UNIVERSIDAD DEL BÍO-BÍO

Facultad de Ciencias Empresariales
Departamento de Ciencias de la Computación y
Tecnologías de la Información

Algoritmos geométricos sobre estructuras de datos compactas

Por
Juan Felipe Castro Arias

Tesis para optar al grado de Magíster
en Ciencias de la Computación

Dirigido por:
Mg. Miguel Romero Vásquez

Co-Dirigido por:
Dr. Gilberto Gutiérrez Retamal

Julio 2016

A mis padres, hermanos y abuelo.

Agradecimientos

A mis padres Mario y Marisol, porque todo lo que he logrado ha sido gracias al sacrificio y esfuerzo constante de ambos. A mis hermanos Henry, Jacobo y Pedro, porque nada toman en serio y a la vez hacen que todo sea más entretenido. A mi abuelo, que siempre me pregunta por la pensadora y si viajé en helicóptero a San Carlos porque piensa que soy millonario (no lo soy).

Agradezco a mis directores de tesis Miguel Romero y Gilberto Gutiérrez, por la disposición que tuvieron durante este año y medio a trabajar muchas veces fuera de su horario laboral e incluso sábados debido a que mi trabajo no permitía otro horario.

A mis amigos y colegas, porque fueron la cuota exacta de relajo ante lo estresante que fue hacer una tesis y tener un trabajo tiempo completo, además agradezco que muchos sin entender bien lo que estaba haciendo, cordialmente preguntaban por el tema y como iba todo.

Resumen

La rápida tendencia de crecimiento que han experimentado los conjuntos de datos espaciales a lo largo de su historia ha derivado en una consecuencia inevitable de buscar nuevas alternativas más eficientes para el tratamiento de los mismos datos. Con la aparición de las Bases de Datos Espaciales, para muchos problemas de Geometría Computacional (búsqueda por rango, el par de vecinos más cercanos, entre otros) se propusieron soluciones que hacen uso de estructuras de datos multidimensionales residentes en memoria secundaria, y que dada la jerarquía de sus estructuras nos permiten cambiar el enfoque de tratamiento de los problemas abordados. Recientemente producto del aumento de la capacidad de la memoria principal y de una serie de técnicas de compresión de datos, es posible aprovecharse positivamente de la jerarquía de memoria de un computador para tratar de manera más eficiente grandes volúmenes de datos. Es un desafío entonces el tratar sobre estas estructuras recientes, algoritmos que permitan interpretarlos como una representación espacial de un conjunto de datos, sin que estos pierdan sus propiedades. En esta tesis se exploran, plantean e implementan algoritmos geométricos para un problema clásico y representativo de la Geometría Computacional como lo es el encontrar la cerradura convexa de un conjunto de puntos discretos en dos dimensiones, todo esto sobre una estructura de datos compacta como lo es el k^2-tree, teniendo como objetivo principal mantener la integridad de la estructura evitando utilizar grandes cantidades de espacio de almacenamiento para el procesamiento y las estructuras auxiliares de los algoritmos, y así mantener la línea de la espacio-eficiencia tanto como para la estructura como para los algoritmos geométricos.

Mediante una serie de experimentos que consideran datos sintéticos y reales evaluamos el rendimiento de nuestros algoritmos. Los resultados de los experimentos dan cuenta de la eficiencia de nuestros algoritmos en términos de almacenamiento y tiempo de ejecución comparado con algoritmos ya existentes en la literatura y que previamente rescatan los puntos desde el k^2-tree. Bajo este enfoque logramos reducir el tiempo de ejecución a sólo un 0,0449 % del necesitado en el caso de extraer todos los puntos de un k^2-tree, lo que avala la eficiencia de nuestra propuesta.

Índice

Parte I

Motivación

Capítulo 1

Motivación

1.1. Introducción

La Geometría Computacional [19, 17, 53, 41] es un área específica de las Ciencias de la Computación que se enfoca en el estudio de la representación de los espacios geométricos como también de dar solución a los problemas que se presentan en el mismo a través de la computación.

Entre las múltiples aplicaciones que se puede dar a la Geometría Computacional, una de las más importantes y conocidas son las bases de datos espaciales y los GIS (Geographic Information Systems), los cuales hacen una fiel representación de la realidad geográfica y soportan diversos problemas geométricos a través de algoritmos y estructuras de datos diseñadas para tales fines. También destacan la utilizada en los sistemas tipo CAD/CAM (Computer Aided Design y Computer Aided Manufacturing). Los CAD/CAM [56] son sistemas computacionales enfocados en agilizar el diseño y la manufactura de variados objetos, ya sea como una silla, un motor, un automóvil, un avión e incluso un cohete espacial. Esto muchas veces representado a través de computación gráfica, la cual es en su base definida por la Geometría Computacional. También, dentro de la Geometría Computacional, se aplica la triangulación de Delaunay que consiste en constituir un objeto a base de múltiples triángulos interiores, y que entre las muchas aplicaciones que se le da, destaca la del refinamiento de mallas [46, 52]. Otra importante aplicación para la Geometría Computacional, es la que se le da en la minería de datos, donde a través de algoritmos geométricos permite mejorar distintas técnicas de minería como por ejemplo el clustering (agrupamiento).

La representación computacional que se hace de los objetos geométricos es el primer punto a tratar cuando se enfrenta un problema geométrico. Para esto, se utilizan distintos índices espaciales como estructuras adicionales para la representación espacial, como por ejemplo el *K-D-B-tree* [45], el *R-tree* [30], el R^+*-tree* [49], el R^**-tree* [6] y el *SR-tree* [33], los cuales, dependiendo del problema (o el conjunto de problemas) a tratar, serán más o menos eficiente según sea el caso. El lograr una estructura adecuada y un algoritmo eficiente no siempre es una tarea fácil. Por ejemplo, supongamos una estructura tipo matriz booleana para representar un espacio de dos dimensiones tipo $\mathbb{N}^2$, donde las columnas representan las x y las filas representan las y, y las coordenadas (x_i, y_j) están dadas por los 1 que intersectan la columna C[i] y la fila F[j]. Claramente es una representación fiel y simple al plano cartesiano, que es fácil de comprender, sin embargo, para

representar regiones de baja densidad y de puntos extremadamente lejanos entre sí, se estará incurriendo en un uso ineficiente del espacio de almacenamiento para los 0's, sin mencionar además lo tedioso que es computar matrices $n \times n$ por su orden cuadrátrico e incluso cúbico para algunos algoritmos. Ahora, en cambio, suponiendo que sólo almacenamos las coordenadas en dos vectores de enteros, lo que significa un tipo de dato que necesita más espacio de almacenamiento que el booleano, sin embargo no incurrimos en un gasto innecesario por parte de los puntos no existentes. La segunda representación necesita menos espacio de almacenamiento que la primera, pero requiere un poco más de atención para comprenderla por parte de nosotros, ahora imaginemos abordar este problema con un R-tree donde claramente se necesitará mucha más atención para su comprensión pero que a la vez nos entregará mayores beneficios. Supongamos ahora que el problema geométrico a abordar para las estructuras anteriores es el de encontrar el punto más cercano dado un punto, donde claramente el R-tree es la mejor estructura para abordar el problema considerando todos los posibles casos, teniendo en el otro extremo la representación matricial como la peor estructura. Es así como podemos ver que la elección de una buena estructura, dependiendo del problema a abordar, es relevante en la eficiencia de la propuesta a realizar para dar solución a la problemática. Sin embargo, también es justo considerar el hecho de que muchas estructuras e índices han sido pensados para ser alojados en memoria secundaria y otras en memoria principal, y además creadas con la finalidad específica de tratar no sólo puntos sino también polígonos y polilíneas, con toda la topología que conllevan los mismos, dando una clara ventaja por sobre estructuras regulares que no han sido creadas con tales fines pero que pueden tratar los mismos problemas.

Los problemas geométricos nacen tanto de nuestras necesidades específicas dentro del campo, como también de la observación de problemas que existen en la vida real, y que pueden ser extrapolados al campo de la geometría. A su vez, estos problemas se pueden clasificar en cinco tipos distintos [17]: los de convexidad, los de proximidad, los de búsqueda geométrica, los de intersección y por último los de optimización.

Uno de los problemas más conocidos en Geometría Computacional es el del cálculo de la cerradura convexa (Convex Hull), el cual consiste en que dado un conjunto de puntos en un espacio, se pide obtener la lista de los puntos que forman el menor polígono convexo que logra contener a todos los puntos. Este problema ha sido tratado por variados algoritmos [26, 34, 11] donde se ha logrado mejorar de una complejidad temporal de $O(n \log n)$ a una de $O(n)$ para el caso especial de los R-tree con baja cantidad de overlap entre sus cuadrantes. Por lo tanto, creemos que un cambio de paradigma en el ordenamiento de los datos, la estructura de datos a utilizar, y la búsqueda a través de la misma, enfocado al objetivo, puede resultar en una mejor solución.

Entre las diferentes estructuras de datos que existen en la actualidad, están las estructuras compactas, las cuales aparecen como una alternativa para almacenar grandes cantidades de información de una manera eficiente en cuanto al espacio de almacenamiento, sin perder las propiedades de los datos contenidos pero, dándose en algunos casos, un mayor costo de procesamiento, en un llamado *trade-off* entre procesamiento y espacio[57]. La gran ganancia de este *trade-off* es el evitar tantos accesos de datos entre la memoria secundaria y la principal, lo cual se traduce no sólo en menos espacio sino también en menor tiempo de procesamiento total. Sin embargo, no todos los conjuntos de datos son candidatos a estructuras compactas, ya que por lo general las

mejoras son más notorias en grandes cantidades de datos, y que cumplen ciertas condiciones. Es así como los datos espaciales son un buen candidato para las estructuras compactas en cuanto a almacenamiento se trata [9, 13, 14, 15, 20, 31].

El lograr procesar algoritmos geométricos sobre una estructura de datos compacta, no sólo impactaría en el ahorro de espacio de almacenamiento para el conjunto de datos, sino que debiese favorecer la eficiencia de los algoritmos aplicados, siempre y cuando estos aprovechen las propiedades que ofrecen las estructuras. Por esta razón, en esta tesis se aborda el desafío de la cerradura convexa sobre un k^2-tree, buscando aprovechar al máximo los beneficios de la estructura tanto por su jerarquía como por la compactación de datos que logra.

A continuación presentamos formalmente la hipótesis y los objetivos de esta investigación.

1.2. Hipótesis y objetivos

Hipótesis

Es posible beneficiarse de las propiedades de las estructuras de datos compactas para diseñar algoritmos geométricos que permitan resolver la cerradura convexa sobre grandes conjuntos de datos espaciales discretos y de dos dimensiones, tal que sean más eficientes que resolverlos directamente con algoritmos tradicionales considerando que los datos se encuentran almacenados en memoria secundaria.

Objetivo general

Utilizar estructuras de datos compactas para el cálculo de la cerradura convexa sobre espacios geométricos discretos y de dos dimensiones para así lograr una mejora en la eficiencia de los algoritmos tanto a nivel de almacenamiento como a nivel de procesamiento.

Objetivos específicos

- Estudiar en profundidad la cerradura convexa y los distintos algoritmos existentes para poder dar una solución sobre una estructura de datos compacta.
- Estudiar las propiedades de las estructuras compactas que pueden ser útiles para la resolución de problemas de Geometría Computacional, en especial el k^2-tree ya que se ha demostrado su utilidad en el contexto espacial [12].
- Mejorar o por lo menos igualar la eficiencia temporal de los algoritmos existentes para problemas de Geometría Computacional.
- Evaluar experimentalmente los algoritmos propuestos, comparándolos con las soluciones existentes en Geometría Computacional.

1.3. Alcance de la investigación

Para la implementación de los algoritmos se utilizará el lenguaje de programación C++ junto a la librería de estructuras de datos compactos LIBCDS[1]. Para la experimentación de los algoritmos se utilizarán datos de prueba generados aleatoriamente como también se tomarán datos reales.

1.4. Metodología de trabajo

La metodología de trabajo a utilizar para la realización de esta tesis constará de una etapa de revisión bibliográfica, una etapa de diseño, una etapa de implementación y una etapa de experimentación. Las etapas definidas son secuenciales.

- **Revisión bibliográfica.-** Se hará una exhaustiva revisión de la literatura para determinar los trabajos relevantes y que puedan ser de utilidad a la tesis. Se enfocarán los esfuerzos en el uso de estructuras de datos para la resolución eficiente de problemas geométricos y se explorará a fondo las soluciones existentes para la cerradura convexa. Se hará también una revision exhaustiva de la aplicación de las estructuras de datos compactas en diversos problemas computacionales, para poder determinar propiedades que puedan ser útiles a la búsqueda de la cerradura convexa.
- **Diseño de algoritmos.-** En base a lo estudiado, y las propiedades de las estructuras de datos compactas que sean definidas como candidatas y prometedoras para la resolución de los problemas geométricos, se formularán los algoritmos correspondientes. Se definirán también los resultados esperados de la implementación de los algoritmos.
- **Implementación.-** Se codificarán los algoritmos planteados, además de medir su desempeño con datos generados aleatoriamente y datos reales. En el caso de no existir la codificación de algoritmos planteados en la literatura se procederá a codificar previo análisis de factibilidad y de importancia del algoritmo para el posterior análisis. Se contrastarán los resultados obtenidos con los resultados esperados.
- **Experimentación.-** Se diseñarán y ejecutarán una serie de experimentos que considerarán datos reales y sintéticos, que permitan evaluar el rendimiento de los algoritmos propuestos.

1.5. Organización de la tesis

A partir de este punto, la tesis contempla 6 capítulos distribuidos en 3 partes: la parte II mostrará una revisión de los contenidos más importantes referentes a la Gemoetría Computacional (capítulo 2) y las estructuras de datos compactas (capítulo 3). La parte III expondrá los algoritmos que proponemos e implementamos, detallándose la representación espacial sobre un k^2-tree y el cómo recuperar coordenadas (capítulo 4), luego se explicarán los algoritmos propuestos e implementados (capítulo 5), y luego se presentarán todos los datos relevantes a la experimentación

[1] https://github.com/fclaude/libcds

(capítulo 6). Por último, la parte IV contendrá las conclusiones y expectativas de trabajo futuro (capítulo 7) que se pueden recoger de este trabajo de tesis.

Parte II

Estado del arte

Capítulo 2

Geometría Computacional

La geometría ha estado presente en la historia del ser humano por más de 2.500 años, como base fundacionaria de las artes, la arquitectura, y las matemáticas, jugando además un rol central en muchas otras áreas [50]. En la actualidad, su uso e influencia ha aumentado su cobertura alcanzando las más diversas áreas de aplicación tales como la militar y la astronomía.

Por ejemplo, para calcular la distancia de una estrella cercana a nuestro sol se utiliza la técnica del paralaje, la cual es sólo aplicar trigonometría. Para esto se considera la inclinación de la estrella a la tierra por ejemplo en enero y en julio, luego teniendo como base de un triángulo isósceles el doble de la distancia de la tierra hacia el sol se tienen los datos necesarios para calcular la altura del triángulo, que corresponderá a la distancia entre la estrella analizada y el sol. Sin embargo, las distancias entre los involucrados pueden llegar a ser ridículamente grandes, y difíciles de manejar con exactitud por cualquiera de nosotros.

La utilidad de la geometría es prácticamente indiscutible, es por esto que cada vez se hace más necesario poder aplicar los conocimientos derivados de la misma con mayor eficiencia, rápidez y precisión, siendo el computador una herramienta que otorga un poder de procesamiento en constante mejora y que facilita las tareas tediosas y repetitivas. Es así como en los años 70 el campo de la Geometría Computacional (GC) emergió, tratando con problemas geométricos de uso cotidiano.

La GC entonces, puede ser definida como el estudio sistemático de algortimos y estructuras de datos para objetos geométricos, enfocado en lograr algoritmos que sean asintóticamente rápidos [19].

2.1. Estructuras de datos para la representación espacial

Representar un espacio bi-dimensional puede ser un tema sencillo si pensamos que sólo nos basta con definir los límites del espacio y para cada objeto (coordenada) presente en el conjunto de datos podemos utilizar por ejemplo dos variables $double(x, y)$. Sin embargo, a medida que los espacios crecen el tratamiento de sus datos se va volviendo cada vez más dificultoso, tanto por el tamaño que estos alcanzan en espacio de almacenamiento como por no contar con un orden espacial. Es por esto que se han creado estructuras de datos capaces de indexar objetos, adecuadas

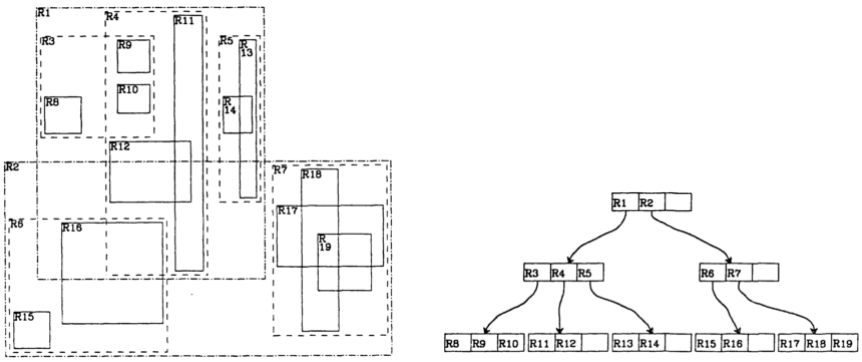
para el tratamiento espacial de datos, tales como el *K-D-B*-tree, *R*-tree, R^+-tree, R^*-tree entre otros. A continuación se explicará brevemente el *R*-tree, puesto que consideramos escencial su entendimiento debido a los algoritmos presentados más adelante sobre esta misma estructura.

2.1.1. *R*-tree

Es una extensión natural del *B*-tree. Cada nodo hoja contiene un MBR (por Minimun Bounding Rectangle) y un oid (por object identifier). Los nodos internos contienen un MBR y una referencia del nodo hijo [30] (ver Fig. 2.1).

Un MBR es, como dice su definición, el menor rectángulo que contiene un polígono (el polígono representa un conjunto de datos dentro del espacio), por lo que en las aristas de este rectángulo siempre se encontrarán puntos (cuando exploremos las hojas).

La organización del *R*-tree se basa en dividir el espacio a representar en rectángulos, los cuales deben contener todos los elementos existentes en el espacio. El tamaño de los rectángulos será mayor a medida que estemos más cerca de la raíz, y para cada rectángulo existirá un sub-división del mismo en rectángulos menores, los cuales corresponderán a los nodos hijos en el árbol. Esto se repite hasta llegar a las hojas donde existen todos los elementos del espacio organizados y de eficiente acceso por las divisiones hechas a mayor altura sobre el árbol (ver Fig. 2.1).



(a) División del espacio (b) Representación del *R*-tree

Fig. 2.1: *R*-tree [30]

La búsqueda en el *R*-tree es de manera recursiva, buscando candidatos y descartando regiones, pudiendo retornar valores vacíos como también conjuntos de punteros que cumplen el objetivo de la consulta. Sea T el nodo raíz, entonces la búsqueda de todos los índices que solapan el área de búsqueda S se define a través de los siguientes pasos:

- **S1.** Si T no es una hoja, revisar cada entrada E para determinar cuales solapan a S. Para cada entrada E que solapa S realizar búsqueda recursiva con T como $E.p$ (puntero al nodo hijo).
- **S2.** Si T es una hoja, revisar cada entrada E para determinar cuales $E.I$ (identificador de la tupla) solapan S. Las que cumplan la condición son parte de la respuesta final.

La inserción y eliminación son similares a la del B-tree pero con una complejidad mucho mayor.

2.2. Problemas geométricos

2.2.1. Consulta por rangos.

Es un problema de Geometría Computacional donde dado un conjunto de puntos o coordenadas $S = \{p_1, p_2, ..., p_n\}$ se requiere encontrar todos los puntos que estén en el rango indicado por S', siendo S' un objeto geométrico formado dentro de S. Los puntos pueden darse en un espacio k-dimensional, donde el rango de búsqueda puede ser representado por un hiper-rectángulo sobre el mismo espacio. A su vez, el proceso de recuperar todos los puntos que cumplan con las condiciones establecidas por la consulta por rango se denomina como *range searching* [8], que, por su parte, estudia las distintas formas en que los algoritmos satisfacen la consulta. Bajo esta definición es que la consulta por rango puede ser extrapolada a otras áreas, como por ejemplo la consulta de objetos que cumplan un set de características especiales en común para un conjunto de objetos. Un ejemplo simple sería obtener todos los registros de una base de datos relacional en que para la tabla "*personas*" los apellidos sean "*Castro Arias*".

Este problema tiene muchas aplicaciones útiles en la actualidad, siendo una de las más relevantes la empleada en las consultas a bases de datos y los conjuntos de las mismas. Dada la importancia del problema, es que se han enfocado esfuerzos en desarrollar estructuras de datos que manejen tanto de manera eficiente el tiempo de consulta como también el espacio de almacenamiento, generando un *trade-off* entre ambos aspectos [2, 29, 20]. La concepción de la estructura de datos ha llegado incluso a considerar el ambiente en que se desenvolverá el algoritmo, como es el caso del HD Tree [28] que es un árbol pensado para computarse a través de una red P2P para consultas por rango en espacios multidimensionales.

2.2.2. K vecinos más cercanos.

También conocido como kNN (por sus siglas en inglés k Nearest Neighbor) y considerado como uno de los 10 algoritmos más importantes en la minería de datos [54]. Es un problema geométrico que dado un punto, busca dar con los k puntos existentes en el espacio que estén más cerca, basados en la distancia, la cual puede ser euclidiana, manhattan, u otras según lo especifique el problema a tratar.

Una de las formas de dar solución a este problema es hacer una búsqueda exhaustiva calculando la distancia entre el punto de búsqueda y los puntos existentes, para lograr así una solución determinista. Sin embargo, también se puede enfocar la solución a un plano estocástico, a través de algoritmos probabilísticos para dar soluciones más rápidas pero con una probabilidad de error. Por ejemplo en [1] postulan una solución con un algoritmo que combina algoritmos genéticos con redes bayesianas, a través de máquinas de aprendizaje, donde a base de simulaciones de conjuntos aleatorios de soluciones se genera un vector de soluciones posibles que son tratadas luego con algoritmos genéticos para refinar la solución. Todo esto, con el fin de evitar las búsquedas exhaustivas, sin embargo, como ya se ha visto, existen estructuras de datos que permiten descartar

zonas enteras del espacio [30, 49, 6, 33]. Aunque para este problema no se pueden descartar zonas, lo que se puede hacer es asignar un orden de búsqueda.

La resolución de los kNN se puede clasificar en dos categorías distintas, pero no excluyentes, según [10]: las libres de estructura y las basadas en estructuras. Las libres de estructuras se refiere a todas los algoritmos que se desarrollan pensando en un conjunto de datos sin mayores particularidades, donde el problema a solucionar son las comparaciones y las mediciones de las distancias sin importar mayormente la estructura de datos que almacena los objetos. Un ejemplo de esta categoría son los WkNN [4] (por sus siglas en inglés Weighted kNN) que designan un peso a los puntos según su distancia con el punto de búsqueda, y en base a estos pesos se definen clases para clasificar los puntos y facilitar la búsqueda. La segunda categoría se refiere a los algoritmos que se hacen pensados en estructuras de datos específicas para dar solución al problema, como por ejemplo el k-d tree [40] y el SR tree [33], siendo esta última estructura ideada especialmente para dar solución a este problema.

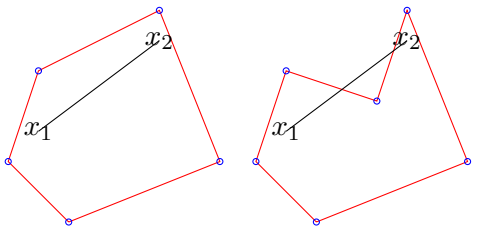
Otro problema clásico de la GC es la cerradura convexa, el cual dada su importancia para esta tesis se tratará como una sección completa a continuación.

2.3. Cerradura Convexa

Dado un conjunto S de puntos x_i en un espacio $\mathbb{R}^n$, se dirá que este conjunto es convexo si y sólo si cumple que:

$$\forall x_1, x_2 \in S \rightarrow \lambda x_1 + (1-\lambda)x_2 \in S, \lambda \in [0,1]$$

esto se refiere, a que dados dos puntos pertenecientes al conjunto S y que forman un segmento cerrado, este segmento se encuentra siempre totalmente contenido dentro del conjunto S (ver Fig. 2.2).



(a) Convexo (b) No convexo

Fig. 2.2: Conjunto Convexo y no Convexo

Algunos conjuntos especiales que cumplen la propiedad convexa son:

- $S = \emptyset$
- $S = x | \forall x \in S \ \rightarrow \ x_i = x_j, \ \forall i, j \in \mathbb{R}$
- $S = \mathbb{R}^n$

- Si S' y S'' son convexos, entonces si $S = S' \bigcap S''$, S es convexo

La convexidad es una propiedad deseable frente a problemas de optimización, pero que no se da en todos los conjuntos. Sin embargo, cuando un conjunto no es convexo, se habla de la cerradura convexa, que es por definición el menor polígono o politopo convexo que encierra todos los puntos pertenecientes al espacio (ver Fig. 2.3).

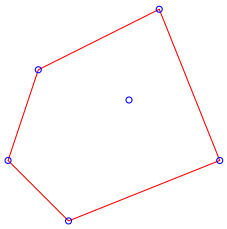


Fig. 2.3: Polígono convexo para un conjunto no convexo

Según la definición de convexidad, dice que un conjunto es convexo si y sólo si todos sus subconjuntos son convexos, lo cual permite buscar la convexidad de un conjunto a través del clásico divide y vencerás, separando el espacio en zonas y buscando polígonos convexos para después mezclarlos en un polígono convexo para el conjunto total.

Los ángulos interiores de un polígono convexo no pueden medir más de 180°, en caso de que un polígono no cumpla está cualidad no se considera como convexo.

Giro Reloj. Un concepto importante antes de adentrarnos en los algoritmos para la cerradura convexa es el llamado giroreloj. Si tenemos 3 puntos: $A(x_1, y_1)$, $B(x_2, y_2)$ y $C(x_3, y_3)$, y queremos saber que giro realiza la secuencia $\overline{ACB}$ aplicamos el producto vectorial sobre los segmentos cerrados formados por $\overline{AC}$ y $\overline{BC}$. Para el cálculo del producto vectorial se utiliza la siguiente fórmula:

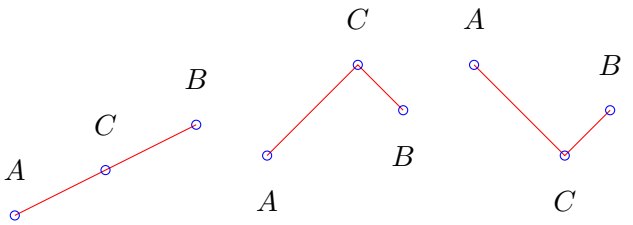
$$(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$$

Ahora, si el producto vectorial da como resultado 0 significa que $\overline{ACB}$ es una recta; si el producto vectorial da como resultado un valor mayor a 0 significa que el giro es a la izquierda; y por último si el resultado es un valor menor a 0 el giro es a la derecha (ver Fig. 2.4).

Distancia de un punto a un segmento. Para calcular la distancia de un punto C a un segmento $\overline{AB}$, siendo $A(x_1, y_1)$, $B(x_2, y_2)$ *y* $C(x_3, y_3)$, se utiliza una fórmula similar a la del cálculo del producto vectorial:

$$\frac{(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

El **giroreloj** y la **distancia de un punto a un segmento** no son propiedades exclusivas para la búsqueda de la cerradura convexa de un conjunto de datos, sino que son propiedades de la geometría euclidiana que pueden ser aplicadas a otros problemas, sin embargo son de vital



(a) giroreloj = 0 (b) giroreloj > 0 (c) giroreloj < 0

Fig. 2.4: Giroreloj

importancia para nuestro enfoque e implementación de un algoritmo para la cerradura convexa sobre una estructura de datos compacta.

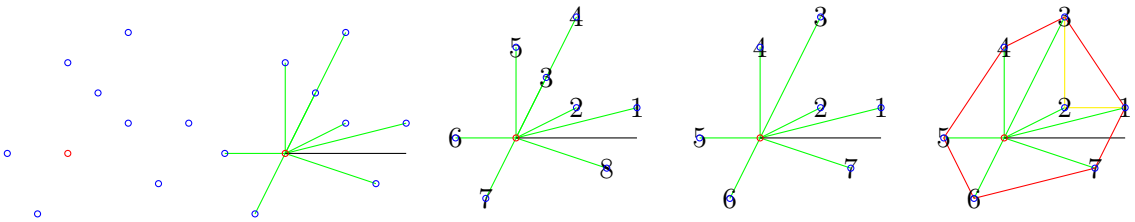
2.4. Algoritmos para la Cerradura Convexa

2.4.1. Graham Scan

En el año 1972 Ronald Graham propuso un algoritmo eficiente para dar solución a la cerradura convexa con una complejidad de $O(n \log n)$ en el peor escenario [25]. Dado un conjunto S de puntos s_i, con $i > 0$, el algoritmo propuesto por Graham resuelve el problema a través de los siguientes 5 pasos (ver Fig. 2.5):

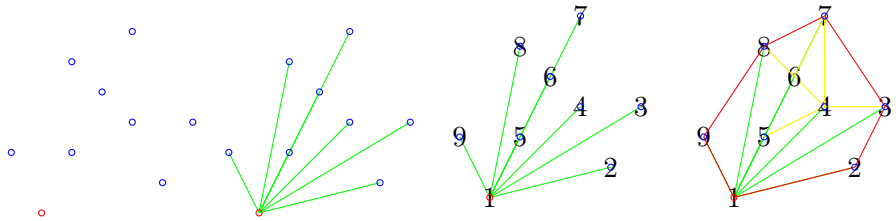
- **Paso 1**: Buscar y definir un punto que pueda ser considerado como centroide del conjunto S, es decir, que no se encuentre en las aristas del polígono convexo.
- **Paso 2**: Definir cada punto como una coordenada polar respecto al centroide y a una recta definida arbitrariamente.
- **Paso 3**: Ordenar los valores de S de manera creciente según su coordenada polar.
- **Paso 4**: Si dos puntos son colineales respecto al centroide, se elimina el de menor amplitud al mismo.
- **Paso 5**: Se define una terna según el orden de los datos, y se verifica el giroreloj que efectúa, si el giro es contrario se vuelve al último giro correcto, sino se avanza a la siguiente terna. Este paso se repite hasta analizar todas las ternas consecutivas posibles.

Del algoritmo propuesto por Graham (Ver Alg. 2.1) se deriva el *Graham Scan* el cual en vez de tomar un centroide, escoge el punto con menor valor para y, y a través del mismo calcula los ángulos a los demás puntos del conjunto S, ordenándolos por el ángulo formado. Este algoritmo puede o no eliminar los valores colineales, aún así se mantiene el formato de las ternas de búsqueda y el giroreloj para ver la pertinencia de los puntos al polígono convexo (ver Fig. 2.6).



(a) Paso 1 (b) Paso 2 (c) Paso 3 (d) Paso 4 (e) Paso 5

Fig. 2.5: Algoritmo de Graham



(a) Paso 1 (b) Paso 2 (c) Paso 3 (d) Paso 4

Fig. 2.6: Graham Scan

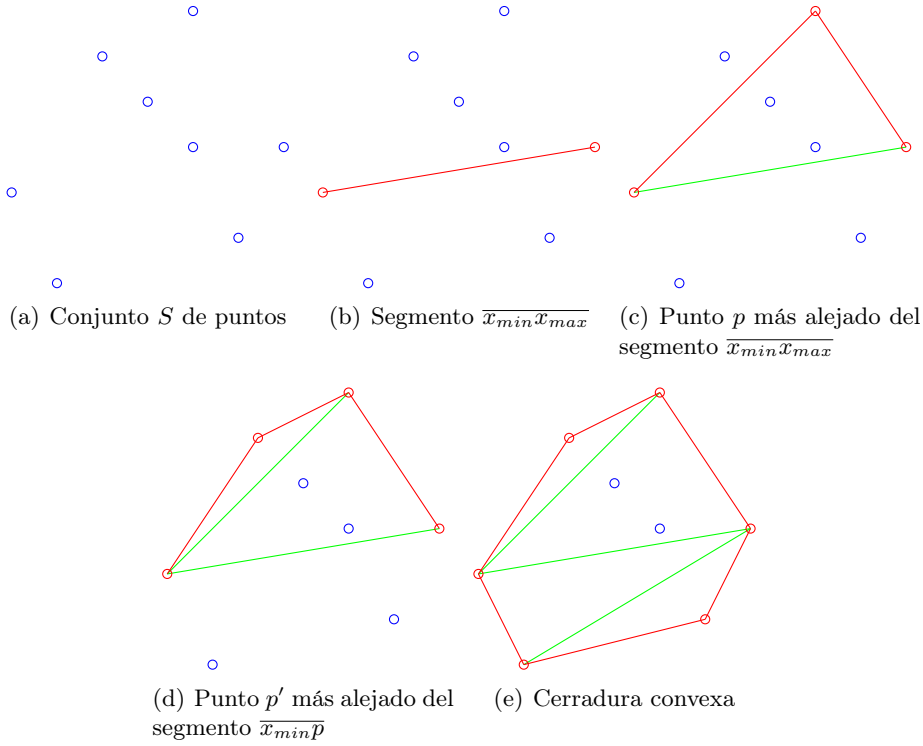
```
1  Algoritmo: GrahamScan()
2  P = p_0, p_1, ..., p_n
3  Encontrar un punto p_pivote que sea extremo inferior;
4  Definir p_pivote como p_0;
5  Ordenar todos los puntos ascendentemente respecto al ángulo formado por p_i y p_pivote;
6  PilaS = {p_0, p_1} = {p_t, p_{t-1}};
7  t = índice del tope de la pila;
8  i = 2;
9  while i < n do
10     if p_i está a la izquierda del segmento p_{t-1}p_t then
11         Push(p_i, S);
12         i++;
13     else
14         Pop(S);
```

Alg. 2.1: Algoritmo Graham Scan

2.4.2. Quickhull

El quickhull [42, 5], es un algoritmo que se basa en la búsqueda de puntos extremos, ya que los puntos extremos (para la ordenada y la abcisa) son siempre vértices del polígono convexo. Primero define un segmento formado por el menor valor de x (x_{min}) y el mayor valor de x (x_{max}).

No es excluyente que sean los valores extremos de x, pueden ser también los valores extremos de y o una combinación de ambos. Luego calcula el punto p más distante sobre o bajo el segmento (dependiendo la orientación de la búsqueda). Una vez encontrado el punto p más distante se forma un nuevo segmento con uno de los extremos primeramente definidos (x_{min} o x_{max}) y el nuevo punto p obtenido, luego se repite el proceso en base a los nuevos segmentos formados hasta conseguir el polígono convexo (ver Fig. 2.7).



(a) Conjunto S de puntos (b) Segmento $\overline{x_{min}x_{max}}$ (c) Punto p más alejado del segmento $\overline{x_{min}x_{max}}$

(d) Punto p' más alejado del segmento $\overline{x_{min}p}$ (e) Cerradura convexa

Fig. 2.7: Quickhull

Este algoritmo funciona mejor cuando existen pocos valores extremos, ya que así los segmentos formados por los valores extremos descartan mayor cantidad de puntos, no así cuando la mayor cantidad de los puntos son valores extremos.

Si bien el peor caso para el quickhull es de complejidad $O(n^2)$, se considera un algoritmo eficiente ya que en promedio, para datos aleatorios, su complejidad es $O(n \log n)$.

2.4.3. Cerradura convexa sobre un R-tree

Este problema también se ha tratado en base a la definición de estructuras propias y adecuadas al problema que tratan de manera eficiente ambos recursos (tiempo y almacenamiento). Una de las estructuras que se ha utilizado para su tratamiento, y que no es exactamente una estructura que se ha creado con la finalidad de resolver este problema en particular, es el R-tree[11]. Los nodos internos contienen MBRs y las correspondientes referencias a los nodos hijos. Esta estructura

permite la exploración por regiones, lo que facilita enormemente la búsqueda de objetos en un plano espacial. Los algoritmos propuestos en [11] se basan en la separación del plano en cuadrantes, definidos por las coordenadas que contengan los mínimos y los máximos x e y del conjunto de objetos o puntos.

El primer algoritmo (Distance priority) se basa en la comparación del punto más cercano, partiendo de los extremos, y viendo si es que éste no gira en sentido contrario dependiendo hacia donde se están concatenando los puntos para la cerradura convexa (ver Alg. 2.2).

```
1  Algoritmo: distance_priority_ch(raiz,pagina)
2  apl = raiz
3  minxpoint = buscar en base de datos por el mínimo x;
4  minypoint = buscar en base de datos por el mínimo y;
5  tch = {minxpoint, minypoint};
6  while apl no esté vacío do
7      b = elemento de apl con mayor distancia a tch;
8      desencolar b desde el disco;
9      cargar b desde el disco;
10     if b es una página de datos then
11         foreach v = elemento de b do
12             if esta_a_la_izquierda(v, tch) then
13                 tch = tch ∪ b;
14                 while tch no sea convexo do
15                     remover next(v) desde tch;
16     else
17         s = conjunto de páginas hijas de b;
18         apl = apl ∪ s;
19     forall Páginas p = elementos de apl do
20         if esta_a_la_derecha(p, tch) then
21             apl = apl/p;
```

Alg. 2.2: Algoritmo distance priority

El algoritmo *Distance priority*, lo primero que hace es definir un índice que apunte a la raíz del árbol (línea 2), el cual por lo demás se encuentra almacenado en memoria secundaria; luego define el mínimo valor de x y el mínimo valor de y (línea 3 y 4), a través de los cuales forma un segmento llamado *tch* (línea 5). Definidas las variables necesarias, comienza a explorar el árbol a través del índice *apl*, donde, para el elemento b con mayor distancia al segmento *tch* lo cargará desde el disco (líneas 7, 8 y 9); si el elemento b cargado es una hoja, explorará cada uno de los datos v presentes en b (línea 11), verificando que estos se encuentren hacia la izquierda del segmento *tch*, y en tal caso los agregará al segmento *tch* (línea 13) el cual será refinado hasta formar un segmento convexo (línea 14 y 15). Para el caso en que el elemento b cargado no sea una hoja (línea 16) se tomarán los valores hijos de b y se agregarán al índice de búsqueda *apl*

(línea 17 y 18); siendo para *apl*, p sus elementos páginas, que serán explorados y en caso de estar totalmente a la derecha del segmento *tch*, ser eliminados del índice de búsqueda *apl*.

El segundo algoritmo (Depth-First) busca el punto más alejado a los extremos que definen el cuadrante y procede a verificar si su giro es en el sentido de la cerradura (ver Alg. 2.3).

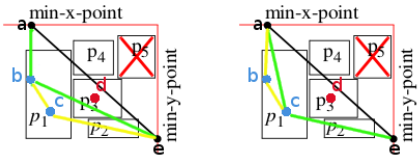
```
1  Algoritmo: depth_first_ch(b :página)
2  Cargar b desde el disco;
3  if b es una página de datos then
4      V = conjunto de puntos almacenados en b;
5      Ordenar ascendentemente V por la coordenada x;
6      foreach v ∈ V do
7          l = segmento que conecta el último punto de tch con minypoint;
8          if esta_a_la_izquierda(v, l) then
9              m = segmento que conecta v con el punto antes del último punto de tch;
10             while esta_a_la_derecha(último punto de tch, m) do
11                 Eliminar el último punto desde tch;
12             Agregar v a tch;
13 else
14     P = conjunto de páginas hijas de b;
15     //restringir P a páginas frontales
16     foreach p ∈ P do
17         foreach q ∈ P do
18             if excluye(p, q) then
19                 Eliminar q desde P;
20     foreach p ∈ P do
21         l = segmento que conecta el último punto de tch con minypoint;
22         if no_esta_a_la_derecha(p, l) then
23             depth_first_ch(p);
```

Alg. 2.3: Algoritmo depth first

Lo primero que el algoritmo *Depth first* hace es cargar una página b desde el disco, en el caso que ésta sea una hoja (línea 3) considera todos los elementos presentes como el conjunto V, y para cada $v \in V$ forma un segmento l entre el último valor de *tch* con *minypoint* (línea 7), para verificar luego si el valor v se encuentra a la izquierda del segmento l (línea 8), en tal caso forma un nuevo segmento m entre v y el penúltimo valor de *tch* y evalúa si el último punto de *tch* está a la derecha de m (líneas 9 y 10). Se eliminan los últimos puntos de *tch* hasta encontrar uno que no esté a la derecha de m (línea 11). Hecho esto se agrega v como último elemento de *tch* (línea 12). Si la página b extraída no es una hoja, se define P como un conjunto de páginas o nodos hijas de b y se evalúa para cada $p \in P$ y a la vez para cada $q \in P$ si son páginas excluyentes (líneas 16, 17 y 18). En el caso que sean excluyentes se elimina q desde P. Del conjunto P resultante se evalúa

para cada p si no está a la derecha del segmento l (segmento formado por el último punto de *tch* con *minypoint*). Si p no está a la derecha se hace un llamado recursivo a la misma función así hasta llegar a las hojas.

Las estructuras como los R-trees permiten para ambos algoritmos descartar regiones enteras de puntos cuando ya se ha seleccionado un miembro de la cerradura convexa, debido a las propiedades de la estructura que permiten saber la ubicación posicional referente a un punto a través de la navegación de esta (ver Fig. 2.8).

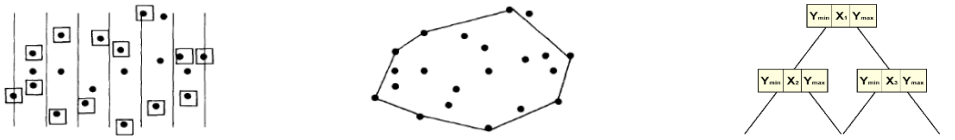


(a) Distance priority (b) Depth-First

Fig. 2.8: Algoritmos convex hull sobre un R-tree [11]

2.4.4. Otros algoritmos y estructuras para la cerradura convexa

No sólo el R-tree es la estructura tipo árbol utilizada en la resolución de la cerradura convexa, sino que también el árbol binario propuesto en [51], donde en cada nodo se guardarán 3 datos. El dato central en cada nodo será un valor para x, ya que el plano se procederá a dividir en tantas columnas como $x's$ existan en el plano, y para cada una habrá un nodo. Luego, para cada columna se buscan los 2 y extremos y se almacenan en los nodos correspondientes completando los 3 datos (ver Fig. 2.9). Una vez que se ha completado el árbol, se procede a explorar dos veces de manera in-order. El plano se divide en dos hemisferios, y cada recorrido del árbol será para encontrar los puntos de la cerradura convexa para el hemisferio norte, como también para el hemisferio sur, suponiendo la posicionalidad de los objetos en base a los puntos cardinales. Este método no asegura la cerradura convexa, sin embargo, y como postulan sus creadores, sirve de base para ser utilizado por otros algoritmos de cerradura convexa, facilitándoles el trabajo. Uno de los algoritmos que se puede utilizar para lograr la cerradura a base del árbol binario es el quickhull [5], el cual presenta un enfoque que también es utilizado en los algoritmos propuestos por [11] para el R-tree.



(a) Extremos para cada ordenada (b) Polígono de extremos (c) Estructura del árbol

Fig. 2.9: Cerradura convexa basado en un árbol binario [51]

Existe también un enfoque similar a los ya descritos, y es el propuesto en [34] donde se basan en el razonamiento de una persona al ver un conjunto de puntos en un espacio y que sabe cuales despreciar y cuales explorar con solo mirar. Lo más probable es que si miramos un conjunto de puntos, cualquiera sea, descartaremos de inmediato una gran zona ubicada en el centro de los puntos. Para simular esto, se define primero una cerradura convexa basada en puntos extremos, con ocho es un buen comienzo. Luego se define un círculo cuyo radio está dado por la distancia de un centroide y una de las aristas de la cerradura convexa, sin que el círculo salga del polígono. Luego, a partir del centroide se extienden partes del círculo en base a los puntos del polígono, descartando así más puntos para refinar la cerradura convexa (ver Fig. 2.10).

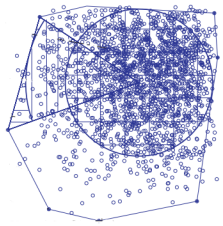


Fig. 2.10: Cerradura convexa en base al mayor círculo inscrito [34]

Capítulo 3

Estructuras de datos compactas

Los grandes volúmenes de datos que se manejan en la actualidad parecen no ser un problema cuando hablamos de almacenamiento en memoria secundaria, ya que estos dispositivos han alcanzado una capacidad casi ilimitada y en constante crecimiento. Pero quedarse sólo en la arista del almacenamiento en memoria secundaria es no ver los problemas que esto acarrea. Los accesos desde memoria secundaria a la principal son costosos y el poder reducirlos es una ganancia en eficiencia. Es así como hay estructuras de datos que intentan replicar grandes volúmenes de datos de manera compacta, es decir, sin perder las propiedades de accesibilidad a los datos dentro de la misma estructura (a diferencia de la compresión de datos). Éstas estructuras buscan aprovechar al máximo la jerarquía de memoria, evitando lo más posible los accesos a memoria secundaria e incluso intentando almacenar todos los datos en caché para un mayor provecho de los recursos.

Todas las estructuras presentes en esta tesis responden al modelo *word-RAM*, el cual se basa en la existencia de un número constante de registros operables y direccionables en la RAM, tal que cada registro es de tamaño $\Theta(\log n)$[1] bits, siendo n el tamaño de la RAM en bits. La memoria es dividida en bloques contiguos de tamaño $\Theta(\log n)$ bits que corresponden a las palabras (*words*), y que son de carga y almacenamiento en tiempo constante.

3.1. Estructuras de datos compactas

3.1.1. Entropía

La entropía es una medida que nos permite cuantificar el contenido de información en un mensaje [21]. Cuando hablamos de entropía es necesario comprender el concepto de alfabeto fuente Σ de tamaño σ y alfabeto destino Σ' de tamaño σ'. Entonces, al calcular la entropía obtendremos la cantidad promedio de simbolos x'_i de Σ' necesaria para codificar un símbolo x_i de Σ. Este promedio será la cota mínima para cualquier algoritmo de compresión. El cálculo se basa en la probabilidad de ocurrencia de un símbolo de Σ y está dado por la siguiente fórmula:

[1] A menos que se especifíque, cada vez que hablemos de logaritmos nos referiremos a uno en base 2, es decir, $\log_2 n$.

$$H \;=\; -\sum_{i=1}^{\sigma} p(x_i)\log_{\sigma'} p(x_i)$$

el cual es también llamado modelo de orden 0. Sin embargo, existen distintos modelos de entropía en base a la probabilidad de los símbolos, como por ejemplo el modelo orden base (H_{-1}) donde la probabilidad de ocurrencia de cada símbolo x_i en Σ es idéntica y por tanto $H_{-1} = \log_D \sigma$.

El modelo de primer orden (H_1) calcula la probabilidad de ocurrencia de un símbolo x_i en base a la ocurrencia del símbolo x_j anterior, y se calcula así: $H_1 = \sum_{j=1}^{n} p(x_j) \sum_{i=1}^{n} P_{x_i|x_j} \log_{\sigma'}(P_{x_i|x_j})$.

El modelo de segundo orden (H_2) es aquél en que la probabilidad de ocurrencia de un símbolo x_i se calcula en base a la ocurrencia de los dos símbolos anteriores. Los modelos de mayor orden siguen la misma idea de las secuencias previas.

Para comprender mejor esto definamos un alfabeto Σ el cual tendrá un largo σ que es la cantidad de caracteres no repetidos que contiene. Se define un conjunto S como el mensaje (con todos sus símbolos contenidos en Σ) el cual tiene un largo n. σ no puede ser mayor a n. Además hay que considerar un alfabeto destino Σ' de tamaño σ'.

- $S = \{$*alabar a la alabarda*$\}$, de tamaño $n = 20$
- $\Sigma = \{$*",a,b,d,l,r*$\}$, de tamaño $\sigma = 6$
- $n > \sigma$
- $\Sigma' = \{0, 1\}$, de tamaño $\sigma' = 2$

Como en este ejemplo conocemos la probabilidad real de ocurrencia de los símbolos de Σ en S (cabe mencionar que S es un ejemplo recurrente cuando se habla de compresión [38])podemos calcular la entropía empírica. La entropía empírica se calcúla en base a la cantidad de ocurrencias n_c de los elementos de Σ en S y está dada por la siguiente fórmula:

$$H_0(S) = \sum_{c\in\Sigma}^{\sigma}(\frac{n_c}{n}\log_{\sigma'}\frac{n}{n_c})$$

Si nos fijamos en el conjunto S y la cantidad de ocurrencias para cada elemento de Σ en S podemos calcular que la entropía empírica para este conjunto es de 2,219 (ver Tabla 3.1). Este valor nos indica que en promedio necesitamos 2,219 bits para cada elemento de S, es decir, para codificar S con Σ' necesitariamos $nH_0(S)$ o bien $\lceil 44{,}38 \rceil$ bits como mínimo. La cantidad de bits utilizados dependerá del sistema de codificación empleado. Por ejemplo si utilizamos un codificación **ASCII** se necesitan $n * 8 = 160$ bits. Como la entropía es la cantidad de información del mensaje, la diferencia entre la entropía y los bits utilizados realmente se denomina redundancia. Así los sistemas de compresión lo que hacen es reducir la redundancia del mensaje, pero no pueden ir más allá de la entropía sin pérdida de información.

Caracter $c \in \Sigma$	**Ocurrencias de c en S**	**Entropía**
a	9	$(\frac{9}{20}\log_2\frac{20}{9}) = 0{,}517$
l	3	$(\frac{3}{20}\log_2\frac{20}{3}) = 0{,}411$
”	3	$(\frac{3}{20}\log_2\frac{20}{3}) = 0{,}411$
b	2	$(\frac{2}{20}\log_2\frac{20}{2}) = 0{,}332$
r	2	$(\frac{2}{20}\log_2\frac{20}{2}) = 0{,}332$
d	1	$(\frac{1}{20}\log_2\frac{20}{1}) = 0{,}216$
	Entropía de S	$2,219$

Tabla 3.1: Cálculo de la entropía

3.1.2. Estructuras de datos sucintas, compactas e implícitas

Nos referiremos al óptimo teórico de información (OPT) como el menor espacio requerido para una estructura de una manera espacio-eficiente. Es así como se definen 3 estructuras en base a este criterio:

Estructuras sucintas. Son aquellas estructuras que permiten representar datos usando un espacio de almacenamiento de orden $OPT + o(OPT)$, cercano a la cota inferior teórica de información y además soportar de manera eficiente las operaciones de navegación sobre la misma estructura [32].

Estructuras compactas. Son aquellas que pueden ser representadas usando un espacio de almacenamiento de orden $O(OPT)$, manteniendo las operaciones de navegación sobre la misma.

Estructuras implícitas. Son aquellas estructuras de orden $OPT + O(1)$ en las cuales el orden relativo de los valores almacenados está implícito en los patrones de los elementos retenidos, no requeriéndose una estructura auxiliar para las operaciones de navegación sobre la estructura original [37].

3.1.3. Bitmap

Una de las estructuras más importantes en la compactación de datos y otras estructuras son los bitmaps o vectores de bits, los cuales tal como lo dice su nombre son vectores unidimensionales de bits. Sin embargo, estas estructuras cuentan además de otras estructuras adicionales, las que permiten aplicar operaciones fundamentales en tiempo constante.

Las operaciones fundamentales sobre un bitmap son **rank**, **select** y **access** (ver Fig. 3.1).

- **Rank:** calcula la cantidad de $1's$ o $0's$ (según se le indique) que existen a la posición m de un bitmap.
- **Select:** calcula la posición del bitmap en que se encuentran el *n-ésimo* 1 o 0 (según se le indique).

- **Access:** indica si el índice accesado sobre el bitmap es un 1 o un 0.

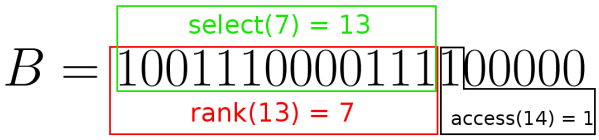


Fig. 3.1: Operaciones fundamentales sobre un bitmap

El bitmap como estructura de datos tiene variadas implementaciones, entre las que destacan la de Munro [36]; la de Raman, Raman y Rao [44]; y la de Okanohara y Sadakane [39] (ver Tabla 3.2), donde cada uno presenta diferentes estructuras auxiliares para dar solución a las operaciones *rank* y *select* eficientemente. Estas estructuras auxiliares son tratadas a través del problema de indexación de diccionarios.

Se define m la cantidad de 1's en un bitmap y n como la cantidad total de elementos en un bitmap.

Método	**Tamaño (bits)**	**rank**	**select**
Munro [36]	$n + O(\frac{n \log\log n}{\log n})$	$O(1)$	$O(1)$
Raman et al. [44]	$nH_0(B) + O(\frac{n \log\log n}{\log n})$	$O(1)$	$O(1)$
Okanohara y Sadakane [39]	$m \log \frac{n}{m} + 1{,}92m + o(m)$	$O(\log \frac{n}{m} + \frac{\log^4 m}{\log n})$	$O(\frac{\log^4 m}{\log n})$

Tabla 3.2: Representaciones de bitmaps [18]

Munro. Implementa distintas estructuras para *rank* y *select*, siendo la utilizada para *select* más compleja, pero ambas de respuesta en tiempo constante [36].

Para *rank* lo primero que hace es dividir el bitmap en bloques de tamaño $\lceil \log^2 m \rceil$, manteniendo una tabla que contiene el número de 1's en la última posición de cada bloque. Luego divide cada bloque en sub-bloques de tamaño $\lceil \frac{1}{2} \log m \rceil$, manteniendo una tabla que registra la cantidad de 1's en el bloque en la última posición de cada sub-bloque. Por último pre-computa una tabla dando el número de 1's a cada posición en cada sub-bloque existente, en un tamaño de $O(\sqrt{m} \log m \log\log m)$ bits, ocupando así un espacio adicional de $o(m)$ bits.

Para *select* define una estructura o directorio que registra la posición de cada $(\log m \log\log m)$ 1 en el bitmap. Luego establece un rango r entre dos valores en la primera estructura, y considera un sub-directorio para este rango. Si $r \geq (\log m \log\log m)^2$ entonces se almacenan todos los 1's, en un espacio de $O(r/\log\log m)$. Por otro lado, divide el rango r y guarda la posición de cada $(\log r \log\log m)$ 1 en el segundo nivel del directorio. Esto toma $O(r/\log\log m)$ bits para cada rango de tamaño r, en total $O(m/\log\log m)$ bits para el vector completo. Después un nivel más de subdivisión de rango similar, el tamaño del rango se reduce a lo más a $(\log\log m)$. Computando *select* en pequeños rangos con un tabla pre-computada en un espacio total de $O(m)$ bits auxiliares.

Raman, Raman y Rao. Sea S el conjunto de datos o bien dicho el bitmap para nuestro caso. Introduce el concepto de *bucketing por el bit más significativo* (MSB por Most Significative Bit), donde la idea es aplicar funciones hash en un alto nivel para las claves en S, las cuales simplemente toman el valor del t MSB de una clave. A diferencia de Munro que ocupa punteros para la representación, aquí se ocupan representaciones sucintas de los índices de los múltiples buckets [44].

Para dar solución a *rank* se utilizan $n\lceil \log m \rceil + O(\log \log m)$ bits, y $n(\lceil \log m \rceil + \lceil \log n \rceil) + O(\log \log m)$ bits para *select*. Ambas con respuesta en tiempo constante.

Okanohara y Sadakane. Okanohara y Sadakane [39] proponen 5 estructuras auxiliares para el problema de la indexación de diccionarios (ver Tabla 3.3), los cuales están pensado cada uno en distintos aspectos del bitmap como por ejemplo lo disperso de los datos.

Método	**Tamaño (bits)**	**rank**	**select**
esp	$nH_0(S) + o(n)$	$O(1)$	$O(1)$
recrank	$1{,}44m \log \frac{n}{m} + m + o(n)$	$O(\log \frac{n}{m})$	$O(\log \frac{n}{m})$
vcode	$m \log(n/\log^2 n) + o(n)$	$O(\log^2 n)$	$O(\log n)$
sarray	$m \log \frac{n}{m} + 2m + o(m)$	$O(\log \frac{n}{m}) + O(\log^4 m/\log n)$	$O(\log^4 m/\log n)$
darray	$n + o(n)$	$O(1)$	$O(\log^4 m/\log n)$

Tabla 3.3: Estructuras de Okanohara y Sadakane [39]

3.1.4. Árboles sucintos

Existen $\frac{1}{2n+1}\binom{2n+1}{n}$ árboles ordinales diferentes con n nodos. Entonces, el mínimo número de bits necesitados para diferenciar un árbol de otro es de

$$\left\lceil \log \frac{1}{2n+1}\binom{2n+1}{n} \right\rceil \approx \lceil \log 4^n \rceil \approx 2n - O(\log n)\ bits$$

lo que significa que necesitamos sólo 2 bits para cada nodo. El árbol sucinto busca construir una estructura que con sólo 2 bits por nodo mantenga las propiedades de navegación sobre el mismo árbol.

BP. El BP [32] (por Preorder Balancing Parentheses) es una representación sucinta de un árbol ordinal a través de paréntesis abiertos y cerrados, donde para cada paréntesis abierto (cerrado) existe otro cerrado (abierto) en una posición tal que entre ambos existe una cantidad igual de parentesis abiertos y cerrados (ver Fig. 3.2). El par de paréntesis abierto y cerrado representan el nodo, y los nodos hijos son los que se encuentran contenidos en otros pares de paréntesis.

LOUDS. El LOUDS [43] (por Level-Ordered Unary Degree Sequence) es otra representación sucinta de un árbol ordinal. Esta representación se basa en una exploración por amplitud del

árbol ordinal, en el cuál cada nodo es representado por un 1 seguido de la cantidad de hijos que tiene, estos representados con 1 a excepción del último hijo que es representado con un 0 para así a la vez representar el término del nodo. En esta representación todas las hojas están al final del arreglo (ver Fig. 3.2).

DFUDS. El DFUDS [7] (por Depth-First Unary Degree Sequence) es una representación sucinta de un árbol ordinal basada en el BP y en el LOUDS. Representada a través de paréntesis y basada en una exploración por profundidad del árbol en que, como el DFUDS, representa cada nodo seguido de la cantidad de sus hijos (delimitado por un paréntesis cerrado), en el que los primeros nodos hijos siguen inmediatamente al paréntesis delimitador (ver Fig. 3.2).

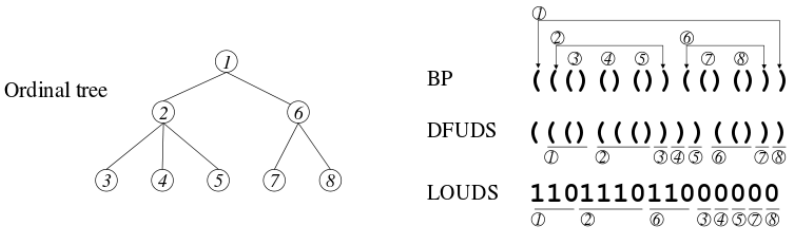


Fig. 3.2: Representaciones sucintas de un árbol ordinal [47]

3.1.5. Wavelet tree

El Wavelet tree [27, 38] es una estructura de datos compacta tipo árbol binario en que cada nodo es un bitmap, y que además posee múltiples aplicaciones, entre las que destacan los *Full-text indexes*, *Positional inverted indexes*, *Non-positional inverted indexes*, *Generic numeric sequences* entre otros.

Dado un alfabeto $\Sigma = \{a_1, a_2, ..., a_n\}$ de σ símbolos diferentes, se denomina a S como una secuencia de n símbolos (distintos o repetidos) donde todos ellos pertenecen a Σ. Entonces, el Wavelet tree representará la secuencia S en su nodo raíz a través de un bitmap de longitud n en donde cada s_i es representado por un 1 o por un 0 dependiendo de su posición en el alfabeto, es decir, si s_i está dentro de la primera mitad $\sigma/2$ del alfabeto Σ será un 1, en caso contrario será un 0. Así mismo, se irá aplicando recursivamente el mismo criterio para las mitades de los alfabetos correspondientes hasta llegar a las hojas que corresponden a cada elemento del alfabeto. La altura del árbol está dada por el tamaño del alfabeto y es $\lceil \log \sigma \rceil$ y el tamaño total de almacenamiento de la estructura es de $n\lceil \log \sigma \rceil$. Para ejemplificar mejor supongamos el siguiente caso (ver Fig. 3.3):

- $S = \{$*alabar a la alabarda*$\}$, de tamaño $n = 20$
- $\Sigma = \{$*",a,b,d,l,r*$\}$, de tamaño $\sigma = 6$

- Altura del Wavelet tree $\lceil \log \sigma \rceil = 3$
- Tamaño de la estructura $n \lceil \log \sigma \rceil = 120$ bits.
- Tamaño de la secuencia considerando tipo de dato char $n * 8\ bits = 160$ bits

En la raíz, todos los caracteres que estén presentes en la primera mitad del alfabeto se representan a través de un 0, los restantes son representados con un 1. Luego, en el mismo orden en que aparecen en la raíz, todos los $0's$ se irán por una rama hacia la izquierda y los $1's$ hacia la derecha. Cada nodo hijo tendrá como alfabeto las letras que siguen en la rama correspondiente, y en base a este nuevo alfabeto se representarán con un 0 los caracteres que correspondan a la primera mitad del nuevo alfabeto, y con un 1 los que no. Esto recursivamente hasta las hojas que corresponden a cada carácter.

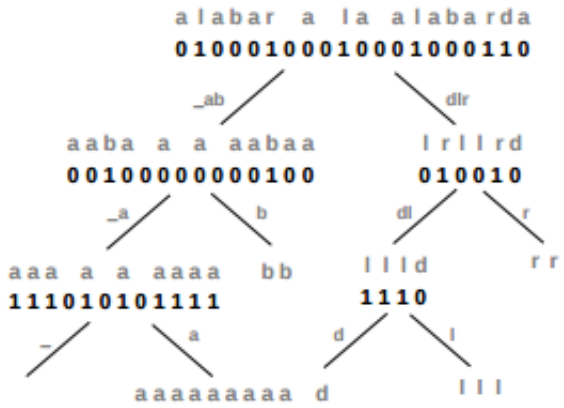


Fig. 3.3: Wavelet tree para la secuencia *"alabar a la alabarda"* [38]

Para recuperar cualquier elemento de la secuencia S se debe recorrer el árbol desde la raíz hasta la hoja a la que corresponde el índice accesado, no superando este el tiempo de acceso $O(\log \sigma)$. El acceso a las distintas posiciones del bitmap en cada nodo se hace a través de las funciones *rank* y *select*. La función *rank*, de acceso constante gracias a una estructura extra, entregará la cantidad de 1's o 0's dada cierta posición del bitmap, la función *select*, de mismas características que *rank*, entregará la posición del i-ésimo 1 o 0 dentro del bitmap dado un número previo.

Otra estructura de datos compacta es el k^2-tree, que dada su importancia para esta tesis se tratará en una sección aparte, a continuación.

3.2. k^2-tree

Es una estructura de datos compacta utilizada para representar matrices binarias como por ejemplo las matrices de adyacencia empleadas por los grafos web, por lo tanto, esta estructura soporta también representaciones de espacios geométricos bi-dimensionales.

Al igual que el quadtree [48], el k^2-tree [12] divide la matriz en k^2 cuadrantes de igual tamaño a cada nivel de exploración, es decir, si definimos un $k = 2$ la matriz será dividida en cuatro cuadrantes de iguales dimensiones, en el orden NorOeste - NorEste - SurOeste - SurEste, y para cada cuadrante resultante, el mismo será dividido en k^2 sub-cuadrantes de iguales dimensiones entre sí. Esto recursivamente hasta llegar a los valores atómicos de la matriz representada (ver Fig. 3.4). Para este caso no existen los nodos oscuros y claros como en el quadtree. Aquí si un cuadrante está conformado sólo por 1's será representado hasta las hojas a través de un 1, sin embargo cada vez que encuentre un cuadrante completo de 0's pasará a ser una hoja con valor 0. Aún así existe una versión del k^2-tree que comprime 1's [9] al mismo estilo que el quadtree para evitar la redundancia de datos.

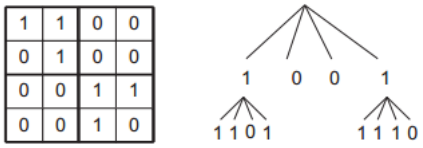


Fig. 3.4: k^2-tree [12]

La estructura de un k^2-tree se codifica a través de dos bitmaps, T y L. T almacenará la información desde los primeros nodos hijos hasta los últimos nodos padres (desde la raíz hacia las hojas). L mantendrá la información de todas las hojas. Para Fig. 3.4 se tendría entonces los siguientes bitmaps:

T : 1001

L : 11011110

Los nodos hijos se enumerarán desde el 0 hasta k^2-1, y para recorrer el árbol se deben utilizar las funciones *rank* y *select*. Supongamos que queremos acceder al primer hijo del cuarto nodo hijo de la raíz (1), para esto debemos utilizar la función "hijo", la cual está dada por la siguiente fórmula: $hijo_i(n) = rank_1(T, n) * k^2 + i$, donde i es el i-ésimo hijo a buscar y n es el n-ésimo nodo considerando los índices del bitmap T. Entonces, para conocer el primer hijo del cuarto nodo hijo de la raíz aplicamos la ecuación $hijo_0(3) = rank_1(T, 3) * 2^2 + 3$, con $rank_1(T, 3) = 2$ y dando como resultado final $hijo_0(3) = 2 + 4 + 3 = 9$. Ahora, como el bitmap T sólo llega hasta el índice 3, esto significará que el $hijo_0(3)$ corresponde a una hoja, y para saber a que bit corresponde en L se le resta, al resultado, la cantidad de bits de T (4), entonces con $hijo_0(3) - 4 = 5$. El resultado final corresponde al índice del $hijo_0(3)$ en L, que como se puede corroborar corresponde al cuarto 1, que es el primer hijo del cuarto nodo hijo de la raíz.

Existen distintas variantes del k^2-tree, entre ellas destaca la implementación de dos valores de k sobre un mismo conjunto de datos. Esto se hace con el fin de optimizar los recursos de la estructura. Para esto se define un mayor valor de k para los primeros niveles del árbol, lo que acorta su altura y mejora la navegación en profundidad. Para los últimos niveles define un valor menor para k lo que permite optimizar el recurso del espacio de almacenamiento, al tener hojas más pequeñas se evita por ejemplo tener que implementar un bitmap b de largo $|b| = 4$ para un $l = 1$, siendo l la cantidad de 1's en el último nivel explorado por rama (ver Fig. 3.5).

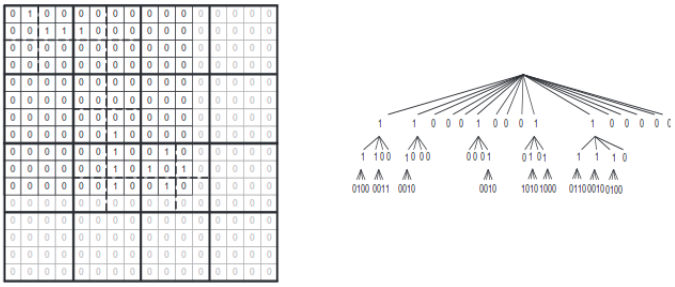


Fig. 3.5: Implementación híbrida de un k^2-tree, con $k = 4$ para el primer nivel y $k = 2$ para los restantes [12]

El k^2-tree es una estructura que ha demostrado funcionar mejor cuando los espacios a representar son datos dispersos como los grafos web, sin embargo, se han implementado extensiones de esta misma estructura las cuales son eficientes para la indexación espacial de datos [9]. Para esta tesis se utilizará el k^2-tree básico propuesto en [12] ya que se espera después poder ajustar los algoritmos a todas las extensiones del k^2-tree que eficientemente representen datos espaciales.

Si bien un Wavelet-tree para indexación espacial permite representar un espacio de dos dimensiones, tiene la limitante que para tratar más de un punto sobre la misma ordenada o abscisa se hace más complejo su tratamiento. Creemos por lo demás que el k^2-tree al jerarquizar las zonas sobre el espacio representado (de manera similar a un R-tree) permite una búsqueda más eficiente en cuanto al problema de la cerradura convexa.

Como el k^2-tree representa una matriz booleana, podemos asumir que es posible representar un espacio de dos dimensiones, sin embargo, dada la importancia para esta tesis, de la representación espacial que se hace a través de esta estructura, es que trataremos el tema en el capítulo siguiente.

3.3. Geometría computacional sobre estructuras de datos espacio-eficientes

La historia de la representación espacio-eficiente de datos ha evocado esfuerzos en su mayor parte a dar soluciones a problemas de datos estáticos, como por ejemplo grandes bases de datos de archivos literarios. Sin embargo, en cuanto al campo de las representaciones espaciales, en el último tiempo se han presentado algunos proyectos que buscan dar solución a problemas clásicos presentes en el área.

3.3.1. Estructuras sucintas e implícitas en Geometría Computacional

La mayoría de las estructuras de datos espacio-eficientes que soportan operaciones sobre la Geometría Computacional han demostrado su capacidad de dar soluciones a problemas geométricos como el reporte de distancia ortogonal, el contador de distancia ortogonal o problemas de localización de puntos [31]. Algunas de las estructuras mencionadas son las siguientes:

Representación implícita del *kd*-tree. Es una estructura implícita propuesta por Munro en 1979 [35], la cual consiste en una estructura formada por registros multi-claves capaces de soportar pareos parciales, y que puede ser modificada de tal forma que da solución a consultas geométricas como el reporte de distancia ortogonal.

Wavelet tree generalizado. A diferencia del wavelet tree, el wavelet tree generalizado [23] es una estructura sucinta en que cada nodo, v, tiene $\sigma = \log^{\epsilon} n$ hijos, con ϵ constante y menor a 1. En vez de construir un array de bits para cada nodo v, se construye una cadena sobre el alfabeto Σ, y cada símbolo presente en Σ corresponde a un subrango representado por un hijo de v.

Representación sucinta de mapas planares. Es una estructura sucinta construida a partir de un marco de trabajo definido el cual toma un objeto (en este caso un árbol plano sin raíz) el cual es dividido y catalogado en múltiples pequeñas piezas de tamaño $O(\log n)$ y codifica las relaciones de incidencias como piezas más pequeñas en un grafo $\mathcal{G}$. La relación que se da entre los distintos tamaños de las piezas permite crear punteros que definen al grafo $\mathcal{G}$ como una estructura sucinta [3].

KDB$_{KD}$-tree Es una representación compacta de un KDB-tree, estructura utilizada para representación de espacios multi-dimensionales. Esta estructura elimina la información redundante existente en el KDB-tree. Para esto reemplaza la política de división *first division splitting* (FD) la cual produce una redundancia al dividir espacios en dos espacios menores que se encontrarán en uno o el otro lado del espacio contenedor, lo que provoca la redundancia de los datos y que simplifica el KDB_{KD}-tree [55].

Faster Compressed Quadtrees. Es una estructura que compacta un Quadtree, para esto se basa en la técnica utilizada por el k^2-tree, el cual luego traspasa a un árbol binario, en el cual los 1's representan las zonas con elementos y los 0's los que no, pudiéndose presentar a lo menos 1 nodo positivo (con valor 1) en cada rama. Luego de esto, codifica los caminos existentes desde la raíz hasta las hojas a través de un 0 (hacia la izquierda) y un 1 (hacia la derecha). Una vez codificados los caminos los ordena de tal forma que al agruparlos en un solo vector de bits se reutilizan trazos de los caminos obtenidos [24].

Al revisar la literatura nos encontramos con múltiples estructuras espacio-eficientes que pueden ser utilizadas para la representación espacial, de las cuales destacamos el k^2-tree y el Wavelet tree. Ambas estructuras pueden ser usadas para representar un espacio geométrico e indexar puntos [15, 12].

3.3.2. Wavelet tree para indexación espacial

La aplicación geométrica que se le da a los Wavelet trees deriva de la representación hecha por Chazelle [16] quien define un espacio geométrico como una grilla de puntos. En base a esto, [15] plantea la indexación del espacio basada en el Wavelet tree, la cual considera para el caso más simple un vector de enteros donde los índices del vector corresponden a las coordenadas x y los valores del vector corresponden a las coordenadas y. Este vector luego es representado

como un Wavelet tree de la misma forma en que se representa una secuencia S con alfabetos no necesariamente numéricos. Cabe destacar que para éste caso se trabaja sobre un espacio discreto en que no existen más de dos coordenadas por cada incidencia en x y en y. Para conjuntos de datos continuos pero no repetidos por abcisa ni ordenada, se utilizan tres estructuras auxiliares, que almacenan cada una todas las x y las y presentes en el espacio, la tercera estructura almacena los ID's ya sea de las x o de las y, siendo más eficiente tratar con las x. Este tipo de estructuras auxiliares permite realizar consultas por rangos [14], ya que a través de cuatro operaciones de *rank* y *select* se puede definir los límites de la búsqueda, tanto para x como para y, dentro de los índices del bitmap y así podar el árbol evitando inspecciones innecesarias (ver Fig. 3.6).

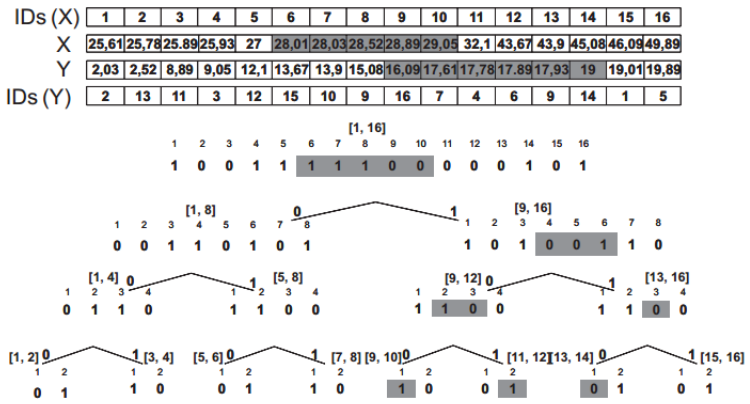


Fig. 3.6: Wavelet tree con indexación espacial [15]

3.3.3. MBRs compactos con Wavelet tree

La compactación de MBRs [13] se presenta para dar solución a un problema específico como lo es la consulta por rangos, para esto utiliza una estructura basada en el wavelet tree con la cual representa cada MBR y así logra entregar una respuesta en tiempo logarítmico para la consulta.

Para su implementación se asume que ningún MBR se proyecta a través del eje x intersectando a la proyección de otro MBR, formando así un conjunto de MBRs que se denomina como maximal. Si el conjunto total de puntos no es máximal debe ser descompuesto en múltiples conjuntos maximales. Sea n el número de MBRs existentes en el conjunto maximal, donde cada uno es descrito por dos vértices contrarios del MBR representado. En la estructura compacta se puede representar a través de una grilla de $2n \times 2n$ y con sólo un punto en cada fila y columna (ver Fig. 3.7).

El objetivo de todo esto es desarrollar nuevos índices espaciales que puedan ser localizados en los niveles más altos sobre la jerarquía de memoria.

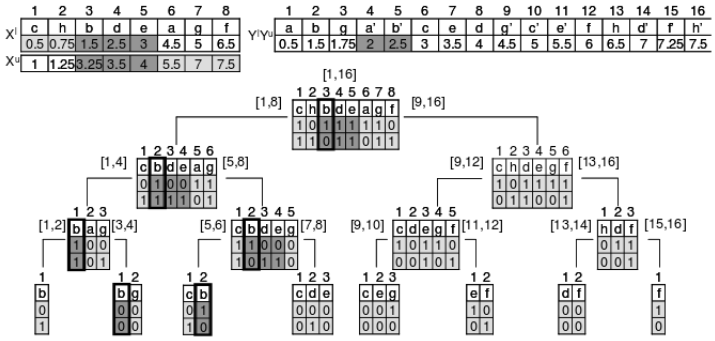


Fig. 3.7: MBRs representados con wavelet trees [13]

Parte III

Algoritmos para el cálculo de la cerradura convexa sobre k^2-tree

Capítulo 4

Coordenadas sobre un k^2-tree

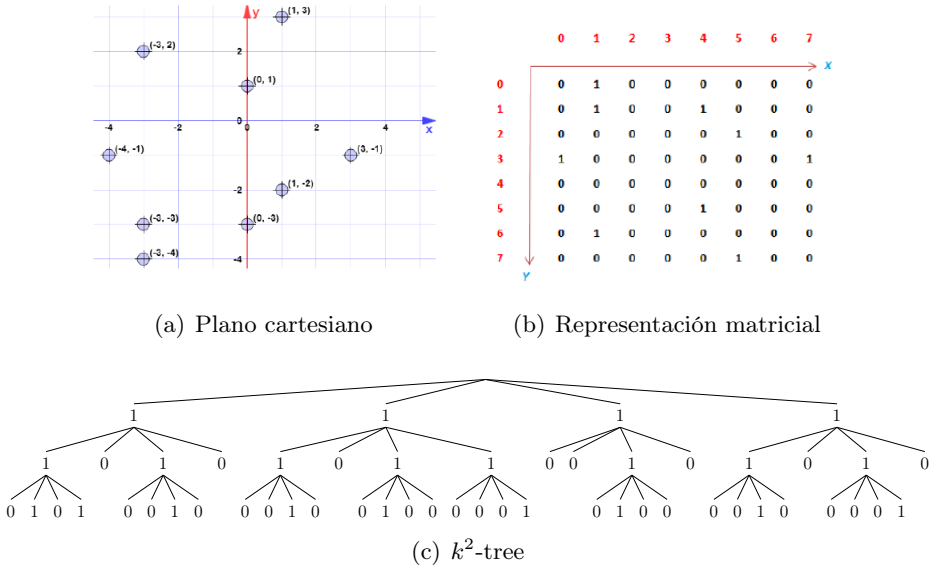
Para esta tesis se ha decidido utilizar como estructura de datos compacta el k^2-tree, ya que cumple el objetivo de lograr la compresión de datos y a la vez es una estructura jerárquica, lo que hace pensar en la posibilidad de extrapolar propiedades de la búsqueda de la cerradura convexa sobre un R-tree [11]. Además, pensamos y demostramos de manera empírica que las propiedades jerárquicas del k^2-tree facilitan y aceleran el proceso de búsqueda dado el enfoque que se le da al problema tratado.

Si bien el k^2-tree desarrolla su máximo potencial para la representación de grafos web, pensamos que al ser un derivado del quadtree puede representar sin mayores problemas un espacio bi-dimensional, que es donde trataremos el problema de la cerradura convexa.

La representación que hace del espacio es simple si se ve desde el ángulo de la matriz booleana, la cual se utilizará como referencia para comprender el espacio representado en un k^2-tree. Es sobre esta representación que proponemos distintos métodos para la obtención de las coordenadas correspondientes sobre un k^2-tree.

4.1. Representación espacial en un k^2-tree

Supongamos un conjunto de datos definido por cuatro coordenadas (x_1, y_1), (x_2, y_2), (x_3, y_3) y (x_4, y_4), en que la proyección de los puntos forma un rectángulo que contiene a todos los puntos presentes en el espacio. Este rectángulo puede ser representado por una matriz booleana de tamaño $n \times n'$, la cual es sólo ejemplificatoria y jamás llega a ser implementada, sin embargo se mencionará repetidamente a lo largo de esta tesis para una mejor comprensión de lo explicado. Para poder representar esta matriz a través de un k^2-tree es necesario que $n = n'$, por lo que para cuando no se cumpla esta condición se completará la matriz con columnas y filas de $0's$ hasta satisfacer la condición. Las coordenadas en una matriz booleana irán desde el $(0, 0)$ hasta el $(n-1, n-1)$, pudiendo representar cualquier espacio discreto en $\mathbb{R}^2$ (ver Fig. 4.1). Ahora que la representación del espacio está definida para la matriz booleana, y siendo que esta es de tamaño $n \times n$ (o $n' \times n'$ si $n' > n$) podemos proceder a la implementación del k^2-tree. Sin embargo, antes es necesario explicar que para la implementación que usa el k^2-tree, dada la forma en que se construye (en orden NorOeste - NorEste - SurOeste - SurEste) y el orden de los índices de



(a) Plano cartesiano (b) Representación matricial

(c) k^2-tree

Fig. 4.1: Representación del espacio de dos dimensiones con un k^2-tree, con $k = 2$.

las matrices a nivel de programación es que se considera el espacio representado a través de la matriz como si el espacio original hubiese sido girado 180° en torno al eje x, esto considerando la traslación de todos los puntos en el espacio hacia el primer cuadrante del plano cartesiano. Esto implica en distintos factores al momento de interpretar cálculos hechos sobre el espacio, los que serán explicados en el momento adecuado.

Los bitmaps L y T correspondientes a Fig. 4.1 se dan de la siguiente forma:

$$T : 1111\ 1010\ 1011\ 0010\ 1010$$

$$L : 0101\ 0010\ 0010\ 0100\ 0001\ 0100\ 0010\ 0001$$

4.2. Obtención de coordenadas sobre un k^2-tree

4.2.1. Identificador bi-dimensional (x, y) de un nodo

Consideremos cada cuadrante por sí sólo como un conjunto distinto, siendo sus dimensiones definidas por la cantidad de sub-cuadrantes contenidos pero sólo un nivel más abajo. Entonces, para cada cuadrante se tendrá una dimensión de $k \times k$, exceptuando las hojas. Es en este escenario en que nos conviene saber a que coordenadas corresponde un sub-cuadrante dentro de su cuadrante padre, y es a estas coordenadas a las que llamamos *identificadores bi-dimensionales (x, y) de un nodo o cuadrante* (ver Fig. 4.2).

El identificador bi-dimensional (x, y) de un nodo puede variar su valor desde $(0, 0)$ hasta $(k - 1, k - 1)$ y puede ser calculado con sólo saber la posición en el bitmap del nodo explorado.

Sean i la posición del nodo en el bitmap T o L, con $i = [0, 1, ..., |T| + |L| - 1]$, donde $|T|$ y

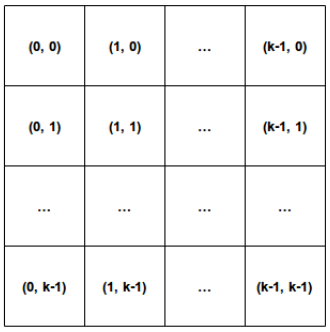


Fig. 4.2: Identificadores bi-dimensionales (x, y) de un nodo para un cuadrante.

$|L|$ corresponden a la cantidad de elementos para cada bitmap T y L respectivamente.

La suma de la cantidad de elementos en los bitmaps T y L da el total de nodos del k^2-tree, para el cual sabemos que el peor caso nos costará $k^2 m \left(\log_{k^2} \frac{n^2}{m} + O(1)\right)$ bits [13], donde m es la cantidad de 1's a representar desde la matriz binaria y n es el tamaño de la matriz (también aplica la fórmula $\log_2 \binom{n^2}{m}$ [22] que considera todas las matrices posibles de tamaño n con m 1's). Sin embargo, no nos es posible saber con exactitud la cantidad de nodos que contendrá el k^2-tree, esto debido a las múltiples posibles combinaciones de los 1's sobre la matriz. Lo que si sabemos con certeza es que el crecimiento del árbol será siempre en bloques de tamaño k^2.

Ahora, sabiendo que cada bloque es de tamaño k^2, entonces podemos afirmar que:

$$i \,\%k^2 = [0, 1, .., k^2 - 1]$$

ya que la posición del nodo puede ser expresado como una descomposición en base a los bloques ya mencionados, de la siguiente forma:

$$i = \{\gamma k^2 + i' \mid \gamma = [0, 1, .., m\lceil \log_k n \rceil - 1] \wedge i' = [0, 1, .., k^2 - 1]\}$$

Supongamos una división sin resto

$$\frac{i}{k^2} = \frac{\gamma k^2}{k^2} \rightarrow \frac{i}{k^2} = \gamma \rightarrow i = \gamma k^2$$

esto implica en que el valor de i' sea 0. Ahora, si i' tomáse un valor superior a $k^2 - 1$ significará que puede ser descompuesto en $\gamma k^2 + i''$, demostrándose que la operación $i \,\%k^2$ siempre dará un valor entre 0 y $k^2 - 1$.

Una vez que sabemos a qué n-hijo nos estámos refiriendo, nos hace falta calcular el valor de x e y que este representa dentro del cuadrante padre, es decir, el identificador bi-dimensional (x, y) del nodo que representa el n-hijo. Por ejemplo, si un k^2-tree es de valor $k = 2$ esto significará que los identificadores bi-dimensionales (x, y) de todos su cuadrantes hijos para con sus padres serán un valor entero entre 0 y $k - 1$, es decir, para este caso un total de cuatro coordenadas: $(0, 0), (1, 0), (0, 1), (1, 1)$, las cuales corresponden a los cuadrantes según el ordenamiento en que

se dan los hijos en el k^2-tree.

Teorema 1. *El identificador bi-dimensional (x, y) de un cuadrante hijo referenciado por el nodo se obtiene como:*

$$Identificador\ bi-dimensional\ (x,y)\ de\ i\ =\ \left(i\,\%k, \left\lfloor \frac{i\,\%k^2}{k} \right\rfloor\right)$$

donde el nodo i está dado por la posición explorada dentro de los bitmaps T y L.

Demostración. Cada fila de un cuadrante tendrá k elementos (independiente de su valor) en su siguiente nivel, estando estos ordenados de tal forma que los primeros k hijos de un cuadrante corresponden su valor con su identificador bi-dimensional (x) dentro del cuadrante padre (ver Fig. 4.2). Siendo k un divisor de k^2 que no deja resto, significa que cada bloque de k^2 puede ser representado por k bloques iguales, dándose entonces las mismas propiedades para $i\,\%k^2$. Por lo tanto es correcto afirmar que $i\,\%k = [0, .., k-1]$ y a la vez que como cada cuadrante hijo puede ser expresado como múltiplo de k esta propiedad aplica a todos los hijos de un cuadrante y por tanto esta fórmula es suficiente para calcular el identificador bi-dimensional (x) de un cuadrante hijo.

Para el identificador bi-dimensional (y) el proceso es más complejo debido a que el orden en que se registran los cuadrantes no es igual de directo como para el identificador bi-dimensional (x). A priori, podemos decir que el valor de y presenta un incremento cada k elementos recorridos en los bitmaps (ver Fig. 4.2). De la misma forma podemos decir entonces que cada k hijos, el valor de y aumenta en 1. Por lo tanto podemos aplicar primeramente la fórmula para saber a qué hijo nos referimos $(i\,\%k^2)$, y luego ver cuántos bloques k conforman el resto obtenido. El valor se redondeará hacia abajo debido a que las posiciones de los nodos comienzan desde el 0 así como también lo hacen las identificadores bi-dimensionales de los cuadrantes.

□

4.2.2. Coordenada de una hoja

Para saber el valor al que corresponde una hoja en el plano original, es necesario conocer todo el recorrido desde el nodo raíz hasta la hoja. A este recorrido le llamaremos r, donde $r = \{r_0, r_1, \ldots, r_{(\lceil \log_k n \rceil - 1)}\}$, siendo cada elemento representado como r_h. El nivel en que estemos en el árbol determinará el conjunto de coordenadas a los que estamos accesando y es fundamental a la hora de calcular el valor de (x, y) al que corresponde la hoja accesada.

Teorema 2. *Los valores (x, y) de una hoja están dados por las siguientes fórmulas:*

$$x = \sum_{h=0}^{\lceil \log_k n \rceil - 1} (r_h\,\%k)k^h$$

$$y = \sum_{h=0}^{\lceil \log_k n \rceil - 1} \left\lfloor \frac{r_h\,\%k^2}{k} \right\rfloor k^h$$

donde siempre r_h corresponderá a la posición de un nodo en T o L.

Demostración. Podemos demostrar que éstas sumatorias jamás nos darán una coordenada que no exista dentro del espacio representado sabiendo que los valores para $i\,\%k$ (recordemos que $i = r_h$) y para $\lfloor \frac{i\,\%k^2}{k} \rfloor$ son valores enteros en el rango $[0, .., k-1]$. Entonces, siendo un espacio de $n \times n$, comenzando desde el 0 la coordenada más alejada del origen será $(n-1, n-1)$.

Primero reemplazamos los valores de $i\,\%k$ y $\lfloor \frac{i\,\%k^2}{k} \rfloor$ por a_h, ya que sabemos que es un valor definido en un rango $[0, .., k-1]$ (la demostración se hace sólo para la coordenada x, ya que para la coordenada y el reemplazo hace que la sumatoria quede idéntica).

$$\sum_{h=0}^{\lceil \log_k n \rceil - 1} (r_h\,\%k)k^h \rightarrow \sum_{h=0}^{\lceil \log_k n \rceil - 1} (a_h)k^h$$

Asumimos todos los valores de a_h iguales ya que estamos buscando casos extremos, lo que nos permite despejar la constante $a = [a_0, ..., a_h]$ y expandir la sumatoria

$$a \sum_{h=0}^{\lceil \log_k n \rceil - 1} k^h = a(k^0 + k^1 + .. + k^{\lceil \log_k n \rceil - 1})$$

y reemplazamos la constante por el mayor valor posible $(k-1)$

$$(k-1)(k^0 + k^1 + .. + k^{\lceil \log_k n \rceil - 1})$$

aplicamos el producto

$$k - 1 + k^2 - k + k^3 - k^2 + ... + k^{\lceil \log_k n \rceil} - k^{\lceil \log_k n \rceil - 1}$$

eliminamos los valores que se anulan y nos queda

$$k^{\lceil \log_k n \rceil} - 1$$

por ahora asumamos $k^{\lceil \log_k n \rceil} = n$ (demostración en sección 4.2.3), entonces podemos afirmar que el mayor valor posible para la sumatoria es $n-1$. El menor valor es 0, ya que al reemplazar la constante a por 0 anula la sumatoria. □

Para ejemplificar, supongamos el primer elemento en el bitmap L, para esto calculamos $rank(L, 1) = 1$ y aplicamos la fórmula $padreL(i)$, sobre el nodo obtenido. Luego aplicamos sucesivamente la fórmula $padreT(i)$ sobre los nodos que se vayan obteniendo hasta que la posición del nodo obtenido sea menor a k^2, es decir, estemos en la raíz.

$$padreL(i) = select\left(T, \frac{|T| + i - i\,\%k^2}{k^2}\right)$$

$$padreT(i) = select\left(T, \frac{i - i\,\%k^2}{k^2}\right)$$

Siguiendo con el ejemplo, calculemos el recorrido del nodo con posición 1, a priori sabemos

que el recorrido es $r = \{1, 4, 0\}$, yendo desde la hoja hacia la raíz, siendo los primeros dos nodos pertenecientes a T y el último a L. Ahora verificamos desde la hoja con la posición del nodo 1.

$$padreL(1) = select\left(T, \frac{|T| + 1 - 1\,\%2^2}{2^2}\right) = 4$$

$$padreT(4) = select\left(T, \frac{4 - 4\,\%2^2}{2^2}\right) = 0$$

Ahora, con el recorrido definido, procedemos a calcular el valor (x, y) de la hoja con nodo 1. Para esto aplicamos las fórmulas ya mencionadas. Primero para x:

$$h = 0 \rightarrow (r_0\,\%2)2^0) = 1$$

$$h = 1 \rightarrow (r_1\,\%2)2^1) = 0$$

$$h = 2 \rightarrow (r_2\,\%2)2^2) = 0$$

$$x = 1 + 0 + 0 = 1$$

ahora para y:

$$h = 0 \rightarrow \lfloor \frac{r_0\,\%k^2}{k} \rfloor k^0 = 0$$

$$h = 1 \rightarrow \lfloor \frac{r_1\,\%k^2}{k} \rfloor k^1 = 0$$

$$h = 2 \rightarrow \lfloor \frac{r_2\,\%k^2}{k} \rfloor k^2 = 0$$

$$y = 0 + 0 + 0 = 0$$

Por lo tanto, la hoja correspondiente al nodo 1 de L, equivale a las coordenadas $(1, 0)$. Ahora, como las coordenadas no van de 0 a n en el plano representado, se deben calcular los valores reales a través de la diferencia entre el menor valor de x en el plano cartesiano contra el menor valor de x en la matriz (0) y la diferencia sumarla a todos los valores de x obtenidos en el k^2 tree. Se debe hacer lo mismo con los valores de y.

$$Planocartesiano(x_m, y_m) = (-4, -4)$$

$$Matriz(x'_m, y'_m) = (0, 0)$$

$$Resto = (x'_m, y'_m) - (x_m, y_m) = (-4, -4)$$

$$Coordenada\ real\ de\ rank(L, 1) = (1 - 4, 0 - 4) = (-3, -4)$$

4.2.3. Bounding Box

Nos referiremos al término *Bounding Box* (de ahora en adelante BB) cuando estemos tratando con alguno de los cuadrantes representados por el k^2-tree. También, para un mejor entendimiento del concepto, nos referimos a un BB cuando hablamos del cuadrado paralelo a los ejes que

corresponde a un nodo del k^2-tree dentro del espacio representado.

El mínimo tamaño que se considera para un BB es de 1×1, lo cual corresponde a una hoja, y el máximo es de $n \times n$, considerando el caso de una raíz que contemple la completitud de la matriz como un cuadrante. Así mismo, el conjunto total (por eje, sea x o y) es divisible en un máximo de $\lceil \log_k n \rceil$ BBs de valor entero, lo cual corresponde a la altura del árbol. Si n es divisible por k recursivamente $\lceil \log_k n \rceil$ veces, esto significa que $n = k^{\lceil \log_k n \rceil}$.

Consideremos la primera división del conjunto de largo n

$$\frac{n}{k}$$

donde podemos reemplazar el valor de n por $k^{\lceil \log_k n \rceil}$

$$\frac{k^{\lceil \log_k n \rceil}}{k} = k^{\lceil \log_k n \rceil} * k^{-1} = k^{\lceil \log_k n \rceil - 1}$$

así mismo se demuestra que la mínima dimensión entera para el BB es de 1×1

$$k^{\lceil \log_k n \rceil} * k^{-\lceil \log_k n \rceil} = k^0 = 1$$

y la máxima es de valor $n \times n$

$$\frac{k^{\lceil \log_k n \rceil}}{k^0} = k^{\lceil \log_k n \rceil} = n$$

por lo tanto se puede definir la fórmula para el ancho (o alto, ya que ambos son iguales en un cuadrado) de un BB basados en la altura h, de la siguiente forma:

$$anchoBB = \{\frac{n}{k^h} \times \frac{n}{k^h} \mid 2 \leq k \leq n \wedge 0 \leq h < \lceil \log_k n \rceil \ \wedge n \,\%k = 0\}$$

Sin embargo, esta fórmula sólo nos permite saber el ancho y alto de un BB, pero no así las coordenadas que lo conforman, para esto se plantean dos métodos distintos.

Bounding Box por recorrido

Si comenzamos a recorrer un árbol desde la raíz, podemos de manera paralela ir calculando las coordenadas que forman los cuadrantes explorados. Para esto es necesario ir manteniendo registro de la coordenada superior izquierda obtenida para cada BB explorado, ya que por cada nivel del árbol que exploremos necesitamos saber la coordenada del BB padre o contenedor para así poder determinar sus propias coordenadas. Esta forma de calcular es prácticamente un fragmento del cálculo de una coordenada para una hoja, comenzando siempre el valor de la altura en $\lceil \log_k n \rceil - 1$ y disminuyendo de manera constante a cada nivel explorado. Entonces, sea h' la altura del nivel en que se encuentra el BB del que se desea saber sus coordenadas

$$x = \sum_{h=h'}^{\lceil \log_k n \rceil - 1} (r_h \,\%k) k^h$$

$$y = \sum_{h=h'}^{\lceil \log_k n \rceil - 1} \left\lfloor \frac{r_h \,\%k^2}{k} \right\rfloor k^h$$

Con esta fórmula obtenemos el valor de la coordenada de la esquina superior izquierda, las otras fácilmente son calculables según la Fig. 4.3.

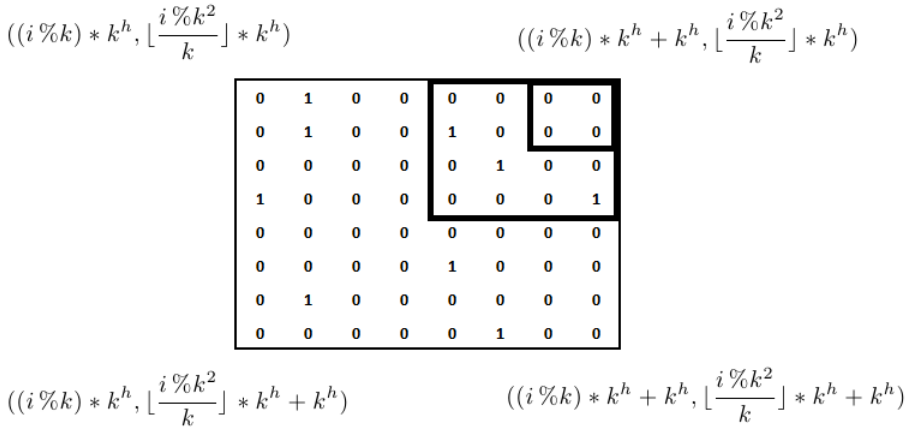


Fig. 4.3: Coordenadas de un bounding box por recorrido.

Bounding Box por coordenada hoja

Supongamos que tenemos una coordenada definida según el recorrido desde la raíz hasta una hoja. Si bien, para el cálculo de la coordenada correspondiente a una hoja es necesario ir calculando alguna de las coordenadas conformantes del cuadrante superior por nivel según la altura del árbol y el cuadrante que se está explorando, una vez obtenida la coordenada final no es necesario mantener almacenadas las coordenadas del recorrido.

Para obtener una coordenada de un cuadrante superior y contenedora es necesario conocer la coordenada de la hoja correspondiente, el valor de k y el valor de h' correspondiente a la altura donde se encuentra el cuadrante del que se quiere obtener sus coordenadas de un BB.

Para el cálculo de las coordenadas de un BB es necesario sólo la obtención de una de las cuatro coordenadas que componen el cuadrilátero, la cual se obtiene con cualquiera de las siguientes fórmulas:

$$Coordenada\ de\ un\ BB\ = \left(\left\lfloor \frac{x}{k^{h'}} \right\rfloor * k^{h'}, \left\lfloor \frac{y}{k^{h'}} \right\rfloor * k^{h'} \right)$$

$$Coordenada\ de\ un\ BB\ = (x - x\,\%k^{h'}, y - y\,\%k^{h'})$$

la cual nos dará como resultado la coordenada correspondiente a la esquina superior izquierda

del cuadrante explorado.

Para la primera fórmula, demostramos su correctitud a través del reemplazo de la variable x por la sumatoria para su cálculo.

$$\left\lfloor \frac{x}{k^{h'}} \right\rfloor k^{h'} \rightarrow \left\lfloor \frac{\sum_{h=0}^{\lceil \log_k n \rceil - 1} (r_h \,\%k) k^h}{k^{h'}} \right\rfloor k^{h'}$$

si sabemos que $r_h \,\%k = [0, .., k-1]$ lo podemos reemplazar por a_h

$$\left\lfloor \frac{\sum_{h=0}^{\lceil \log_k n \rceil - 1} a_h k^h}{k^{h'}} \right\rfloor k^{h'}$$

si extendemos un poco la sumatoria nos quedará así:

$$\left\lfloor \frac{a_0 k^0 + a_1 k^1 + .. + a_{h'} k^{h'} + .. + a_{\lceil \log_k n \rceil - 1} k^{\lceil \log_k n \rceil - 1}}{k^{h'}} \right\rfloor k^{h'}$$

ahora aplicamos la división y las correspondientes restas de exponentes

$$\left\lfloor a_0 k^{0-h'} + a_1 k^{1-h'} + .. + a_{h'} k^0 + .. + a_{\lceil \log_k n \rceil - 1} k^{\lceil \log_k n \rceil - 1 - h'} \right\rfloor k^{h\prime}$$

siendo todos los valores desde $a_0 k^{-h'}$ hasta $a_{h'-1} k^{-1}$ menores a 1 y no pudiendo sumar entre todos tampoco un valor mayor o igual a 1. Como se les está aplicando el redondeo hacia abajo, estos valores desaparecen

$$\left(a_{h'} k^0 + .. + a_{\lceil \log_k n \rceil - 1} k^{\lceil \log_k n \rceil - 1 - h'} \right) k^{h'}$$

y al aplicar la multiplicación por $k^{h'}$ a los valores restantes nos damos cuenta que es la misma fórmula para calcular el recorrido de un BB desde la raíz

$$a_h k^{h'} + .. + a_{\lceil \log_k n \rceil - 1} k^{\lceil \log_k n \rceil - 1} = \sum_{h=h'}^{\lceil \log_k n \rceil - 1} (r_h \,\%k) k^h$$

por lo tanto la fórmula es correcta.
El cálculo de las restantes coordenadas de un BB se hace sumando k^h al componente x e y según corresponda (ver Fig. 4.4).

Por ejemplo, para la hoja correspondiente a la posición 31 en L se tiene como coordenada el valor $(5, 7)$, el cual se encuentra en el primer nivel en el cuarto cuadrante de coordenadas de un BB $(4, 4), (7, 4), (4, 7), (7, 7)$ y en el segundo nivel en el tercer cuadrante de coordenadas de un BB $(4, 6), (5, 6), (4, 7), (5, 7)$ y en el último nivel (las hojas) de coordenadas simples $(5, 7)$ como ya se mencionó. Ahora, para el correcto cálculo se considera el incremento de la altura desde la hoja hasta la raíz, considerándose al nivel de las hojas como altura 0 e incrementando nivel a nivel de uno en uno. Para una mejor ejemplificación en el primer nivel se ocupará la primera fórmula de coordenadas de un BB, y para el segundo nivel se ocupará la segunda fórmula.

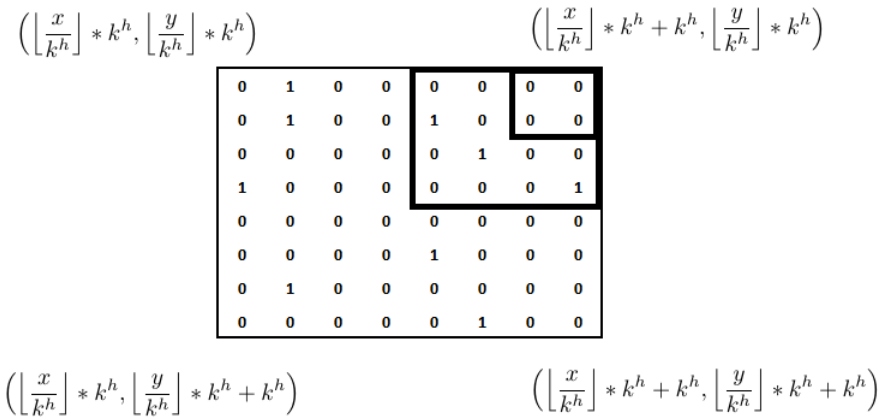


Fig. 4.4: Coordenadas de un bounding box por coordenada hoja.

$$Primer\ nivel,\ coordenadas(5,7),\ altura = 2$$
$$Coordenada\ de\ un\ BB\ = \left(\left\lfloor \frac{5}{2^2} \right\rfloor * 2^2, \left\lfloor \frac{7}{2^2} \right\rfloor * 2^2\right) = (4,4)$$

$$Segundo\ nivel,\ coordenadas(5,7),\ altura = 1$$
$$Coordenada\ de\ un\ BB\ =\ (5 - 5\,\%2^1, 7 - 7\,\%2^1)\ =\ (4,6)$$

Capítulo 5

Algoritmos

5.1. Introducción

En este capítulo se presentan los aportes realizados a través de esta tesis, enfocándose los mismos en el diseño de dos algoritmos para la obtención de la cerradura convexa sobre un k^2-tree. En el presente capítulo se explicará la estrategia utilizada y se demostrará la correctitud de los algoritmos propuestos.

Dado un conjunto S de coordenadas discretas y positivas en un espacio de tamaño $n \times n$, estando todos los elementos de S almacenados sobre un k^2-tree, se necesita encontrar el menor polígono convexo que contenga todos los elementos del conjunto S.

La estrategia que utilizaremos para encontrar los puntos que conforman la cerradura convexa será similar a la utilizada en [11] y que utiliza un R-tree donde a partir de cuatro puntos extremos se conforma un polígono convexo que es una aproximación a la cerradura convexa definitiva y que llamaremos Cerradura Convexa Candidata (CCC). Luego, mediante un procedimiento similar al definido por el algoritmo quickhull y que básicamente consiste en ubicar por cada arista de la CCC el punto más alejado perpendicular a la arista, se refinará la CCC hasta obtener la cerradura convexa definitiva.

En la sección 5.2 se presenta el algoritmo para la obtención de los puntos extremos sobre un k^2-tree, el cual es parte fundamental de los demás algoritmos planteados. En la sección 5.3 se presenta el algoritmo principal de esta tesis, a través del cual se logra obtener la cerradura convexa sobre un k^2-tree. Para finalizar el capítulo, en la sección 5.4 se presenta una variante del algoritmo principal, el cual busca una respuesta más eficiente a través de una heurística propuesta.

5.2. Puntos extremos sobre un k^2-tree

Para la obtención de los puntos debemos revisar potencialmente todo el k^2-tree, sin embargo para optimizar la búsqueda se explorará de forma secuencial sobre las k filas o columnas candidatas dependiendo el extremo a buscar. Por ejemplo, supongamos que queremos encontrar el menor valor de y, el cual corresponderá a uno de los 1's almacenados en la fila poblada más cercana al origen. En la Figura 5.1 correspondería al punto indicado con el 1 en la fila 0 y la columna 1.

Se puede dar el caso de la existencia de dos o más extremos, lo cual significará que estamos en presencia de una arista del polígono convexo.

La búsqueda del menor y sobre el k^2-tree se describe en el Algoritmo 5.1, el cual aprovecha las propiedades de la estructura para optimizar la búsqueda. Básicamente, lo que el algoritmo hace es explorar el k^2-tree mediante un recorrido en profundidad considerando los cuadrantes (hijos) adecuados. Por ejemplo si buscamos el punto con el menor y el algoritmo parte explorando el cuadrante azul de más a la izquierda y luego el cuadrante azul de más a la derecha. En caso de que ambos cuadrantes no tengan hijos se deben explorar los otros k cuadrantes. Una vez explorado por completa una rama del árbol y posicionados ya en las hojas, se seleccionará la primera aparición de un 1 a través de un *rank* sobre los k^2 elementos correspondientes. Si el valor obtenido es además uno de los primeros k hijos significará que el punto encontrado es un extremo absoluto, ya que estamos buscando el menor valor de y presente, y por tanto termina nuestra búsqueda. Si el punto encontrado no es uno de los primeros k hijos significará que existe la posibilidad de no ser un extremo, ya que puede haber otro cuadrante de los ya definidos como candidatos, que contenga un hijo con un y menor.

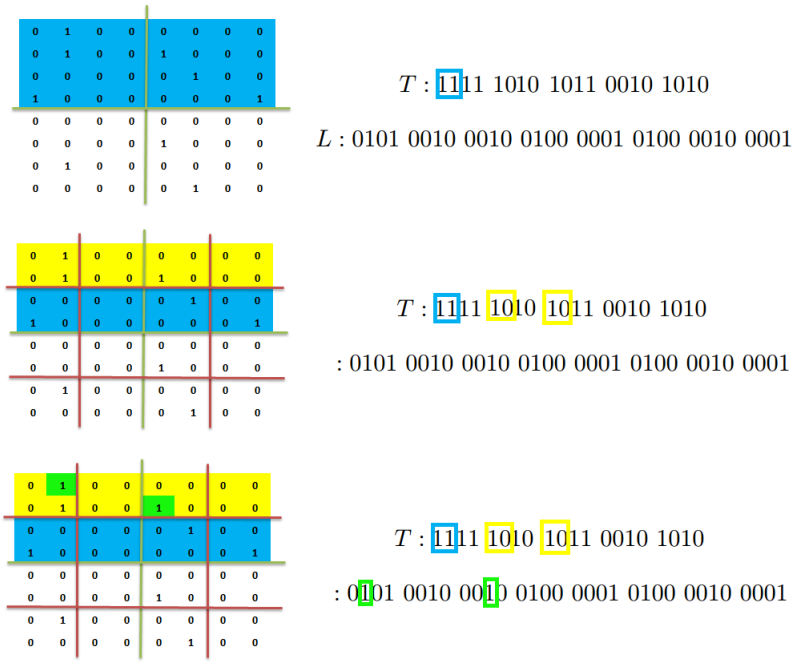


Fig. 5.1: Búsqueda del y_{min} para el k^2-tree.

Si al tomar los primeros k cuadrantes estos están vacíos, significará que a lo menos debe existir uno de los cuadrantes restantes, para el nodo explorado, que contenga elementos candidatos, sin embargo su prioridad es baja, ya que son los primeros k elementos los que son más prometedores. Aun así no se deben descartar, ya que no sabemos si esto se repite para todos los cuadrantes a explorar en el nivel del árbol en que nos encontremos, por lo tanto el nodo se almacenará en una

pila extra, la cual será utilizada sólo en el caso de no encontrar un extremo al vaciarse la primera pila, es decir, que todos los nodos explorados no tengan valores positivos dentro de sus primeros k nodos hijos. Si llegamos a utilizar la pila extra, exploraremos los siguientes k cuadrantes del nodo, y a la vez explorando los primeros k de estos nuevos candidatos. El proceso se repite hasta encontrar un extremo. Podemos asegurar que de esta forma encontraremos un extremo, ya que dadas las propiedades del k^2-tree podemos saber de la existencia de un punto desde el momento en que estamos sobre lo más alto de sus ramas.

Este proceso se repite para cada uno de los cuatro extremos, variando en el algoritmo el proceso de navegación sobre el k^2-tree, el cual está definido por el extremo a buscar.

El peor caso para la búsqueda del extremo a través de este enfoque es el cual en el que el punto extremo se encuentra en el tope inferior de los últimos k cuadrantes explorados, sin embargo, la jerarquía que presenta la estructura facilita la exploración y el algoritmo no se ve afectado en gran medida sobre este escenario. Otro punto a favor, es que el peor caso para la búsqueda de un extremo es el mejor caso para el extremo contrario, es decir, si el peor caso para la búsqueda del mínimo valor de y se presenta, esto significará que estamos en presencia del mejor caso para la búsqueda del mayor valor de y, dándose además que ambos extremos sean el mismo valor, lo que es un escenario muy poco probable. Lo mismo aplica a los extremos de x.

Consideramos para el problema un espacio total de $n \times n = \mu$, el cual es dividido en k^2 partes iguales, de las cuales en el peor de los casos hacemos k llamadas recursivas. En el caso de existir muchos 0's se harán a lo máximo k^2 comparaciones para decidir cual explorar, por lo que la ecuación de recurrencia nos quedará como:

$$t(\mu) = kt\left(\frac{\mu}{k^2}\right) + k^2$$

donde el primer k corresponde a la cantidad de llamadas recursivas, $t\left(\frac{\mu}{k^2}\right)$ es la llamada recursiva sobre una fracción del problema original, y k^2 son las comparaciones necesarias en cada llamada recursiva para llegar a la solución.

Aplicamos el teorema maestro sobre la ecuación de recurrencia para obtener la complejidad del algoritmo. Sea $a = k$, $b = k^2$ y $k1 = 0$. Se explica $k1 = 0$ debido a que en la última expresión de la ecuación "k^2"no existe una función $f(\mu)$.

Como $a > b^{k1}$, entonces $t(\mu) = O(\mu^{\log_b a})$, es decir, $O(\mu^{\log_{k^2} k})$. Haciendo cambio de base de logaritmo tenemos $O(\mu^{\frac{\log_2 k}{2\log_2 k}}) = O(\mu^{\frac{1}{2}}) = O(\sqrt{\mu})$, pero como $\mu = n^2$ entonces $t(n) = O(n)$.

Calculamos el mejor caso de una complejidad $O(k\lceil\log_k n\rceil)$ en el que a la primera bajada y explorándo los k elementos correspondientes por nivel del k^2-tree accesado, encontramos un extremo absoluto en las primeras hojas visitadas, sin importar la distribución de los demás nodos.

La búsqueda de los extremos *Este* y *Oeste* es más compleja, ya que al hacer una exploración de los elementos de un cuadrante de manera vertical no podemos aplicar directamente funciones como *rank* y *select* para acelerar el proceso, ya que estos no se encuentran de manera consecutiva dentro de la implementación de la estructura como si lo están al hacer una búsqueda horizontal. La diferencia será notoria cuando el valor de k sea mucho mayor del que se acostumbra, por lo tanto no ha de considerarse un agravante en la eficiencia de la búsqueda vertical, sin embargo hace que su implementación sea más compleja al tener que ir manteniendo registro de los avances tanto verticales como horizontales cuando se de el caso.

```
1  Algoritmo: y_min(k²-tree)
2  n = raíz del k²-tree;
3  Pila⟨nodo n, posicion p⟩ explora = {⟨n, 0⟩}, auxiliar = {};
4  ⟨nodo n, posicion p⟩e = ⟨⟩;
5  VectorExtremo = {};
6  if n está vacío then
7      return "Árbol vacío.";
8  while Extremo esté vacía do
9      while explora no esté vacía do
10         e = pop(explora);
11         flg = 0;
12         if los hijos de e.n son hojas then
13             ex= primer hijo positivo de e.n;
14             if ex es un extremo absoluto then
15                 return ex;
16             else
17                 push(Extremo,ex);
18         foreach h_i hijo de e, con i = [e.p, ..., e.p + k] do
19             if access(h_i)==1 then
20                 push(explora,⟨h_i, 0⟩);
21                 flg = 1;
22         if flg == 0 then
23             push(auxiliar,⟨e.n, e.p + (k − 1)⟩);
24     if Extremo está vacía then
25         explora = auxiliar
26 return mayorExtremo(Extremo);
```

Alg. 5.1: Algoritmo búsqueda extremo y_{min}

5.3. Convex Hull sobre un k^2-tree (CHk^2)

En esta sección abordamos el algoritmo principal de esta tesis, el Convex Hull sobre un k^2-tree (CHk^2), a través del cual buscamos dar solución a un problema clásico de la Geometría Computacional como lo es la cerradura convexa para un espacio bi-dimensional, el cual representamos a través de una estructura compacta como lo es el k^2-tree. Para lograr esto, es que nos apoyamos en la Capítulo 4 donde definimos las bases para la exploración del k^2-tree y la interpretación de sus datos como coordenadas.

Nuestro algoritmo CHk^2 aprovecha la partición recursiva de la matriz binaria y la organización jerárquica de las mismas mantenidas en el k^2-tree. Esto le permite calcular la Cerradura Convexa del conjunto de puntos accediendo solamente a una pequeña fracción de la estructura del k^2-tree,

sin necesidad de descompactarla completamente.

En primer lugar CHk^2, obtiene los cuatro extremos en cada una de las direcciones del cuadrante que contiene a S, es decir, los puntos: (1) $Norte$, punto más al norte (más arriba); (2) Sur, punto más al sur (más abajo); (3) $Este$, punto más al Este (más a la derecha) y (4) $Oeste$, punto más al Oeste (más a la izquierda). Para ello se utiliza el algoritmo descrito en la Sección 5.2. Estos cuatro puntos son los vértices de un polígono convexo cuyas aristas son $(Oeste, Norte), (Norte, Este), (Este, Sur)$ y $(Sur, Oeste)$ (ver Figura 5.2). A este polígono en adelante le llamaremos Cerradura Convexa Candidata (CCC). Notar que los cuatro puntos son vértices de CCC y de CC (cerradura convexa definitiva).

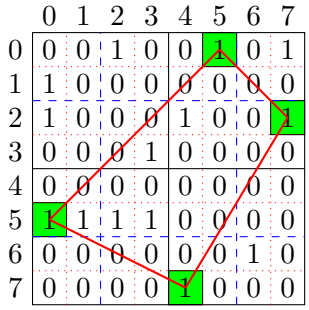


Fig. 5.2: Cerradura Convexa Candidata formada por los extremos

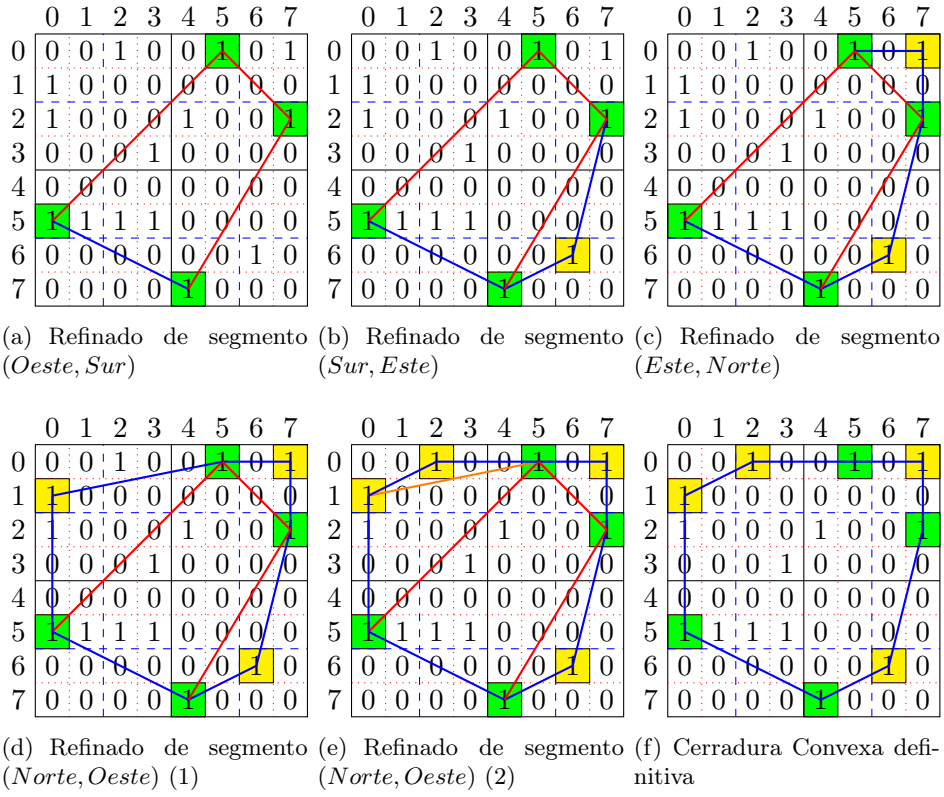
A partir del CCC inicial, y de manera recursiva el algoritmo refina CCC de tal menera de lograr CC, es decir, la cerradura convexa definitiva. Para ello se procede de manera similar a QuickHull, es decir, por cada arista a de vértices (v_1, v_2) de CCC se busca el punto más alejado y que se encuentra por sobre/debajo (dependiendo de la orientación de la búsqueda) de a. Sea p_a dicho punto. El algoritmo elimina la arista a de CCC y en su lugar agrega las arista con vértices (v_1, p_a) y (p_a, v_2) (ver Figura 5.3). El procedimiento continua hasta que ya no sea posible eliminar/agregar aristas a CCC siendo esta la CC definitiva del conjunto de puntos.

El quickhull demuestra que el punto más alejado de un segmento formado por extremos o también formado por puntos del polígono convexo, es un punto vértice de la cerradura convexa. Es en base a esta definición que se establece la estrategia de obtención de los puntos conformantes de la cerradura convexa para el conjunto de datos. Entonces, si ya tenemos definidos los extremos y los segmentos que forman los mismos, sólo basta establecer el modus operandi de la búsqueda de los puntos más alejados sobre el k^2-tree.

Antes de comenzar con la explicación en detalle del algoritmo es necesario definir la estructura del Bounding Box (BB), el cual estará compuesto por dos enteros (coordenadas x, y del extremo superior izquierdo), un entero sin signo para la altura (h) en que se encuentra el BB y un long int para el índice o nodo (i) accesado sobre el k^2-tree (ver Algoritmo 5.2).

$$BB = \langle (x, y); h; i \rangle$$

La búsqueda del punto más alejado de una arista a con vértices (v_1, v_2) se realiza comenzando desde la raíz hacia las hojas del k^2-tree descartando en el camino cuadrantes o nodos de la misma



(a) Refinado de segmento $(Oeste, Sur)$ (b) Refinado de segmento $(Sur, Este)$ (c) Refinado de segmento $(Este, Norte)$

(d) Refinado de segmento $(Norte, Oeste)$ (1) (e) Refinado de segmento $(Norte, Oeste)$ (2) (f) Cerradura Convexa definitiva

Fig. 5.3: Refinado del CCC al CC

```
1  Algoritmo: Cálculo del BB(bb,indice)
2  BoundingBox bbRetorno;
3  if indice == 0 then
4  |   bbRetorno.x = 0;
5  |   bbRetorno.y = 0;
6  |   bbRetorno.h = ⌈log_k n⌉;
7  |   bbRetorno.i = 0;
8  else
9  |   bbRetorno.x = bb.x + (indice %k)k^(bb.h−1);
10 |   bbRetorno.y = bb.y + ⌊(indice %k^2)/k⌋ k^(bb.h−1);
11 |   bbRetorno.h = bb.h − 1;
12 |   bbRetorno.i = indice;
13 return bbRetorno;
```

Alg. 5.2: Algoritmo para el cálculo de un Bounding Box

estructura. En primer lugar se necesita obtener la direccionalidad de la arista a como se indica en la Tabla 5.1 y la cual se obtiene mediante la pendiente que forman los puntos v_1 y v_2 de a (ver Algoritmo 5.3).

```
1  Algoritmo: Direccionalidad(Vertice v1, Vertice v2)
2  Pendiente = (v2.y − v1.y) / (v2.x − v1.x);
3  if Pendiente > 0 then
4  |  if v2.x > v1.x then
5  |  |  Direccion = 2;
6  |  else
7  |  |  Direccion = 1;
8  if Pendiente < 0 then
9  |  if v2.x > v1.x then
10 |  |  Direccion = 3;
11 |  else
12 |  |  Direccion = 4;
13 if Pendiente == 0 then
14 |  Direccion = 0;
15 return Direccion;
```

Alg. 5.3: Algoritmo para la pendiente y direccionalidad de un segmento $\overline{v_1 v_2}$

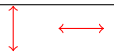
Segmento	Dirección	Búsqueda
	1	Esquina superior izquierda
	2	Esquina superior derecha
	3	Esquina inferior izquierda
	4	Esquina inferior derecha
	0	No hay que buscar

Tabla 5.1: Direccionalidad

Por ejemplo, sea un segmento $\overline{Ex1Ex2}$ formado por los puntos extremos $Ex1(x_1, y_1)$ y $Ex2(x_2, y_2)$, con pendiente negativa y con valor de $x_2 > x_1$, entonces la direccionalidad será 1. Aquí es importante mencionar que al estar representando un espacio del primer cuadrante de un plano cartesiano pero rotado 180° se puede generar confusión con respecto a las pendientes y la direccionalidad ya que es inverso a como se presenta en el espacio original. Ahora, como la direccionalidad es 1 significará que debemos buscar los cuadrantes cuya coordenada correspondiente a la esquina superior izquierda efectúe un giro reloj negativo para con las coordenadas del

segmento (ver Algoritmo 5.4).

```
1 Algoritmo: GiroReloj
2 Ex1 = (x1, y1);
3 Ex2 = (x2, y2);
4 C = (x3, y3);
5 if ((x1 == x3) AND (y1 == y3)) OR ((x2 == x3) AND (y2 == y3)) then
6     return 1;
7 gr = (x2 − x1)(y3 − y1) − (y2 − y1)(x3 − x1);
8 return gr;
```

Alg. 5.4: Algoritmo para el cálculo del giroReloj

Por lo general los cuadrantes de mayor tamaño (los más cercanos a la raíz) suelen ser intersectados por los segmentos definidos por la búsqueda. Esto hace que el considerar una coordenada del cuadrante puede hacer que el giroReloj entre los extremos del segmento y la coordenada escogida nos de un valor positivo y por tanto habría que descartar este cuadrante. Sin embargo, el segmento formado por los extremos también define el área de búsqueda, y si esta área se intersecta con un cuadrante significa que el cuadrante es un candidato a explorar.

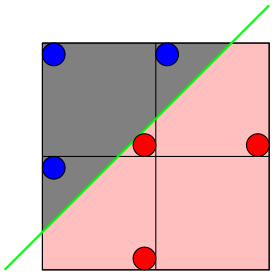


Fig. 5.4: Criterio de selección de coordenada por segmento

Lo anterior lo podemos ver en la Figura 5.4, donde la recta de color verde representa el segmento $\overline{Ex1Ex2}$ y los cuadrados representan los cuadrantes intersectados por el segmento. Si la direccionalidad del segmento fuese 1 significaría que nuestra área de búsqueda es la de color gris. En cambio, si la direccionalidad fuera 4, el área de búsqueda sería la de color rosa. Ahora, para el área de búsqueda gris vemos que los círculos azules indican las esquinas (coordenadas) pertinentes para evaluar con el segmento, y las rojas las que no. De manera inversa son los círculos rojos los que indican la esquina (coordenada) pertinente para el área rosa y así mismo los azules no lo son para este caso. Es por esto la importancia de la direccionalidad (ver Algoritmo 5.5), ya que a mayor altura en el árbol no podemos saber la ubicación exacta de los valores atómicos, entonces los cuadrantes deben ser considerados si se da aunque sea la más remota posibilidad de éxito bajo los criterios de búsqueda definidos.

Una vez que se ha definido un cuadrante como candidato se requiere calcular la distancia perpendicular de la coordenada correspondiente al cuadrante con el segmento formado por extremos. Para esto sólo es necesario dividir el resultado del giroReloj por la distancia entre los extremos,

```
1  Algoritmo: coordenadaPertinente
2  BoundingBox BB;
3  direccion d;
4  if d == 1 then
5  |  return ⟨BB.x, BB.y⟩;
6  if d == 2 then
7  |  return ⟨BB.x + k^{BB.h}, BB.y⟩;
8  if d == 3 then
9  |  return ⟨BB.x, BB.y + k^{BB.h}⟩;
10 if d == 4 then
11 |  return ⟨BB.x + k^{BB.h}, BB.y + k^{BB.h}⟩;
```

Alg. 5.5: Algoritmo para el cálculo de la coordenada pertinente a explorar de un BB

lo cuál nos dará un valor negativo (ver Algoritmo 5.6).

```
1  Algoritmo: distanciaPuntoSegmento
2  Ex1 = (x_1, y_1);
3  Ex2 = (x_2, y_2);
4  C = (x_3, y_3);
5  gr = giroReloj(Ex1, Ex2, C);
6  return gr / sqrt((x_2 − x_1)^2 + (y_2 − y_1)^2);
```

Alg. 5.6: Algoritmo para el cálculo de la distancia entre un punto y un segmento

Por cada nivel explorado del árbol se presentará un máximo de k^2 cuadrantes candidatos, por lo tanto es necesario establecer un orden para la exploración de los siguientes niveles. Creemos que una exploración en amplitud del árbol resultaría ineficiente, por lo que hemos decidido establecer una cola de prioridades basada en la distancia entre el cuadrante candidato y el segmento explorado. Recordemos que estamos con una representación del espacio rotada 180°, por lo tanto la cola de prioridades tendrá siempre a tope el cuadrante con la menor distancia entre el candidato y el segmento.

Una vez explorado un nivel, se procederá a explorar los hijos del cuadrante candidato que haya quedado a tope en la cola de prioridades. De manera iterativa se irán ingresando los cuadrantes hijos correspondientes en la cola según la distancia de los mismos hacia el segmento de extremos. Sin embargo, la exploración de un cuadrante en particular no asegura que todos sus hijos, que correspondan a un área de datos no vacía, tengan una distancia tal que los haga quedar a tope en la cola de prioridades. Es en tal caso que avanzado cierta cantidad de niveles en una rama del árbol nos demos cuenta que el espacio no vacío no está más distante que un cuadrante de nivel superior. Es por esto que la cola de prioridad juega un rol fundamental definiendo los cuadrantes a explorar sin generar pérdida de posibles candidatos.

Si al extraer un dato de la cola de prioridades, éste elemento es una hoja significa que hemos

dado con un vértice del polígono convexo, ya que al ser un elemento del tope de la cola nos aseguramos que es el más distante al segmento explorado. En tal caso, se debe vaciar la cola de prioridades para una nueva exploración, tomar el punto vH encontrado y definirlo como extremo ubicándolo entre $Ex1$ y $Ex2$. Con estos datos comenzamos una nueva exploración por segmento, sólo que ahora el segmento está formado por $Ex1$ y vH.

Si una cola de prioridad queda vacía por la búsqueda sin haber extraído una hoja de la misma significa que los cuadrantes candidatos eran falsos positivos. Los falsos positivos se dan cuando a medida que descendemos por el k^2-tree nos vamos dando cuenta que los puntos están concentrados todos bajo el segmento de exploración (ver Fig. 5.5) y por tanto no hay puntos de la cerradura convexa entre los puntos extremos.

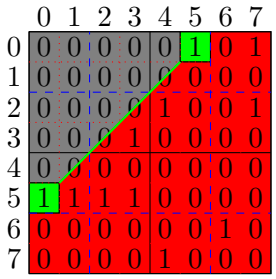


Fig. 5.5: Candidatos con falsos positivos

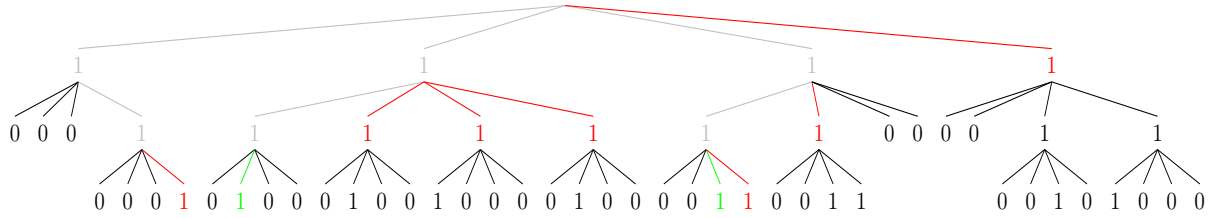


Fig. 5.6: k^2-tree para la Figura 5.5

Podemos notar en la Figura 5.6 que, para el segmento formado por las hojas de color verde, los nodos explorados son los de color gris y rojo. Como buscamos ejemplificar los falsos positivos y la búsqueda a través de la cola de prioridades no presentamos el caso de la consecucíon de un vértice del polígono convexo. Entonces, los nodos y hojas grises, rojos y verdes son los índices interpretados a través de un BB, de los cuales pasan a la cola de prioridades sólo los grises. Podemos observar que las hojas de color rojo no lo son si no hasta que son hojas, es decir, todo el recorrido hasta antes de dar con la hoja es gris y por tanto un candidato posible, es por esto que no sabemos que es un falso positivo hasta explorar el nodo padre de la hoja roja. Otro caso es el de un nodo de color rojo, que al ser explorado en un alto nivel del árbol nos damos cuenta que no es candidato y por tanto no pasa a la cola de prioridades. Las hojas verdes están a una distancia nula del segmento, por tanto no son candidatas y no pasan a la cola de prioridad.

Cuando un punto sale de la pila de extremos tiene dos destinos posibles, la lista de los vértices del polígono convexo, o volver a la misma pila de extremos. Cada vez que extraemos

dos puntos de la pila de extremos por lo menos uno volverá a la estructura, ya que se dan dos posibilidades, la obtención de un vértice del polígono convexo o no. Para la consecución de un vértice v significará que el extremo definido como $Ex2$ volverá a la pila, y v pasará a convertirse en $Ex2$, manteniéndose el valor de $Ex1$. En el caso de la no consecución de un vértice $Ex1$ pasará directamente al set de datos que conforma la cerradura convexa, $Ex2$ pasará a ser el nuevo $Ex1$ y se extraerá un nuevo $Ex2$ de la pila de extremos. Este proceso se repite hasta que la pila de extremos quede vacía, es decir que haya encontrado todos los puntos que conforman la cerradura convexa del conjunto de datos S (ver Algoritmo 5.7).

```
1  Algoritmo: CerraduraConvexa
2  PilaExtremos extremos = {EO', EN, EE, ES, EO}, convexo = {};
3  n = BB para la raíz del k²-tree;
4  ColaPrioridades⟨BB, distancia⟩ Cola = {⟨n, 0⟩};
5  Ex1 = pop(extremos);
6  Ex2 = pop(extremos);
7  while extremos no esté vacío do
8      direccion = direccion(Ex1, Ex2);
9      while Cola no esté vacía do
10         cuadrante = Cola.top();
11         Cola.pop();
12         if cuadrante es hoja then
13             break;
14         foreach h hijo positivo de cuadrante do
15             BB = BB(cuadrante, h);
16             C = coordenadaPertinente(BB, direccion);
17             if giroReloj(Ex1, Ex2, C) < 0 then
18                 distancia = distanciaPuntoSegmento(Ex1, Ex2, C);
19                 Cola.add(⟨BB, distancia⟩);
20     if cuadrante.hoja() then
21         push(extremos, Ex2);
22         Ex2 = cuadrante;
23     else
24         push(convexo, Ex1);
25         Ex1 = Ex2;
26         Ex2 = pop(extremos);
27     Cola.vaciar();
28     Cola.add(⟨n, 0⟩);
29 return convexo;
```

Alg. 5.7: Algoritmo para la cerradura convexa sobre un k^2-tree

Se considera como el peor caso para nuestro algoritmo el escenario en que todos los puntos

del conjunto S son vértices de la cerradura convexa. La consecución de los puntos, sin embargo, no puede ser un peor caso para cada exploración, ya que se considera una búsqueda por tramos. La peor búsqueda por tramos es aquella en que todos los puntos se encuentran sobre el segmento explorado (sin considerar los puntos formados por el segmento). Para este caso se calcula en orden de $O(k^{2^{\lceil \log_k n \rceil + 1}})$, ya que un k^2-tree lleno (una matriz repleta de 1's) tendrá un total de $\sum_{l=1}^{\lceil \log_k n \rceil} k^{2^l}$ que equivale a $\frac{k^{2^{\lceil \log_k n \rceil + 1}} - 1}{k^2 - 1} - 1$ y por tanto una complejidad de $O(k^{2^{\lceil \log_k n \rceil + 1}})$. El peor caso, entonces es la exploración de todos los nodos existentes para dar con el último candidato como el acertado. Sin embargo, este caso es poco realista, ya que en la realidad un k^2-tree completo haría que el primer descenso nos diera el candidato acertado, aún así, debido a que dependemos de la distribución de los datos dentro del k^2-tree y además del sentido de la búsqueda, la que además variará a cada resultado es que se hace altamente complejo calcular un peor caso estándar. Es por esto que estimamos que el peor caso tiene una cota superior estimada $O(k^{2^{\lceil \log_k n \rceil + 1}})$, pero que en la práctica está muy por debajo de este límite.

5.4. Convex Hull sobre un k^2-tree con traslación de segmento (CHk^2t)

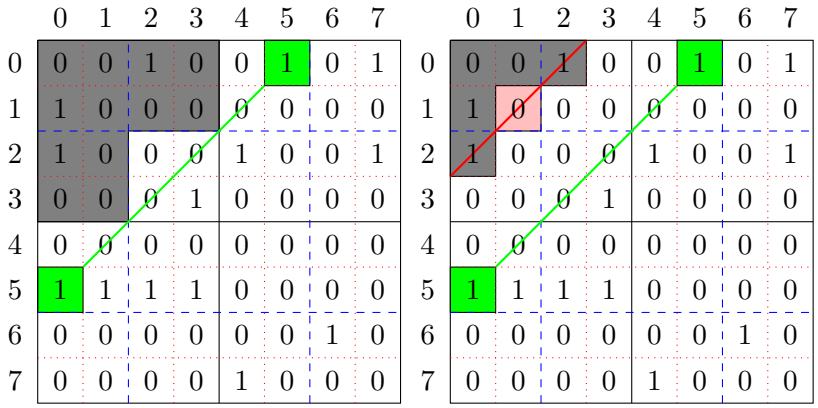
Una variante que se puede aplicar a nuestra propuesta del CHk^2 es la de trasladar el segmento formado por las aristas de la cerradura convexa candidata. Si observamos el peor caso para un cuadrante respecto al segmento ya mencionado, existe la posibilidad de que este esté sobre el mismo segmento, lo que significa que en el peor de los casos (en que el cuadrante sólo sea 1 en ese punto) para el cuadrante explorado ese será el punto más alejado del segmento. En base a esto se define un nuevo segemento imaginario trasladado a partir del original, que establece que ningún elemento de otros cuadrantes que esté por debajo de él será un mejor candidato que el peor candidato del cuadrante explorado (ver Fig. 5.7 [b]).

Cuando mencionamos la traslación de un segmento nos referimos a definir un nuevo segmento de exploración que sea paralelo al primeramente definido, y a la vez que esté más cercano a las coordenadas de búsqueda según la direccionalidad abordada. Para lograr la traslación del segmento se debe partir por un cuadrante candidato en el primer nivel del k^2-tree, luego, en base al cuadrante seleccionado que debe ser el más alejado del segmento, y según corresponda la direccionalidad en que se esté haciendo la búsqueda se seleccionara la coordenada del cuadrante diagonalmente inversa a la que se está utilizando para la definición del giroReloj. Por ejemplo, si la coordenada superior izquierda es la utilizada para definir el giroReloj con el segmento de exploración, entonces la diagonalmente inversa será la inferior derecha.

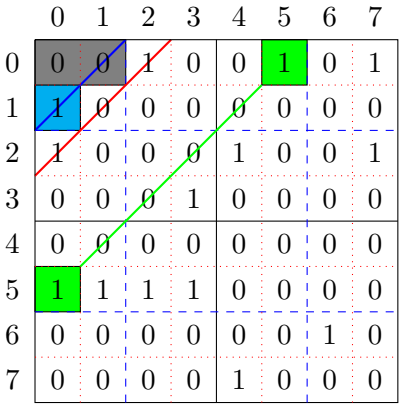
Con la coordenada del cuadrante escogida se debe trasladar el segmento formado por los extremos de tal forma que mantenga su pendiente pero que a la vez la nueva coordenada escogida forme parte del nuevo segmento (ver Fig. 5.7).

Para una mayor ejemplificación definamos un cuadrante candidato como cc$[(x_1, y_1), (x_2, y_2)]$, donde (x_1, y_1) es el vértice superior izquierdo y (x_2, y_2) el vértice inferior derecho que nos bastan para conocer las dimensiones y ubicación del cuadrante. Ahora, vemos que para la Figura 5.7 [a] el segmento (línea verde) define a cc1[(0,0),(1,1)], cc2[(2,0),(3,1)] y cc3[(0,2),(1,3)] como los cuadrantes candidatos que estan por sobre el segmento, siendo cc1 el más distante. El mejor caso

para cc1 es el punto $(0, 0)$, y el diagonalmente inverso o peor caso es el punto $(1, 1)$. Definimos entonces el punto $(1, 1)$ para nuestro nuevo segmento trasladado a partir del original, el cual se puede ver en la Figura 5.7 [b]. Con esto sabemos que ningún cuadrante o punto que se encuentre por debajo del nuevo segmento (línea roja) será el vértice que estamos buscando. Si calculamos la distancia de los puntos que están por sobre el segmento el mayor será el punto $(0, 1)$, el cual nos da la pauta para el nuevo segmento trasladado (línea azul en Figura 5.7 [c]) y así define nuestra nueva área de búsqueda. Como no hay valores positivos sobre el punto $(0, 1)$ y el último segmento formado, significa que éste punto es un vértice de la cerradura convexa.



(a) Primer segmento de exploración (b) Primera traslación de segmento



(c) Segunda traslación de segmento

Fig. 5.7: Traslación de segmento

Si la coordenada diagonalmente inversa se encuentra por debajo del segmento de exploración no se produce una mejora en el descenso del k^2-tree.

Para obtener el nuevo segmento de exploración es necesario calcular la ecuación de la recta del primer segmento. Siendo $Ex1(x_1, y_1)$ y $Ex2(x_2, y_2)$, recta definida por la ecuación $y = mx + b$, donde m es el valor de la pendiente $(\frac{y_2 - y_1}{x_2 - x_1})$, x e y pueden tomar el valor de $Ex1$ o $Ex2$ y b es la incógnita a calcular para poder trasladar la ecuación. Una vez calculado el valor de b se

deben reemplazar los valores de x e y por los de la coordenada diagonalmente inversa escogida sobre la ecuación, a través de la cual podemos calcular las nuevas coordenadas del segmento de exploración, siempre manteniendo los valores de x del primer segmento de exploración (ver Algoritmo 5.8).

```
1  Algoritmo: traslacionSegmento
2  Ex1 = (x_1, y_1);
3  Ex2 = (x_2, y_2);
4  BoundinBox = BB;
5  CoordenadaInversa = CoordenadaPertinente(BB, direccionInversa);
6  if giroReloj(Ex1, Ex2, CoordenadaInversa) < 0 then
7      b = CoordenadaInversa → y − pendiente ∗ CoordenadaInversa → x;
8      nuevaY = (pendiente ∗ CoordenadaInversa → x + b);
9      nuevaX = (pendiente ∗ CoordenadaInversa → y + b);
10     if direccion == 1 OR direccion == 4 then
11         nuevoEx1 = ⟨Ex1 → x, nuevaY⟩;
12         nuevoEx2 = nuevaX, Ex2 → y⟩;
13     if direccion == 2 OR direccion == 3 then
14         nuevoEx1 = nuevaX, Ex2 → y⟩;
15         nuevoEx2 = ⟨Ex1 → x, nuevaY⟩;
```

Alg. 5.8: Algoritmo para la traslación del segmento

La aplicación de este criterio debe ser realizada al final del ciclo en la línea 13 en el algoritmo 5.7.

A pesar de definir nuevos segmentos de exploración, se debe mantener el cálculo de la distancia al primer segmento explorado, ya que la cola de prioridades comenzó a utilizarse con esos valores y debe mantenerse a lo largo de su vida útil para impedir la obtención de datos contaminados.

Éste método nos permite ir acotando la búsqueda innecesaria en cuadrantes irrelevantes, y a la vez mantener una cola de prioridades más reducida y por tanto más fácil de reordenar.

Capítulo 6

Experimentación

Los algoritmos propuestos *Convex Hull sobre un k^2-tree* (desde ahora CHk^2) y *Convex Hull sobre un k^2-tree con traslación de segmento* (desde ahora CHk^2t), más un algoritmo que se presenta en este capítulo al cual denominamos *Graham Scan sobre un k^2-tree* (desde ahora GSk^2), fueron implementados y ejecutados para diferentes escenarios, para poder evaluarlos y compararlos en términos de tiempo y almacenamiento.

Cada resultado de la ejecución de alguno de estos algoritmos fue contrastada, para el mismo set de datos, con el algoritmo Graham Scan para verificar la correctitud de las respuestas.

En los experimentos se midieron el tiempo y el almacenamiento requerido por los algoritmos para cada conjunto de datos.

6.1. Entorno de experimentación

Para la implementación de los experimentos se utilizó el lenguaje de programación C++, junto con la librería *libcds* para la utilización de estructuras de datos compactas. Además todas las ejecuciones de los algoritmos fueron sobre un computador Intel(R) Core(TM) i7-4770 CPU @ 3.40 GHz 3.40 GHz, con 4 GB de RAM. El sistema operativo utilizado fue Ubuntu 14.04.2 LTS, bajo un kernel Linux 3.16.0-3-generic. El compilador utilizado fue gcc 4.8.4 (Ubuntu 4.8.4-2 ubuntu 14.04).

6.2. Conjuntos de datos

Para la experimentación de los algoritmos se utilizaron en total 39 set de datos distintos, de los cuales 15 corresponden a datos reales y los 24 restantes fueron generados aleatoriamente a través de la herramienta *GNU Octave*[1].

[1] https://www.gnu.org/software/octave/

6.2.1. Datos sintéticos

Los datos sintéticos generados aleatoriamente se dividen en dos grandes grupos según su distribución. Las distribuciones seleccionadas fueron la gaussiana o distribución normal y la distribución uniforme. Para cada conjunto se definieron distintos tamaños (n) de los conjuntos a generar y sobre tres espacios distintos para cada tamaño (ver Tabla 6.1).

Tabla 6.1: Distribución de datos sintéticos

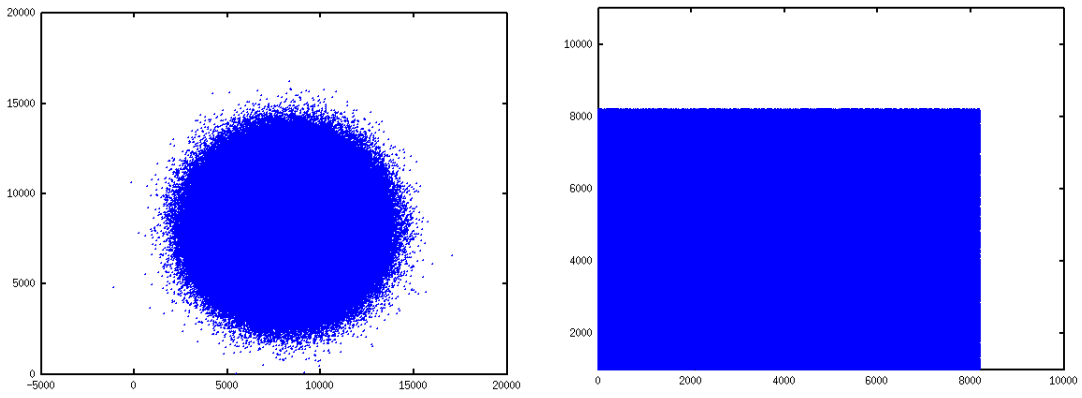
Cantidad de puntos (Millones)	Tamaño de la matríz					
	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
1	•		•		•	
2		•	•		•	
4			•	•	•	
8			•		•	•

Siendo np el número de puntos a generar y e el tamaño del espacio, los datos de distribución gaussianos (ver Fig. 6.6 [a]) fueron generados a través del siguiente comando (los valores negativos fueron ignorados dado que son mínimos en consideración al tamaño del set):

$$floor(\frac{e}{10} * randn(n, 2) + \frac{e}{2})$$

Para los datos de distribución uniforme (ver Fig. 6.6 [b]) se utilizó el siguiente comando (para este caso no hubo generación de datos negativos):

$$randi([0\ e-1], n, 2)$$



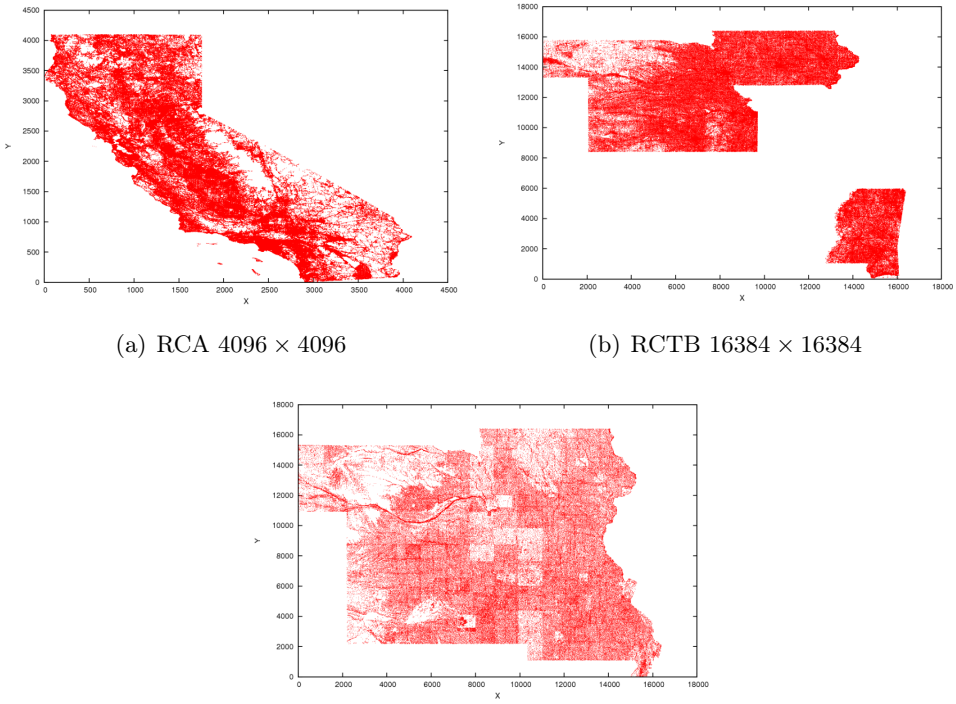
(a) Distribución gaussiana con $np = 8000000$ y $e = 16384$.

(b) Distribución uniforme con $np = 4000000$ y $e = 8192$.

Fig. 6.1: Representación gráfica de distribuciones de puntos sintéticos

6.2.2. Datos reales

En los experimentos también usamos datos reales con el propósito de verificar el rendimiento de nuestros algoritmos en la práctica. Usamos tres set de datos: RCA de 2224727 puntos, RTS de 194971 puntos y RCTB de 556696 puntos. Para todos los set de datos se utilizaron matrices de cuatro tamaños diferentes: 2^{11}, 2^{12}, 2^{13} y 2^{14}. En la Fig. 6.2 se muestran las distribuciones de los puntos de los conjuntos RCA, RTS y RCTB.



(a) RCA 4096×4096

(b) RCTB 16384×16384

(c) RTS 16384×16384

Fig. 6.2: Set de datos espaciales

Con respecto al problema de la compactación quisimos probar sobre conjuntos de datos que presentarán clusterización en sus elementos. Para esto se utilizaron 2 grafos web. El primer grafo web llamado CNR en su versión del año 2000, el que cuenta con un total de 3216152 datos en un espacio de 325577×325577 (ver Fig. 6.3).

El segundo grafo llamado EU en su versión del año 2005 cuenta con un total de 19235140 datos en un espacio de 862664×862664 (ver Fig. 6.4 [a]), de este grafo se extrajeron dos grafos más pequeños respecto a la diagonal principal los cuales fueron los utilizados para los experimentos. Supongamos una división en cuatro cuadrantes de igual tamaño sobre el conjunto EU tal como lo haría un k^2-tree con $k = 2$. El primer grafo extraído de EU se llamó c2EU2005 ya que corresponde al segundo cuadrante (ver Fig. 6.4 [b]). El segundo grafo extraído se llamó c3EU2005 por corresponder al tercer cuadrante (ver Fig. 6.4 [c]). Ambos grafos fueron trasladados al origen del plano y dimensionados en un espacio de 431332×431332.

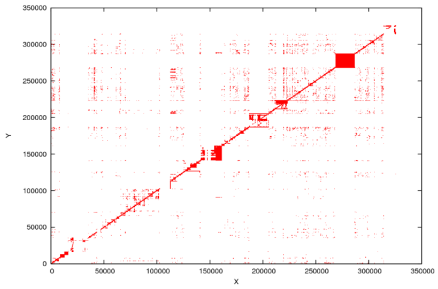
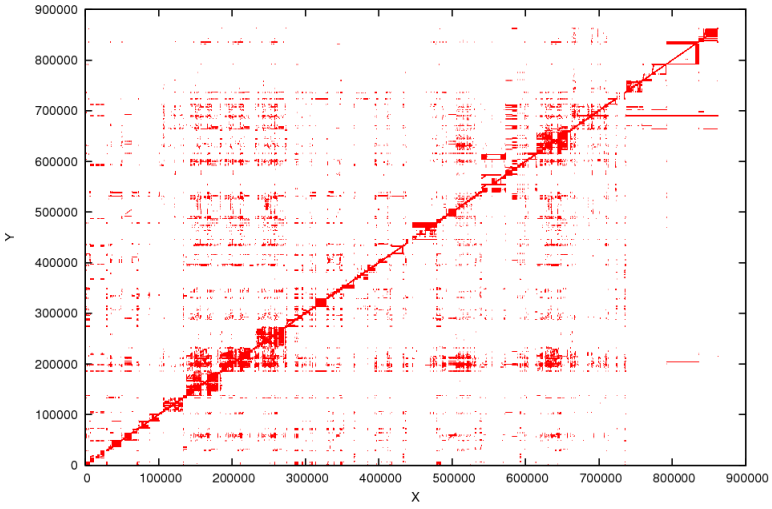
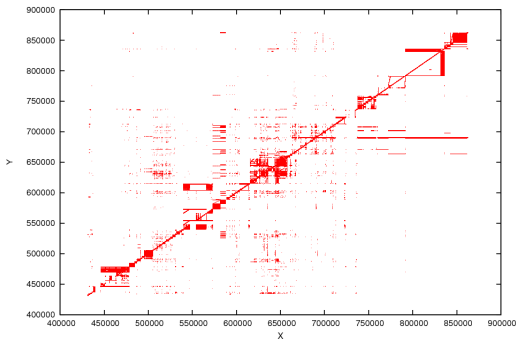


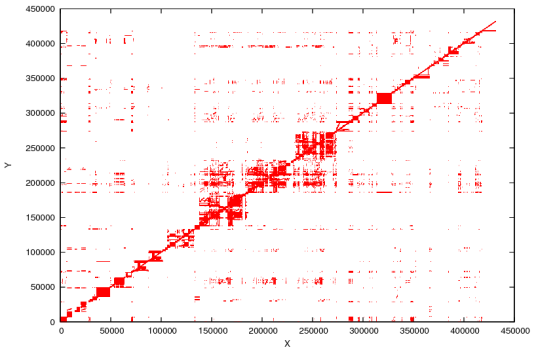
Fig. 6.3: CNR 2000



(a) EU 2005



(b) c2EU2005 (datos sin trasladar al origen)



(c) c3EU2005 (datos sin trasladar al origen)

Fig. 6.4: Set de datos clusterizados

Antes de continuar presentamos un tercer algoritmo para la cerradura convexa, el cual está implementado con la única finalidad de hacer una transición de una estructura compacta a una estructura regular y así poder aplicar un algoritmo ya implementado para la cerradura convexa.

6.3. Graham Scan sobre un k^2-tree

Un enfoque distinto a la búsqueda de la cerradura convexa a base de puntos extremos es la que proponemos aquí. Cuando aplicamos el algoritmo de Graham Scan (ver Sección 2.4.1) no es necesario saber cuales son los valores extremos dentro del conjunto de datos. Sin embargo, dada la implementación a base de bitmaps del k^2-tree nos es imposible aplicar el Graham Scan sobre la estructura. Por lo tanto proponemos recorrer el k^2-tree en su completitud para poder saber cuales son los puntos almacenados, extraer cada uno de ellos a un vector de enteros y sobre ese vector aplicar el Graham Scan. Para extraer las coordenadas sobre un k^2-tree aplicamos la fórmula de las coordenadas de una hoja, para lo cual utilizaremos los Bounding Box para mantener registro del recorrido de los puntos desde la raíz hasta las hojas.

El k^2-tree al ser una estructura de datos jerárquica basada en un árbol ordinal, nos permite hacer un recorrido pre-order sobre la misma, recorrido que por lo demás favorece el cálculo de las coordenadas de una hoja basadas en la sumatoria de los nodos directamente relacionados entre la hoja y la raíz (ver Algoritmo 6.1).

```
1  Algoritmo: Extracción de puntos sobre un k^2-tree
2  raiz = raíz del k^2-tree;
3  Pila nodo;
4  Vector⟨BB⟩ ret;
5  push(nodo, BB(raiz, 0));
6  while nodo no esté vacía do
7      n = pop(nodo);
8      if h_i = hijos de n son hojas then
9          foreach h_i hijos de n distintos de 0 do
10             ret.add(BB(n,h_i));
11     else
12         foreach h_i hijos de n distintos de 0 do
13             push(nodo,BB(n,h_i));
14 return ret;
```

Alg. 6.1: Algoritmo para la extracción de puntos sobre un k^2-tree

Este método claramente no es el ideal, ya que sólo estamos adecuando la estructura a un formato ya conocido para trabajar, perdiendo así todas las propiedades del k^2-tree de las cuales podemos beneficiarnos. Sin embargo, creemos necesaria su implementación para la comparación de los resultados contra los demás algoritmos.

La complejidad calculada para esta solución es de $O(n \log n + k^{2^{\lceil \log_k n \rceil + 1}})$, donde $n \log n$ co-

rresponde a la complejidad del Graham Scan y $k^{2^{\lceil \log_k n \rceil + 1}}$ corresponde a la cota superior en el caso de la exploración total de un k^2-tree lleno.

6.4. Set de experimentos

Tal como indicamos, en los experimentos medimos dos parámetros: el tiempo y el almacenamiento requerido por el algoritmo.

En el primer grupo de experimentos medimos el tiempo consumido por los algoritmos. Para esto se generó para cada set de datos (sintéticos y reales) un total de 10 ejecuciones para cada algoritmo (CHk^2, CHk^2t y GSk^2), lo que en total nos da 30 ejecuciones para cada uno de los 39 set de datos. Se consideró prudente establecer 10 ejecuciones para cada set de datos debido a que los tiempos de respuestas pueden variar de una ejecución a otra, y al tener 10 respuestas diferentes (dentro de un rango razonable) podemos calcular un promedio de tiempo de respuesta.

El segundo grupo de experimentos se enfocó en medir el almacenamiento requerido por los algoritmos. En este grupo, buscamos indicadores respecto al espacio de almacenamiento usado tanto por las estructuras de los set de datos, como el k^2-tree, como también de las estructuras auxiliares utilizadas por los algoritmos, como lo son el caso de la cola de prioridades para el CHk^2 y el CHk^2t, y el vector de puntos para el GSk^2. Para cada set de datos se efectuaron 2 ejecuciones por cada algoritmo, es decir, 6 ejecuciones para cada uno de los 39 set de datos. Ya que los set de datos no varían por cada ejecución no es necesario repetir los experimentos para un mismo set de datos, es por esto que la segunda ejecución sólo fue para corroborar los datos obtenidos en la primera.

6.5. Resultados

6.5.1. Resultados temporales

Datos sintéticos

En las Tablas 6.2, 6.3, 6.4 y 6.5 podemos ver el tiempo[2] que le toma a cada uno de los algoritmos para conjuntos de 1, 2, 4 y 8 millones de puntos respectivamente con distribución Gaussiana y Uniforme y posicionados sobre los espacios definidos en la Tabla 6.1. En la última fila se presenta $\%\Delta$ que indica la variación porcentual entre el menor y el mayor valor por columna.

Para los resultados obtenidos en las tablas 6.2, 6.3, 6.4 y 6.5 podemos observar a través de los gráficos correspondientes 6.5, 6.6, 6.7 y 6.8 que uno de los mayores indicadores de cambio en el desempeño temporal de los algoritmos es el espacio en que se generaron los datos, dándose una relación de a mayor tamaño el espacio, mayor será el tiempo de respuesta. Algo previsto, ya que al ser un mayor espacio dimensional y al mantenerse el valor de k significará que la altura del árbol se incrementará y por tanto la cantidad total de cuadrantes también, lo que hace más costosa la tarea de los algoritmos.

[2]Todos los tiempos calculados en este Capítulo están medidos en segundos.

1 millón de puntos							
Espacio		**Gauss**			**Uniforme**		
		CHk^2	**GSk^2**	**CHk^2t**	**CHk^2**	**GSk^2**	**CHk^2t**
1024		0,0005777	0,7037591	0,0004825	0,000116	0,7485737	0,0000928
4096		0,000625	0,7950698	0,0006672	0,0004838	0,9631146	0,0005776
16384		0,0007381	1,0763793	0,00076	0,0005679	1,2640053	0,0006031
% Δ	*1600*	*127,77*	*152,95*	*157,51*	*489,57*	*168,86*	*649,89*

Tabla 6.2: Resultados para 1 millón de puntos

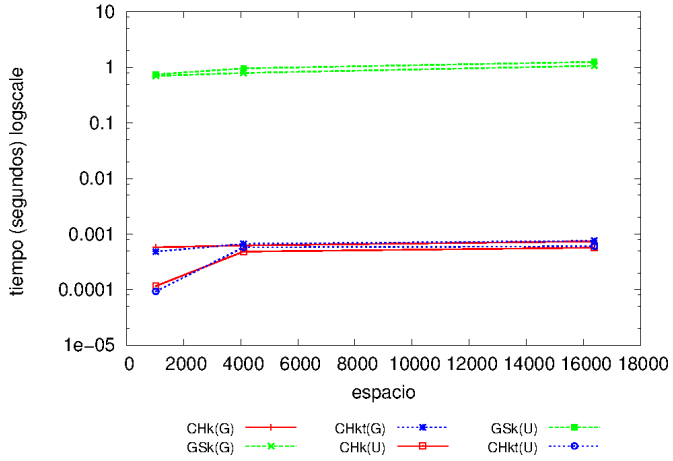


Fig. 6.5: Desempeño temporal para 1 millón de puntos

2 millones de puntos							
Espacio		**Gauss**			**Uniforme**		
		CHk^2	**GSk^2**	**CHk^2t**	**CHk^2**	**GSk^2**	**CHk^2t**
2048		0,0001543	0,2727297	0,0001319	0,00023	1,4366531	0,0001178
4096		0,0004586	1,1495548	0,0003387	0,0004377	1,8443667	0,0002908
16384		0,0004247	2,234218	0,0005117	0,0006963	2,5980554	0,0007409
%Δ	*800*	*275,24*	*819,21*	*387,95*	*302,74*	*180,84*	*628,95*

Tabla 6.3: Resultados para 2 millones de puntos

Si contrastamos los algoritmos es notorio el bajo desempeño del GSk^2 con respecto a las demás implementaciones, esto claramente se debe a que en realidad se están aplicando dos algoritmos para calcular la cerradura convexa, y que por demás transforma una estructura compacta a una estructura de datos normal, que pierde los beneficios tanto de la jerarquía del k^2-tree como los beneficios derivados de la jerarquía de memoria que aprovechan las estructuras de datos compactas.

Al evaluar el CHk^2 y el CHk^2t vemos que su desempeño en cuanto a tiempo de respuesta es muy similar, dándose una mayor similitud en los conjuntos de datos uniforme por sobre los de

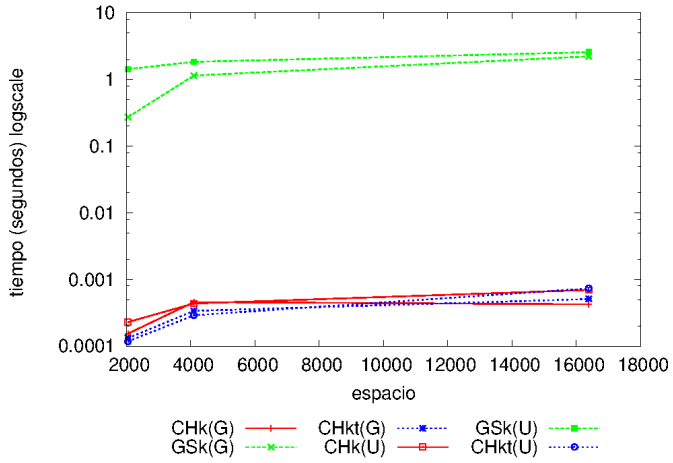


Fig. 6.6: Desempeño temporal para 2 millones de puntos

4 millones de puntos							
Espacio		**Gauss**			**Uniforme**		
		CHk^2	**GSk^2**	**CHk^2t**	**CHk^2**	**GSk^2**	**CHk^2t**
4096		0,0005786	1,725197	0,0005885	0,0002862	3,4819604	0,0002281
8192		0,0005637	3,0870408	0,0007054	0,0003245	4,1985538	0,0002616
16384		0,0005783	4,0286308	0,0005588	0,0004627	5,032145	0,000425
%Δ	*400*	*99,95*	*233,52*	*94,95*	*161,67*	*144,52*	*186,32*

Tabla 6.4: Resultados para 4 millones de puntos

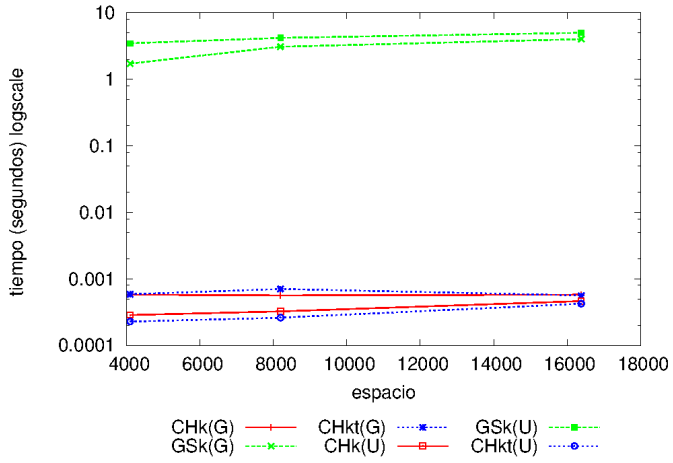


Fig. 6.7: Desempeño temporal para 4 millones de puntos

distribución gaussiana. Esto nos lleva a concluir que la densidad de los datos influye directamente en una baja existencia de falsos positivos dentro del set, siendo el fuerte de CHk^2t la eliminación de

8 millones de puntos							
Espacio		**Gauss**			**Uniforme**		
		CHk^2	**GSk^2**	**CHk^2t**	**CHk^2**	**GSk^2**	**CHk^2t**
	4096	0,0007154	6,9922325	0,0002811	0,0002281	7,6230904	0,0002281
	16384	0,0004802	7,6689264	0,0006022	0,0004977	9,4006375	0,0002616
	32768	0,0007689	8,9881001	0,0004868	0,0004992	10,5095948	0,000425
%Δ	*800*	*128,54*	*111,40*	*107,48*	*137,87*	*173,18*	*218,85*

Tabla 6.5: Resultados para 8 millones de puntos

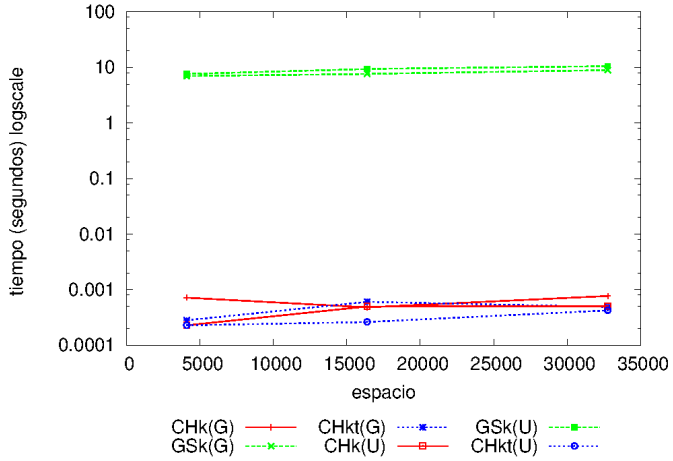


Fig. 6.8: Desempeño temporal para 8 millones de puntos

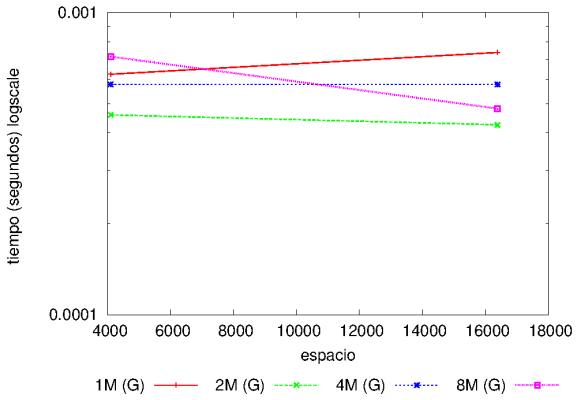
estos falsos positivos. Es por esto que no vemos una variación relevante entre estos dos algoritmos para estos escenarios en que los datos están densamente concentrados.

No sólo el espacio influye en el desempeño del algoritmo, sino también la cantidad de datos, aunque suene obvio, es necesario mencionar que a mayor tamaño sea el conjunto de datos más costoso será para el algoritmo resolver el problema. Sin embargo, para los conjuntos con distribución Gaussiana se da el caso en que a una mayor cantidad de datos en un espacio también mayor, la densidad del conjunto baja de manera significativa, y por tanto las variaciones porcentuales entre distintos escenarios se amortigua cada vez más (ver Fig. 6.9).

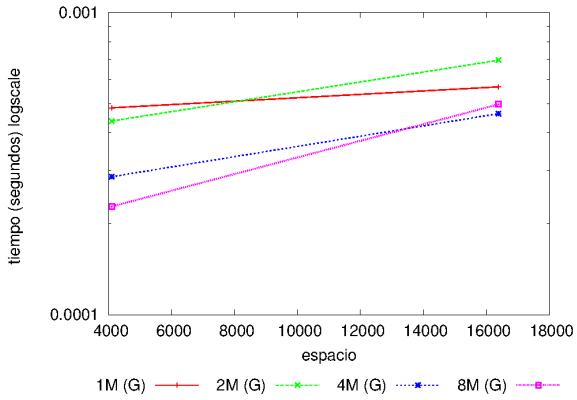
Datos reales

En las Tablas 6.6, 6.7 y 6.8 podemos ver el tiempo que le toma a cada uno de los algoritmos para los conjuntos de puntos espaciales RCA, RTS y RCTB respectivamente, posicionados sobre distintos tamaños de espacios ya definidos. En la última fila se presenta %Δ que indica la variación porcentual entre el menor y el mayor valor por columna.

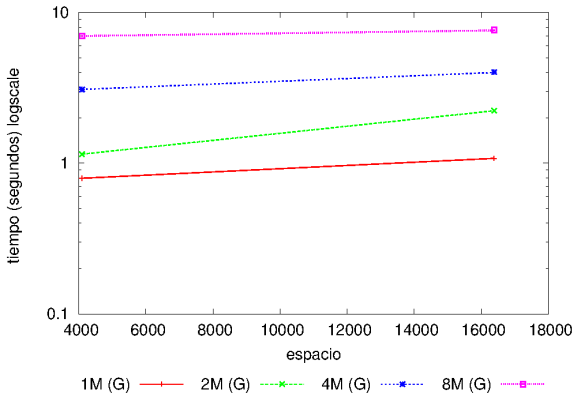
Al analizar los datos respecto a los set de datos espaciales, se comienza a hacer notorio una relación entre la cantidad de datos y la dimensión del espacio en que está representado. Podemos



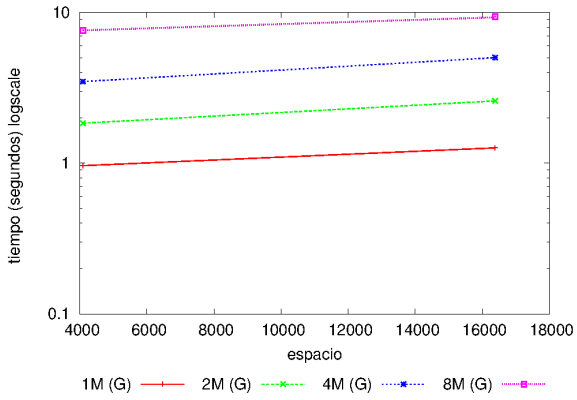
(a) CHk^2 para distribución Gaussiana



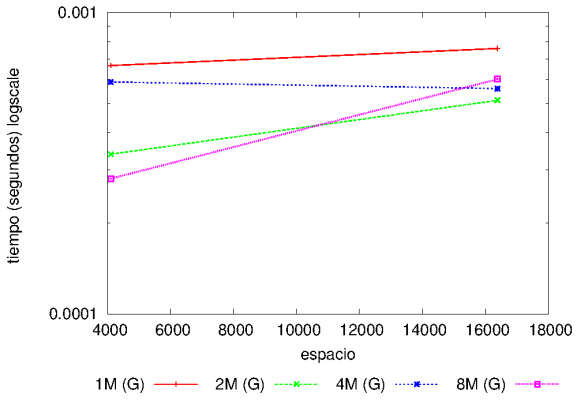
(b) CHk^2 para distribución Uniforme



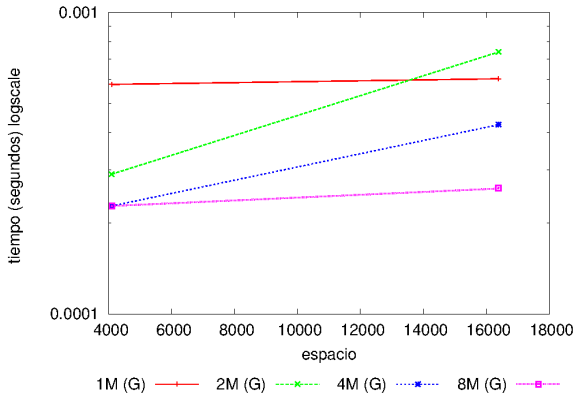
(c) GSk^2 para distribución Gaussiana



(d) GSk^2 para distribución Uniforme



(e) CHk^2t para distribución Gaussiana



(f) CHk^2t para distribución Uniforme

Fig. 6.9: Desempeño temporal para CHk^2, GSk^2 y CHk^2t

RCA				
Cantidad de puntos	Espacio	CHk^2	GSk^2	CHk^2t
2249727	2048	0,0017867	0,3449493	0,0016952
	4096	0,0026176	0,6902825	0,0023664
	8192	0,0030774	1,2573215	0,0025862
	16384	0,0039684	1,8819714	0,0028764
% Δ	*800*	*222,11*	*545,58*	*169,68*

Tabla 6.6: Resultados para el set RCA

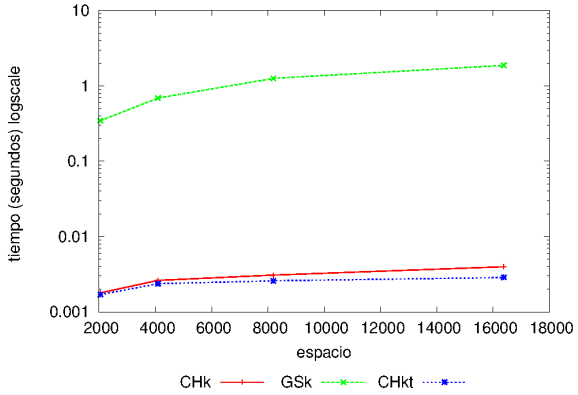


Fig. 6.10: Desempeño temporal para RCA

RCTB				
Cantidad de puntos	Espacio	CHk^2	GSk^2	CHk^2t
556696	2048	0,0004971	0,4346185	0,0004877
	4096	0,0009329	0,5000092	0,0005599
	8192	0,0009864	0,5719448	0,0007031
	16384	0,0018521	0,6696978	0,001308
% Δ	*800*	*372,58*	*154,09*	*268,20*

Tabla 6.7: Resultados para el set RCTB

RTS				
Cantidad de puntos	Espacio	CHk^2	GSk^2	CHk^2t
194971	2048	0,000735	0,1788208	0,0007549
	4096	0,0011472	0,2112017	0,0008337
	8192	0,0009728	0,2434636	0,0010053
	16384	0,0016721	0,277284	0,0016461
% Δ	*800,00*	*227,50*	*155,06*	*218,06*

Tabla 6.8: Resultados para el set RTS

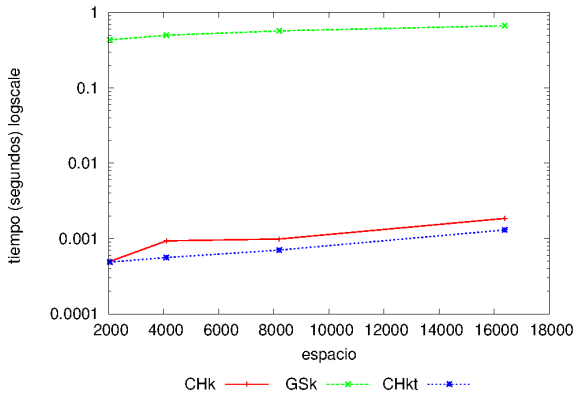


Fig. 6.11: Desempeño temporal para RCTB

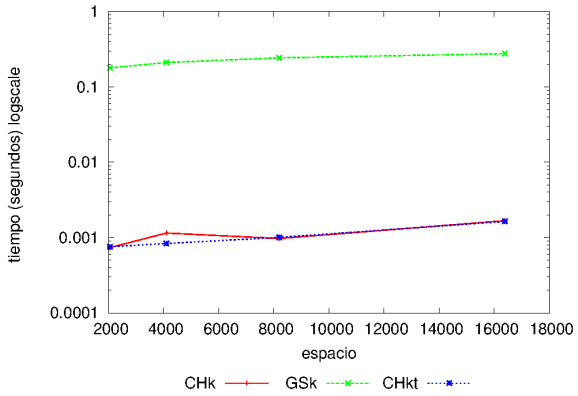


Fig. 6.12: Desempeño temporal para RTS

ver en las gráficos 6.10, 6.11 y 6.12 que a medida que nos alejamos del set formado en un espacio de 2048×2048 y que nos acercamos al espacio (para la misma cantidad de datos) de 16384×16384 que los tiempos de respuesta tienden a mantenerse constantes, como si se tratara de una función logarítmica acercándose a su cota superior. Si bien sería necesario extender los experimentos a mayores dimensiones para establecer esta relación. El hecho de que la situación se repita en los tres conjuntos de datos sienta una base para establecer una relación entre la densidad del conjunto respecto al desempeño de los algoritmos.

En la Tabla 6.9 se muestran los resultados obtenidos para cada uno de los grafos web al medir su desempeño temporal, además indicando la cantidad de puntos como también el espacio en que se dimensionaron los puntos.

Si analizamos la tabla 6.9 ya podemos definir una tendencia en cuanto a un peor desempeño del GSk^2 en contra de los otros algoritmos. Ya es evidente que este algoritmo es el menos eficiente en cuanto a tiempo de respuesta, y dista mucho de lo logrado por las demás implementaciones.

Podemos notar también que la brecha entre los tiempos del CHk^2 y el CHk^2t se incrementa considerablemente respecto a los casos anteriormente analizados, tomándole al CHk^2t en el mejor

Grafos Web					
Grafo	**Cantidad de puntos**	**Espacio**	**CHk^2**	**GSk^2**	**CHk^2t**
CNR (2000)	3216152	325577	0,8878030	2,8232602	0,0069516
c2EU2005	310230	431332	0,0185854	0,2582101	0,0009690
c3EU2005	429050	431332	0,0958267	0,3507733	0,0033567

Tabla 6.9: Resultados para los grafos web

escenario planteado tan sólo un 0,8 % del tiempo total necesario para el CHk^2, demostrándo así la utilidad de la traslación del segmento, y además de establecer la relación entre la densidad de los conjuntos con respecto a la aparición de los falsos positivos.

6.5.2. Almacenamiento requerido por los algoritmos

Datos sintéticos

Para cada uno de los algoritmos implementados y las distribuciones utilizadas en esta tesis se midió también el espacio de almacenamiento requerido por los mismos, tanto por la estructura contenedora de los puntos, como las estructuras adicionales requeridas por los distintos algoritmos. Para los conjuntos de datos sintéticos se presentan los resultados en las Tablas 6.10, 6.11, 6.12 y 6.13, siendo todos los resultados medidos en Bytes.

1 millón de puntos							
Espacio		**Gauss**			**Uniforme**		
		CHk^2	**GSk^2**	**CHk^2t**	**CHk^2**	**GSk^2**	**CHk^2t**
1024	k^2-tree	57320	57320	57320	180680	180680	180680
	Est. Adicional	1080	2523648	800	1200	7738032	1200
	Total	58400	2580968	58120	181880	7918712	181880
4096	k^2-tree	426408	426408	426408	986148	986148	986148
	Est. Adicional	960	9567480	760	1280	11651952	1200
	Total	427368	9993888	427168	987428	12638100	987348
16384	k^2-tree	1338684	1338684	1338684	2016928	2016928	2016928
	Est. Adicional	1000	11813964	760	1280	11977416	1280
	Total	1339684	13152648	1339444	2018208	13994344	2018208

Tabla 6.10: Espacio de almacenamiento de estructuras para 1 millón de puntos

Con respecto a los resultados de espacio de almacenamiento detallados en las tablas 6.10, 6.11, 6.12 y 6.13 y sus respectivos gráficos 6.13, 6.14, 6.15 y 6.16 (con el total de espacio de almacenamiento alcanzado por cada algoritmo) podemos notar una clara superioridad en cuanto a eficiencia del recurso espacio de almacenamiento por parte de los algoritmos CHk^2 y CHk^2t contra el GSk^2, ya que este último convierte una estructura compacta a una estructura regular como un vector de puntos requiriendo así mayor almacenamiento.

Existe una gran diferencia para los mismos algoritmos sobre distintas distribuciones para un conjunto del mismo tamaño, sin embargo esto puede ser atribuido completamente al hecho de

que un conjunto de tamaño n en un espacio de $x \times x$ tiene mayor probabilidad de generar datos repetidos en una distribución gaussiana que en una uniforme, ya que la primera distribución concentra los datos en un área menor a la que utiliza la segunda.

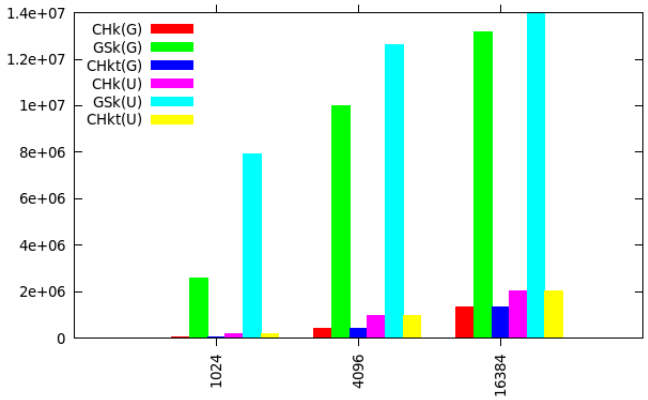


Fig. 6.13: Espacio de almacenamiento total para 1 millón de puntos

2 Millones de puntos							
Espacio		**Gauss**			**Uniforme**		
		CHk^2	**GSk^2**	**CHk^2t**	**CHk^2**	**GSk^2**	**CHk^2t**
2048	k^2-tree	138608	138608	138608	652596	652596	652596
	Est. Adicional	1000	4021284	1000	1200	19102620	1200
	Total	139608	4159892	139608	653796	19755216	653796
4096	k^2-tree	548056	548056	548056	1487804	1487804	1487804
	Est. Adicional	880	15862020	840	1280	22625772	1280
	Total	548936	16410076	548896	1489084	24113576	1489084
16384	k^2-tree	2174632	2174632	2174632	3512612	3512612	3512612
	Est. Adicional	960	23274180	800	1360	23913492	1360
	Total	2175592	25448812	2175432	3513972	27426104	3513972

Tabla 6.11: Espacio de almacenamiento de estructuras para 2 millones de puntos

Es relevante mencionar, que al observar los gráficos podemos notar que en muchos casos a través de todos los set de datos sintéticos los algoritmos CHk^2 y CHk^2t no superan en espacio de almacenamiento a un GSk^2 a través de distintos conjuntos, lo que es fácilmente observable al analizar los datos entre los conjuntos de 8 millones de datos contra los de 1 millón de datos, donde el mayor CHk^2 del primer set (8700160 bytes) no supera al mayor GSk^2 del segundo set (13152648 bytes), alcanzando el CHk^2 sólo un 66,15 % del total utilizado por el GSk^2 para un espacio que en teoría es 8 veces menor.

En la Tabla 6.14 podemos notar el detalle en cuanto a qué porcentaje representa un algoritmo sobre otro para los mismos conjuntos de 1 millón de datos con distribuciones gaussianas (G) y uniformes (U). Las comparaciones se hacen de la forma *la columna A ocupa un x % de la fila B*,

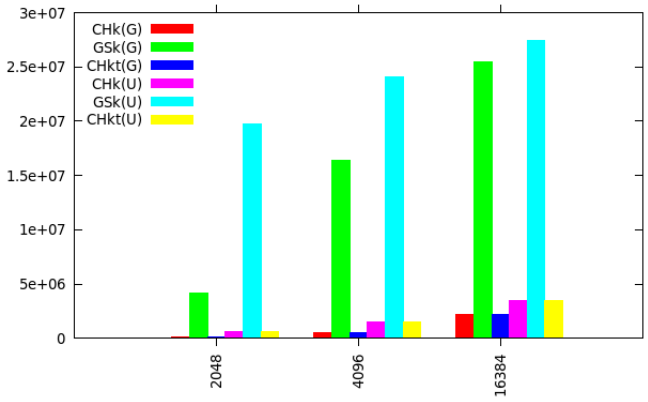


Fig. 6.14: Espacio de almacenamiento total para 2 millones de puntos

4 Millones de puntos							
Espacio		**Gauss**			**Uniforme**		
		CHk^2	**GSk^2**	**CHk^2t**	**CHk^2**	**GSk^2**	**CHk^2t**
4096	k^2-tree	669800	669800	669800	2075088	2075088	2075088
	Est. Adicional	1040	23665452	800	1360	32000000	1360
	Total	670840	24335252	670600	2076448	34075088	2076448
8192	k^2-tree	1705440	1705440	1705440	3943772	3943772	3943772
	Est. Adicional	960	23665452	720	1320	42690612	1320
	Total	1706400	25370892	1706160	3945092	46634384	3945092
16384	k^2-tree	3380188	3380188	3380188	5983580	5983580	5983580
	Est. Adicional	960	45170424	840	1280	47644644	1280
	Total	3381148	48550612	3381028	5984860	53628224	5984860

Tabla 6.12: Espacio de almacenamiento de estructuras para 4 millones de puntos

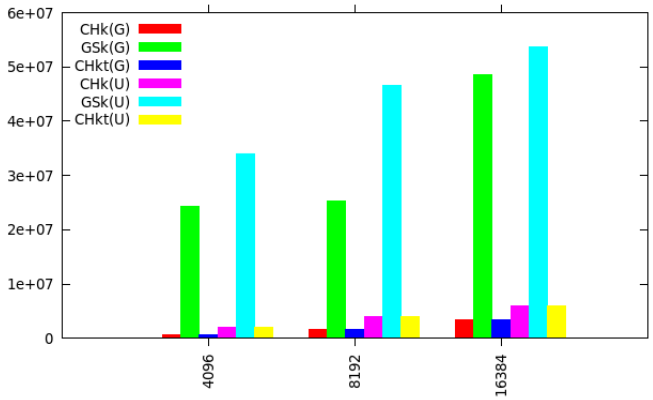


Fig. 6.15: Espacio de almacenamiento para 4 millones de puntos

8 Millones de puntos							
Espacio		**Gauss**			**Uniforme**		
		CHk^2	**GSk^2**	**CHk^2t**	**CHk^2**	**GSk^2**	**CHk^2t**
4096	k^2-tree	730984	730984	730984	2609080	2609080	2609080
	Est. Adicional	1080	29990148	840	1280	76351644	1280
	Total	732064	30721132	731824	2610360	78960724	2610360
16384	k^2-tree	4812596	4812596	4812596	9908896	9908896	9908896
	Est. Adicional	680	84640788	600	1400	94592280	1400
	Total	4813276	89453384	4813196	9910296	104501176	9910296
36768	k^2-tree	8699320	8699320	8699320	14046252	14046252	14046252
	Est. Adicional	840	93095148	720	1560	95644644	1560
	Total	8700160	101794468	8700040	14047812	109690896	14047812

Tabla 6.13: Espacio de almacenamiento de estructuras para 8 millones de puntos

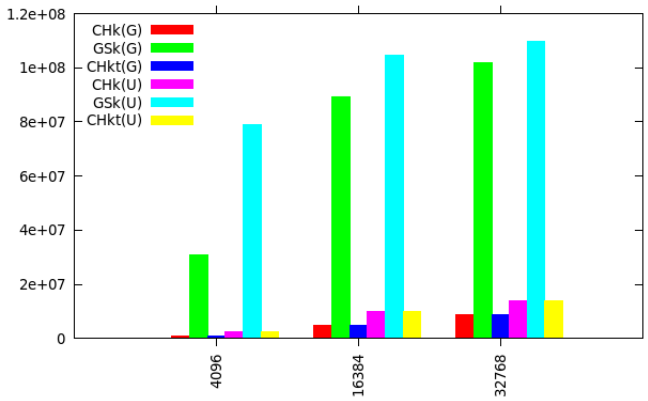


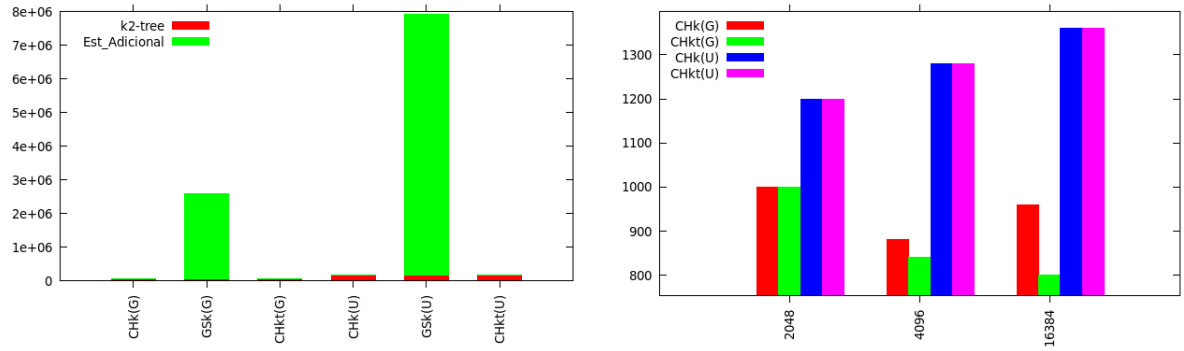
Fig. 6.16: Espacio de almacenamiento para 8 millones de puntos

Comparación porcentual de espacio de almacenamiento para 1 Millón de puntos						
	CHk^2 (G)	**GSk^2 (G)**	**CHk^2t (G)**	**CHk^2 (U)**	**GSk^2 (U)**	**CHk^2t (U)**
CHk^2 (G)		*981,77*	*99,98*	*150,65*	*1044,60*	*150,65*
GSk^2 (G)	*4,28*		*4,27*	*15,34*	*106,40*	*15,34*
CHk^2t (G)	*100,02*	*981,95*		*150,68*	*1044,79*	*150,68*
CHk^2 (U)	*66,38*	*651,70*	*66,37*		*693,40*	*100,00*
GSk^2 (U)	*150,65*	*93,99*	*9,57*	*14,42*		*14,42*
CHk^2t (U)	*66,38*	*651,70*	*66,37*	*100,00*	*693,40*	

Tabla 6.14: Comparación porcentual de espacio de almacenamiento utilizado por los distintos algoritmos para 1 millón de puntos

presentando el porcentaje en la intersección de ambas. Si analizamos los datos tenemos como un representativo de la compactación el caso del GSk^2(U) que ocupa un 1044, 79 % del CHk^2t(G), y

aunque sean conjuntos distintos (por la repetición de datos y la distribución) se puede notar que el GSk^2(G) ocupa un $981,95\,\%$ del CHk^2t(G), y por el otro lado que el GSk^2(U) corresponde a un $693,40\,\%$ del CHk^2t(U), cifras que no están tan alejadas considerando el tamaño de las mismas y que dejan evidencia del poder de compactación del k^2-tree y el bajo espacio de almacenamiento temporal utilizado por los algoritmos CHk^2 y CHk^2t. Este dato también nos refleja que el mejor algoritmo en cuanto a espacio de almacenamiento es el CHk^2t y por otro lado nos confirma que el GSk^2 es el peor.



(a) Detalle de espacio de almacenamiento para 1 millón de datos en un espacio de 1024×1024

(b) Comparación de estructuras adicionales para algoritmos CHk^2 y CHk^2t para el set de 2 millones de datos

Fig. 6.17: Análisis de espacio de almacenamiento de datos sintéticos

Con respecto al uso de las estructuras adicionales podemos notar que para los algoritmos CHk^2 y CHk^2t, el espacio de almacenamiento utilizado por estas (ver Fig. 6.17 [a]) es tan sólo una pequeña fracción del total utilizado por la estructura, lo que significa que podemos mantener la compactación de los datos controlada, a través de la interpretación de datos justa y necesaria. Ahora, si entramos en el detalle de las estructuras adicionales para los algoritmos CHk^2 y CHk^2t (ver Fig. 6.17 [b]) notamos que la variación es prácticamente nula en los conjuntos uniformes (debido a la casi inexistente presencia de falsos positivos producto de la uniformidad del conjunto), sin embargo presentando una mejora considerable para los conjuntos con distribución gaussiana, donde el aumento del espacio (dimensión) reduce el espacio de almacenamiento para el CHk^2t.

Datos reales

Al igual que para los datos sintéticos, para los datos reales se midió el espacio de almacenamiento requerido tanto por la estructura contenedora de los puntos, como las estructuras adicionales de los distintos algoritmos. A continuación se presentan los resultados en las Tablas 6.15, 6.16 y 6.17, siendo todos los resultados medidos en Bytes.

Como era de esperar, para los datos espaciales (RCA, RCTB y RTS) el espacio de almacenamiento utilizado por el GSk^2 crece muy velozmente a través de los distintos tamaños de las matrices representadas sobre las que se implementa el conjunto de puntos, manteniendo siempre una tendencia incremental a lo largo de los experimentos para todos los conjuntos. Sin embargo, esta tendencia es mayor para los conjuntos que representan mayores cantidades de datos como el

RCA				
Espacio	Estructura	CHk^2	GSk^2	CHk^2t
2048	k^2-tree	169024	169024	169024
	Est. Adicional	4360	5197704	2240
	Total	173384	5366728	171264
4096	k^2-tree	396428	396428	396428
	Est. Adicional	4800	9424548	2520
	Total	401228	9820976	398948
8192	k^2-tree	808760	808760	808760
	Est. Adicional	4920	15816348	2840
	Total	813680	16625108	811600
16384	k^2-tree	1500732	1500732	1500732
	Est. Adicional	5000	21952992	2960
	Total	1505732	23453724	1503692

Tabla 6.15: Resultados espacio de almacenamiento para set de datos RCA

RTS				
Espacio	Estructura	CHk^2	GSk^2	CHk^2t
2048	k^2-tree	174332	174332	174332
	Est. Adicional	1640	2225916	1480
	Total	175972	2400248	175812
4096	k^2-tree	271724	271724	271724
	Est. Adicional	1240	2299620	1480
	Total	273364	2571344	273204
8192	k^2-tree	372336	372336	372336
	Est. Adicional	1960	2326932	1800
	Total	374296	2699268	374136
16384	k^2-tree	474144	474144	474144
	Est. Adicional	2000	2335368	1840
	Total	476144	2809512	475984

Tabla 6.16: Resultados espacio de almacenamiento para el set de datos RTS

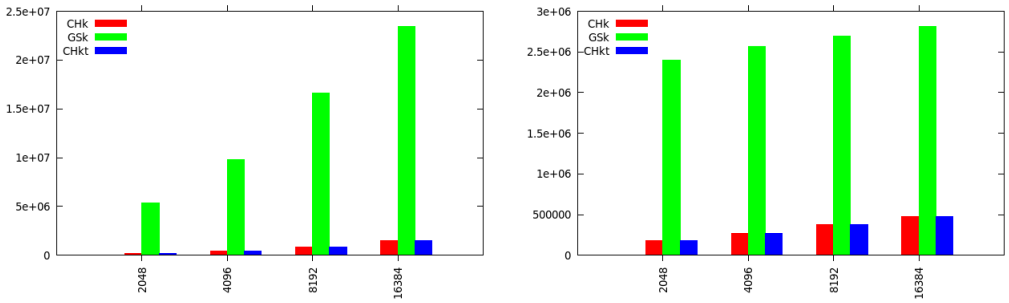
RCA que los otros que representan menores cantidades de datos como el RCTB y el RTS, lo que es observable en la diferencia de tamaño de las barras en los gráficos 6.5.2 [a], [b] y [c].

El comportamiento de los algoritmos CHk^2 y CHk^2t a lo largo de las distintas áreas de representación mantiene la misma tendencia que el GSk^2, considerando siempre las grandes diferencias entre el espacio de almacenamiento usado por los primeros algoritmos versus el GSk^2, reforzando el hecho de una mejor eficiencia tanto temporal como en espacio de almacenamiento por parte de estos algoritmos.

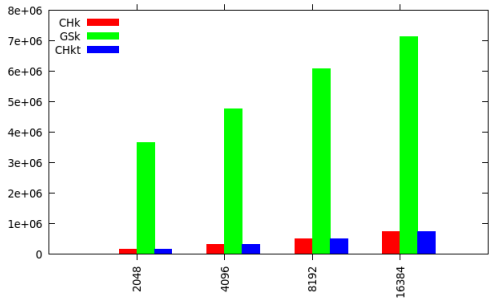
Al ver el desempeño de las estructuras de datos adicionales (ver Fig. 6.19 [b]) para los algoritmos CHk^2 y CHk^2t podemos notar una gran diferencia entre los tamaños de estas, dándose

RCTB				
Espacio	Estructura	CHk^2	GSk^2	CHk^2t
2048	k^2-tree	157072	157072	157072
	Est. Adicional	1160	3494076	1080
	Total	158232	3651148	158152
4096	k^2-tree	309948	309948	309948
	Est. Adicional	1440	4467408	1120
	Total	311388	4777356	311068
8192	k^2-tree	505404	505404	505404
	Est. Adicional	1440	5567772	1160
	Total	506844	6073176	506564
16384	k^2-tree	749000	749000	749000
	Est. Adicional	1140	6386628	1200
	Total	750440	7135628	750200

Tabla 6.17: Resultados espacio de almacenamiento para el set de datos RCTB



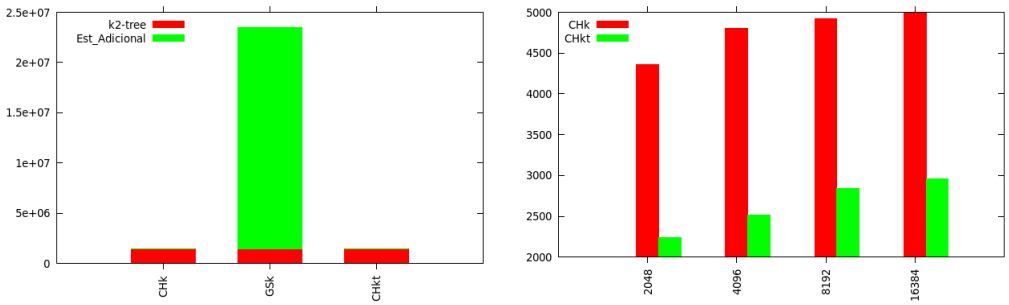
(a) Espacio de almacenamiento para el set RCA (b) Espacio de almacenamiento para el set RTS



(c) Espacio de almacenamiento para el set RCTB

Fig. 6.18: Espacio de almacenamiento para datos espaciales.

en promedio un uso del 50 % de espacio de almacenamiento por parte del algoritmo traslador respecto al primero, y cumpliéndose entonces el objetivo de la traslación de segmento, dejando



(a) Detalle de espacio de almacenamiento para el set de datos RCA en un espacio de 16384 × 16384

(b) Comparación de estructuras adicionales para algoritmos CHk^2 y CHk^2t para el set RCTB

Fig. 6.19: Análisis de espacio de almacenamiento de datos espaciales

en evidencia la aparición de falsos positivos para el set de datos estudiado. También es necesario mencionar el hecho de que el crecimiento de la estructura adicional para el CHk^2t es sostenido y parece estar directamente relacionado (más allá de lo evidente que deriva de estar trabajando sobre el mismo conjunto) con la estructura adicional del CHk^2. El porcentaje de las estructuras adicionales con respecto al uso total de espacio de almacenamiento (ver Fig. 6.19 [a]) es despreciable en los algoritmos CHk^2 y CHk^2t (1, 19 % y 0, 64 % respectivamente), no así para el GSk^2 donde la estructura adicional corresponde al mayor porcentaje del tamaño dándose casi de manera inversa a los primeros casos que este represente un 95, 96 % del total, reafirmando la idea de lo ineficiente de esta implementación si la comparamos con los otros algoritmos implementados.

Grafos Web						
Grafo	**Cant. de puntos**	**Espacio**	**Estructura**	**CHk^2**	**GSk^2**	**CHk^2t**
CNR	**3216152**	**325577**	k^2-tree	1793128	1793128	1793128
			Est. Adicional	86280	38593824	55080
c2EU2005	**310230**	**431332**	k^2-tree	249196	249196	249196
			Est. Adicional	1960	3722760	1480
c3EU2005	**429050**	**431332**	k^2-tree	294508	294508	294508
			Est. Adicional	7480	5148600	6480

Tabla 6.18: Resultados de espacio de almacenamiento para grafos web

Al analizar los datos obtenidos para los grafos web (ver Fig. 6.20 y 6.21) sólo podemos confirmar las interpretaciones que se han venido haciendo desde el comienzo de los experimentos, no encontrando nada nuevo respecto de la comparación de estos conjuntos con otros como el de los espaciales.

6.6. Conclusiones

Finalizado los experimentos podemos concluir lo siguiente:

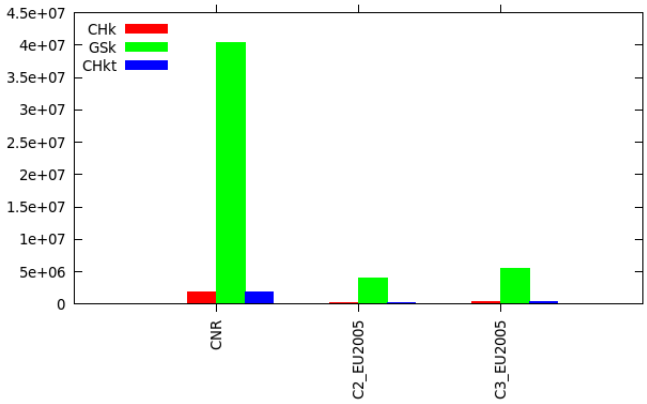
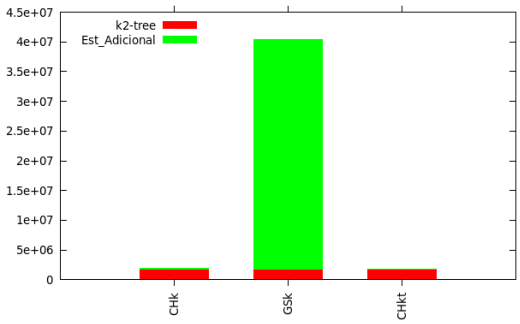
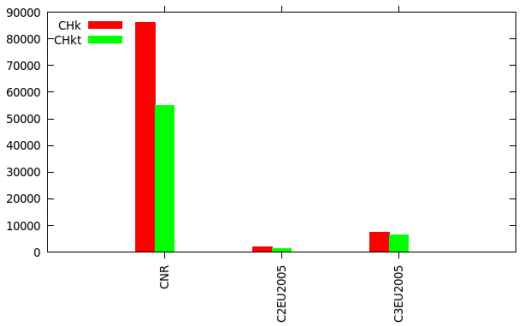


Fig. 6.20: Espacio de almacenamiento para los grafos web



(a) Detalle de espacio de almacenamiento para el grafo web CNR



(b) Comparación de estructuras adicionales para algoritmos CHk^2 y CHk^2t para los distintos grafos web

Fig. 6.21: Análisis de espacio de almacenamiento de grafos web

- La forma más sencilla y a la vez menos eficiente de abordar un problema sobre una estructura de datos compacta es transformar la misma a una estructura regular.
- Si bien lo ideal es encontrar relaciones y propiedades que nos permitan trabajar de manera directa sobre una estructura de datos compacta, la interpretación controlada del mínimo de datos accesados necesario aparece como una estrategia totalmente válida y de una eficiencia mucho más que aceptable.
- Los conjuntos de datos con distribuciones uniformes afectan negativamente la aparición de falsos positivos, produciéndose en muchos casos una paridad respecto a la eficiencia tanto temporal como en espacio de almacenamiento entre los algoritmos CHk^2 y CHk^2t.
- Los conjuntos de datos con distribuciones no uniformes afectan positivamente la aparición de falsos positivos, potenciándose así el algoritmo CHk^2t por sobre el CHk^2 en estos escenarios más realistas.

- Para los datos sintéticos con distribución Gaussiana, en cuanto al desempeño temporal, el peor caso a través de diferentes tamaños de dimensión para el CHk^2 fue un aumento de 275,24 % en contraste a un aumento de 819,21 % presentado por el GSk^2, lo que supone una mejor adaptabilidad del algoritmo CHk^2 a través de distintos tamaños dimensionales.

- Para los datos sintéticos con distribución Uniforme, en cuanto al desempeño temporal, el peor caso a través de diferentes tamaños de dimensión para el CHk^2 fue un aumento de 489,57 % en contraste a un aumento de 180,84 % presentado por el GSk^2, lo que supone una mejor adaptabilidad del algoritmo GSk^2 a través de distintos tamaños dimensionales, sin embargo, para todos los casos CHk^2 es más rápido que GSk^2, respondiendo en el peor de los casos CHk^2 en un 0,0449 % del tiempo necesario por GSk^2.

- Para los datos sintéticos con distribución Gaussiana y Uniforme, en el peor de los casos las estructuras para el algoritmo CHk^2 sólo necesitaron un 10,185 % y un 14,421 % respectivamente del espacio de almacenamiento necesario por el GSk^2, lo que demuestra la gran eficiencia del algoritmo preservando la compactación de los datos.

- Para los datos reales, en el peor de los casos las estructuras para el algoritmo CHk^2 sólo necesitaron un 16,947 % del espacio de almacenamiento necesario por el GSk^2, diferenciándose en un pequeño porcentaje de los datos sintéticos, siendo aún muy eficiente en cuanto al espacio de almacenamiento.

- Para los datos reales, en el peor de los casos el tiempo de respuesta del CHk^2 sólo necesito un 31,446 % del tiempo necesario por el GSk^2. Esto demuestra que sobre todos los escenarios planteados, el CHk^2 es más eficiente por mucho sobre el GSk^2.

- La densidad y distribución de los conjuntos afecta directamente el desempeño temporal de los algoritmos, dándose una relación tipo: *a mayor área de representación para la misma cantidad de datos, los tiempos de respuesta comenzarán a estancarse como una función logarítmica acercándose a su cota superior.*

Parte IV

Conclusiones y trabajo futuro

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

La Geometría Computacional, a lo largo de su historia, ha demostrado empíricamente su utilidad a través de múltiples aplicaciones que se han dado a los problemas y soluciones planteadas en esta área. No es una conclusión esta observación respecto a la Geometría Computacional, sin embargo, es destacable lo influyente que puede llegar a ser este campo de estudio sobre otros relacionados. Por su parte, las estructuras de datos compactas cumplen lo que prometen, es decir, logran reducir el espacio de almacenamiento de manera considerablemente eficiente, aprovechando la jerarquía de memoria, y además presentándose en una forma tal que nos permite aprovechar propiedades de estructuras más complejas como un plus a lo que ya ofrecen de partida. En base a esto, es evidente la utilidad que presenta la mezcla de ambas disciplinas o campos de estudio, siempre y cuando pretendamos aprovechar el máximo posible de cada una sin interferir en la otra, buscando el equilibrio ideal entre ambas.

Nuestro objetivo general era el uso de estructuras de datos compactas para el cálculo de la cerradura convexa, lo cual fue cumplido a cabalidad con la propuesta e implementación de los algoritmos CHk^2 y CHk^2t, los que además demostraron ser altamente eficientes tanto a nivel de almacenamiento como a nivel de procesamiento, lo cual fue reflejado a través de la experimentación. Así mismo, consideramos que los objetivos específicos fueron todos cumplidos, y que es a lo largo de esta tesis que se puede ver el cumplimiento de los mismos. Por ejemplo, entre los objetivos específicos se plantea el estudio de las estructuras compactas, lo que es reflejado en el capítulo 3 donde presentamos en total 8 estructuras espacio-eficientes capaces de representar conjuntos de puntos y que para el k^2-tree, en el capítulo 4, es abordada en profundidad a través de propuestas claras y fundamentales para el desarrollo de esta tesis como lo son la obtención de las coordenadas de una hoja.

Según nuestra hipótesis planteamos que era posible desarrollar un algoritmo eficiente sobre una estructura de datos compacta que además se beneficiara de las propiedades de la misma estructura y que a la vez fuese más eficiente que tratar un algoritmo tradicional sobre una estructura de datos compacta para calcular la cerradura convexa de un conjunto de puntos. Fue a través de la experimentación que pudimos demostrar este punto, siendo además todos los algoritmos propuestos, implementados pensando en el máximo aprovechamiento de las propiedades

del k^2-tree, las que por su jerarquía benefició enormemente la búsqueda. Otro punto destacable fue que el k^2-tree nos permite saber en altos niveles del árbol si una área en específica contiene a lo menos un punto, lo que potenció significativamente la búsqueda de aristas para la cerradura convexa.

Afirmamos entonces, que los algoritmos geométricos sobre estructuras de datos compactas son un enfoque inteligente y audaz, que aprovecha lo mejor de ambos campos, para dar una solución eficaz, eficiente y altamente competitiva dentro de los límites auto-impuestos sobre esta tesis.

7.2. Trabajo futuro

La implementación de la cerradura convexa sobre una estructura compacta como lo es el k^2-tree es sólo el comienzo de lo que puede ser un campo de estudio muy basto en investigación tal como el de los algoritmos geométricos sobre estructuras de datos compactas. Aún queda mucho por explorar, plantear, implementar y descubrir con respecto a estas dos áreas de estudio combinadas, sin embargo creemos pertinente remitirnos a un futuro cercano en base a lo ya implementado en esta tesis. Por lo tanto planteamos los siguientes puntos como trabajo futuro:

- **Cerradura convexa sobre distintos tipos de k^2-tree:** lo que consideramos un punto de partida para la continuación del trabajo aquí presentado, buscando implementar algoritmos como los propuestos sobre k^2-trees que logren compactar 1's y también sobre k^2-tree dinámicos. Con respecto a los k^2-tree que compactan 1's (de manera similar a los quadtree) nos parece interesante el hecho de que la modificación de la estructura pueda afectar lo teórico de los algoritmos planteados, teniendo quizás que bajar el nivel de abstracción de los mismos para adaptarse a este escenario, sin embargo considerando el hecho de que una estructura más compacta aprovechará de mejor manera la jerarquía de memoria frente a conjuntos cada vez de mayor tamaño. Por otro lado, la implementación de la cerradura convexa sobre un k^2-tree dinámico nos permitiría trabajar sobre un conjunto de puntos variable en el tiempo, una ventaja totalmente deseable y que con lo implementado no podemos satisfacer en el presente inmediato.

- **Aislamiento de zonas en un k^2-tree:** el hecho de que los k^2-tree potencien su utilidad sobre los grafos web nos hace notar que estos conjuntos son del tipo clusterizados, por lo que sería de gran utilidad poder aislar distintas zonas dentro de la estructura, para así poder, con los algoritmos implementados, calcular la cerradura convexa de distintos subconjuntos pertenecientes a uno mayor, estando todos estos contenidos sobre la misma implementación de la estructura de datos compacta.

- **Algoritmos geométricos sobre estructuras de datos compactas**: en esta investigación sólo se implementa un algoritmo geométrico sobre una estructura de datos compactas, pero como se mencionó en los capítulos 2 y 3 existen otros algoritmos geométricos y también otras estructuras de datos compactas que sería interesante abordar. Si nos remitimos a lo más inmediato planteamos la implementación de la cerradura convexa sobre un wavelet tree y sobre un faster compressed quadtree, lo que además nos permitiría comparar el desempeño de una estructura por sobre otras en diferentes aspectos como el temporal y el espacio

de almacenamiento. Sin duda nos parece interesante, lo que además reforzaría la idea de la implementación de algoritmos geométricos sobre estructuras de datos espacio-eficientes.

- **Buscar relaciones matemáticas sobre las estructuras compactas que minimíce el uso de estructuras auxiliares**: lo que consideramos un trabajo más ambicioso y a la vez más complejo, que sin embargo potenciaría de manera considerable el desempeño de algoritmos de este tipo sobre una estructura de datos compactas.

Bibliografía

[1] Aci, M., İnan, C., Avci, M.: A hybrid classification method of k nearest neighbor, bayesian methods and genetic algorithm. Expert Systems with Applications 37(7), 5061–5067 (2010)

[2] Agarwal, P.K., Erickson, J., et al.: Geometric range searching and its relatives. Contemporary Mathematics 223, 1–56 (1999)

[3] Aleardi, L.C., Devillers, O., Schaeffer, G.: Succinct representations of planar maps. Theoretical Computer Science 408(2), 174–187 (2008)

[4] Bailey, T., Jain, A.: A note on distance-weighted k-nearest neighbor rules. IEEE Transactions on Systems, Man, and Cybernetics (4), 311–313 (1978)

[5] Barber, C.B., Dobkin, D.P., Huhdanpaa, H.: The quickhull algorithm for convex hulls. ACM Transactions on Mathematical Software (TOMS) 22(4), 469–483 (1996)

[6] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles, vol. 19. ACM (1990)

[7] Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)

[8] Bentley, J.L., Friedman, J.H.: Data structures for range searching. ACM Computing Surveys (CSUR) 11(4), 397–409 (1979)

[9] de Bernardo Roca, G.: New data structures and algorithms for the efficient management of large spatial datasets. Ph.D. thesis, Universidade da Coruña (2014)

[10] Bhatia, N., et al.: Survey of nearest neighbor techniques. arXiv preprint arXiv:1007.0085 (2010)

[11] Böhm, C., Kriegel, H.P.: Determining the convex hull in large multidimensional databases. In: Data Warehousing and Knowledge Discovery, pp. 294–306. Springer (2001)

[12] Brisaboa, N.R., Ladra, S., Navarro, G.: k2-trees for compact web graph representation. In: String Processing and Information Retrieval. pp. 18–30. Springer (2009)

[13] Brisaboa, N.R., Luaces, M.R., Navarro, G., Seco, D.: A fun application of compact data structures to indexing geographic data. In: Fun with Algorithms. pp. 77–88. Springer (2010)

[14] Brisaboa, N.R., Luaces, M.R., Navarro, G., Seco, D.: Space-efficient representations of rectangle datasets supporting orthogonal range querying. Information Systems 38(5), 635–655 (2013)

[15] Brisaboa, N.R., Luaces, M.R., Navarro, G., Seco, D.: Indexación espacial de puntos empleando wavelet trees. In: JISBD. pp. 225–236 (2009)

[16] Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM Journal on Computing 17(3), 427–462 (1988)

[17] Chen, J.: Computational geometry: Methods and applications. (1996)

[18] Claude, F.: Space-efficient data structures for information retrieval (2013)

[19] De Berg, M., Van Kreveld, M., Overmars, M., Schwarzkopf, O.C.: Computational geometry. Springer (2000)

[20] Durocher, S., El-Zein, H., Munro, J.I., Thankachan, S.V.: Low space data structures for geometric range mode query. Theoretical Computer Science 581, 97–101 (2015)

[21] Farina, A.: New compression codes for text databases. Ph.D. thesis, PhD thesis, Database Laboratory, University of A Coruna. http://coba. dc. fi. udc. es/˜ fari/phd (2005)

[22] Farzan, A., Gagie, T., Navarro, G.: Entropy-bounded representation of point grids. In: International Symposium on Algorithms and Computation. pp. 327–338. Springer (2010)

[23] Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms (TALG) 3(2), 20 (2007)

[24] Gagie, T., González-Nova, J.I., Ladra, S., Navarro, G., Seco, D.: Faster compressed quadtrees. In: 2015 Data Compression Conference. pp. 93–102. IEEE (2015)

[25] Graham, R.L.: An efficient algorith for determining the convex hull of a finite planar set. Information processing letters 1(4), 132–133 (1972)

[26] Graham, R.L., Yao, F.F.: Finding the convex hull of a simple polygon. Journal of Algorithms 4(4), 324–331 (1983)

[27] Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 841–850. Society for Industrial and Applied Mathematics (2003)

[28] Gu, Y., Boukerche, A.: Hd tree: A novel data structure to support multi-dimensional range query for p2p networks. Journal of Parallel and Distributed Computing 71(8), 1111–1124 (2011)

[29] Gupta, P., Janardan, R., Kumar, Y., Smid, M.: Data structures for range-aggregate extent queries. Proc. 20th CCCG pp. 7–10 (2008)

[30] Guttman, A.: R-trees: a dynamic index structure for spatial searching, vol. 14. ACM (1984)

[31] He, M.: Succinct and implicit data structures for computational geometry. In: Space-Efficient Data Structures, Streams, and Algorithms, pp. 216–235. Springer (2013)

[32] Jacobson, G.: Space-efficient static trees and graphs. In: Foundations of Computer Science, 1989., 30th Annual Symposium on. pp. 549–554. IEEE (1989)

[33] Katayama, N., Satoh, S.: The sr-tree: An index structure for high-dimensional nearest neighbor queries. In: ACM SIGMOD Record. vol. 26, pp. 369–380. ACM (1997)

[34] Liu, R., Fang, B., Tang, Y.Y., Wen, J., Qian, J.: A fast convex hull algorithm with maximum inscribed circle affine transformation. Neurocomputing 77(1), 212–221 (2012)

[35] Munro, J.I.: A multikey search problem. In: Proceedings of the 17th Allerton Conference on Communication, Control and Computing, University of Illinois. pp. 241–244 (1979)

[36] Munro, J.I.: Tables. In: Foundations of Software Technology and Theoretical Computer Science. pp. 37–42. Springer (1996)

[37] Munro, J.I., Suwanda, H.: Implicit data structures for fast search and update. Journal of Computer and System Sciences 21(2), 236–250 (1980)

[38] Navarro, G.: Wavelet trees for all. Journal of Discrete Algorithms 25, 2–20 (2014)

[39] Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: ALENEX. SIAM (2007)

[40] Ooi, B.C., McDonell, K.J., Sacks-Davis, R.: Spatial kd-tree: An indexing mechanism for spatial databases. In: IEEE COMPSAC. vol. 87, p. 85 (1987)

[41] o'Rourke, J.: Computational geometry in C. Cambridge university press (1998)

[42] Preparata, F.P., Hong, S.J.: Convex hulls of finite sets of points in two and three dimensions. Communications of the ACM 20(2), 87–93 (1977)

[43] Rahman, N., Raman, R., et al.: Engineering the louds succinct tree representation. In: Experimental Algorithms, pp. 134–145. Springer (2006)

[44] Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 233–242. Society for Industrial and Applied Mathematics (2002)

[45] Robinson, J.T.: The kdb-tree: a search structure for large multidimensional dynamic indexes. In: Proceedings of the 1981 ACM SIGMOD international conference on Management of data. pp. 10–18. ACM (1981)

[46] Ruppert, J.: A delaunay refinement algorithm for quality 2-dimensional mesh generation. Journal of algorithms 18(3), 548–585 (1995)

[47] Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms. pp. 134–149. Society for Industrial and Applied Mathematics (2010)

[48] Samet, H.: The quadtree and related hierarchical data structures. ACM Computing Surveys (CSUR) 16(2), 187–260 (1984)

[49] Sellis, T., Roussopoulos, N., Faloutsos, C.: The r+-tree: A dynamic index for multi-dimensional objects (1987)

[50] Shamos, M.I.: Computational Geometry. Ph.D. thesis, New Haven, CT, USA (1978), aAI7819047

[51] Sharif, M., Khan, S., Khan, S.J., Raza, M.: An algorithm to find convex hull based on binary tree. In: Multitopic Conference, 2009. INMIC 2009. IEEE 13th International. pp. 1–6. IEEE (2009)

[52] Shewchuk, J.R.: Delaunay refinement algorithms for triangular mesh generation. Computational geometry 22(1), 21–74 (2002)

[53] Toth, C.D., O'Rourke, J., Goodman, J.E.: Handbook of discrete and computational geometry. CRC press (2004)

[54] Wu, X., Kumar, V., Quinlan, J.R., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G.J., Ng, A., Liu, B., Philip, S.Y., et al.: Top 10 algorithms in data mining. Knowledge and Information Systems 14(1), 1–37 (2008)

[55] Yu, B., Bailey, T., Orlandic, R., Somavaram, J.: Kdb kd-tree: a compact kdb-tree structure for indexing multidimensional data. In: Information Technology: Coding and Computing [Computers and Communications], 2003. Proceedings. ITCC 2003. International Conference on. pp. 676–680. IEEE (2003)

[56] Zeid, I.: CAD/CAM theory and practice. McGraw-Hill Higher Education (1991)

[57] Ziviani, N., De Moura, E.S., Navarro, G., Baeza-Yates, R.: Compression: A key for next-generation text retrieval systems. Computer (11), 37–44 (2000)