria RAM para las diferentes configuraciones de matriz presentadas en la Sección 6.2. Para realizar los cálculos de memoria, se consideró la siguiente información:

- El dato punto en el sistema tiene dos elementos: coordenada x y la coordenada y.
- Para representar x e y en la matriz de 65.536 $\times$ 65.536 se necesitan 16 bits por coordenada lo que equivale a 4 bytes por punto.

De esta manera, los datos representados en las Tablas 6.2 y 6.3 ilustran los resultados de compactación de la estructura de datos compacta k^2-tree, que corresponde al bitmap más un 5 % de memoria adicional para responder las consultas de rank y select, frente a tener los puntos cargados en memoria RAM. .

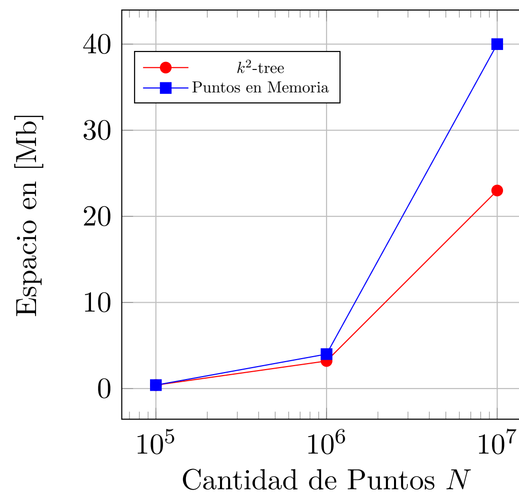
Tabla 6.2: Memoria utilizada por la estructura de datos compacta k^2-tree frente a memoria en RAM en distribución aleatoria

N = 100.000		N = 1.000.000		N = 10.000.000	
k^2-tree	Puntos en RAM	k^2-tree	Puntos en RAM	k^2-tree	Puntos en RAM
0.4 Mb	0.4 Mb	3.2 Mb	4 Mb	23 Mb	40 Mb

Tabla 6.3: Memoria utilizada por la estructura de datos compacta k^2-tree frente a memoria en RAM en distribución Gaussiana

N = 100.000		N = 1.000.000		N = 10.000.000	
k^2-tree	Puntos en RAM	k^2-tree	Puntos en RAM	k^2-tree	Puntos en RAM
0.4 Mb	0.4 Mb	3.1 Mb	4 Mb	21 Mb	40 Mb

Como se puede apreciar en la Tabla 6.2, la compactación mejora cuando se tiene mayor cantidad de puntos. Con mayor detalle, se puede apreciar que para un N de 1.000.000 de puntos se logró compactar alrededor de un 20 % versus el espacio requerido que implica tener los puntos en RAM, y para un N de 10.000.000 de puntos se logró compactar un 42.5 % respecto de tener los puntos en RAM. De esta manera se observa un incremento del doble en la compactación de los datos en función de N para los conjuntos con distribución aleatoria como también para conjuntos con distribución Gaussiana. Sin embargo, compactar se vuelve poco útil cuando la cantidad de puntos es pequeña, observándose un límite de N de 100.000 en donde la compactación es de 0. La Figura 6.2 muestra la gráfica del espacio utilizado para una distribución aleatoria de puntos sobre conjuntos de datos sintéticos.



(a) Distribución aleatoria

Figura 6.2: Uso de memoria de la estructura compacta k^2-tree v/s puntos en RAM

6.4. Resultados en Tiempos de Ejecución para KNN

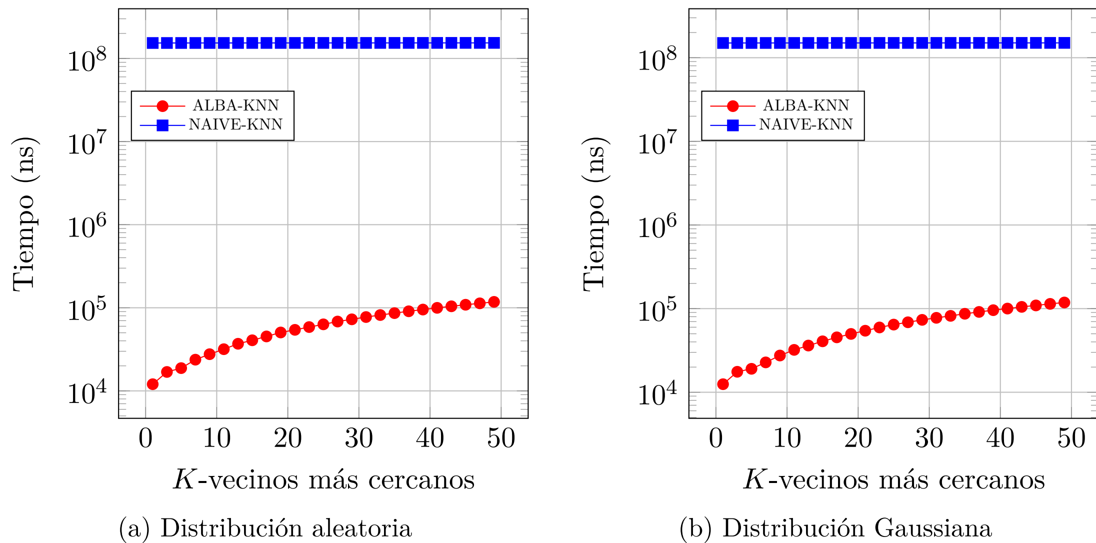
A continuación se presentan gráficas que ilustran el comportamiento de los algoritmos *KNN-ALBA* y *KNN-NAIVE* para conjuntos de datos sintéticos y reales, junto con los cálculos de distancia para datos sintéticos.

6.4.1. Tiempos de Ejecución para KNN sobre Datos Sintéticos

En esta Sección se presentan los resultados para el algoritmo *ALBA-KNN* versus el algoritmo *NAIVE-KNN* en función del tiempo de respuesta medido en Nano Segundos.

Las Figuras 6.3, 6.4 y 6.5 ilustran el comportamiento del algoritmo *ALBA-KNN* frente al algoritmo *NAIVE-KNN*, para distintos valores de K bajo conjuntos de datos de N =100.000, N =1.000.000 y N =10.000.000. La consulta por rango es la consulta que se

realiza para obtener todos los puntos desde la estructura de datos compacta k^2-tree para que el algoritmo *NAIVE-KNN* pueda operar. Por consiguiente, se observa que el algoritmo *ALBA-KNN* tiene mejor rendimiento en tiempo para los distintos K, sobre 3 órdenes de magnitud, que el algoritmo *NAIVE-KNN* en todas las situaciones evaluadas, y esto se debe a que éste no necesita recuperar los puntos desde la estructura compacta k^2-tree, dado que opera directamente en ella, a diferencia del algoritmo *NAIVE-KNN*, que necesita recuperar los puntos para poder procesarlos posteriormente. Esa recuperación de puntos, llamada consulta por rango, tiene un costo considerable en tiempo y por tanto afecta al comportamiento general del algoritmo *NAIVE-KNN*. Sin embargo, se muestra que para mientras crece N y K la consulta *ALBA-KNN* mantiene sus tiempos en la cota de 10^5 nango segundos.



(a) Distribución aleatoria

(b) Distribución Gaussiana

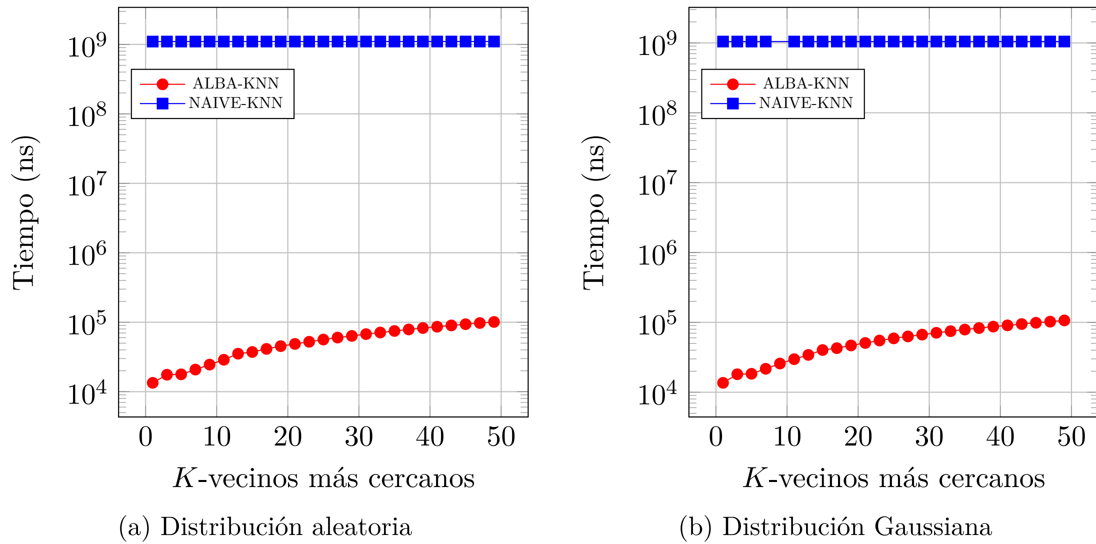
Figura 6.3: Tiempos de ejecución de los algoritmos KNN con $N = 100{,}000$

Por otra parte, las gráficas representadas en la Figura 6.6 ilustra el comportamiento en tiempo de los algoritmos *ALBA-KNN* y *NAIVE-KNN* frente a distintos valores de N, para un valor de K fijo y distribuciones aleatoria y Gaussiana. Semejante a las conclusiones anteriores, se evidencia que el algoritmo *ALBA-KNN* tiene mejor comportamiento a medida que aumenta N.

6.4.2. Cálculos de Distancia para KNN sobre Datos Sintéticos

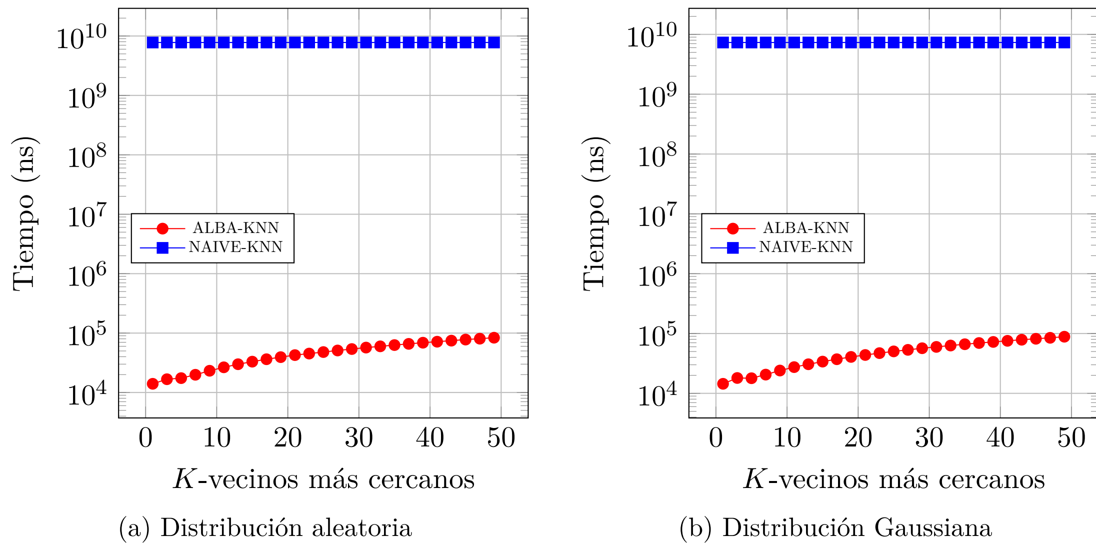
En esta Sección se ilustran los resultados para la medida de cálculos de distancia ejecutadas por cada algoritmo. En particular se midió el número de veces que cada algoritmo llamó a la función de cálculo de distancia para resolver las operaciones.

Para el parámetro de cálculo de distancias el algoritmo *ALBA-KNN* supera ampliamente al algoritmo *NAIVE-KNN*. Las gráficas ilustradas en las Figuras 6.7 y 6.8 muestran la amplia brecha entre la cantidad de cálculos realizados para responder la consulta KNN.



(a) Distribución aleatoria (b) Distribución Gaussiana

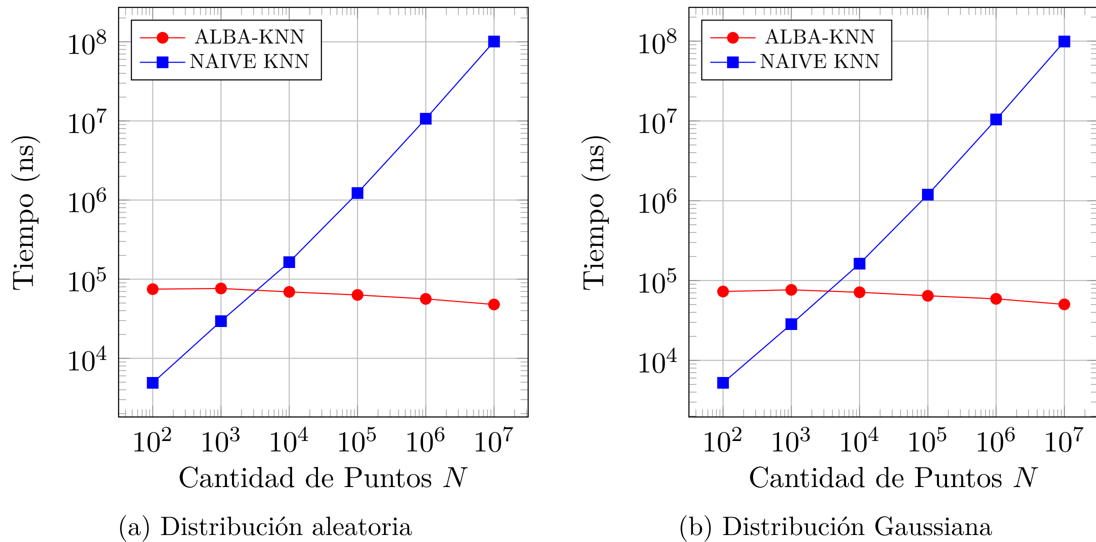
Figura 6.4: Tiempos de ejecución de los algoritmos KNN con $N = 1{,}000{,}000$



(a) Distribución aleatoria (b) Distribución Gaussiana

Figura 6.5: Tiempos de ejecución de los algoritmos KNN con $N = 10{,}000{,}000$

El algoritmo *NAIVE-KNN* necesita calcular la distancia a todos los puntos para lograr una respuesta. Sin embargo, el algoritmo *ALBA-KNN* realiza un ínfimo número de cálculos para resolver el problema, notar que los valores del algoritmo *ALBA-KNN* en las Tablas 6.4 y 6.5, lo que denota un logro importante en eficiencia bajo este enfoque. Por otra parte, se evidencia el incremento de cálculos de distancia del algoritmo *ALBA-KNN* a medida que el número de K vecinos aumenta, no obstante el aumento de puntos en la matriz no implica un aumento proporcional de cálculos de distancia, más bien una disminución



(a) Distribución aleatoria

(b) Distribución Gaussiana

Figura 6.6: Tiempos de ejecución de los algoritmos KNN con un K fijo de $K = 25$

de estos a medida que aumenta N. Lo anterior se puede explicar debido a que mientras aumenta N y se mantiene el mismo espacio (tamaño de la matriz), la distancias entre los puntos disminuye y por lo tanto es probable que se encuentre de manera temprana mejores candidatos que permiten descartar cálculos de distancias futuros.

Tabla 6.4: Tabla de cálculos de distancia para consulta KNN en conjuntos de datos con distribución aleatoria

K	N = 100.000		N = 1.000.000		N = 10.000.000	
	ALBA-KNN	NAIVE-KNN	ALBA-KNN	NAIVE-KNN	ALBA-KNN	NAIVE-KNN
5	199	100.000	196	1.000.000	191	10.000.000
15	406	100.000	370	1.000.000	330	10.000.000
25	601	100.000	532	1.000.000	459	10.000.000
35	792	100.000	689	1.000.000	582	10.000.000
45	979	100.000	844	1.000.000	703	10.000.000

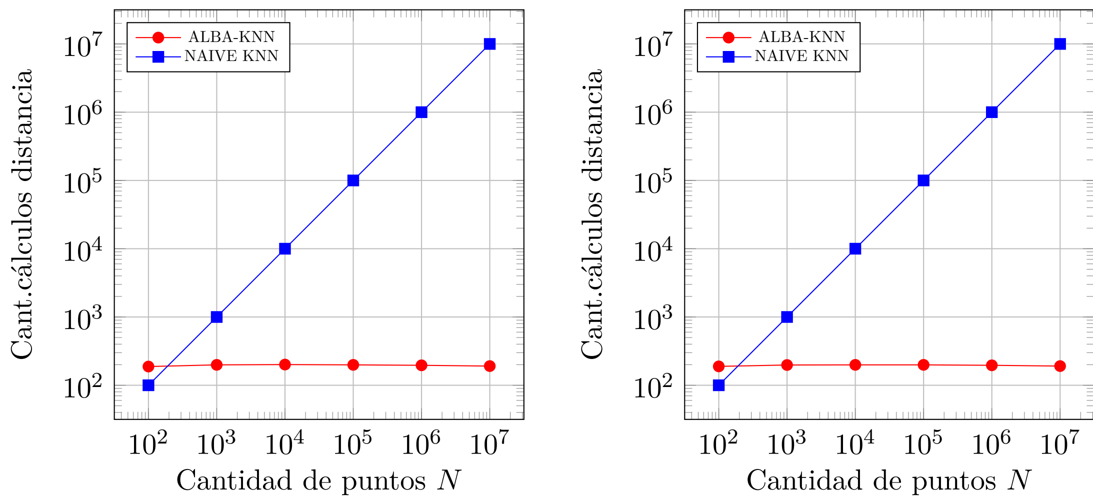
6.4.3. Tiempos de Ejecución para KNN sobre Datos Reales

En esta sección se presentan gráficas que ilustran el comportamiento en tiempo de ejecución de los algoritmos *ALBA-KNN* y *NAIVE-KNN* mientras aumenta el valor de K para conjuntos de datos reales.

Semejante a los resultados obtenidos con datos sintéticos, las gráficas en las Figuras 6.10 y 6.11 ilustran que el algoritmo *ALBA-KNN* se mantiene en el rango de 10^4 a 10^5 Nano Segundos, logrando mejor comportamiento que su contraparte para todos los conjuntos de dato. No obstante, mantiene mayores cercanías con el algoritmo *NAIVE-KNN* cuando los

Tabla 6.5: Tabla de cálculos de distancia para consulta KNN en conjuntos de datos con distribución Gaussiana

K	N = 100.000		N = 1.000.000		N = 10.000.000	
	ALBA-KNN	NAIVE-KNN	ALBA-KNN	NAIVE-KNN	ALBA-KNN	NAIVE-KNN
5	199	100.000	196	1.000.000	191	10.000.000
15	411	100.000	376	1.000.000	337	10.000.000
25	610	100.000	544	1.000.000	472	10.000.000
35	806	100.000	706	1.000.000	601	10.000.000
45	998	100.000	867	1.000.000	727	10.000.000

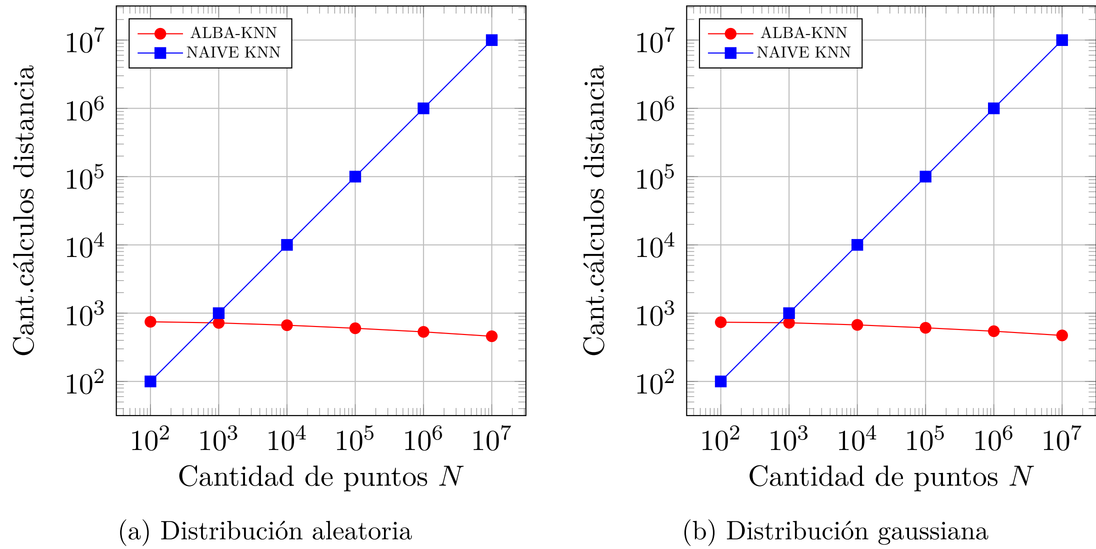


(a) Distribución aleatoria

(b) Distribución gaussiana

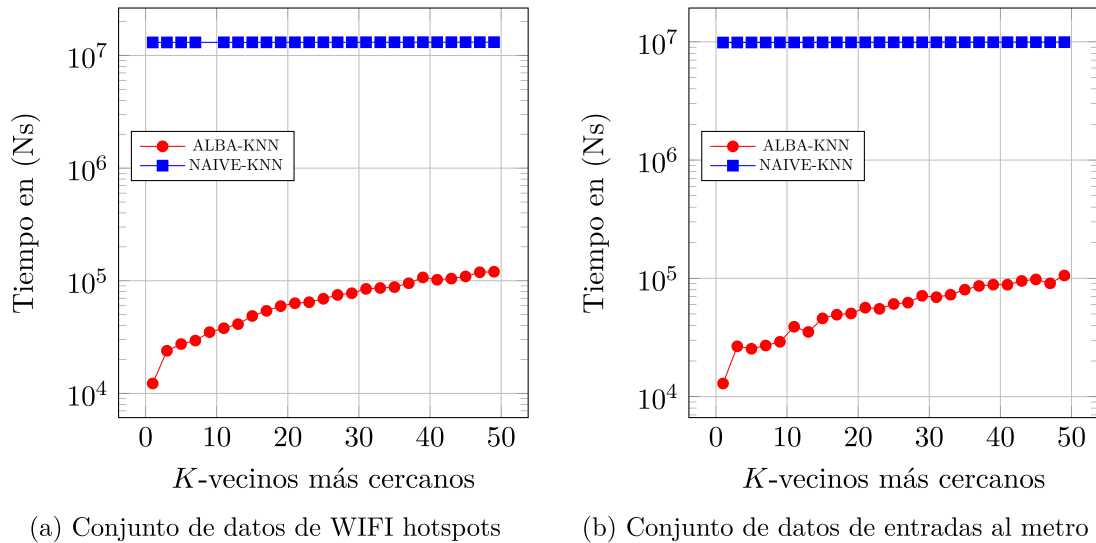
Figura 6.7: Comportamiento en cálculos de distancia de KNN para distintos N y un $K = 5$

conjuntos de datos son pequeños. Esta cercanía es de un comportamiento esperable, dado que los conjuntos de datos son pequeños, por tanto, el algoritmo *NAIVE-KNN* es más eficiente cuando los sets son más pequeños.



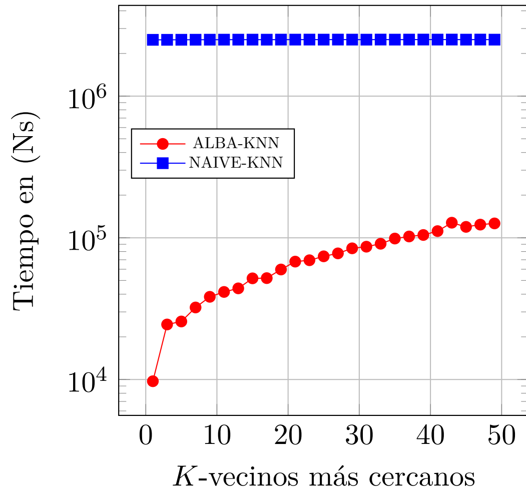
(a) Distribución aleatoria (b) Distribución gaussiana

Figura 6.8: Comportamiento en cálculos de distancia de KNN para distintos N y un $K = 25$

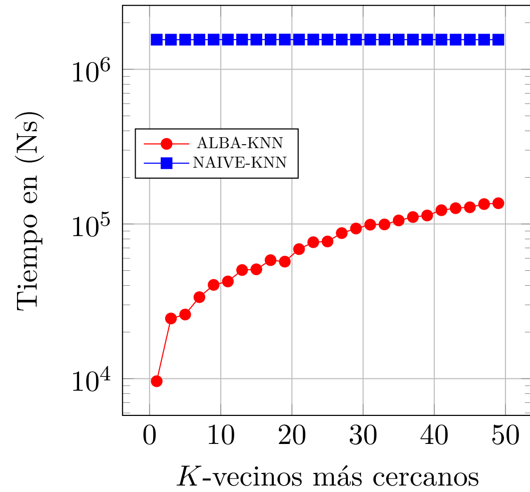


(a) Conjunto de datos de WIFI hotspots (b) Conjunto de datos de entradas al metro

Figura 6.9: Tiempo de ejecución en Nano Segundos para conjuntos de datos reales

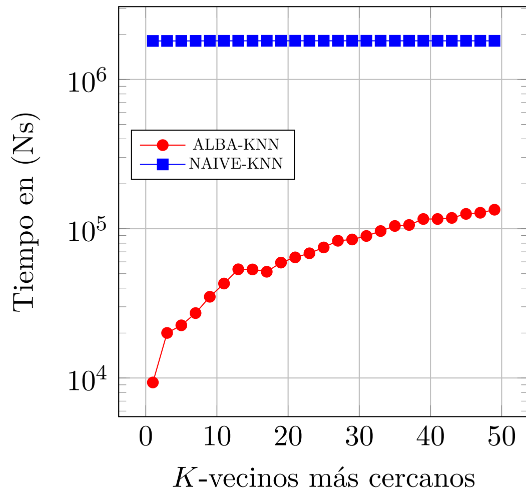


(a) Conjunto de datos de museos

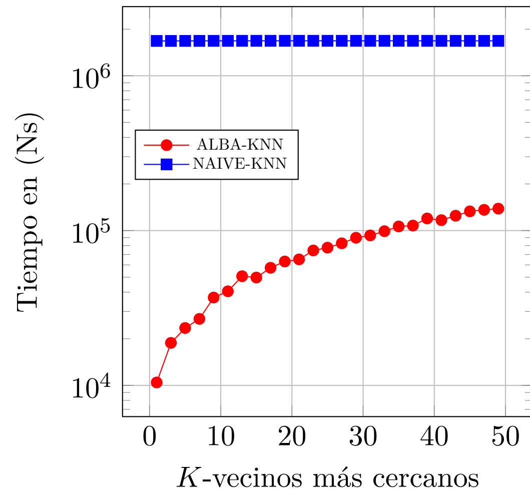


(b) Conjunto de datos de hospitales

Figura 6.10: Tiempo de ejecución en Nano Segundos para conjuntos de datos reales



(a) Conjunto de datos de Universidades



(b) Conjunto de datos de centros de evacuación

Figura 6.11: Tiempo de ejecución en Nano Segundos para conjuntos de datos reales

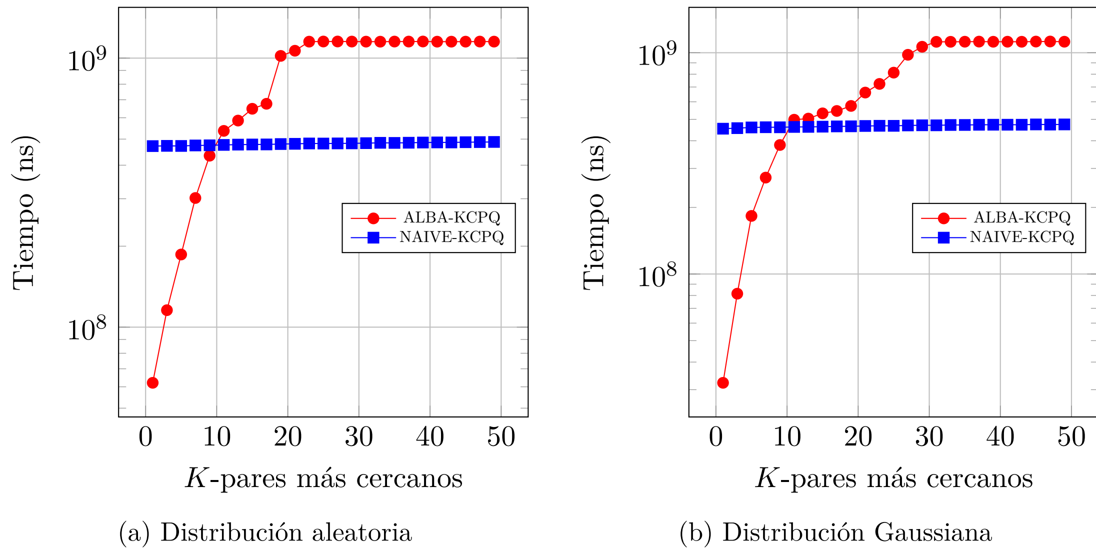
6.5. Resultados para KCPQ

A continuación se presentan gráficas que ilustran el comportamiento de los algoritmos *KCPQ-ALBA* y *KCPQ-NAIVE* para conjuntos de datos sintéticos y reales, y el cálculo de distancia para datos sintéticos.

6.5.1. Tiempos de Ejecución para KCPQ Sobre Datos Sintéticos

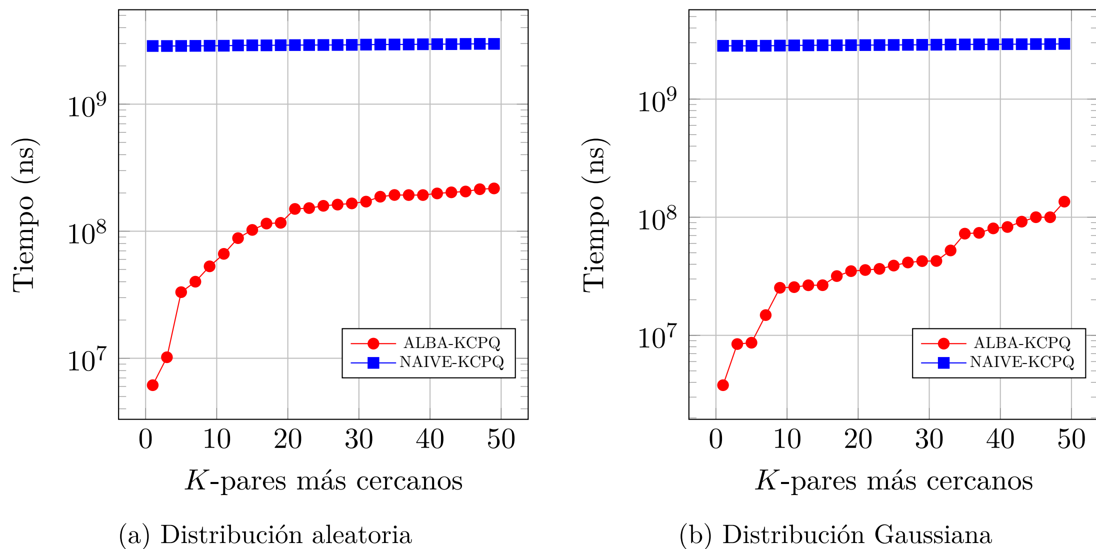
Las gráficas presentes en las Figuras 6.12, 6.13 y 6.14 evidencian un comportamiento similar al algoritmo KNN en tiempo de ejecución. Sin embargo, aunque el algoritmo *ALBA-KCPQ* tiene mejor comportamiento que el algoritmo *NAIVE-KCPQ* en la mayoría de las gráficas, los tiempos aumentan, para ambos algoritmos, respecto de las consultas KNN con los mismos tamaños de N. El aumento del tiempo de respuesta es debido a que se necesitan procesar dos conjuntos de puntos y de alguna manera se deben hacer más comparaciones (de un punto contra varios del otro conjunto), y por lo tanto se ejecutan más cálculos de distancia para responder la consulta.

Por otra parte, la Figura 6.12 ilustra un comportamiento totalmente distinto al resto de los gráficos. Se observa que el algoritmo *ALBA-KCPQ* tiene un comportamiento más costoso en tiempo que el algoritmo que no utiliza EDCs y, más aún, esto es confirmado por los valores de cálculo de distancia presentes en la Tabla 6.6. Se hacen muchos cálculos de distancia respecto de la implementación ingenua y por tanto el tiempo de ejecución aumenta. Sin embargo, y en particular para este algoritmo *ALBA-KCPQ*, con el conjunto de datos con N=100.000, ocurre esta situación. El comportamiento de la Figura 6.12 se debe a que el algoritmo *ALBA-KCPQ* es afectado por la densidad de la matriz. Es decir, la cantidad de puntos bajo el tamaño de matriz establecido. Note que para un N=1.000.000 los tiempos y los cálculos de distancia mejoran considerablemente, y para un N=10.000.000 mejoran aún más. Lo anterior es por que mientras se aumenta la cantidad de puntos N, las matrices se hacen mas densas, es decir, los puntos están más cerca unos de otros y por tanto, los criterios de poda del algoritmo *ALBA-KCPQ* se hacen más efectivos. En el conjunto de puntos donde N es de 100.000, los puntos están más distantes que en las matrices con más puntos, por lo tanto, al algoritmo le cuesta más encontrar un candidato prometedor que pode más ramas en la estructura compacta.



(a) Distribución aleatoria (b) Distribución Gaussiana

Figura 6.12: Tiempos de ejecución de los algoritmos KCPQ con $N = 100{,}000$



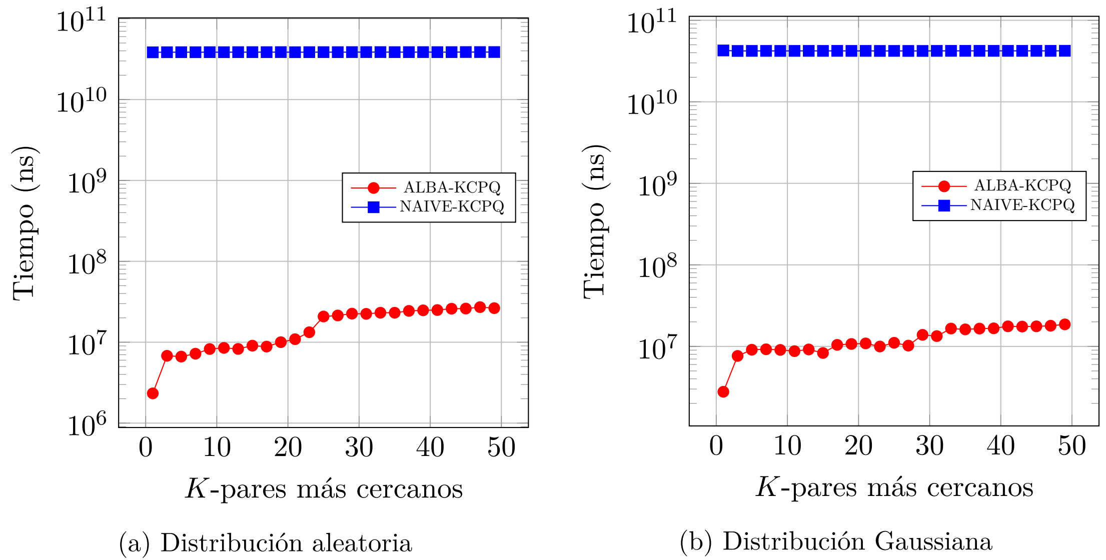
(a) Distribución aleatoria (b) Distribución Gaussiana

Figura 6.13: Tiempos de ejecución de los algoritmos KCPQ con $N = 1{,}000{,}000$

6.5.2. Cálculos de Distancia para KCPQ sobre Datos Sintéticos

La presente sección ilustra los resultados para la medida de cálculos de distancia ejecutados por los algoritmos *ALBA-KCPQ* y *NAIVE-KCPQ*.

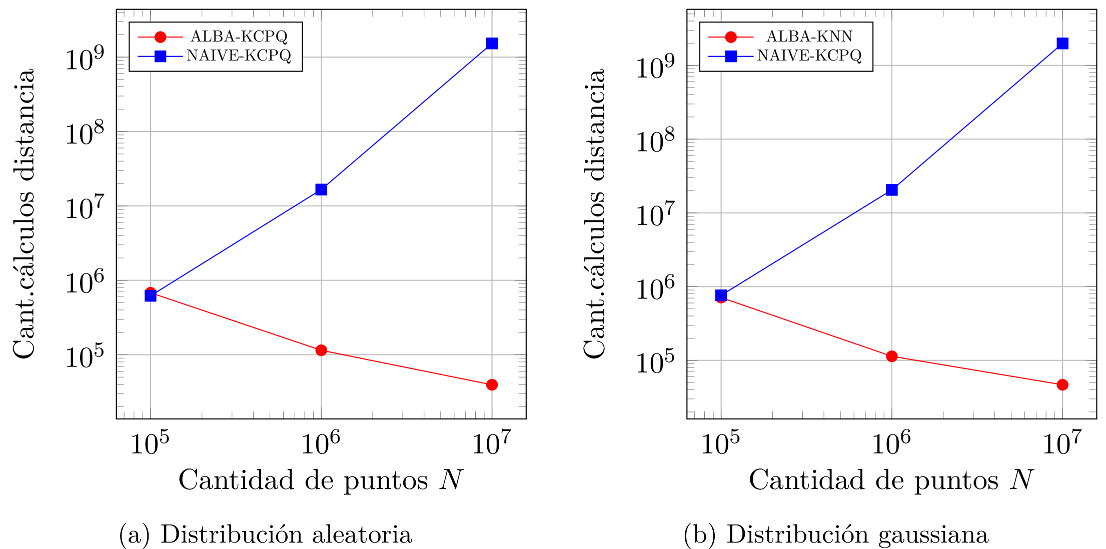
Las gráficas expuestas en las Figuras 6.15 y 6.16 y los valores de las Tablas 6.6 y 6.7 para conjuntos de puntos con distribución aleatoria y Gaussiana, respectivamente, ilustran que el algoritmo *ALBA-KCPQ* mejora la cantidad de cálculos a medida que crece la densidad



(a) Distribución aleatoria (b) Distribución Gaussiana

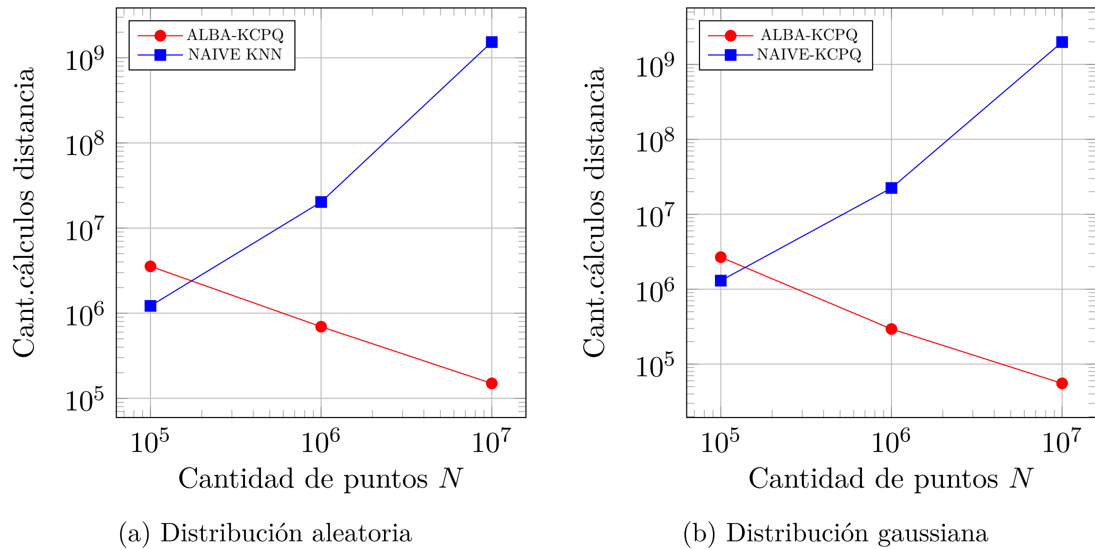
Figura 6.14: Tiempos de ejecución de los algoritmos KCPQ con $N = 10{,}000{,}000$

en la matriz. Sin embargo, para el algoritmo que no utiliza EDCs ocurre todo lo contrario. Las densidades y tamaños de matriz no son variables que afecten al algoritmo que no utiliza EDCs, siendo N (cantidad de puntos), la única variable que merma su funcionamiento a medida que crece logarítmicamente.



(a) Distribución aleatoria (b) Distribución gaussiana

Figura 6.15: Comportamiento en cálculos de distancia de KCPQ para distintos N y un $K = 5$



(a) Distribución aleatoria (b) Distribución gaussiana

Figura 6.16: Comportamiento en cálculos de distancia de KCPQ para distintos N y un $K = 25$

Tabla 6.6: Tabla de número de cálculos de distancia para consulta KCPQ en conjuntos de datos con distribución aleatoria

K	N = 100.000		N = 1.000.000		N = 10.000.000	
	ALBA-KCPQ	NAIVE-KCPQ	ALBA-KCPQ	NAIVE-KCPQ	ALBA-KCPQ	NAIVE-KCPQ
5	680.495	619.082	114.752	16.592.577	39.463	1.527.375.560
15	2.131.915	934.833	259.870	18.628.258	50.712	1.528.701.488
25	3.553.460	1.218.702	694.275	20.235.992	149.922	1.530.135.874
35	3.553.460	1.388.239	778.617	21.718.445	158.920	1.531.165.972
45	3.553.460	1.531.593	1.027.040	22.918.416	167.989	1.532.363.681

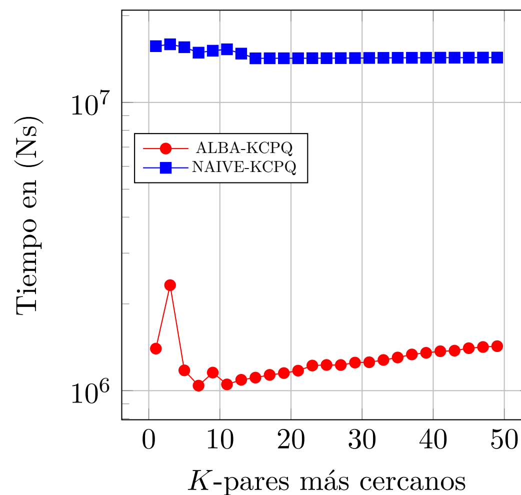
Tabla 6.7: Tabla de número de cálculos de distancia para consulta KCPQ en conjuntos de datos con distribución Gaussiana

K	N = 100.000		N = 1.000.000		N = 10.000.000	
	ALBA-KCPQ	NAIVE-KCPQ	ALBA-KCPQ	NAIVE-KCPQ	ALBA-KCPQ	NAIVE-KCPQ
5	710.343	762.032	113.544	20.486.433	46.671	1.985.830.718
1.5	1.848.944	991.307	248.043	21.370.508	47.243	1.987.033.303
2.5	2.678.011	1.301.658	294.349	22.443.278	55.367	1.988.292.510
3.5	3.583.484	1.554.548	593.804	24.225.438	84.968	1.989.616.617
4.5	3.583.484	1.677.464	705.679	25.771.882	90.500	1.990.558.473

6.5.3. Tiempos de Ejecución para KCPQ Sobre Datos Reales

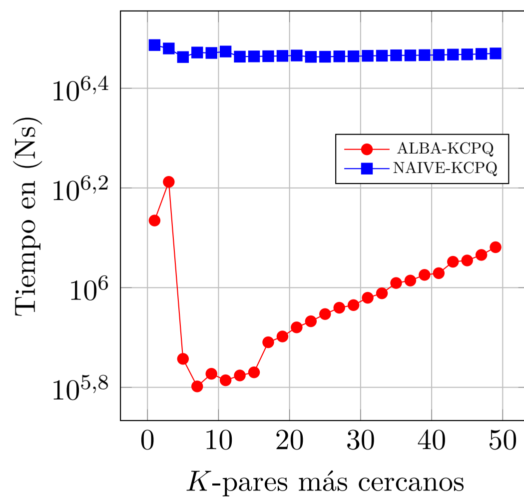
La presente sección ilustra el resultado en tiempo de ejecución de los algoritmos *ALBA-KCPQ* y *NAIVE-KCPQ* para conjuntos con datos reales. Se tomó como ejemplo tres tipos de consultas con los datos reales explicados en la Sección **??**. Una de ellas es consultar, por ejemplo, aquellos accesos públicos de Internet (WIFI Hotspots) más cercanos a Universidades dentro de la ciudad de New York. La Figura 6.17 muestra este resultados para distintos valores de K. Otro ejemplo, es consultar los centros de evacuación que estén mas cerca de universidades, representados en la Figura 6.18 (a). Finalmente, el tercer ejemplo, es representado en la Figura 6.18(b), que ilustra la consulta de aquellas salidas de estaciones del metro más cercanas a museos de la ciudad de New York.

De manera semejante a los resultados con datos sintéticos para la consulta *ALBA-KCPQ* y *NAIVE-KCPQ*, se aprecia que el algoritmo *ALBA-KCPQ* tiene un comportamiento aceptable en comparación al algoritmo *NAIVE-KCPQ*. Este comportamiento puede parecer contradictorio respecto al conjunto de puntos sintéticos explicados en la Sección 6.5.1. Sin embargo, es importante mencionar que estos conjuntos de puntos reales son pequeños, y que en este caso en particular, las matrices tienen más cuadrantes con valores vacíos (grandes sectores sin puntos), los cuales son descartados rápidamente por el algoritmo dado que en la estructura de datos compacta son marcados con un 0 (en el bitmap) y no se sigue recorriendo esa rama. Los tiempos de ejecución son aproximadamente un orden de magnitud inferior respecto a los conjuntos de puntos sintéticos, básicamente, porque los conjuntos de puntos reales contienen menos puntos y por lo tanto, los algoritmos se comportan con mejor desempeño en tiempo de ejecución.

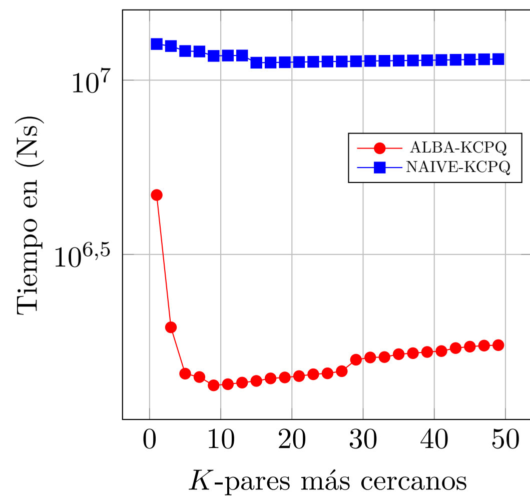


(a) Consulta de puntos WIFI con universidades

Figura 6.17: Tiempo de ejecución en Nano Segundos para conjuntos de datos reales



(a) Consulta de centros de evacuación con universidades



(b) Consulta de salidas del metro con museos

Figura 6.18: Tiempo de ejecución en Nano Segundos para conjuntos de datos reales

Capítulo 7

Conclusiones y Trabajo Futuro

La aplicación de algoritmos de proximidad espacial, como obtener los K-vecinos más cercanos a un punto (KNN), u obtener los K-pares de vecinos más cercanos entre dos conjuntos de puntos (KCPQ), en general, es amplia en la literatura, sin embargo, la utilización de estructuras de datos compactas para representar grandes volúmenes de datos y responder este tipo de consultas ha sido casi inexplorado. Hoy el manejo de grandes volúmenes de información en memorias volátiles representa un reto importante para las ciencias de la computación, dado que existe una brecha que todavía no se puede acortar o definitivamente eliminar. Acceder a disco sigue siendo caro y los precios de las memorias volátiles todavía siguen altos. Por lo tanto, el estudio y aplicación de estructuras de datos compactas para dar solución a estas problemáticas, es sin duda, una opción interesante.

En esta tesis, se propone una solución específica para almacenar grandes cantidades de puntos geográficos representados en una estructura de datos compacta k^2-tree, se definen algoritmos para responder a dos consultas interesantes de proximidad espacial, en tiempos acotados. Lo anterior es de gran relevancia para los sistemas de información geográfica, para las aplicaciones que utilizan proximidad espacial como google mapts, entre otras. Los algoritmos desarrollados en esta tesis, quedan disponibles para la comunidad en una librería implementada en lenguaje JAVA[1].

Los experimentos presentados sobre datos sintéticos y reales, muestran que, utilizando la estructura de datos compacta k^2-tree se puede ahorrar espacio de almacenamiento en memoria principal cuando se utilizan grandes conjuntos de datos y que al realizar consultas de proximidad espacial KNN y KCPQ se puede responder más eficientemente (en la mayoría de los casos), que usando algoritmos ingenuos de $O(n \log n)$. Por otra parte, se muestra la eficiencia de los algoritmos KNN y KCPQ a la hora de realizar cálculos de distancia para resolver las consultas. Esto último es un logro importante, dado que las operaciones de cálculo de distancia son costosas en términos de tiempo CPU. Se evidencia también que estas estructuras de datos a pesar de estar diseñadas para trabajar con grandes conjuntos de datos, están afectas a variables tales como: la distribución de los datos en el espacio y la densidad de los conjuntos de datos. Estas variables pueden hacer que las podas de

[1] http://arrau.chillan.ubiobio.cl/fernando/

los algoritmos de proximidad espacial no se aprovechen correctamente, o en su defecto, la construcción de una estructura compacta saturada y ineficiente.

Como trabajos futuros existen bastantes alternativas interesantes que se pueden abordar, como por ejemplo, implementar otras consultas de proximidad espacial, como por ejemplo, el problema de obtener los vecinos reversos de un punto. Por otro lado, es necesario optimizar la construcción del k^2-tree. Además, dado los resultados prometedores de nuestras implementaciones, sería interesante hacer una comparativa con otras estructuras de datos compactas para procesar información espacial, esto considerando otros conjuntos de datos reales más voluminosos.

Por otra parte, la programación de la librería en lenguaje JAVA permitirá que esta se extienda a otras plataformas, como Apache Spark, permitiendo paralelizar las implementaciones y hacer más eficiente aspectos de ésta.

Esta tesis se encuentra enmarcada en el proyecto exploratorio 2030 titulado "*Estructuras de Datos Compactas para procesar eficientemente datos espaciales y espacio-temporales en el contexto del Big Data*", código 1638, adjudicado por el grupo de investigación ALBA, código GI 160119/EF, de la Universidad del Bío-Bío.

Referencias

David Aha, Dennis Kibler, y Marc Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, 1991.

Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, y Kunihiko Sadakane. Succinct trees in practice. En *Proceedings of the Workshop on Algorithm Engineering and Experiments*, págs. 84–97. 2010.

Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, y Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, 1990.

Stefan Berchtold, Christian Böhm, y Hans-Peter Kriegal. The pyramid-technique: towards breaking the curse of dimensionality. En *Proceedings of SIGMOD*, págs. 142–153. 1998.

Stefan Berchtold, Daniel Keim, y Hans-Peter Kriegel. The x-tree: an index structure for high-dimensional data. En *Proceedings of the 22th International Conference on Very Large Data Bases*, págs. 28–39. 1996.

Nieves Brisaboa, Ana Cerdeira-Pena, Narciso López-López, Gonzalo Navarro, Miguel Penabad, y Fernando Silva-Coira. Efficient representation of multidimensional data over hierarchical domains. En *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval*, págs. 191–203. 2016.

Nieves Brisaboa, Guillermo De Bernardo, Gilberto Gutiérrez, Susana Ladra, Miguel Penabad, y Brunny Troncoso. Efficient set operations over k2-trees. En *Proceedings of the Data Compression Conference*, págs. 373–382. 2015.

Nieves Brisaboa, Guillermo De Bernardo, Roberto Konow, y Gonzalo Navarro. K2-treaps: range top-k queries in compact space. En *Proceedings of the 21st International Symposium on String Processing and Information Retrieval*, págs. 215–226. 2014a.

Nieves Brisaboa, Susana Ladra, y Gonzalo Navarro. k2-trees for compact web graph representation. En *Proceedings of the International Symposium on String Processing and Information Retrieval*, págs. 18–30. 2009.

Nieves Brisaboa, Susana Ladra, y Gonzalo Navarro. Dacs: bringing direct access to variable-length codes. *Information Processing and Management*, 49(1):392–404, 2013a.

Nieves Brisaboa, Susana Ladra, y Gonzalo Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39:152–174, 2014b.

Nieves Brisaboa, Miguel Luaces, Gonzalo Navarro, y Diego Seco. Space-efficient representations of rectangle datasets supporting orthogonal range querying. *Information Systems*, 38(5):635–655, 2013b.

Yun Chen y Jignesh Patel. Efficient evaluation of all-nearest-neighbor queries. En *Proceedings of the 23rd International Conference on Data Engineering*, págs. 1056–1065. 2007.

Francisco Claude y Gonzalo Navarro. A fast and compact web graph representation. En *Proceedings of the International Symposium on String Processing and Information Retrieval*, págs. 118–129. 2007.

Francisco Claude y Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. En *Proceedings of the International Symposium on String Processing and Information Retrieval*, págs. 176–187. 2008.

Thomas Cormen, Charles Leiserson, Ronald Rivest, y Clifford Stein. *Introduction to algorithms*. The MIT Press, 2009.

Antonio Corral. *Algorithms for processing of spatial queries using R-trees. The closest pairs query and its application on spatial databases*. Tesis Doctoral, Universidad de Almeria, 2002.

Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, y Michael Vassilakopoulos. Closest pair queries in spatial databases. En *Proceedings of SIGMOD*, págs. 189–200. 2000.

Belur Dasarathy. *Nearest neighbor (NN) norms: nn pattern classification techniques*. IEEE Computer society press, 1991.

Guillermo De Bernardo, Sandra Álvarez-García, Nieves Brisaboa, Gonzalo Navarro, y Oscar Pedreira. Compact querieable representations of raster data. En *Proceedings of the International Symposium on String Processing and Information Retrieval*, págs. 96–108. 2013.

Andrea De Mauro, Marco Greco, y Michele Grimaldi. What is big data? a consensual definition and a review of key research topics. En *Proceedings of the American Institute of Physics*, págs. 97–104. 2015.

Antonio Fariña, Susana Ladra, Oscar Pedreira, y Ángeles Places. Rank and select for succinct data structures. *Electronic Notes in Theoretical Computer Science*, 236:131–145, 2009.

Jerome Friedman, Forest Baskett, y Leonard Shustek. An algorithm for finding nearest neighbors. *IEEE Transactions on Computers*, 24(10):1000–1006, 1975.

Volker Gaede y Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

Travis Gagie, Javier González-Nova, Susana Ladra, Gonzalo Navarro, y Diego Seco. Faster compressed quadtrees. En *Proceedings of the Data Compression Conference*, págs. 93–102. 2015.

Alexander Golynski, Roberto Grossi, Ankur Gupta, Rajeev Raman, y Satti Rao. On the size of succinct indices. En *Proceedings of the European Symposium on Algorithms*, págs. 371–382. 2007.

Rodrigo González, Szymon Grabowski, Veli Mäkinen, y Gonzalo Navarro. Practical implementation of rank and select queries. En *Proceedings of the 4th Workshop on Efficient and Experimental Algorithms*, págs. 27–38. 2005.

Ralf Güting. An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399, 1994.

Antonin Guttman. R-trees: a dynamic index structure for spatial searching. En *Proceedings of SIGMOD*, págs. 47–57. 1984.

Kazuki Ishiyama, Koji Kobayashi, y Kunihiko Sadakane. Succinct quadtrees for road data. En *Proceedings of the International Conference on Similarity Search and Applications*, págs. 262–272. 2017.

Guy Jacobson. Space-efficient static trees and graphs. En *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, págs. 549–554. 1989.

Norio Katayama y Shiníchi Satoh. Sr-tree: an index structure for nearest-neighbor searching of high-dimensional point data. *Systems and Computers in Japan*, 29(6):59–73, 1998.

You Jung Kim y Jignesh Patel. Performance comparison of the r*-tree and the quadtree for knn and distance join queries. *IEEE Trans. on Knowl. and Data Eng.*, 22(7):1014–1027, 2010.

Nicholas Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, y Andrew Campbell. A survey of mobile phone sensing. *IEEE Communications magazine*, 48(9), 2010.

King-Ip Lin, Hosagrahar Jagadish, y Christos Faloutsos. The tv-tree: an index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–542, 1994.

Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos Papadopoulos, y Yannis Theodoridis. *R-trees: theory and applications.* Springer Science and Business Media, 2010.

George Mavrommatis, Panagiotis Moutafis, Michael Vassilakopoulos, Francisco García-García, y Antonio Corral. Slicenbound: solving closest pairs and distance join queries in apache spark. En *Proceedings of the Advances in Databases and Information Systems Conference*, págs. 199–213. 2017.

Natalia Miranda, Edgar Chávez, María Fabiana Piccoli, y Nora Reyes. Very fast all k-nearest neighbors in metric and non metric spaces without indexing. En *Proceedings of the International Conference on Similarity Search and Applications*, págs. 300–311. 2013.

Gonzalo Navarro. Spaces, trees, and colors: the algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):52:1–52:47, 2014.

Gonzalo Navarro. *Compact data structures: a practical approach*. Cambridge University Press, 2016.

Gonzalo Navarro y Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):16:1–16:39, 2014.

Sameer Nene y Shree Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, 1997.

Dimitris Papadias y Yannis Theodoridis. Spatial relations, minimum bounding rectangles, and spatial data structures. *International Journal of Geographical Information Science*, 11(2):111–138, 1997.

Apostolos Papadopoulos y Yannis Manolopoulos. Performance of nearest neighbor queries in r-trees. En *Proceedings of the International Conference on Database Theory*, págs. 394–408. 1997.

Miguel Romero. *Un ’indice espacio temporal para puntos móviles basado en estructuras de datos compactas*. Tesis Doctoral, Universidad de la Coruña, 2017.

Nick Roussopoulos, Stephen Kelley, y Frédéric Vincent. Nearest neighbor queries. En *Proceedings of SIGMOD*, págs. 71–79. 1995.

Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

Robert Sedgewick y Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 2011.

Shashi Shekhar y Sanjay Chawla. *Spatial databases: a tour*. Prentice Hall, 2003.

Yannis Theodoridis, Jefferson Silva, y Mario Nascimento. On the generation of spatio-temporal datasets. En *Proceedings of the 6th International Symposium on Advances in Spatial Databases*, págs. 147–164. 1999.

Cristian Vallejos, Mónica Caniupán, y Gilberto Gutierrez. K^2-treaps to represent and query data warehouses into main memory. En *36th International Conference of the Chilean Computer Science Society, SCCC 2017, Arica, Chile, October 16-20, 2017*, págs. 1–12. 2017.

Prayaag Venkat y David Mount. A succinct, dynamic data structure for proximity queries on point sets. En *Proceedings of the Canadian Conference on Computational Geometry*. 2014.

Sebastiano Vigna. Broadword implementation of rank/select queries. En *Proceedings of the International Workshop on Experimental and Efficient Algorithms*, págs. 154–168. 2008.

Mark Weiss. *Data structures and algorithm analysis in Java*. Addison-Wesley, 2007.

Dietrich Wettschereck, David Aha, y Takao Mohri. A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. *Artificial Intelligence Review*, 11(1):273–314, 1997.

David White y Ramesh Jain. Similarity indexing with the ss-tree. En *Proceedings of the Twelfth International Conference on Data Engineering*, págs. 516–523. 1996.

Bin Yao, Feifei Li, y Piyush Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. En *Proceedings of the 26th International Conference on Data engineering*, págs. 4–15. 2010.

Hai-Da Zhang, Zhi-Hao Xing, Lu Chen, y Yun-Jun Gao. Efficient metric all-k-nearest-neighbor search on datasets without any index. *Journal of Computer Science and Technology*, 31(6):1194–1211, 2016.