Universidad de Concepción
Dirección de Postgrado
Facultad de Ingeniería, Programa de Magíster en Ciencias de la Computación

Representaciones Compactas de Matrices con Localidad Espacial

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

PATRICIO ALEJANDRO PINTO MEDINA
CONCEPCIÓN-CHILE
2016

Profesor Guía: Diego Seco Naveiras
Comisión: Gilberto Gutiérrez
Andrea Rodríguez
Dpto. de Ingeniería Informática y Ciencias de la Computación, Facultad de Ingeniería
Universidad de Concepción

Agradecimientos

Doy gracias primeramente a Dios, por estar conmigo en cada etapa de mi vida, dándome un camino a seguir, mostrándome que siempre está conmigo y permitiéndome finalizar este trabajo.

A mi profesor guía, Diego, le agradezco la ayuda que me brindó durante todo este trabajo, además del ánimo que me dio cuando me veía frustrado o desalentado.

Además quisiera agradecer a mi familia, que me apoya incondicionalmente frente a los desafíos que día a día enfrento, en particular a mis padres que, con su esfuerzo, me muestran un ejemplo a seguir.

Agradezco también a Nataly, que estuvo conmigo en todo este proceso y me apoyo de infinitas formas, viviendo esta etapa conmigo, asimilándola como suya.

Finalmente, agradezco a aquellos amigos que con una palabra de ánimo o un consejo contribuyeron, en parte, a lo que este documento significa.

Resumen

El objetivo de este trabajo de tesis es diseñar e implementar representaciones eficientes en espacio de matrices con localidad espacial, es decir, matrices cuyas celdas cercanas almacenan valores similares entre sí. Esto es de utilidad en muchas aplicaciones prácticas como por ejemplo, en el dominio de los sistemas de información geográfica, se necesitan almacenar y consultar mapas de temperatura, precipitaciones, elevación, etc. en donde los datos están relacionados y tienen localidad espacial. En este trabajo se explorarán las representaciones existentes para abordar dicho problema, las propiedades del dominio de datos abordado y, a partir de esto, se diseñarán nuevas estructuras, tomando como base varias técnicas conocidas en los dominios de bases de datos espaciales y estructuras de datos compactas, como son las curvas de llenado del espacio, árboles binarios de búsqueda en su versión comprimida, árboles sucintos, *Wavelet Trees* comprimidos y k^2*-trees*. Las operaciones que nuestras soluciones soportan son: responder consultas por el valor de una celda en particular y consultas por una subregión (con y sin rango de valores posibles restringido). Un dominio donde este tipo de consultas resulta interesante, es en los mapas climáticos, en los cuales se quiere conocer la temperatura de una de una coordenada en particular (consulta de acceso), realizar zoom sobre un mapa (consulta por una subregión) o conocer la temperatura de una zona que excede un umbral (consulta por una subregión con rango).

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

1.1 Motivación

Los Sistemas de Información Geográfica (SIG) permiten organizar, almacenar, manipular, analizar y modelar grandes cantidades de datos procedentes de un dominio real, que están vinculados a una referencia espacial. Estos sistemas permiten a los usuarios crear consultas interactivas de manera que puedan analizar información espacial, editar datos y mapas, y presentar los resultados de estas operaciones. También pueden ser usados para la investigación científica, la gestión de recursos, la planificación urbana, la sociología, entre otras áreas [1].

Existen dos formas de almacenar los datos [1] en un SIG: el modelo vectorial y el modelo de *raster*. En esta tesis nos centraremos en el segundo de ellos. Un *raster* es cualquier tipo de imagen digital representada como una grilla. En otras palabras, es una matriz de dos dimensiones con valores numéricos en sus celdas. El tamaño de esta matriz define la resolución de la imagen capturada: mientras más celdas representan una misma región, más nivel de detalle se tiene de la misma. Ejemplos comunes de sistemas de información geográfica donde es habitual usar el modelo de *raster*, son: mapas de temperatura, de altitud, de superficie, de población, etc. Habitualmente los *rasters* son almacenados en diferentes formatos: TIFF, GEO-TIFF, JPEG, BLOB, etc.

Una propiedad importante de los sistemas de información geográfica, y por tanto de los *rasters*, es que los valores, almacenados en estos sistemas, suelen cumplir con la primera ley de la geografía: "*Las cosas más próximas en el espacio, tienen una relación mayor que las distantes, aunque todas están relacionadas*" [2]. En otras palabras, los datos modelados como *rasters*, son matrices que tienen localidad espacial. Uno de los principales problemas de estos, es que son grandes volúmenes de información. Eso, sumado a que en algunos casos se tienen matrices con una alta

resolución y que se tomen muchas muestras en el tiempo de un determinado dominio, hace que se requiera mucho espacio para poder almacenarlos. Por ejemplo, un *raster* de información climática de Chile (superficie de 756.096 km^2) requiere de una grilla de alrededor de 870x870 celdas para una resolución de 1 km^2 por celda. Si aumentamos la resolución a 1 m^2 por celda, la grilla requiere de 27.500x27.500 celdas. Ahora, como es un mapa climático, sería razonable pensar en una razón de captura de una hora por mapa, lo cual supone aproximadamente 70 GB de datos generados al día. Entonces, se hace necesario reducir ese costo de almacenamiento y, para ello, se requieren estructuras eficientes en espacio para representar matrices con localidad espacial, que es el objetivo principal de este trabajo.

1.2 Hipótesis

La hipótesis de este trabajo es que es posible explotar la localidad espacial existente en matrices bidimensionales de diferentes dominios, como SIG, mediante el diseño de estructuras de datos compactas que requieran poco espacio y ofrezcan tiempos de consulta competitivos con el estado del arte.

1.3 Objetivos

1.3.1 Objetivo General

Diseñar e implementar estructuras de datos eficientes en espacio para representar matrices con localidad espacial, permitiendo hacer consultas de acceso, consultas por subregiones y por subregiones con rango restringido. Además, comparar y evaluar experimentalmente, con las estructuras existentes para el mismo propósito.

1.3.2 Objetivos Específicos

- Estudiar las estructuras existentes para el almacenamiento de *rasters*, además de estructuras de datos compactas y técnicas de mapeo que conserven la localidad espacial.

- Diseñar estructuras de datos compactas, a partir de los conceptos estudiados.

- Implementar las estructuras de datos propuestas y los algoritmos de consulta sobre las mismas.
- Comparar, en términos de espacio y tiempo de consulta, las soluciones propuestas con las soluciones existentes en el estado del arte.

Capítulo 2

Discusión Bibliográfica

Este apartado está destinado a los conceptos necesarios para el diseño de las soluciones que en esta tesis se proponen, además de algunos trabajos existentes que se enfocan en problemas similares.

2.1 Conceptos Previos

Este trabajo de tesis, combina distintos métodos y técnicas para generar estructuras de datos compactas para representar *rasters*. Es necesario, por tanto, tener una noción de aquellos métodos y técnicas que emplearemos en nuestras soluciones propuestas, los cuales son brevemente descritos en este apartado.

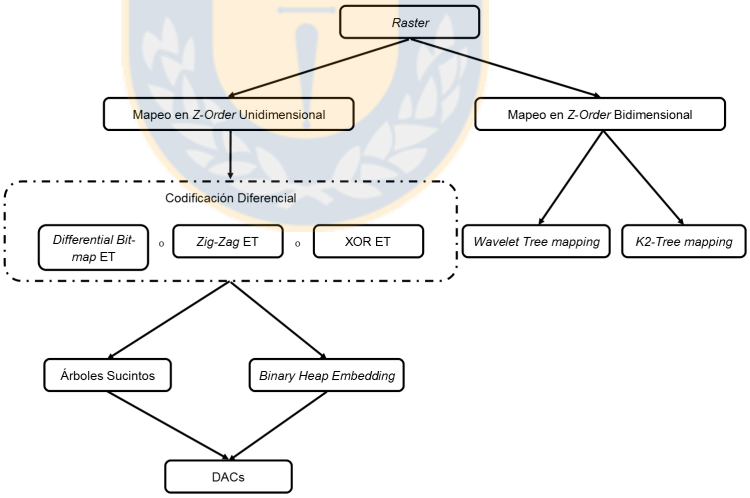


Figura 2.1: Diagrama de relación de conceptos del capítulo.

La figura 2.1, muestra cómo la mayoría de los temas de este capítulo se relacionan para crear representaciones eficientes en espacio de un *rasters*.

2.1.1 Modelos de Información Geográfica

Un sistema de información geográfica (SIG)[1] es un conjunto de herramientas que permiten trabajar con datos de un origen real que están vinculados espacialmente. Por ejemplo, mapas de temperatura, de altitud de terreno, de densidad de población, entre otros. Estos sistemas permiten, a usuarios de diversas áreas, realizar consultas interactivas, permitiéndoles analizar datos para luego ser presentados. Los datos de un SIG son almacenados usualmente bajo dos modelos: el modelo vectorial (usando figuras geométricas, polígonos, etc.) y el modelo matricial usando celdas con información de un punto del espacio representado (*rasters*). En la figura 2.2 se muestra un modelo de *raster* y uno de representación vectorial. Nuestro trabajo se centra en el modelo matricial, explicado a continuación.

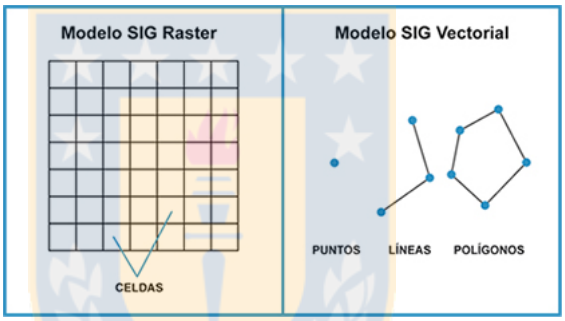
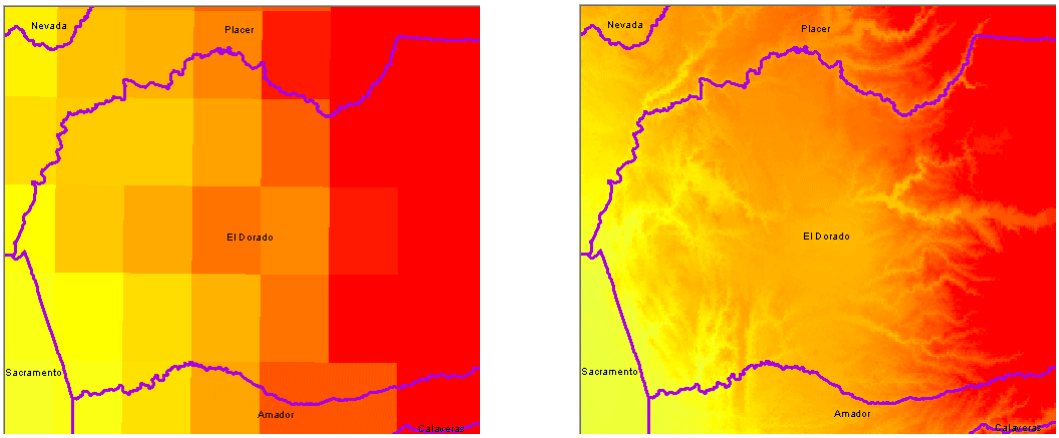


Figura 2.2: Modelo *Raster* y Vectorial. Fuente: Sistema de Información Geográfica[1]

2.1.1.1 *Raster*

Un *raster* [1] es una representación matricial de un espacio de datos, donde los puntos tienen una referencia geográfica. Tiene aplicaciones en conjuntos de datos que representan algún plano: un mapa de densidad de población, de temperatura, etc. Su estructura es básicamente una matriz de 2 dimensiones, donde el valor de cada celda representa una zona de los datos. Tomando como ejemplo un mapa de temperatura, cada posición en la matriz representa una zona de un determinado lugar y, el valor de la celda, la temperatura de dicha zona. El número de celdas destinadas a representar una misma localidad es llamada resolución. Mientras más celdas estén dedicadas a

[1]url: http://sistemainformaciongeografica.blogspot.cl/



(a) Baja resolución (b) Alta resolución

Figura 2.3: Dos *rasters* climáticos con diferentes resoluciones. Fuente: CA LCC[2]

representar una región, mejor resolución se tiene y, por lo tanto, una imagen de mejor calidad. Sin embargo, a mayor resolución, mayor es el uso de espacio, siendo este el problema principal de un *raster*. La figura 2.3 muestra 2 mapas climáticos que se almacenan bajo el modelo de *raster* con diferentes resoluciones.

2.1.2 *Z-Order Curve* / Curva de Orden Z

Una curva de relleno del espacio [14] es una función matemática que mapea datos de un espacio multidimensional a uno de una sola dimensión. La curva de orden Z, o *Morton-Code* [15], como también se la conoce, es una curva de relleno del espacio, con la propiedad de mantener la localidad de los datos. Es esta propiedad la que nuestro trabajo necesita y por ello esta curva es de nuestro interés. En la figura 2.4 se muestra con flechas el recorrido de una curva *Z-Order* de una matriz de 8×8, partiendo de la celda superior izquierda.

El valor *Z-Order* de un punto multidimensional (*Z-Order-Index*($\vec{x}$)), en el arreglo unidimensional, es calculado tomando la representación binaria de los índices en la estructura multidimensional. La operación es sencilla, se intercalan los bits de un índice con los bits del otro. Tomando como caso particular una matriz de 2 dimensiones, sean x e y las coordenadas de un punto en la matriz. Para conocer el índice i_{xy} del punto en el arreglo unidimensional, se toma el primer bit (menos significativo)

[2]url: http://climate.calcommons.org/article/about-raster-resolutions

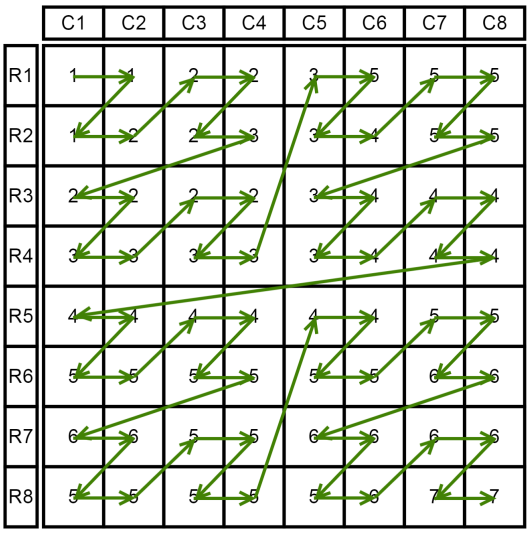


Figura 2.4: Recorrido en *Z-Order* de una matriz de 8×8.

de la representación binaria de y, que será el bit menos significativo de i_{xy}; el primer bit de x, se transformará en el segundo bit de i_{xy}; y así sucesivamente, hasta crear el código i_{xy}. En la figura 2.5 se muestra un ejemplo de este proceso.

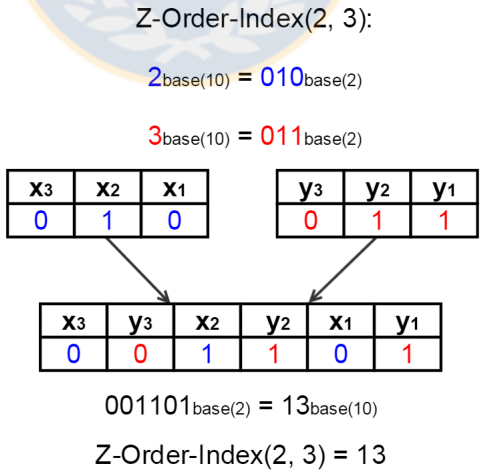


Figura 2.5: Ejemplo del cálculo *Z-Order-Index* para $i_{2,3}$.

2.1.3 Descomposición en *Quadboxes*

Una descomposición en *quadboxes* [16] [17] es la división de una región rectangular en bloques cuadrados de lado potencia de 2. En nuestro dominio, este procedimiento tiene la particularidad de coincidir con las "Z"s del recorrido en *Z-Order*. La figura 2.6 muestra la descomposición en *quadboxes* de una región rectangular, donde se marca cuáles son las "Z" del recorrido correspondientes. Esta técnica suele utilizarse en bases de datos espaciales basadas en *quadtrees*.

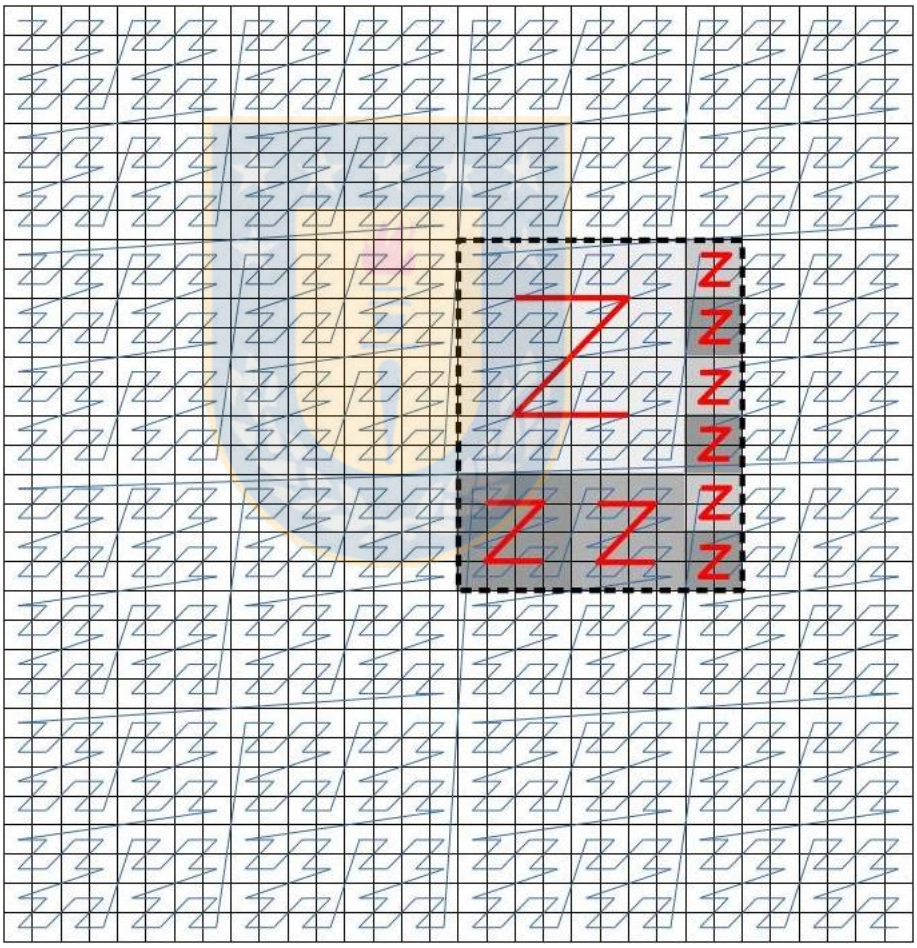


Figura 2.6: Descomposición en *quadboxes* de una región rectangular. Las "Z"s, coinciden con los *quadboxes*.

2.1.4 *Diferencial Encoding Search Tree* (DEST)

DEST [3] es una estructura de datos eficiente en espacio para representar secuencias monótonas de números que usa tiempo logarítmico para el acceso a los datos. La estructura toma como base un arreglo de enteros ordenados no decreciente y lo transforma en un árbol binario. La transformación se hace tomando el valor almacenado en la posición central del arreglo como la raíz del árbol, el rango izquierdo como subárbol izquierdo y el derecho como subárbol derecho. Se aplica el mismo principio recursivamente para cada subárbol hasta llegar a las hojas. En la figura 2.7 la parte superior muestra un arreglo no decreciente de enteros el cual es descompuesto en 3 regiones vistas en la parte central de la figura: el rango izquierdo, el centro y el rango derecho, que son las secciones que conforman el subárbol izquierdo, la raíz y el subárbol derecho respectivamente, de la representación del arreglo como árbol binario, visto en la parte inferior de la figura.

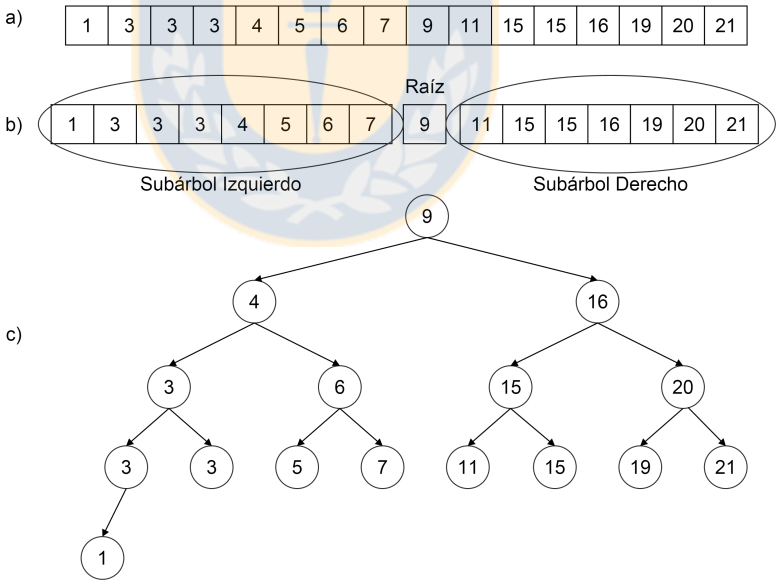


Figura 2.7: Representación de un arreglo como árbol binario: a) Arreglo de valores original; b) Primera etapa de construcción del árbol; c) Árbol representante del arreglo en a).

2.1.4.1 Árbol de Diferencias

Posteriormente se calcula un árbol de diferencias, que es un árbol semejante al árbol binario de la etapa anterior. Los nodos de este tienen las diferencias de los valores del árbol original, esto es, la raíz tiene el mismo valor que en el árbol original y cada hijo izquierdo v'_i tiene el valor de la diferencia entre el padre v_p y el hijo izquierdo v_i del árbol original, esto es $v'_i = v_p - v_i$. De manera similar, cada hijo derecho v'_d tiene la diferencia entre el hijo derecho v_d y el padre v_p del árbol original, esto es $v'_d = v_d - v_p$. De esta forma este árbol contiene valores más pequeños que el árbol original. En la figura 2.8 se muestra el árbol de diferencias T' generado en base al árbol T de la figura 2.7 c), en donde sobre cada nodo se muestra la operación que genera el valor del nodo en este nuevo árbol.

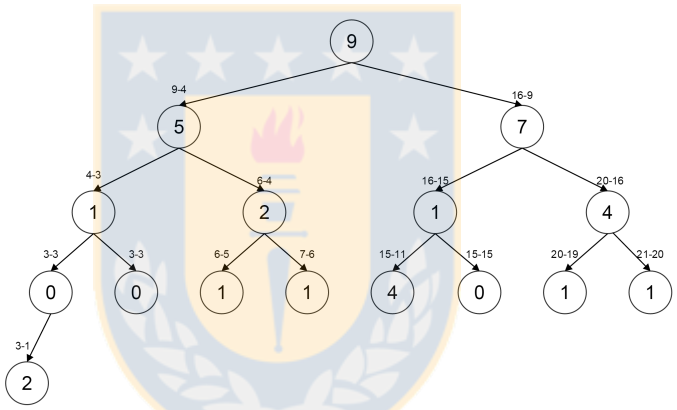


Figura 2.8: Árbol de diferencias del árbol mostrado en la figura 2.9 c).

2.1.4.2 *Binary Heap Embedding*

El árbol descrito anteriormente es almacenado en un arreglo, pues la representación clásica de árboles (con punteros) es muy costosa en términos de memoria. La raíz del árbol se almacena en la posición 1 y los hijos izquierdo y derecho de un padre en la posición i del arreglo se almacenan en la posición $2i$ y $2i + 1$, respectivamente. De esta manera el árbol se almacena por niveles en el arreglo. Este almacenamiento de un árbol en un arreglo se conoce como *Binary Heap Embedding*. En la figura 2.9 se muestra un arreglo que representa el *Binary Heap Embedding* asociado al árbol

de diferencias que aparece en la figura 2.8. Nótese que para que la representación funcione, el primer índice del arreglo debe ser 1.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
9	5	7	1	2	1	4	0	0	1	1	4	0	1	1	2

Figura 2.9: Árbol de diferencias de la figura 2.11, en su representación como arreglo.

El nuevo arreglo de diferencias es codificado con *DACs: Direct Access Codes* [19], un tipo de algoritmo de compresión de longitud variable, que permite acceso directo a cualquier posición en tiempo constante en la práctica (ver sección 2.1.6.2). La ventaja de usar esta estructura para nuestro trabajo recae en el hecho de que el arreglo, por tener localidad espacial y al ser este creado a partir de una matriz que cumple con que las posiciones próximas tienen valores próximos, los valores son muy similares, de manera que sus diferencias son valores muy pequeños que se codificar más eficientemente que valores grandes. Cabe mencionar que la compresión de estos árboles es más eficiente en las hojas (que son la mitad de los datos).

2.1.5 *ZigZag Encoding*

La representación estándar de números enteros en binario consiste en transformar el número a base 2 usando k bits si es positivo y, si es negativo, restarle al valor máximo representable con k bits, el valor absoluto del número que se quiere representar menos uno. Este modelo permite representar valores entre $[-2^{k-1}, 2^{k-1}[$. Esto implica que, para representar números negativos, se utilizan muchos bits significativos. Para solucionar este problema existe una técnica llamada *ZigZag Encoding* [20] que consiste en que, a cada valor n, se le representará con $2n$ si es positivo y los valores negativos con $2n-1$. La operación es equivalente a $n = (n << 1) XOR (n >> w)$, donde $>>$ es un corrimiento de bits a la derecha, $<<$ es un corrimiento a la izquierda y w representa el número de bits que se usa para representar un entero (usualmente 32). Por ejemplo, el valor 25 es mapeado a $2 \cdot |25| = 50$; el valor -31 es mapeado a $2 \cdot |-31| - 1 = 61$.

2.1.6 Compresión de Enteros Pequeños

Para la compresión de enteros pequeños destacan para nuestro trabajo los *DACs* [19]. *DACs* es un tipo de codificación donde la longitud de la representación binaria de los datos es variable, de esta forma, los valores pequeños usan menos bits para ser representados que en una tradicional de longitud fija. *DACs* está basado en la idea principal de los *VBytes* [18] pero tiene la ventaja de brindar acceso directo a datos, es decir, permite acceder a una posición cualquiera del arreglo comprimido sin tener que descomprimir la secuencia de valores previa (lo cual no es posible en los *VBytes*).

2.1.6.1 *VBytes*

La representación usual de datos en una máquina es usar una determinada cantidad fija w de bits (usualmente 32 ó 64) para almacenar el valor de un dato en base binaria. Si fuese un arreglo de n valores, se necesitan para su almacenamiento $n \cdot w$ bits. Esto es demasiado cuando se tienen pocos recursos o n es muy grande.

Una técnica para abordar el problema son los *VBytes* [18]. Su idea es, dado un valor v, tomar su representación en binario y descomponerla en bloques de tamaño $b+1$ bits, donde en el primer bloque se almacenan los primeros b bits menos significativos de v y un bit indicador en 1, si el número continúa, o en 0, si no. Usualmente, los *VBytes* usan un tamaño de bloque de 1 byte. *Vbytes* comprime bien, pero no soportan acceso aleatorio a datos, es decir, encontrar un valor específico requiere buscar dónde termina cada valor antes de él. En la figura 2.10 aparece un ejemplo en el cual se comprime un arreglo de tamaño 4 usando *VBytes*. Primero cada valor es separado en bloques de 4 bits (en el ejemplo se usan bloques de tamaño 4 + 1 bits). Luego cada bloque es reordenado para formar la secuencia que representa a un valor. Tomando como base el valor 434 del ejemplo, se toma el bloque con sus bits menos significativos (en verde), se le agrega un bit en 1 y pasan a ser la primera parte de la secuencia. Luego el segundo bloque menos significativo (en azul) y se agrega un bit en 1 pues la secuencia continua. Finalmente se agrega el tercer bloque menos significativo (y el último necesario para representar el dato 434) y se agrega un 0 que indica que la secuencia termina ahí. En el caso del dato 15 (en amarillo), sólo se requiere un bloque para ser representado, es decir, su secuencia es conformada por los

primeros 4 bits menos significativos y un 0. Las secuencias (*VBytes*) son almacenadas consecutivamente, como se ve en la parte inferior de la figura.

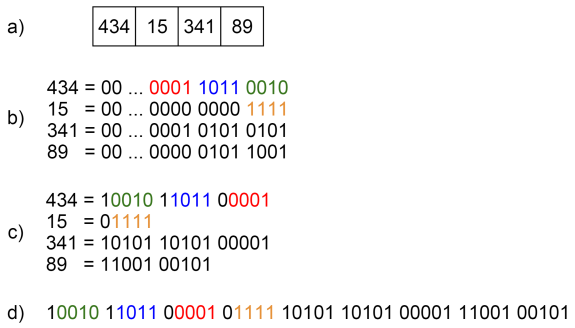


Figura 2.10: Representación de datos usando *VByte* con tamaño de bloque 4+1: a) Arreglo de datos; b) Representación normal de los datos en binario; c) *VByte* de cada dato; d) Nuevo arreglo comprimido.

2.1.6.2 *DACs: Direct Access Codes*

DACs [19] tiene como propósito abordar el problema de acceso aleatorio a los datos. La primera etapa de *DACs* es representar los datos como *VBytes* pero con tamaño de bloque no necesariamente de 1 byte (el tamaño de bloque se calcula con programación dinámica, para determinar el largo óptimo). Luego, el almacenamiento de los datos es por niveles: si el arreglo es de tamaño n se almacenan los primeros bloques de los n elementos del arreglo de forma consecutiva. Posteriormente, se almacenan los segundos bloques de los elementos del arreglo que tengan un segundo bloque, y así sucesivamente. El bit indicador se almacena en un bitmap *bit_dacs* independiente de los datos, en una estructura que soporta operaciones de *rank* (sección 2.1.7.1) y *select* (sección 2.1.7.2). En la figura 2.11 se muestra la compresión usando *DACs*. Tomando como base el dato 341 que aparece en el ejemplo, este es separado en bloques y se le agrega el bit indicador (tal como *VBytes*), pero cada bloque es almacenado en un nivel: en el tercer bloque del nivel L_1 está el primer bloque menos significativo del dato 341, en el segundo bloque de L_2 está almacenado el segundo bloque menos significativo y en el segundo bloque de L_3 está almacenado el tercer bloque menos significativo. Sobre los bloques de cada nivel están los bits indicadores, estos son

almacenados en un bitmap aparte.

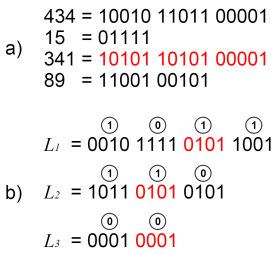


Figura 2.11: Representación de datos usando *DACs*: a) *VBytes* de los datos de la figura 2.13; b) Representación por niveles de *DACs*. Los círculos representan los valores en el bitmap *bit_dacs*.

Finalmente, para el acceso a los datos, dada una posición i, en la posición i del arreglo de *VBytes*, está el *VByte* menos significativo del elemento en la posición i del arreglo original. Además, revisando el bitmap *bit_dacs* en la posición i, podemos saber si el valor continúa o no. Si lo hace, con una operación de $rank_0$ podemos saber cuántos elementos terminaron en el primer nivel y así, en el segundo nivel, la posición del elemento será $rank_1(i) - 1$ del segundo nivel del bitmap y así sucesivamente hasta tener el valor completo del número en la posición i. Con esto, se puede acceder en tiempo $\lceil log_2(M)/b + 1 \rceil$ en el peor caso, donde M es el largo en bits del valor más grande del arreglo. En la práctica, el tiempo de acceso se considera constante.

2.1.7 Operaciones sobre Bitmaps

Existen estructuras de datos sucintas [7] (estructuras comprimidas que requieren espacio $Z + o(Z)$, con Z el mínimo teórico para representar la información, y no necesitan ser descomprimidas para el acceso a datos) que suelen reducirse a operaciones sobre bitmaps [23].

Un bitmap es una estructura de datos que se asemeja a un arreglo pero que sus valores son 0 ó 1. Es decir, cada celda es un bit. Existen implementaciones que usan $n + o(n)$ de espacio y pueden resolver las operaciones básicas sobre bitmaps (*rank, select y access*) en tiempo constante $O(1)$ [23]. Para las siguientes definiciones, se usará como ejemplo el Bitmap $B = [0, 0, 1, 1, 0]$ (el primer índice es 0).

2.1.7.1 *rank*

La operación *rank* devuelve cuántos 0s ó 1s hay en un bitmap hasta una determinada posición. En términos formales, $rank_q(T, x)$ devuelve cuántos valores iguales a q hay en la secuencia T hasta la posición x. Por ejemplo, $rank_0(B, 2) = 2$, $rank_1(B, 3) = 2$.

2.1.7.2 *select*

Esta operación determina cuál es la posición del i-ésimo 1 ó 0 en el bitmap. Se define como $select_p(T, i)$ la posición del i-ésimo valor p en la secuencia T. Por ejemplo, $select_0(B, 1) = 0$, $select_1(B, 1) = 2$.

2.1.7.3 *access*

Determina el valor del bitmap en una posición, esto es $access(T, i)$ el valor de la secuencia T en la posición i. En el ejemplo, $access(B, 0) = 0$, $access(B, 3) = 1$.

2.1.8 Árboles Sucintos

Un árbol normal para su almacenamiento requiere del espacio para los nodos (datos) y las aristas (conexiones). Por cada nodo, hay un espacio para los datos y un conjunto de punteros a sus nodos hijos. Esta representación es muy costosa pues en dominios donde los datos son un problema se requieren además $n - 1$ punteros para cada una de las aristas. En la sección 2.1.4.2 se presentó una alternativa para representar un árbol binario completo, sin la necesidad de almacenar los punteros. Para el caso de un árbol cualquiera, se presentan los árboles sucintos [12]. Estas estructuras necesitan, aparte del espacio para almacenar los nodos, $2n + o(n)$ bits adicionales para ser representados, permitiendo operaciones de navegación en tiempo constante. Una representación sucinta muy conocida de árboles se basa en la idea de paréntesis balanceados. El principio es simple, por cada nodo existirá un par de paréntesis, se recorre el árbol en profundidad: cada vez que se llegue a un nodo se escribe un '(', se recorren sus hijos, y luego se escribe un ')'. Expresado en términos algebraicos, la representación R de un nodo v y sus hijos $v_1, v_2, ..., v_k$ es $R(v) = (R(v_1)R(v_2)...R(v_k))$. Nótese que los paréntesis, en la práctica, son representados por 1s y 0s, por lo que

la secuencia que representa el árbol, no es más que un bitmap. Además, mediante operaciones de *rank/select* sobre el bitmap se soportan operaciones de navegación habituales en árboles (hijo, padre, número de descendientes, etc.). En la figura 2.12 se muestra un árbol balanceado y la representación con paréntesis balanceados descrita anteriormente.

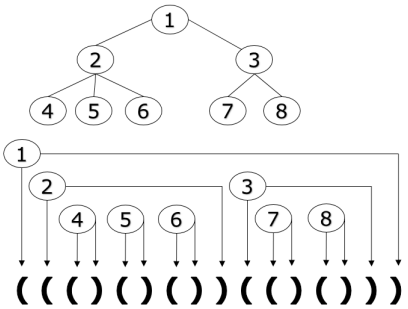


Figura 2.12: Árbol sucinto y una representación con paréntesis balanceados.

2.1.9 *Wavelet Trees*

Un *Wavelet Tree* [8][9] es una estructura de datos sucinta con forma de árbol desarrollada para almacenar secuencias. Su mecanismo es sencillo, se considera una secuencia de símbolos junto a un alfabeto con los símbolos posibles dentro de la secuencia. Luego, se toma la secuencia original como nodo inicial y se divide el alfabeto en 2. Cada símbolo es etiquetado con un bit que estará en 0 si pertenece a la primera mitad del alfabeto o en 1 si pertenece a la segunda. Luego todos los símbolos etiquetados con 0 pasan a formar una nueva secuencia correspondiente al hijo izquierdo del nodo y los con 1 al hijo derecho y se repite el procedimiento en los nuevos nodos hasta tener alfabetos con un solo símbolo. Finalmente se almacenan las secuencias de bits y el alfabeto original.

Esta estructura requiere soporte para las operaciones de *rank* y *select* comentadas anteriormente para tener acceso eficiente a los datos.

La figura 2.13 es la representación gráfica de un *wavelet tree* donde en cada nodo hay una secuencia y un bitmap.

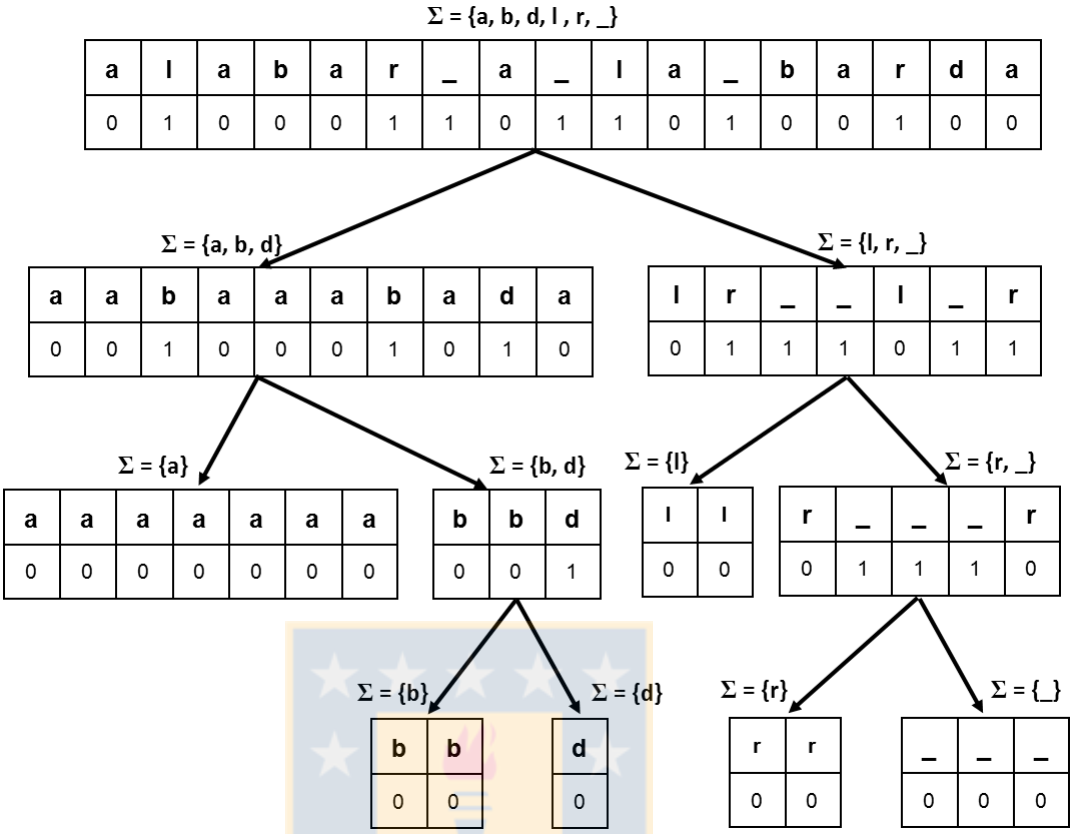


Figura 2.13: *Wavelet Tree* de una secuencia de caracteres.

2.2 Trabajo Relacionado

En [4] se describen los *Compact Querieable Representation of Rasters*, siendo el trabajo que consigue mayor compresión y mejores tiempos de consulta en el estado del arte. Sin embargo, este trabajo, no considera explícitamente la localidad espacial existente en los datos a comprimir, de manera que no aprovechan esta propiedad en la compresión, la cual puede ser aprovechada por una estructura como la nuestra que sí toma ventaja de la localidad espacial. Una falencia en el trabajo del estado del arte, es que se basa en el número de elementos diferentes dentro del *Raster* a comprimir de manera que si el *dataset* con el que se trabaja tuviera muchos elementos diferentes, el comportamiento en espacio de la estructura se vería afectado. Nuestras estructuras, en contraste, tienen poca dependencia de dicha característica. A continuación se describe el trabajo propuesto en [4].

2.2.1 *Compact Querieable Representation of Rasters*

Son representaciones eficientes de *rasters* que utilizan de varias formas el k^2-*tree* (ver 2.2.1.1) para el uso eficiente de memoria. La primera versión que proponen en [4], es una representación en la que se utiliza un k^2-*tree* para cada valor distinto en el *raster*, esta versión soporta consultas de acceso cuyo tiempo depende de un número constante de consultas a un k^2-*tree*, pero por la naturaleza de la estructura, las consultas de rango son más costosas. La segunda usa un k^2-*tree* para cada valor del *raster*, donde el k^2-*tree* representa la celda cuyos valores son menores o iguales al valor representado por tal k^2-*tree*. Esta segunda versión, soporta consultas de rango en tiempo dependiente de consultar a lo más 2 k^2-*tree* y consultas de acceso en tiempo razonable. La última versión que proponen utiliza un k^3-*tree*, que sería una versión multidimensional de un k^2-*tree*, donde una tupla $< x, y, z >$ representa la coordenada (x, y) y el valor z. En general, estas estructuras tienen buen comportamiento en cuanto a compresión, manteniendo buenos tiempos de consulta. Sin embargo, todas las versiones dependen del número de valores diferentes dentro del *raster*. Nuestras estructuras, en cambio, no depende de esta propiedad.

2.2.1.1 k^2-*tree*

Un k^2-*tree* [6] es una representación compacta de matrices esparsas binarias, propuesto originalmente para la compresión del grafo de la Web. Se basa en la partición recursiva de la matriz.

La idea es que, en cada partición, la matriz se divide en k^2 submatrices y se repite el proceso hasta tener una matriz de grado menor que k. Para representar cada submatriz, se usa un sólo bit: en 1 si la matriz tiene algún 1 y 0 en otro caso. Estas particiones son representadas como un árbol, donde cada partición de la matriz es un nodo hijo de la partición anterior. Este árbol puede ser representado con 2 bitmaps, uno que representa todos los niveles menos el último y otro que representa el último nivel. Esto se debe a que el último nivel no necesita soporte para la operación *rank* (los niveles anteriores requieren de esta operación para identificar dónde están los hijos de un nodo) como los niveles anteriores. Nótese que el árbol no almacena submatrices que sólo contienen 0s. En la figura 2.14 aparece una matriz esparsa binaria y su

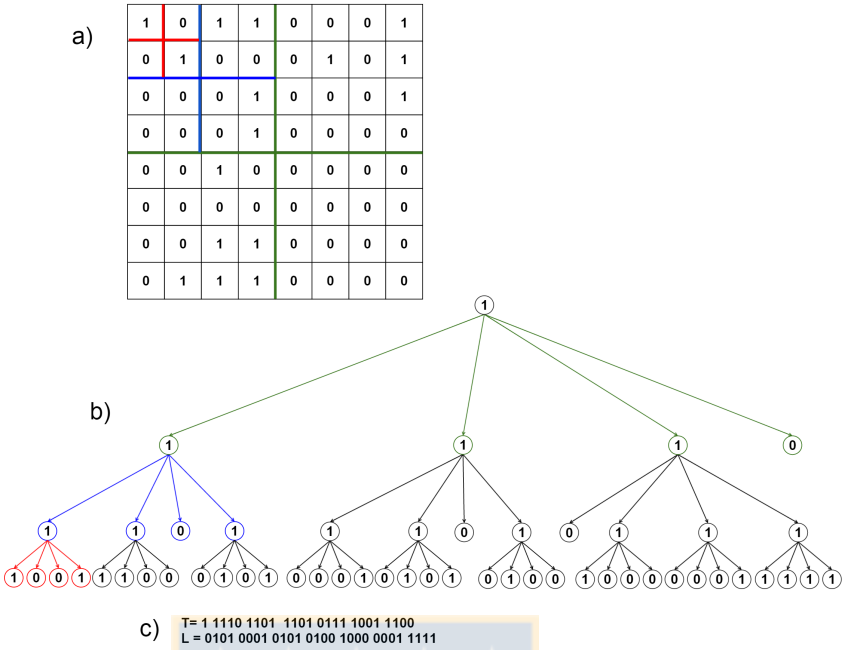


Figura 2.14: Representación de un k^2-*tree*: a) Matriz de 1s y 0s; b) Árbol que representa al k^2-*tree*; c) Representación como bitmap.

representación como k^2-*tree*. Las líneas en verde muestran la primera subdivisión de la matriz que forman las ramas de la raíz. En esta división, el cuadrante inferior derecho solo tiene 0s, de manera que el hijo de más a la derecha de la raíz no tiene hijos. Las líneas azules muestran la subdivisión del cuadrante superior izquierdo resultante de la subdivisión anterior, donde el cuadrante inferior derecho no tiene 1s, de manera que el tercer hijo del hijo izquierdo de la raíz no tiene hijos.

2.2.2 Mejora reciente sobre el estado del arte

Durante la fase final de la escritura de esta tesis se publicó un trabajo [5] que sigue un enfoque alternativo al nuestro para tratar de mejorar la eficiencia de la solución descrita anteriormente cuando el *raster* tiene muchos valores diferentes. Debido a los plazos del magíster no se incluye una evaluación experimental completa de dicha estructura, pero sí algunos resultados previos que pueden guiar una comparación futura.

La estructura propuesta en dicho trabajo, denominada k^2-*raster*, se puede describir de manera resumida empleando los conceptos y técnicas descritas en la sección anterior. La primera parte de la estructura es una variante del k^2-*tree* que realiza el mismo tipo de particionamiento pero que se detiene al llegar a submatrices del *raster* en las cuales todas las celdas contienen el mismo valor. Es decir, los 1s en esta representación significan que la submatriz correspondiente contiene 2 o más valores diferentes, mientras que los 0s significan que toda la submatriz contiene el mismo valor (no necesariamente todos los valores tienen que ser 0 como el k^2-*tree* original). En paralelo a esta estructura se guarda un árbol de valores mín/máx que almacena, por cada submatriz el valor mínimo y máximo de dicha submatriz en el *raster*. Dichos valores no se almacenan en plano sino que se realiza una codificación diferencial similar a la del DEST (también empleando DACs para almacenar las diferencias). Esta representación soporta los tres tipos de consulta que se describen en el Capítulo 3 (*access*, *windowQuery* y *rangeQuery*). Los algoritmos que soportan dichas consultas se basan en recorridos de la raíz a las hojas tanto en $k^2 - tree$ como en el árbol de diferencias min/max.

La evaluación experimental realizada por los autores muestra un comportamiento similar al estado del arte cuando el *raster* tiene pocos valores, pero una mejora drástica cuando el número de valores aumenta. Las mejoras se producen tanto en espacio de almacenamiento como en tiempo de consulta.

Capítulo 3

Soluciones Desarrolladas

El problema se abordó desde varios ángulos y fueron varias las alternativas de solución. Estas se pueden agrupar y jerarquizar. Transversal a todas las alternativas consideradas, se definen 3 tipos de consultas principales enfocadas en el dominio para el cual nuestra solución fue desarrollada. Estas son:

- $access(x, y)$ - Consultas de acceso: Es una consulta sobre una posición en particular de la matriz. Toma como parámetro la celda (x, y) en la matriz - *raster* original y devuelve el valor correspondiente a esa posición.

- $windowQuery(x1, y1, x2, y2)$ - Consulta por una subregión: dadas dos celdas en la matriz $c1 = (x1, y1)$ y $c2 = (x2, y2)$, esta consulta devuelve todos los valores comprendidos en una matriz de tamaño $(x2 - x1) \times (y2 - y1)$ formada por un rectángulo generado por $c1$ y $c2$ como puntos diagonales.

- $rangeQuery(x1, y1, x2, y2, min, max)$ - Consulta por una subregión con rango: esta consulta es similar a la consulta anterior, $windowQuery(x1, y1, x2, y2)$, pero limitando los resultados a valores que sean mayores o iguales que min y menores o iguales que max.

3.1 Estrategia 1: Mapeo de 2 Dimensiones a 1 Dimensión

Esta alternativa de mapeo busca convertir el *raster* de 2 dimensiones en un arreglo de datos unidimesional. Cada valor z en una posición (x, y) es almacenado en una posición $p_{x,y}$ de un nuevo arreglo. Para lograr esto, se usa la curva *Z-Order*.

Podemos dividir esta etapa en 4 pasos:

- Paso 1: mapear de 2D a 1D usando curva *Z-Order*.

- Paso 2: reducir la magnitud de los valores usando codificación diferencial, creando un árbol binario de diferencias (DEST).
- Paso 3: representar la topologia del árbol eficientemente.
- Paso 4: codificar los enteros pequeños en los nodos del árbol eficientemente.

3.1.1 Paso 1: Mapeo en *Z-Order*.

La transformación *Z-Order* devuelve una secuencia de índices asociada a una matriz cuadrada cuyo lado es potencia de 2. Si la matriz M de entrada no cumple este requisito, se asume una nueva matriz ampliada N que tiene 0s a la derecha y hacia abajo hasta completar la potencia de 2 más cercana. La potencia de 2 más cercana queda definida por: $2^{\lceil log_2(max(n,m))\rceil}$, donde n es el número de filas, y m el de columnas de la matriz M. En la figura 3.1 se muestra una matriz de 3×3 ampliada a su potencia de 2 más cercana.

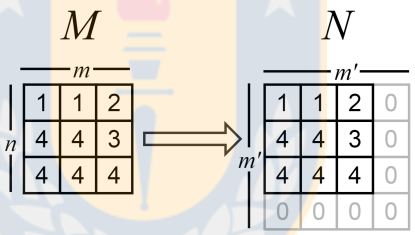


Figura 3.1: Matriz transformada a potencia de 2. $m' = 2^{\lceil log_2(max(3,3))\rceil} = 4$.

Una vez ampliada la matriz, esta es convertida en un arreglo unidimensional aplicando la transformación *Z-Order*. Llamaremos a este arreglo *z-order-array*. La figura 3.2 muestra un ejemplo de este procedimiento con base en la matriz ampliada de la figura 3.1.

3.1.2 Paso 2: Reducción de Magnitud

El arreglo *z-order-array* es convertido en un *binary tree* con codificación diferencial con el fin de disminuir la magnitud de sus valores. Esto se hace tomando el centro del arreglo como raíz y sus regiones izquierda y derecha como subárboles, de la misma forma que en DEST, descrito en la sección 2.1.4. Dado que se utilizan matrices

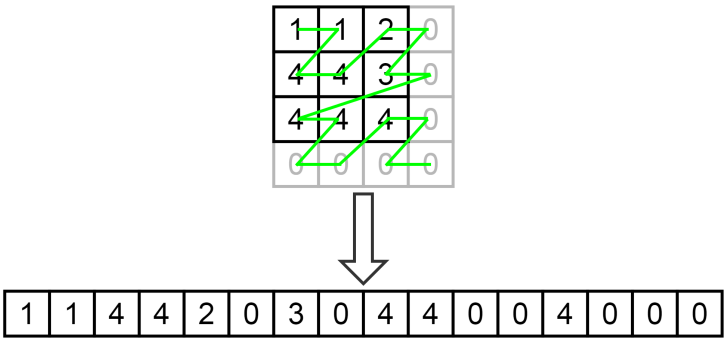


Figura 3.2: Transformación *Z-Order*, sobre la matriz transformada de la figura 3.1

cuadradas de lado potencia de 2, la altura del árbol, queda definida por $h = \lceil log_2(n+1) \rceil$, donde n es el número de elementos del arreglo. La figura 3.3 muestra la forma

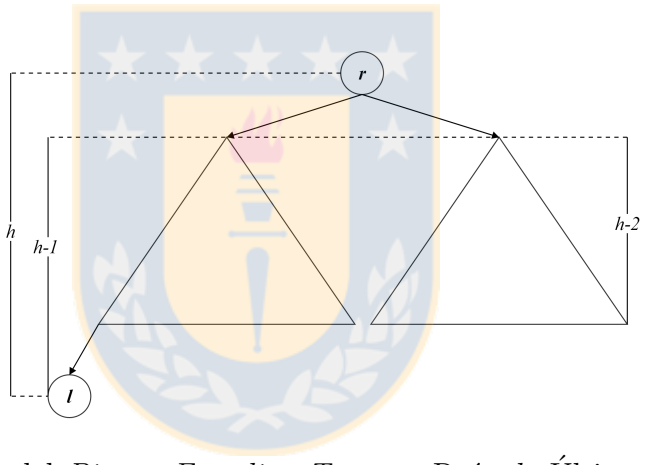


Figura 3.3: Forma del *Binary Encoding Tree*: r: Raíz, l: Último nodo del subárbol izquierdo, h: Altura del árbol.

descrita.

Como el número de elementos es $n = 2^k$, la estructura del árbol son dos subárboles, donde el izquierdo tiene un nodo más que el derecho, es decir, el subárbol izquierdo tiene 2^{k-1} elementos y el derecho $2^{k-1} - 1$ elementos. La raíz del árbol, es el nodo en la posición $n - 2^{h-2} + 1$. La raíz de cada subárbol se determina de la siguiente forma:

$2^{h'-1}$, si $n = 2^{h'} - 1$ o si $n' \geq 3 \times 2^{h'-2}$

$n - 2^{h'-2} + 1$, si $n < 3 \times 2^{h'-2}$

donde n' es el número de nodos en el subárbol y h' es la altura del mismo. La figura 3.4 muestra cómo un arreglo es descompuesto en un rango izquierdo, un centro y un rango derecho, que representan el subárbol izquierdo, la raíz y el subárbol derecho de la representación del arreglo como árbol.

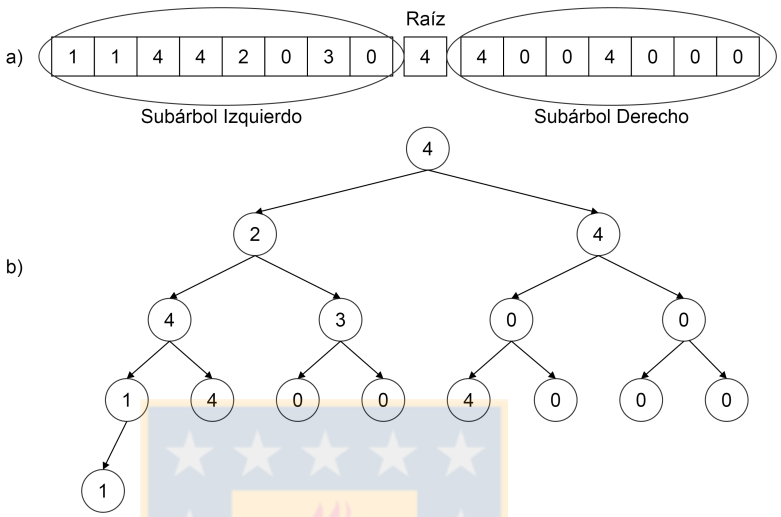


Figura 3.4: Representación del arreglo como árbol binario.

Hasta este punto el procedimiento es el mismo que para el árbol usado en DEST. Sin embargo, a diferencia de DEST, dado que el arreglo no tiene únicamente valores no decrecientes, al calcular las diferencias en el árbol pueden aparecer valores negativos, los cuales hacen que la compresión mediante codificación diferencial no sea posible. Para lidiar con este problema se estudiaron 3 alternativas de solución, las cuales fueron probadas experimentalmente y cuyos resultados están expuestos en el capítulo de experimentación.

3.1.2.1 *XOR ET: XOR Encoding Tree*

Al usar la operación XOR entre dos números enteros similares (cuya distancia es pequeña) el resultado tiende a ser más pequeño que los valores originales. Esto ya que los bits más significativos de los valores deberían ser los mismos. Referente a este caso, la primera alternativa que proponemos es usar la operación binaria XOR en lugar de usar la diferencia para reducir los valores de los nodos. Para ello, dado el árbol T original, un nodo v'_h en T', el árbol operado, tendrá el valor de la operación

XOR entre el nodo v_h y su padre v_p, ambos nodos del árbol T. Esto es $v'_h = v_h XOR v_p$. Nótese que el nodo v'_h en T' es el nodo que está en la misma posición que v_h en T. Lo mismo para v'_p con v_p en T' y T respectivamente. En la figura 3.5 se muestra el árbol resultante de realizar la operación XOR sobre el árbol de la figura 3.4. Sobre cada nodo está la operación que genera el nuevo valor. Por ejemplo, el nodo hijo izquierdo

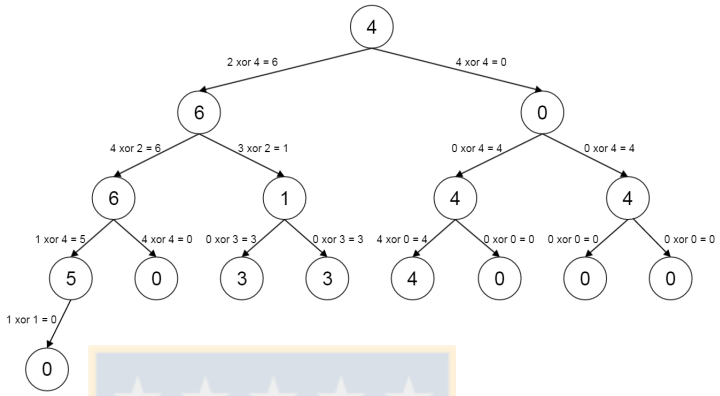


Figura 3.5: *XOR ET*: Árbol operado XOR, resultante del ejemplo en la figura 3.4

de la raíz, tiene el valor 6, resultado de operar con XOR el valor en el hijo izquierdo de la raíz del árbol original (2) y el padre de ese nodo en dicho árbol (4). Si bien en el ejemplo se ve que aumentaron las magnitudes, la regla general dice que estas deberían disminuir para datos donde hay localidad espacial.

3.1.2.2 *D-B ET: Differential-Bitmap Encoding Tree*

La segunda propuesta es, tal como DEST, usar la diferencia como operación entre nodos y separar el signo del dato: sólo almacenar el valor absoluto de la diferencia y usar un bitmap para almacenar los signos de los valores de los nodos resultantes. Esto es $v'_h = |v_h - v_p|$ y $bitmap[pos(v'_h)] = 1$ si $v_h - v_p < 0$ y 0 si no. Nótese que la operación $pos(v)$ es de tiempo constante, ya que es acceder a un arreglo cuya posición es un índice conocido. El inconveniente de esta representación es que requiere de n *bits* adicionales, donde n son los datos (nodos). En la figura 3.6 se muestra el árbol de diferencias descrito en esta sección y su bitmap. Sobre cada nodo está la operación que generó el valor y entre paréntesis el valor del bitmap en la posición del nodo. Por ejemplo, el hijo izquierdo de la raíz tiene el valor absoluto de la resta entre el mismo

nodo en el árbol original (2) y el padre (4), y pone el bitmap en su posición en 1, pues la resta da un valor negativo.

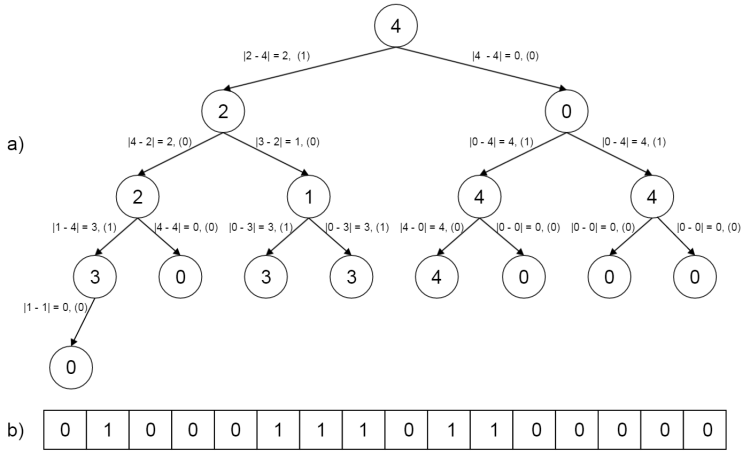


Figura 3.6: *D-B ET*: a) Árbol operado con diferencias, resultante del ejemplo de la figura 3.4; b) Bitmap del árbol.

3.1.2.3 *ZigZag ET: ZigZag Encoding Tree*

La última alternativa que proponemos para generar un árbol operado de valores más pequeños que el árbol original, es usar *ZigZag Encoding* (sección 2.1.5) sobre los valores resultantes en el árbol de diferencias, es decir, calcular T' usando diferencia como operación. Posteriormente, para cada nodo en T', si su valor n es positivo o 0, es reemplazado por $2|n|$ y, si es negativo, es remplazado por $2|n| - 1$. Esto es equivalente a almacenar el bit de signo en el bit menos significativo de la secuencia que conforma n. La figura 3.7 muestra el árbol resultante de operar el árbol original (figura 3.4) usando *ZigZag Encoding* sobre los resultados. Por ejemplo, el nodo hijo izquierdo de la raíz, tiene el valor 3, que es el resultado de tomar el valor en el nodo original (2), el valor en el padre (4), restarlos $(2 - 4)$ y el resultado (-2) mapearlo $(|-2| \times 2 - 1 = 3)$.

3.1.3 Paso 3: Representación de la Topología.

El árbol resultante de las etapas anteriores es un árbol binario completo. La magnitud de los datos es pequeña y además debieran haber subárboles con el mismo valor

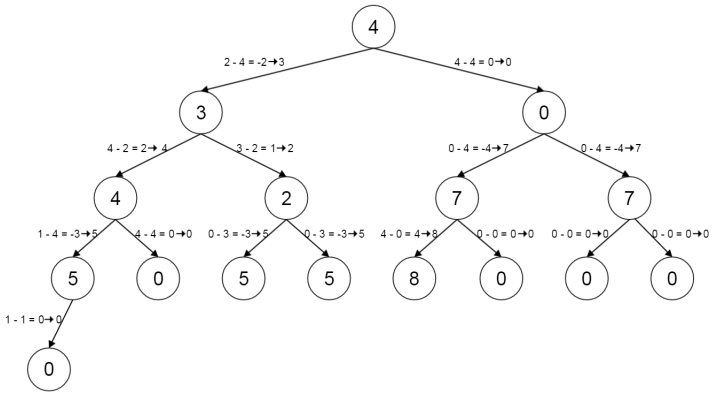


Figura 3.7: *ZigZag ET*: Árbol operado ZigZag, resultante del ejemplo de la figura 3.4, donde los resultados negativos n son mapeados a $2|n| - 1$ y los positivos a $2|n|$.

(en particular, debieran existir subárboles con el valor 0 en sus nodos, resultado de completar la matriz original con 0s). En función de estas características se consideran dos alternativas para la representación de la topología del árbol: *Binary Heap Embedding* y Árboles Sucintos.

3.1.3.1 *Binary Heap Embedding*

Esta alternativa toma ventaja del hecho que el árbol es binario y completo y, como no requiere un uso adicional de espacio para almacenar la topología, resulta muy conveniente considerarla en este trabajo. En la sección 2.1.4.2 hay una descripción de como funciona esta topología. En la figura 3.8, se muestra el *Binary Heap Embedding*, resultante del árbol en ZigZag Encoding, descrito en la sección 3.1.2.3. Nótese que para las otras 2 alternativas el proceso es análogo.

4	3	0	4	2	7	7	5	0	5	5	8	0	0	0	0

Figura 3.8: Representación *binary heap embedding* del árbol de la figura 3.7.

3.1.3.2 Árboles Sucintos

Si bien el *Binary Heap Embedding* es una buena alternativa, no permite realizar un podado del árbol, esto es, quitar algún subárbol de la representación. Esto resultaría útil pues el árbol resultante tiene subárboles con un mismo valor los cuales sería

conveniente podar en vez de almacenar. Ahora bien, al contrario que el *Binary Heap Embedding*, usar árboles sucintos requiere de un espacio adicional a los datos para almacenar la topología.

La aplicación directa de los árboles sucintos descritos en la sección 2.1.8 requiere de $2n+o(n)$ bits para un árbol de n nodos. Sin embargo, a continuación, proponemos una mejora para la topología particular de los árboles que se forman en nuestro dominio.

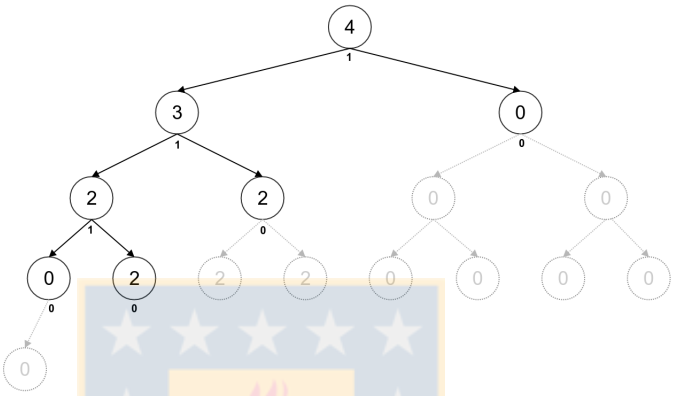


Figura 3.9: Árbol binario podado y etiquetado.

Dado que tenemos un árbol binario y completo, cuando es podado cada nodo resultante tiene 0 o 2 hijos (debido a que cada subárbol cuyos valores son todos iguales es reemplazado por un único nodo). Esto nos permite representar el árbol podado marcando sus nodos asignándole un 1 a cada nodo interno y un 0 a un nodo hoja. La figura 3.9 muestra un ejemplo de este procedimiento. El árbol es distinto al de los ejemplos anteriores con la finalidad de mostrar un mejor escenario de podado de árbol.

Luego, se almacenan tanto los datos como el bitmap (por separado), por niveles (igual que el *Binary Heap Embedding*). En esta representación, el hijo izquierdo de un nodo en la posición i estará ubicado en la posición $2 \times rank_1(i)$ y el hijo derecho en la posición $2 \times rank_1(i) + 1$ (dado que se está realizando un almacenamiento por niveles). Para el recorrido, se asume que al llegar a un nodo hoja todos los valores por debajo de la hoja son iguales al valor de ésta. De esta forma, se recorre un árbol binario completo virtual (cuando se llega a un nodo podado, se asume el mismo valor del nodo al descender por él). La figura 3.10 muestra la secuencia de datos y el bitmap

asociado al árbol de la figura 3.9.

4	3	0	2	2	0	2

1	1	0	1	0	0	0

Figura 3.10: Datos y bitmap asociado al árbol de la figura 3.9

3.1.4 Paso 4: Codificación de valores

La última etapa es codificar los valores del árbol. Estos valores son de magnitud pequeña, por lo que resulta conveniente usar Códigos de Acceso Directo, DACs. Esto por el acceso, en la práctica, de tiempo constante a los datos, lo cual es necesario para soportar las consultas (que se explicarán a continuación). Nótese que la codificación es independiente del tipo de árbol usado (XOR ET, Bitmap ET o ZigZag ET) o de la topología escogida (*Binary Heap Embedding* o Árboles Sucintos).

3.1.5 Algoritmos de Consulta

3.1.5.1 Consultas de Acceso

La consulta de acceso se convierte en: primero transformar (x, y) en $p_{x,y}$ una posición en el *z-order-array*. Luego, se recorre el árbol desde su raíz hasta dar con el nodo correspondiente a la posición $p_{x,y}$. De manera que la complejidad de esta consulta es logarítmica con respecto a los datos, pues depende de la altura del árbol. El ejemplo de la figura 3.11 muestra como sería la consulta para la celda $(1, 1)$.

Como se puede apreciar en el ejemplo, la consulta de acceso está fuertemente ligada a la representación de la topología del árbol. Dado que se ha usado DACs, el acceso a los valores es constante, por lo que nuestra consulta de acceso tiene un tiempo logarítmico (depende de la altura del árbol).

El algoritmo 1 muestra el procedimiento de esta consulta. Las funciones z_order_trans, go_to_left y go_to_right, son funciones para mapear los valores en *Z-Order*, ir al nodo hijo izquierdo e ir al nodo hijo derecho, respectivamente.

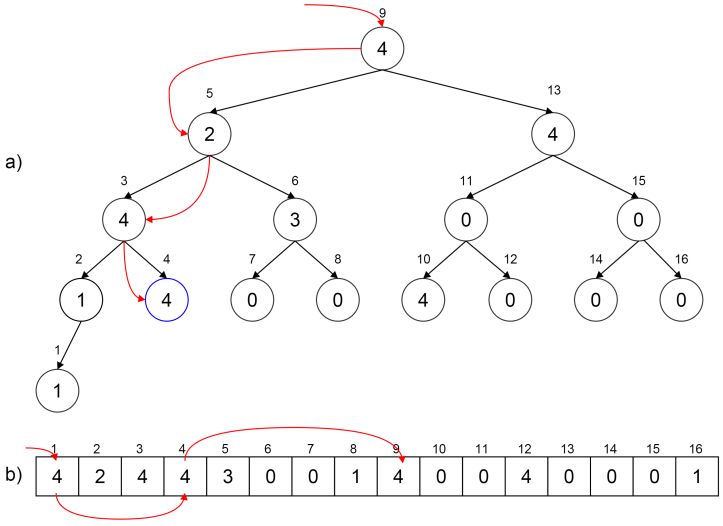


Figura 3.11: Navegación en el árbol de la operación *access(1,1)*

3.1.5.2 Consulta por una Subregión

El comportamiento de la curva *Z-Order* es fundamental para resolver este tipo de consultas. Dados dos puntos, $p1 = (x1, y1)$ y $p2 = (x2, y2)$, todos los puntos del rectángulo formado por $p1$ y $p2$ como diagonal están en el recorrido de la curva. Sin embargo, puntos adicionales también pueden ser visitados en dicho recorrido. La figura 3.12 muestra cómo a partir de un *access* en un punto inicial $p1 = (1, 1)$ y recorriendo hasta un punto $p2 = (2, 2)$ se da la situación antes descrita.

Ahora bien, una descomposición en *quadboxes* (sección 2.1.3) de una región rectangular es un conjunto de Zs del recorrido en *Z-Order*, como se mencionó en la descripción de esa descomposición.

Tomando en cuenta estas 2 premisas, se puede derivar un algoritmo para realizar las consultas por subregiones. Primero, se debe hacer una descomposición en *quadboxes* con base en la región de la consulta y luego hacer una consulta de rango por cada *quadbox-Z*.

Nótese que un recorrido en *Z-Order* sobre el árbol es equivalente a un recorrido en *in-order* sobre el mismo, es decir, visitar subárbol izquierdo, luego nodo padre y finalmente subárbol derecho, desde $p1$ hasta $p2$ (como se aprecia en la figura 3.12). El algoritmo 2 muestra el procedimiento para realizar la consulta por subregión con base en la descomposición en quadboxes, con la función quadbox_desc un algoritmo

Algoritmo 1 access(x, y), consulta de acceso.

Entrada: x, y, las coordenadas de la celda en la matriz.
Salida: v el nodo en el árbol que representa a la celda buscada.

```
1: p_{x,y} ← z_order_trans(x, y)
2: v ← root
3: while p_{x,y} ≠ v.position do
4:    if p_{x,y} < v.position then
5:       v ← go_to_left(v)
6:    else
7:       v ← go_to_right(v)
8:    end if
9: end while
10: return v
```

que toma una región y devuelve un conjunto de *quadboxes*.

Algoritmo 2 windowQuery$(x1, y1, x2, y2)$, con estrategia de descomposición en *quadboxes*.

Entrada: $x1, y1, x2, y2$ las coordenadas de la subregión de consulta.
Salida: M matriz de valores que conforma la subregión consultada.

```
1: M ← null
2: Q ← quadbox_desc(x1, y1, x2, y2)
3: while not Q.isEmpty() do
4:    quad ← Q.element()
5:    node ← access(quad.x1, quad.y1)
6:    p_end ← z_order_trans(quad.x2, quad.y2)
7:    while node.position ≠ p_end do
8:       M.insert(node.value)
9:       node ← node.next()
10:   end while
11:   quad ← Q.next()
12: end while
13: return M
```

Una segunda alternativa para resolver las consultas por una subregión, es realizar el recorrido de la curva y detectar cuando este recorrido ha salido del área de consulta. Cuando esta situación es detectada, se calcula cuál es la siguiente celda donde la curva debe ingresar para continuar su recorrido. Para realizar esto, de forma recursiva, se separa la región consultada en la celda de salida con una línea imaginaria (vertical u horizontal, dependiendo del caso) y se busca, usando operaciones sobre bits, cuál

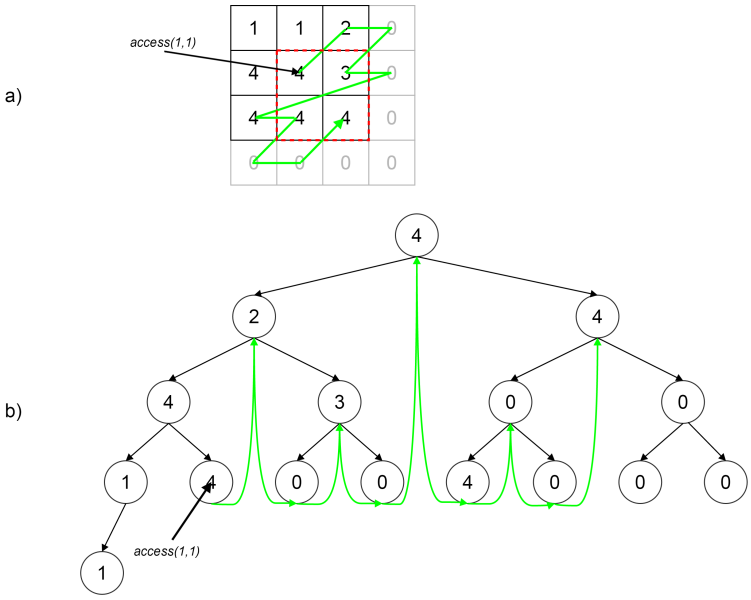


Figura 3.12: Recorrido en *Z-Order* para una consulta de rango.

es el siguiente valor de entrada en la región contraria a donde se salió la curva. Esta técnica se conoce como *range search* en [22]. El algoritmo 3 muestra el procedimiento para resolver la consulta por una subregión usando esta alternativa, donde la función *z-divide* retorna el nodo donde reingresa la curva *Z-Order*.

Para evitar confusiones, llamaremos a nuestra primera alternativa de solución "descomposición en *quadboxes*" (QD) y a la segunda "*z-divide*" (ZD) (este último nombre es el que usamos en la experimentación).

3.1.5.3 Consulta por una Subregión con Rango

La estructura descrita en esta sección fue diseñada pensando en las operaciones de access y windowQuery, y no proporciona un soporte eficiente para consultas de tipo rangeQuery[1] (a diferencia de la solución que proponemos en la siguiente sección), de manera directa.

Una manera ingenua, pero que no requiere espacio extra, de resolver este tipo de consultas consiste en emplear una estrategia de filtrado y refinamiento. Para esto, se puede emplear la solución descrita para consultas de tipo windowQuery en la etapa

[1]Notar que este tipo de consultas estaba fuera del alcance original de la propuesta de trabajo de esta tesis.

Algoritmo 3 windowQuery($x1, y1, x2, y2$), con estrategia *z-divide*.

Entrada: $x1, y1, x2, y2$ las coordenadas de la subregión de consulta.
Salida: M matriz de valores que conforma la subregión consultada.

```
1:  M ← null
2:  node ← access(x1, y1)
3:  p_end ← z_order_trans(x2, y2)
4:  while node.position ≠ p_end do
5:     if not node ∈ window then
6:        node ← zdivide(x1, y1)
7:     end if
8:     M.insert(node.value)
9:     node ← node.next()
10: end while
11: return M
```

de filtrado y, posteriormente, realizar un refinamiento secuencial sobre los resultados obtenidos. Esta solución puede tener buenos resultados cuando el rango de valores sea pequeño, pero su eficiencia se degradará rápidamente a medida que crezca dicho rango.

Como alternativa para paliar este problema, se debe observar que cada nodo en nuestra estructura de datos, además de representar una celda, también se puede asociar a una subregión del espacio original (es decir, el valor almacenado en dicha celda actúa como representante de la región. Por tanto, es posible almacenar información extra en cada nodo que represente el mínimo y el máximo de la subregión, de manera que se pueda filtrar el resultado a medida que se desciende en el árbol. Esta solución es similar a la empleada en [5]. Se puede observar también que el valor almacenado en cada nodo estará entre el mínimo y el máximo, por tanto, dichos valores se pueden codificar diferencialmente (además de estar codificados diferencialmente con respecto a los valores del nodo padre). En [26] se muestra que el almacenamiento de este tipo de información suele requerir muy poco espacio extra.

3.2 Estrategia 2: Mapeo de 3 Dimensiones a 2 Dimensiones

La segunda alternativa busca mapear la tupla $< x, y, z >$, con (x, y) la coordenada en el *raster* y z el valor en el *raster* de aquella coordenada, a una matriz donde un eje es

un valor generado a partir de mapear la coordenada (x, y) (eje de posiciones) y el otro es generado por el rango de valores posibles de z (eje de valores). Esta estrategia fue diseñada con la intención de mejorar la eficiencia de las consultas (en particular las de subregión con rango que no presentan un soporte directo en la estructura propuesta en la sección anterior). Consta de 2 pasos:

- Paso 1: Mapeo de los datos.
- Paso 2: Codificación de valores.

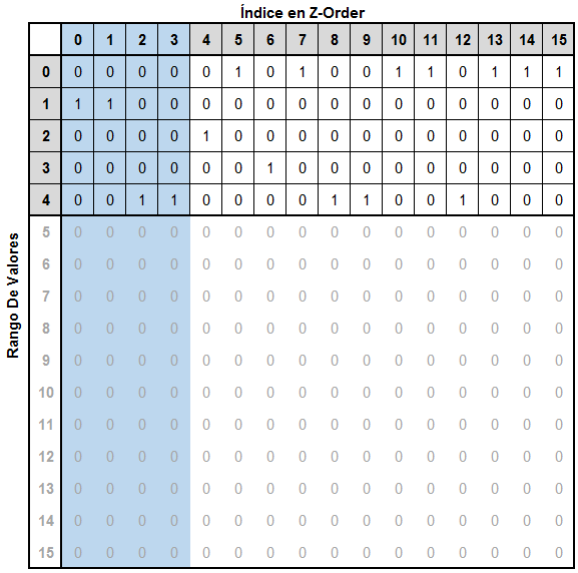
3.2.1 Paso 1: Mapeo de los Datos

Esta etapa usa nuevamente la curva *Z-Order*, para mapear las coordenadas (x, y) a una coordenada $p_{x,y}$, que se convierte en el nuevo eje horizontal (denominado, eje de posiciones) de una matriz, y el rango de valores almacenados, z, es convertido en el eje vertical (eje de valores). Es decir, se crea una matriz, donde en cada columna $p_{x,y}$ habrá un único 1 (que estará en la fila z), con x la fila, y la columna y z el valor almacenado en la posición (x, y) en la matriz original. Esta nueva matriz, es binaria, es decir sus celdas toman el valor 1 cuando (x, y) tiene el valor z en la matriz original y 0 si no. Nótese que en cada columna de esta matriz existirá a lo más una celda con el valor 1. El ejemplo de la figura 3.13 muestra este mapeo en los ejes horizontal (posiciones en *Z-Order* de 0 a 15) y vertical (valores posibles, de 0 a 4) y la grilla de 1s y 0s generada. Observando el ejemplo, en la figura 3.13 hay un 1 en la posición $(3, 4)$, pues en la matriz original (figura 3.1) hay un 4 en la posición $(1, 1)$ pues $p_{1,1} = 3$.

3.2.2 Paso 2: Codificación de valores

Una vez se tiene el mapeo, se consideraron dos alternativas para la representación de los datos (es decir, para la compresión de la matriz binaria):

- k^2*-tree*.
- *Wavelet Tree*s comprimidos.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	1	0	1	0	0	1	1	0	1	1	1
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
4	0	0	1	1	0	0	0	0	1	1	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 3.13: Ilustración matricial del mapeo de 3 dimensiones a 2 dimensiones con base en el ejemplo de la figura 3.1. En color, una consulta por una subregión.

3.2.2.1 k^2-*tree*

Dado que tenemos una matriz binaria, resulta obvio considerar el k^2-*tree* como alternativa, por su buen desempeño tanto en compresión como en tiempo de consulta. Nótese que la matriz resultante del mapeo es mucho más grande en su eje de posiciones que en el de valores (el cual se completa con 0s para formar una matriz cuadrada que se puede almacenar con el k^2-*tree*). Esto sumado a que el mapeo realizado preserva la localidad espacial, resulta en muchas regiones de 0s que son fácilmente representadas en el k^2-*tree*. La figura 3.14 es la representación del k^2-*tree* formado a partir de la matriz de la figura 3.13 (en línea punteada, el recorrido sobre el árbol de la consulta de rango, que se explicará más adelante). En ella se puede apreciar cómo el árbol elimina rápidamente las regiones con 0s, en particular las zonas que se forman debido a que el eje vertical es mayor al horizontal. Nuestras consultas pueden ser fácilmente mapeadas a consultas de rango sobre el k^2-*tree* (descrito en la sección 3.2.3).

3.2.2.2 *Wavelet Tree*

La segunda alternativa es usar un *Wavelet Tree* tomando la secuencia de valores como entrada para el mismo y el eje de valores como alfabeto. La figura 3.15 muestra el

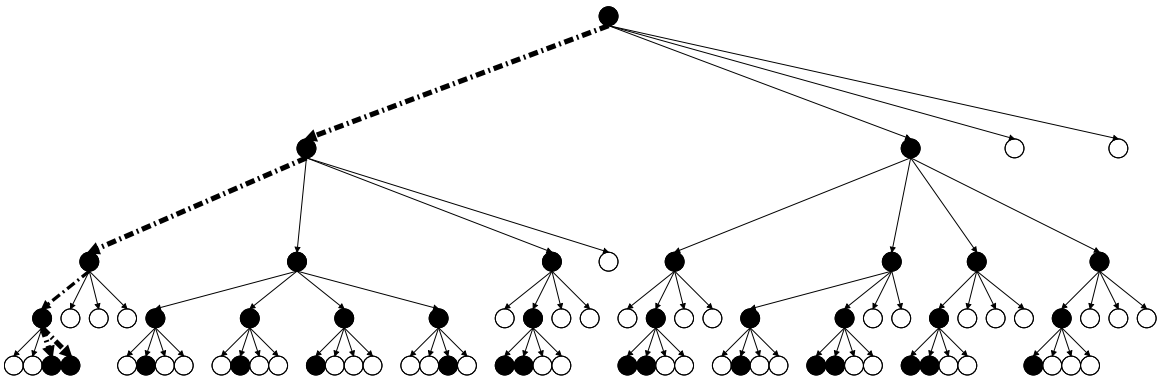


Figura 3.14: k^2-*tree* generado a partir de la matriz de la figura 3.13. Los círculos negros representan 1s y los blancos 0s. La línea discontinua representa la consulta de la figura citada.

Wavelet Tree asociado al ejemplo de la figura 3.1. Observamos de este ejemplo, que la raíz contiene la secuencia completa de la transformación en *Z-Order* (figura 3.2) y el alfabeto son todos los posibles valores en la matriz ampliada (0-4). En la figura está destacada la consulta por una subregión que será explicada más adelante. Dado que usamos la curva *Z-Order* y debido a que ésta preserva la localidad espacial, es muy probable encontrar regularidades, las cuales pueden ser aprovechadas por las distintas técnicas de compresión en los nodos del WT [11], lo que vislumbra las potenciales capacidades de compresión de esta estructura en este dominio.

3.2.3 Algoritmos de Consulta

3.2.3.1 Consultas de Acceso

Tanto para la alternativa de k^2-*tree* como la de *Wavelet Tree* la consulta de acceso se convierte en una consulta de acceso en las respectivas estructuras. Nótese que para el k^2-*tree* se debe consultar por el único 1 en la columna dada por la posición. Esto se hace descendiendo en el árbol hasta dar con una hoja. El algoritmo 4 ilustra este proceso, donde la función get_inverse_list retorna la lista de índices del k^2-*tree* que tengan un 1 en la columna consultada (donde sólo habrá a lo más un 1).

Por otro lado, en el *Wavelet-Tree*, se debe profundizar en el árbol partiendo en la posición indicada por el mapeo en *Z-Order* y luego se va al hijo izquierdo o derecho en función del bit en esa posición y se repite el proceso en la secuencia del hijo

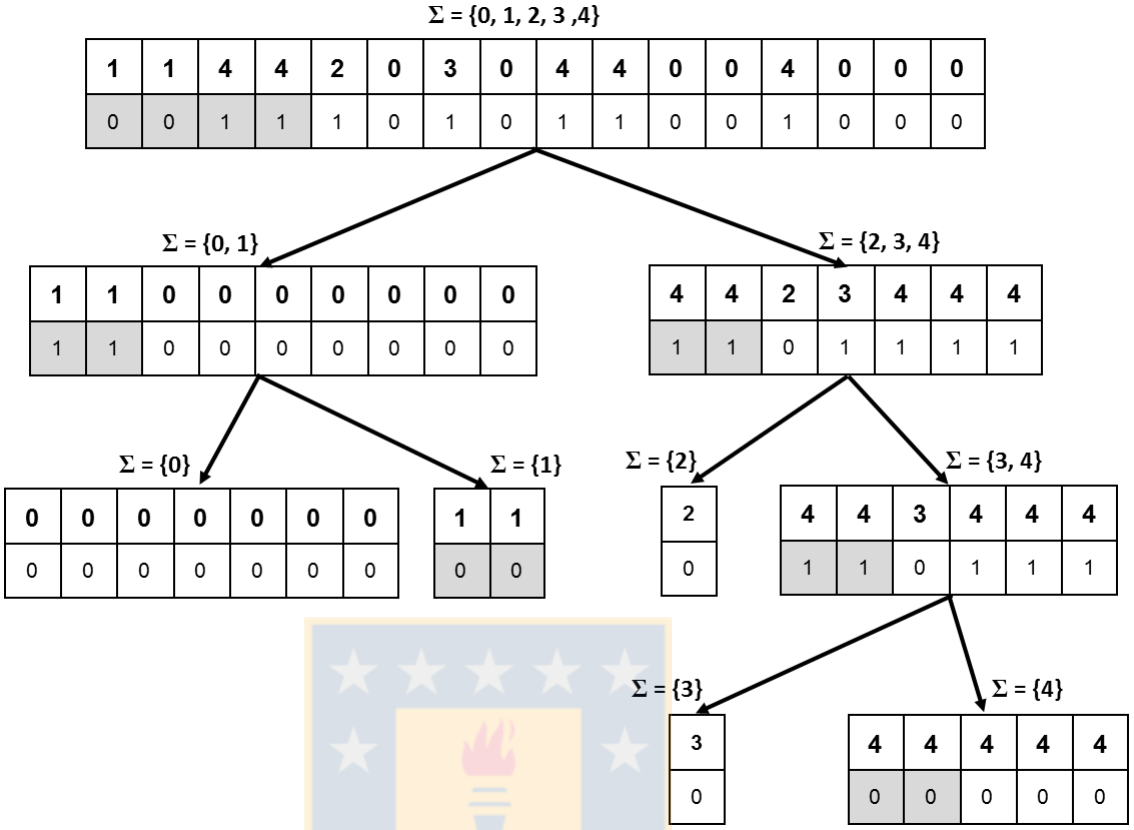


Figura 3.15: *Wavelet Tree* generado a partir del ejemplo de la figura 3.1. En gris la consulta por una subregión subdividida entre sus ramas.

accedido hasta llegar a una hoja. El algoritmo 5 muestra el procedimiento para hacer consultas de acceso con base en el *wavelet tree*. La función accessWT retorna el valor almacenado en la secuencia en la posición consultada.

3.2.3.2 Consulta por una Subregión

En la sección 3.1.5.2 se describió cómo una descomposición en *quadboxes* (sección 2.1.3) se puede utilizar para resolver consultas por una subregión. Usando esta misma idea, una consulta por una subregión es mapeada a k consultas donde cada subregión es un *quadbox*. Luego, para cada subconsulta i, en la alternativa que usa k^2-*trees*, se realiza una consulta de rango sobre el k^2-*tree* [6], esta consiste en, partiendo en la raíz, descender por cada nodo que esté presente en la subregión i. Esto se puede apreciar en la figura 3.14 en líneas discontinuas, mostrando como resultados los nodos

Algoritmo 4 access(x, y), consulta de acceso en el k^2-*tree*

Entrada: x, y, las coordenadas de la celda en la matriz.
Salida: v el valor en la coordenada solicitada.

```
1: p_{x,y} ← z_order_trans(x, y)
2: list ← get_inverse_list(k2tree, p_{x,y})
3: v ← list.element()
4: return v
```

Algoritmo 5 access(x, y), consulta de acceso en el *wavelet tree*

Entrada: x, y, las coordenadas de la celda en la matriz.
Salida: v el valor en la coordenada solicitada.

```
1: p_{x,y} ← z_order_trans(x, y)
2: v ← accessWT(WT, p_{x,y})
3: return v
```

hojas tercero y cuarto de izquierda a derecha. Esta consulta se realiza sobre la región del eje de posiciones que comprende todas las celdas de la consulta i, con respecto a todas las celdas del eje de valores (región destacada en la figura 3.13). El algoritmo 6 muestra el procedimiento para las consultas por subregiones con base en el k^2-*tree*, con la función windowRangeQueryK2Tree como una consulta de rango del k^2-*tree*, MIN_VAL el valor mínimo posible y MAX_VAL el valor máximo posible.

De manera similar, en la alternativa que usa un *Wavelet Tree*, para cada subconsulta i se hace una consulta sobre el rango dado por i con respecto a todo el alfabeto del *Wavelet Tree* [10]. En la figura 3.15, en gris se aprecia esta consulta y cómo esta se distribuye en los nodos del árbol. Nótese que dado el caso que toda la secuencia de un nodo sea parte de una consulta, no es necesario seguir profundizando en el árbol. El algoritmo 7 ilustra el procedimiento de la consulta por una subregión con base en el *wavelet tree*, con la función windowRangeQueryWT, como una consulta por una subsecuencia en el wavelet tree. Nótese que en este caso, MIN_VAL y MAX_VAL no representan una limitación sobre el alfabeto (MIN_VAL y MAX_VAL son el principio y fin del alfabeto, respectivamente).

3.2.3.3 Consulta por una Subregión con Rango

Esta consulta se resuelve similar a la anterior, pero ahora limitando el eje de valores para cada estructura. Esto en el k^2-*tree* se resuelve descartando las ramas que se

Algoritmo 6 windowQuery($x1, y1, x2, y2$), consulta por una subregión, con base en el k^2-*tree*.

Entrada: $x1, y1, x2, y2$ las coordenadas de la subregión de consulta.
Salida: M matriz de valores que conforma la subregión consultada.

```
1:  M ← null
2:  Q ← quadbox_desc(x1, y1, x2, y2)
3:  while not Q.isEmpty() do
4:      quad ← Q.element()
5:      p_ini ← z_order_trans(quad.x1, quad.y1)
6:      p_end ← z_order_trans(quad.x2, quad.y2)
7:      list ← windowRangeQueryK2Tree(k2tree, p_ini, p_end, MIN_VAL, MAX_VAL)
8:      while not list.empty() do
9:          M.insert(list.element())
10:         list ← list.next()
11:     end while
12:     quad ← Q.next()
13: end while
14: return M
```

Algoritmo 7 windowQuery($x1, y1, x2, y2$), consulta por una subregión, con base en el *wavelet tree*.

Entrada: $x1, y1, x2, y2$ las coordenadas de la subregión de consulta.
Salida: M matriz de valores que conforma la subregión consultada.

```
1:  M ← null
2:  Q ← quadbox_desc(x1, y1, x2, y2)
3:  while not Q.isEmpty() do
4:      quad ← Q.element()
5:      p_ini ← z_order_trans(quad.x1, quad.y1)
6:      p_end ← z_order_trans(quad.x2, quad.y2)
7:      list ← windowRangeQueryWT(WT, p_ini, p_end, MIN_VAL, MAX_VAL)
8:      while not list.empty() do
9:          M.insert(list.element())
10:         list ← list.next()
11:     end while
12:     quad ← Q.next()
13: end while
14: return M
```

asocian al eje de valores y que no están en el rango de consulta. En el caso del *Wavelet Tree*, se restringe el alfabeto, lo que se traduce en descartar los elementos de la secuencia, en cada nodo, que no estén en el rango de valores solicitados. Los algoritmos 8 y 9 muestran las consultas por una subregión con rango con base en el k^2-*tree* y el wavelet tree, respectivamente.

Algoritmo 8 windowQuery$(x1, y1, x2, y2, min, max)$, consulta por una subregión con rango, con base en el k^2-*tree*.

Entrada: $x1, y1, x2, y2, min, max$ las coordenadas de la subregión de consulta y límites del rango.

Salida: M matriz de valores que conforma la subregión consultada.

```
1:  M ← null
2:  Q ← quadbox_desc(x1, y1, x2, y2)
3:  while not Q.isEmpty() do
4:     quad ← Q.element()
5:     p_ini ← z_order_trans(quad.x1, quad.y1)
6:     p_end ← z_order_trans(quad.x2, quad.y2)
7:     list ← windowRangeQueryK2Tree(k2tree, p_ini, p_end, min, max)
8:     while not list.empty() do
9:        M.insert(list.element())
10:       list ← list.next()
11:    end while
12:    quad ← Q.next()
13: end while
14: return M
```

Algoritmo 9 windowQuery($x1, y1, x2, y2, min, max$), consulta por una subregión con rango, con base en el *wavelet tree*.

Entrada: $x1, y1, x2, y2, min, max$ las coordenadas de la subregión de consulta y límites del rango.

Salida: M matriz de valores que conforma la subregión consultada.

```
1:  M ← null
2:  Q ← quadbox_desc(x1, y1, x2, y2)
3:  while not Q.isEmpty() do
4:     quad ← Q.element()
5:     p_ini ← z_order_trans(quad.x1, quad.y1)
6:     p_end ← z_order_trans(quad.x2, quad.y2)
7:     list ← windowRangeQueryWT(WT, p_ini, p_end, min, max)
8:     while not list.empty() do
9:        M.insert(list.element())
10:       list ← list.next()
11:    end while
12:    quad ← Q.next()
13: end while
14: return M
```

Capítulo 4

Evaluación Experimental

El trabajo experimental fue desarrollado en una máquina con las siguientes características:

- Procesador Intel Core i7-3820@3.60GHz.
- Memoria RAM de 32GB.
- Sistema operativo Ubuntu server (kernel 3.13.0-35).
- Compilador gnu/g++ versión 4.6.3.

Para evaluar las estructuras desarrolladas se usaron *datasets* que representan Modelos Digitales de Terreno (MDT) que denotan el nivel de elevación de un determinado lugar [21]. En particular, los *datasets* utilizados, MDT05-*, son modelos digitales del terreno con paso de malla de 5m. En particular, se evaluaron los *datasets* MDT05-500 y MDT05-700 debido a que estos son evaluados en el estado del arte [4]. La tabla 4.1 muestra información general acerca de los *datasets* utilizados.

Dataset	Cols	Rows	min value	max value
MDT05-0500	5841	4001	0	914,049
MDT05-0700	5841	3841	-0,085	471,777

Tabla 4.1: Información general de los *dataset*s utilizados.

En el estado del arte truncan los valores para realizar sus mediciones de modo que nosotros hicimos lo mismo para nuestros experimentos.

La estructura propuesta en 3.1 propone 3 alternativas para reducir la magnitud de los valores en los datos (secciones 3.1.2.1, 3.1.2.2 y 3.1.2.3). Se realizó un experimento previo de la compresión que cada una de estas alternativas consigue para determinar

Dataset	D-B ET		XOR ET		ZigZag ET	
	ratio	bit per cell	ratio	bit per cell	ratio	bit per cell
MDT05-0500	0,22	7,12	0,22	6,99	0,20	6,50
MDT05-0700	0,23	7,32	0,22	6,92	0,21	6,59

Tabla 4.2: Relación en espacio entre la estructura comprimida y la estructura original. *ratio*: representa la razón entre el tamaño de la matriz comprimida en bytes y el tamaño de la matriz original. *bit per cell*: bit por celda de la matriz usados en cada una de las representaciones.

cual de ellas es la que analizaremos en profundidad (pues el desempeño en tiempo es similar en las 3).

La tabla 4.2 muestra los resultados de dicho experimento usando como topología el *binary heap embedding*. Del experimento se ve que la alternativa que logra mejor compresión es la que usa *ZigZag encoding*, de modo que esa será la estructura que se estudiará en profundidad (usando como topología tanto el *binary heap embedding* como los árboles sucintos).

Dado que son varias las alternativas de solución, se creó la siguiente nomenclatura para referirnos a cada estructura:

- ZZ BHE: es la alternativa que usa el ZigZag Encoding y un Binary Heap Emmbeding (secciones 3.1.2.3 y 3.1.3.1).
- ZZ ST: es la alternativa que usa el ZigZag Encoding y árboles sucintos (secciones 3.1.2.3 y 3.1.3.2).
- K2T: es la alternativa que realiza un mapeo de 3 dimensiones a 2 dimensiones y que usa un k^2-*tree* (sección 3.2.2.1).
- WT: es la alternativa de 3 dimensiones a 2 dimensiones que usa un *Wavelet Tree* (sección 3.2.2.2).
- k2-acc y k3tree: son las alternativas, propuestas en el estado del arte, que usan varios k^2-*tree* y un K^3-*Tree*, respectivamente [4].

4.1 Uso de Memoria

La tabla 4.3 muestra los bits por celda que requiere cada estructura, además de una columna con los resultados expuestos en el estado del arte.

Dataset	ZZ BHE	ZZ ST	K2T	WT	k3tree	k2-acc
MDT05-500	6,5	2,75	2,61	5,82	1,83	2,30
MDT05-700	6,6	1,89	2,15	4,9	1,38	2,40

Tabla 4.3: Uso de memoria en bits por celda.

De la tabla 4.3, se aprecia que nuestras estructuras ZZ ST y K2T tienen un comportamiento muy cercano en uso de espacio al que expone el estado del arte, mientras que las otras alternativas no son competitivas. Para WT se está empleando compresión basada en *Run-Lenght* (disponible en la SDSL que es la librería que empleamos como base). Es posible que una variante que solo emplee compresión en los primeros niveles del árbol, ofrezca mejores resultados. Esto se plantea como trabajo futuro. La compresión alcanzada por k2-acc, fue directamente extraída de los resultados de [4], pues no se pudo replicar el experimento. De modo que sus resultados no estarán expuestos para ninguna de nuestras consultas. Cabe mencionar que, si bien en [4] muestran que para las consultas por subregión rango k2-acc tiene una leve ventaja que k3tree, en [5] muestran que su desempeño es mucho peor a medida que aumenta la cantidad de valores diferentes en el rango.

4.2 Tiempo de Consulta de Acceso

Para evaluar el tiempo de consultas de acceso de cada estructura se crearon *datasets* con 50000 consultas de acceso. Luego se realizaron las 50000 consultas seguidas y se dividió el tiempo en la cantidad de consultas. En la tabla 4.4 se muestra el tiempo por consulta resultante del experimento descrito en microsegundos (μs).

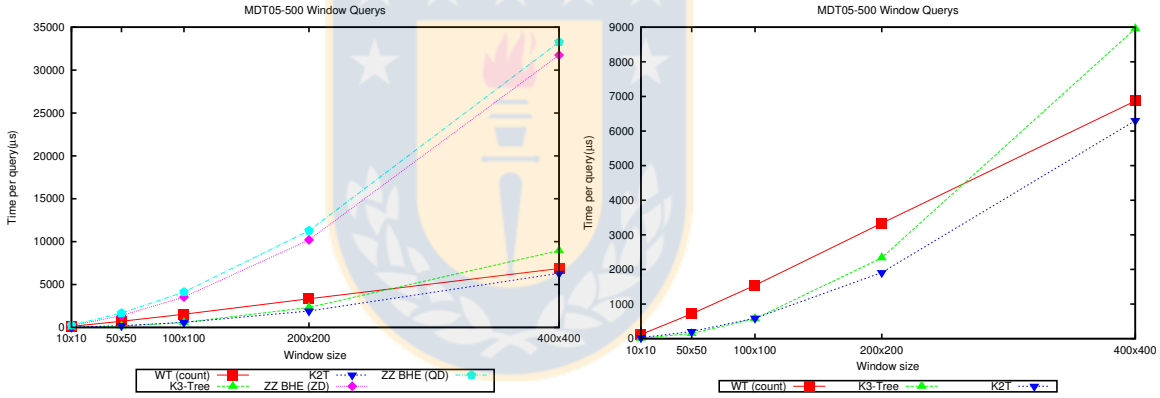
En los resultados expuestos en la tabla 4.4 se ve que no hay gran diferencia en orden de magnitud en los tiempos de acceso a datos (considerando que son microsegundos). Además nuestra alternativa K2T muestra mejores resultados que el estado del arte.

Dataset	ZZ BHE	ZZ ST	K2T	WT	k3tree
MDT05-500	4,40	4,60	1,4	4,00	2,20
MDT05-700	3,20	3,40	1,2	3,00	1,60

Tabla 4.4: Tiempo de consulta de acceso en microsegundos (μs).

4.3 Tiempo de Consulta por una Subregión

Para evaluar el comportamiento de las consultas por una subregión, se tomó como parámetro el tamaño de la ventana de consulta (10x10, 50x50, 100x100, 200x200 y 400x400) y se generaron *querysets* con mil consultas cada uno. Se midió el tiempo, en microsegundos (μs), que tardaba cada estructura en resolver el *queryset* veinte veces y el resultado se dividió en el número de consultas del *queryset* por veinte para obtener el promedio de tiempo por consulta.



(a) Gráfico con ZZ BHE (b) Gráfico sin ZZ BHE

Figura 4.1: Gráficos de tiempo de cosulta por una subregión

Para esta consulta se evaluaron las estructuras ZZ BHE, K2T, WT y K^3-*Tree*. Para la estructura ZZ BHE se evaluaron las dos alternativas de consulta de rango descritas en la sección 3.1.5.2 (descomposición en quadboxes (QD) y función z-divide (ZD)). Por otro lado, la estructura WT, muestra el comportamiento de una consulta que solo reporta la cantidad de resultados contenidos ("count"'), sin devolver estos resultados. [1] Además, la estructura ZZ ST no fue implementada pero se puede

[1]La versión que de verdad reporta los resultados es mucho más lenta que las otras estructuras comparadas, por lo que no resulta competitivo. Esto se debe a que, por cada elemento en el resultado,

ver el comportamiento de ZZ BHE en este tipo de consulta como cota inferior de comportamiento.

Dado que los resultados entre ZZ BHE y las demás estructuras son muy distantes, se presentan 2 gráficos en la figura 4.1, a la izquierda uno que incluye los resultados de ZZ BHE y a la derecha uno que no los incluye. Se muestran en esta sección sólo los resultados del *dataset* MDT05-500, pues los resultados para MDT05-700 son similares.

De los gráficos, se aprecia que a medida que las consultas aumentan su tamaño, nuestra propuesta K2T ofrece mejores resultados que el estado del arte.

4.3.1 Caso Especial: consultas por Quadboxes

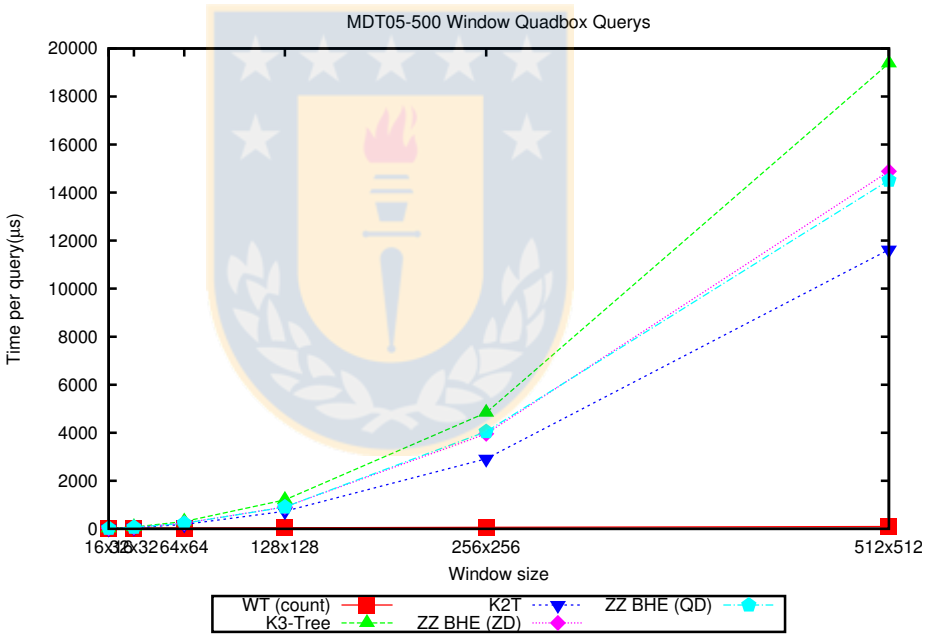


Figura 4.2: Gráfico de tiempo en microsegundos (μs) de consulta por subregiones que son quadboxes.

Dado que gran parte de nuestras alternativas constan de una descomposición en quadboxes, se realizó un experimento donde las regiones consultadas son quadboxes. Esto es justificable en la práctica en dominios como SIG donde muchas veces las

se debe volver a subir en el árbol. Por tanto, el WT sólo tendría aplicación cuando nos interesa contar y no reportar resultados.

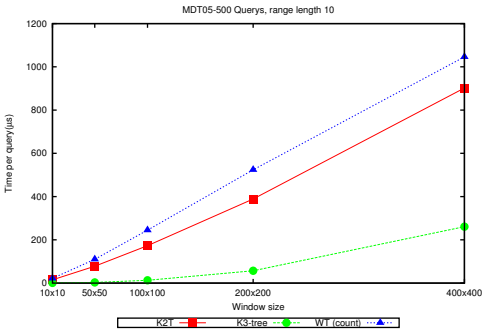
consultas no son definidas por el usuario sino por el sistema (ejemplo, "*tiles*" en un servicio de mapas WMS). Para este experimento, los tamaños de regiones fueron cambiados a una potencia de 2 (16x16, 32x32, 64x 64, 128x128, 256x256, 512x512). Los resultados de la figura 4.2 son el promedio de tiempo, en microsegundos (μs), de realizar consultas por todos los posibles quadboxes del tamaño de la ventana en particular, repitiendo cada consulta 20 veces.

Dado este caso especial, se reporta que, incluso ZZ BHE entrega mejores resultados que el estado del arte. Al igual que en la sección anterior el WT sólo cuenta resultados y no los reporta.

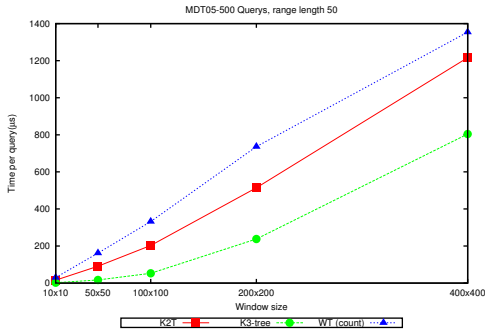
4.4 Tiempo de Consulta por una Subregión con Rango

Para este tipo de consultas se utilizaron como parámetros el tamaño de la ventana de consulta (10x10, 50x50, 100x100, 200x200 y 400x400) y el tamaño del rango (10, 50, 100, 200 y 400). Los experimentos se realizaron de manera similar que en el caso de la consulta por una subregión, pero creando *querysets* distintos para varios tamaños de rango. Es decir, cada uno de los resultados expuestos es el promedio de realizar mil consultas del tamaño de la ventana en particular, repitiendo 20 veces cada consulta. El tiempo se midió en microsegundos (μs).

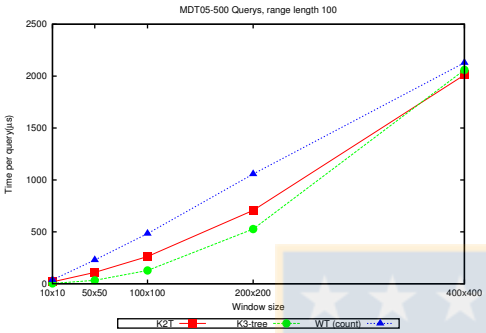
Los gráficos de la figura 4.3 muestran como es el desempeño en tiempo (eje vertical) en función del tamaño de la ventana de consulta (eje horizontal) fijando para cada gráfico un tamaño de rango. En esta sección solo los resultados del *dataset* MDT05-500, dado que los resultados vistos para el *dataset* MDT05-700 son similares.



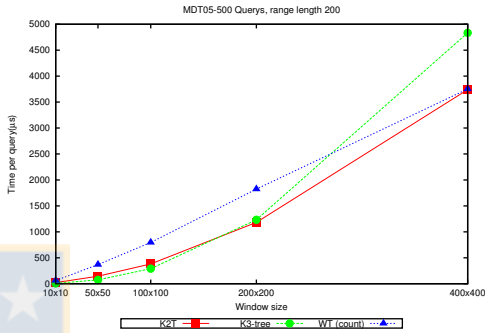
(a) Rango 10



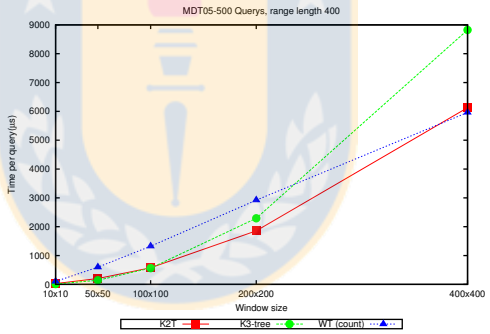
(b) Rango 50



(c) Rango 100



(d) Rango 200



(e) Rango 400

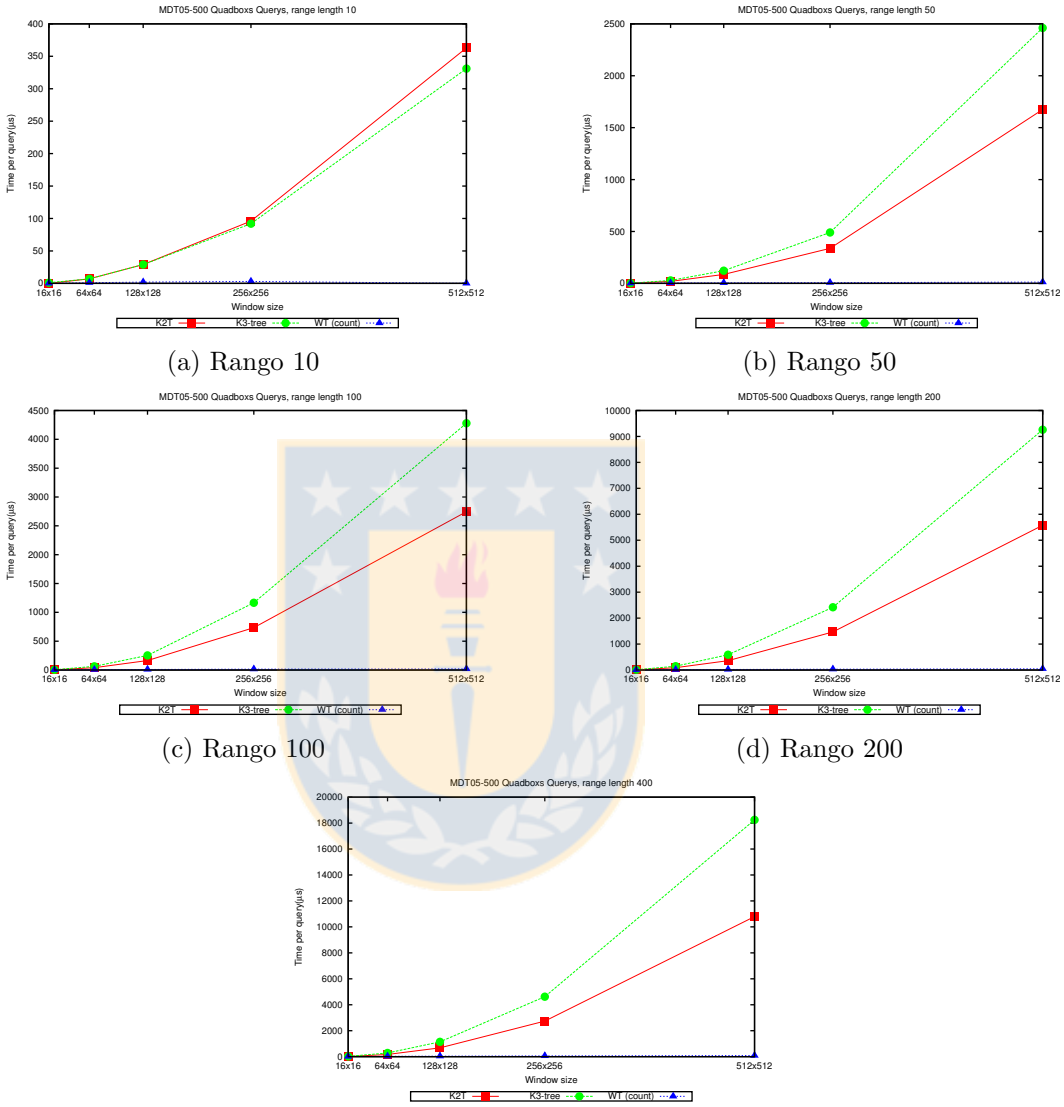
Figura 4.3: Gráficos de consultas por una subregión para varios tamaños de rango.

Nótese que no se realizó un experimento para probar esta consulta en ZZ BHE, pues los resultados serían similares a los mostrados en la consulta anterior. Igualmente, podemos ver el comportamiento de ZZ BHE como una cota inferior a ZZ ST, pues esta consulta realiza un mayor número de operaciones, de modo que no se realizaron experimentos para esta estructura. Además, la estructura WT muestra el comportamiento de la consulta que sólo reporta la cantidad de resultados contenidos.

De los gráficos, se aprecia que a medida que las consultas aumentan su tamaño y

rango, nuestra propuesta K2T ofrece mejores resultados.

4.4.1 Caso Especial: Consultas por Quadboxes



(a) Rango 10

(b) Rango 50

(c) Rango 100

(d) Rango 200

(e) Rango 400

Figura 4.4: Gráficos de consultas por quadboxes para varios tamaños de rango.

Dado que las consultas funcionan a base de una descomposición en quadboxes, se realizaron experimentos donde las ventanas de consulta eran exactamente un quadbox. Los resultados están en la figura 4.4. Nótese que los tamaños de ventana fueron cambiados por tamaños de quadboxes (16x16, 64x64, 128x128, 256x256 y 512x512)

mientras que los tamaños de rango se mantuvieron. Los resultados son el promedio de realizar consultas por todos los posibles quadboxes del tamaño de la ventana en particular, repitiendo cada consulta 20 veces.

En este caso especial se ve que, para consultas por ventanas de más de 50x50, independiente del rango, nuestra estructura K2T reporta mejores resultados que el estado del arte.

4.5 Análisis de los Resultados

Los resultados mostrados en la tabla 4.3 muestran que las alternativas ZZ ST y K2T tienen gran potencial de compresión, siendo muy cercanas a los valores presentados en el estado del arte (superando a k2-acc en el *dataset* MDT05-700).

En cuanto a las consultas de acceso también muestran gran similaridad con el estado del arte obteniendo un mejor desempeño en ambos *datasets* para la alternativa K2T.

Las consultas por una subregión muestran que ZZ BHE y ZZ ST tienden a demorar mucho más que las otras alternativas, incluidas las del estado del arte, a medida que aumenta el tamaño de consulta. Sin embargo K2T tiene un desempeño que supera al estado del arte y crece menos rápido a medida que aumenta el tamaño de la subregión. Si bien la consulta WT-count tiene un buen desempeño, no resuelve la misma consulta que las demás estructuras de modo que su rendimiento no es comparable. Los resultados expuestos por WT-report (la consulta por una subregión que sí devuelve los valores consultados) tiene un muy mal desempeño y no fue incluida en los gráficos para que las demás curvas fueran apreciadas.

Ahora bien, cuando la consulta por una subregión es un quadbox, los resultados cambian favoreciendo el comportamiento de nuestras estructuras por sobre el estado del arte debido principalmente a que, al ser un solo quadbox, en ZZ BHE se traduce en recorrer una Z, operación menos costosa que acceder a varios quadboxes o encontrar la siguiente celda de entrada (operaciones de las 2 alternativas de consulta por una subregión).

Finalmente, las consultas por una subregión con rango muestran que, a medida que el rango aumenta, el comportamiento de K2T supera al entregado en el estado

del arte, esto se puede apreciar en los gráficos c, d y e de la figura 4.3.

En el caso de las consultas por una subregión con rango, donde la región consultada es un quadbox, se puede apreciar que para rangos muy pequeños, el estado del arte tiene un buen comportamiento, pero para rangos más grandes K2T mejora los resultados.

Nótese que en las consultas por quadboxes los resultados de WT-count superan ampliamente a K2T creando la idea de que esta estructura debería ser mejor que K2T en el caso general, pero esto no se ve reflejado en los resultados generales. Esto es debido a que cuando los quadbox son demasiado pequeños (1x1, 2x2, 4x4, 8x8) el K2T tiene un comportamiento mejor y, dado que las consultas son una descomposición en quadboxes, esta descomposición resulta en pocos quadboxes grandes y muchos pequeños, dándole la ventaja al K2T.

Capítulo 5

Conclusiones y Trabajo Futuro

En esta tesis, se exponen varias estructuras de datos que son eficientes en espacio para representar matrices con localidad espacial y mantienen buenos tiempos de consulta para varias operaciones. En particular, proponemos 2 alternativas, K2T y ZZ ST, cuyos usos de espacio son muy similares (aunque ligeramente superiores) a los expuestos en el estado del arte. Además, en el acceso a datos, nuestras estructuras tienen muy buen comportamiento, llegando el K2T a superar en tiempo de acceso a datos a lo propuesto en el estado del arte. En lo que se refiere a consultas por subregiones, presentamos resultados que tienen un buen comportamiento cuando las regiones consultadas son grandes (400x400). Lo mismo sucede para cuando tenemos rangos de valores posibles como restricción a las consultas por subregiones. Un caso particular de las consultas por subregiones (tanto con y sin rango) es cuando éstas son un *quadbox*, en cuyo caso, el desempeño de nuestras estructuras es mucho más eficiente que lo propuesto en el estado del arte.

Como trabajo futuro, queda lidiar con las consultas de rango sobre el ZZ ST, el árbol binario podado. Además, dar soporte a las estructuras de consultas agregadas tales como: máximo, mínimo y promedio de una región.

Actualmente estamos trabajando en la implementación de consultas de rango sobre el ZZ ST, el árbol binario podado, y en una evaluación experimental que incluya tanto la propuesta recientemente publicada en [5] como un estudio de la influencia de la precisión de los valores en las distintas alternativas (esto favorece tanto a nuestra propuesta como a la realizada en [5] con respecto al *baseline* considerado en esta tesis). Como trabajo futuro se proponen varias líneas. En primer lugar se propone implementar y evaluar experimentalmente la propuesta que realizamos en la sección 3.1.5.3 para mejorar la eficiencia de las consultas de rango sobre el ZZ ST cuando el rango de valores es grande. En segundo lugar, tal y como se propone en la sección

4.1 sería interesante estudiar el comportamiento del WT cuando se emplean técnicas de compresión sólo en los niveles superiores del árbol (ya que en los niveles inferiores se pierden las regularidades y los datos son incompresibles, añadiendo el tratar de emplear compresión una sobrecarga en espacio y tiempo innecesaria). Aunque el tiempo de consulta del WT en nuestra evaluación experimental puede no resultar muy convincente, es importante destacar que es la única estructura de las evaluadas que soporta una optimización directa para consultas de tipo conteo ("range counting"), muy utilizadas en varios dominios. Además, sería interesante estudiar cómo se pueden optimizar las operaciones reporte ("range reporting") sobre dicha estructura para poder ofrecer una solución completa con ella. Las consultas de tipo conteo se enmarcan en un dominio más general de consultas de rango agregadas (donde también se incluye mínimo/máximo, promedio, top-k, entre otras). Por tanto, otra línea interesante es el estudio de la generalización de las distintas estructuras descritas en esta tesis para soportar consultas agregadas. Por último, tal y como se menciona en la introducción, hay ciertos dominios donde interesa almacenar varias muestras de la información espacial a lo largo del tiempo (ejemplo, temperatura cada hora del día). En dichos dominios existe no sólo localidad espacial (la cual estudiamos en esta tesis) sino también localidad temporal (es decir, es esperable que, a lo largo del tiempo, los valores varían lentamente). Las estructuras de datos estudiadas en esta tesis consideran cada muestra temporal por separado, por lo que no pueden explotar dicha regularidad, por lo que sería interesante estudiar cómo extender las propuestas realizadas para aprovechar dicha regularidad.

Bibliografía

[1] Worboys, Michael, and Matt Duckham. (2004) GIS: a computing perspective.

[2] Tobler, W. R (1970). A computer model simulation of urban growth in the Detroit region. Economic Geography. p.236.

[3] Claude, F., Nicholson, P. K., & Seco, D. (2014). On the compression of search trees. Information Processing & Management, 50(2), 272-283.

[4] De Bernardo, G., Álvarez-García, S., Brisaboa, N. R., Navarro, G., & Pedreira, O. (2013, October). Compact querieable representations of raster data. In International Symposium on String Processing and Information Retrieval (pp. 96-108). Springer International Publishing.

[5] Ladra, S., Paramá, J. R., & Silva-Coira, F. (2016, July). Compact and queryable representation of raster datasets. In Proceedings of the 28th International Conference on Scientific and Statistical Database Management (p. 15). ACM.

[6] Brisaboa, N. R., Ladra, S., & Navarro, G. (2009, August). k2-trees for compact web graph representation. In International Symposium on String Processing and Information Retrieval (pp. 18-30). Springer Berlin Heidelberg.

[7] Navarro, G. (2016). Compact Data Structures: A Practical Approach. Cambridge University Press.

[8] Grossi, R., Gupta, A., & Vitter, J. S. (2003, January). 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.

[9] Grossi, R., Gupta, A., & Vitter, J. S. (2004, January). When indexing equals compression: Experiments with compressing suffix arrays and applications. In Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms (pp. 636-645). Society for Industrial and Applied Mathematics.

[10] Mäkinen, V., & Navarro, G. (2006, March). Position-restricted substring searching. In Latin American Symposium on Theoretical Informatics (pp. 703-714). Springer Berlin Heidelberg.

[11] Gagie, T., Navarro, G., & Puglisi, S. J. (2012). New algorithms on wavelet trees and applications to information retrieval. Theoretical Computer Science, 426, 25-41.

[12] Navarro, G., & Sadakane, K. (2014). Fully functional static and dynamic succinct trees. ACM Transactions on Algorithms (TALG), 10(3), 16.

[13] Brisaboa, N. R., Ladra, S., & Navarro, G. (2014). Compact representation of web graphs with extended functionality. Information Systems, 39, 152-174.

[14] Sagan, H. (1994). Space-filling curves, vol. 18.

[15] Morton, G. M. (1966). A computer oriented geodetic data base and a new technique in file sequencing. New York: International Business Machines Company.

[16] Proietti, G. (1999). An optimal algorithm for decomposing a window into maximal quadtree blocks. Acta Informatica, 36(4), 257-266.

[17] Tsai, Y. H., Chung, K. L., & Chen, W. Y. (2004). A strip-splitting-based optimal algorithm for decomposing a query window into maximal quadtree blocks. IEEE Transactions on Knowledge and Data Engineering, 16(4), 519-523.

[18] VBytes: Williams, H. E., Zobel, J., 1999. *Compressing integers for fast file access. The Computer Journal 42 (3), 193–201.*

[19] Brisaboa, N. R., Ladra, S., & Navarro, G. (2013). DACs: Bringing direct access to variable-length codes. Information Processing & Management, 49(1), 392-404.

[20] "Encoding - Protocol Buffers", 2012. [Online]. Disponible: https://developers.google.com/protocol-buffers/docs/encoding

[21] Modelos digitales de terreno [Online]. Disponible: http://www.ign.es/ign/layoutIn/modeloDigitalTerreno.do

[22] Tropf, H., & Herzog, H. (1981). Multidimensional Range Search in Dynamically Balanced Trees. ANGEWANDTE INFO., (2), 71-77.

[23] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 233-242. SIAM Press, 2002.

[24] Proietti, G. (1999). An optimal algorithm for decomposing a window into maximal quadtree blocks. Acta Informatica, 36(4), 257-266.

[25] Tsai, Y. H., Chung, K. L., & Chen, W. Y. (2004). A strip-splitting-based optimal algorithm for decomposing a query window into maximal quadtree blocks. IEEE Transactions on Knowledge and Data Engineering, 16(4), 519-523.

[26] Brisaboa, N. R., De Bernardo, G., Konow, R., Navarro, G., & Seco, D. (2016). Aggregated 2D range queries on clustered points. Information Systems, 60, 34-49.

[27] P. Alejandro Pinto. Estructura de datos eficiente en espacio para matrices con localidad espacial (2015). Memoria de título, Universidad de Concepción.