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

Quadtrees eficientes en espacio para conjuntos de datos con clústers

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

JAVIER IGNACIO GONZÁLEZ NOVA
CONCEPCIÓN-CHILE
2017

Profesor Guía: Diego Seco Naveiras
Dpto. de Ingeniería Informática y Ciencias de la Computación, Facultad de Ingeniería
Universidad de Concepción

Resumen

Poder almacenar en memoria principal conjuntos de puntos de dos dimensiones para realizar consultas sobre ellos (por ejemplo, saber si un punto está en el conjunto o qué puntos están en un cierto rango. Un rango son todos los puntos tales que $x \in [x_1, x_2]$ y $y \in [y_1, y_2]$, con x e y coordenadas y $x_1 < x_2; y_1 < y_2$.) es útil en varias áreas de la computación, tales como geometría computacional, sistemas de información geográfica, gráficos, etc.

En esta tesis se explorarán estructuras sucintas (estructuras que no necesitan ser descomprimidas para realizar consultas sobre ellas) porque tener la estructura en niveles superiores de la jerarquía de memoria supone un incremento en la velocidad de procesamiento (que puede llegar a ser órdenes de magnitud en el caso de pasar de disco a RAM), aún cuando realizar operaciones sobre este tipo de estructuras suele ser más costoso que en las estructuras clásicas, asumiendo que ambas estructuras pudiesen estar contenidas en memoria principal. En base a esto, se propondrá una estructura alternativa que mejore las actuales.

La estructura propuesta es una representación comprimida de quadtrees que aprovecha los clústers que forman los elementos. Esta estructura intenta mejorar el problema que presenta el $k^2 - tree$, el cual ocupa mucho espacio para almacenar matrices esparsas. Se espera también mejorar el tiempo de las consultas (buscar puntos en el espacio) con respecto al $k^2 - tree$.

Contenidos

Lista de figuras

Lista de tablas

Capítulo 1

Introducción

Poder almacenar en memoria principal conjuntos de puntos de dos dimensiones para realizar consultas sobre ellos (por ejemplo, saber si un punto está en el conjunto o qué puntos están en un cierto rango. Un rango son todos los puntos tales que $x \in [x_1, x_2]$ y $y \in [y_1, y_2]$, con x e y coordenadas y $x_1 < x_2; y_1 < y_2$.) es útil en varias áreas de la computación, tales como geometría computacional, sistemas de información geográfica, gráficos, etc. Un ejemplo práctico es la localización de lugares geográficos. Estos pueden ser representados por coordenadas en dos dimensiones y luego es posible obtener información acerca de lugares ubicados en una cierta zona geográfica (por zona se entiende un rango de coordenadas).

Considerar el siguiente ejemplo. Se tienen las siguientes zonas geográficas del país Chile: Volcán Copahué $(-37.85, -71.167)$, Cerro Larancagua $(-18.0167, -69.083)$ y Cerro Prieto $(-19.283, -68.63)$. Las coordenadas son definidas por una tupla (Latitud, Longitud). Estas coordenadas pueden ser representadas en una matriz transformando cada punto (Latitud, Longitud) a una posición en la matriz. Suponer que las posiciones en la matriz son $(2, 6)$, $(3, 5)$ y $(0, 5)$, respectivamente. La matriz es como se muestra en la **Figura 1.1**.

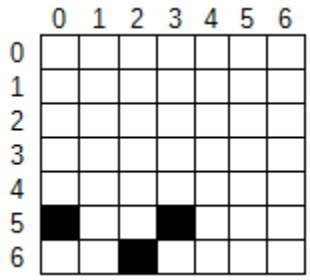


Figura 1.1: Una representación de matriz de una zona geográfica.

Esta matriz puede ser la representación de un mapa. Un usuario puede querer saber qué hay en cierta zona del mapa. Si la consulta es en la zona $(2, 6)$, la respuesta será Volcán Copahué. Notar que un lugar geográfico puede estar compuesto de varias casillas en la matriz. Si la consulta es por el punto $(0, 0)$ entonces la respuesta será que no hay nada en esa zona.

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

Figura 1.2: Matriz de 5x5 donde cada casilla tiene un valor de 1 o 0. Un 1 significa que existe un elemento en esa casilla, mientras que no hay nada si es un 0. Ejemplos de consultas: a) ¿Cuántos elementos existen en la región verde? **Respuesta: 3**. b) ¿Existe un elemento en la posición celeste? **Respuesta: Sí.**

La razón de buscar una estructura que ocupe poco espacio es porque existen varios niveles en la jerarquía de memoria (caches que permiten a la CPU obtener los datos más rápidamente que irlos a buscar a memoria principal) y ser capaces de almacenar una estructura en niveles superiores de la jerarquía supone un incremento en la velocidad de procesamiento (que puede llegar a ser órdenes de magnitud en el caso de pasar de disco a RAM), aún cuando realizar operaciones sobre este tipo de

estructuras suele ser más costoso que en las estructuras clásicas (estructuras sin compresión, como arreglos, listas, etc.), asumiendo que ambas estructuras pueden estar contenidas en memoria principal.

Desde un punto de vista tradicional de diseño de estructuras de datos, se busca diseñar estructuras de datos cuyo tamaño, medido en palabras de máquina, sea lineal al número de puntos. Por ello, se considera que una estructura de datos ocupa espacio óptimo si es capaz de almacenar n puntos de una matriz de universo u (dimensión de mayor tamaño de la matriz, $u = 5$ en el caso de la **Figura 1.2**) en $O(n)$ palabras de tamaño $O(\log u)$ bits cada una. En total la estructura ocupa $O(n \log u)$ bits. Si además se considera que los puntos están agrupados entonces se puede reducir el espacio mediante estructuras compactas.

Diseñar una estructura comprimida implica que para realizar consultas sobre la estructura, ésta se debe descomprimir, ocupando más espacio y tiempo. Para evitar este problema, existen las **estructuras de datos sucintas** que buscan poder realizar consultas directamente sobre la estructura comprimida. En este trabajo, se busca una estructura de datos sucinta que resuelva el problema antes mencionado y que obtenga mejores resultados que el estado del arte para la representación de matrices binarias comprimidas.

Existen en el estado del arte varias estructuras sucintas que resuelven el problema de almacenar matrices binarias. Las representaciones de quadtrees comprimidos tienen buenos resultados. Ejemplos son el Linear quadtree [14] y el k^2-tree [1]. Este último presenta los mejores resultados hasta el momento [11]. Ambas estructuras se basan en representar los índices de la matriz de forma binaria y aprovechar la localidad de referencia (puntos cercanos comparten un prefijo común en su representación binaria, es decir, los primeros n bits de ambos puntos son los mismos).

La propuesta es diseñar una representación alternativa a las actuales de quadtrees

comprimidos. La estructura se basa en la siguiente hipótesis: "Un método para comprimir rutas (secuencias de nodos) o moverse por múltiples aristas a la vez usando estructuras sucintas puede mejorar los algoritmos que se basan en navegar por un quadtree" [15].

Se propone una estructura que además de ocupar poco espacio, aprovecha el hecho de que las coordenadas estén agrupadas en una región en particular. Esto se logra al representar cada coordenada de forma binaria. De esta forma, si dos o más pares de puntos pertenecen a una misma región del universo compartirán parte de su representación binaria [14]. El proceso consiste en representar cada coordenada de tal forma que sea posible aprovechar el hecho de que puntos cercanos entre sí tengan información en común y así ocupar poco espacio. Una vez almacenados debe ser posible realizar consultas (saber si una coordenada está en la estructura, por ejemplo), la idea es que las consultas dependan de la cantidad de puntos en la estructura.

La estructura propuesta demuestra como, abordando el almacenamiento de la matriz desde otro punto de vista, es posible mejorar ciertas falencias de las estructuras del estado del arte. Esto se logra al representar los puntos por sus coordenadas y no por su posición en la matriz (como lo hace el $k^2 - tree$ al describir un punto en base al cuadrante donde se ubica). De esta forma regiones muy esparsas de la matriz (con muy pocos 1's en comparación con los 0's) se pueden representar con mucho menor espacio que el $k^2 - tree$. Además, se agiliza el tiempo de consulta ya que la estructura almacena directamente los puntos, lo que implica no tener que buscar en la región. En una matriz muy esparsa, hay que recorrer un gran espacio vació para encontrar el punto, en cambio si se tienen representados directamente los puntos, solo se tiene que iterar sobre ellos, sin importar que tan grande es la región.

El documento se divide en cinco capítulos. En el primer capítulo, se detallan todas las estructuras, técnicas y algoritmos usados por la solución propuesta. Además se explica el funcionamiento de una de las estructuras de datos que se usa como línea

base de comparación en posteriores experimentos. El siguiente capítulo explica la solución. Aquí se proponen dos alternativas a la estructura de datos diseñada. El tercer capítulo presenta resultados experimentales, el quinto presenta una propuesta de estructura que busca combinar lo mejor del k^2-tree y la ya mencionada estructura HP. Finalmente, el último capítulo concluye el trabajo. Además presenta las líneas de trabajo futuro, que incluyen el extender a tres dimensiones la estructura.

Capítulo 2

Conceptos previos y trabajo relacionado

2.1 Relación d-aria

Se define relación como una tupla o lista ordenada de elementos que cumplen un criterio específico. Por ejemplo, es posible definir una relación binaria ($d = 2$) usando el sistema de coordenadas en dos dimensiones. Dado R como una relación binaria que representa si un punto en el plano tiene o no información relevante se pueden tener casos como $R(2, 3) = F$ (no hay nada en ese punto) o $R(1, 1) = V$ (hay información relevante en dicho punto). Las estructuras y algoritmos propuestos en este documento se basan en relaciones binarias, dejando como trabajo futuro relaciones terciarias ($d = 3$), como coordenadas espaciales o coordenadas bidimensionales en el tiempo, y relaciones de mayor orden de dimensionalidad.

2.2 Trie

Estructura de datos con forma de árbol, útil cuando se almacenan cadenas de texto [5]. Permite obtener todas las palabras que tengan un prefijo en común fácilmente, por ello también se le llama árbol de prefijos. Otra ventaja es el ahorro de espacio cuando el árbol contiene una o más palabras con prefijo común. Por ejemplo, suponer que se quiere armar un trie con las siguientes palabras (asumimos un alfabeto binario): 0010, 0100, 1000, 0011. Al insertar 0010 en el árbol queda como en la **Figura 2.1**.

Figura 2.1: Cuatro nodos tienen que crearse para almacenar el 0010

Al insertar una nueva palabra, ésta no necesariamente ocupará cuatro nodos (o tantos nodos como caracteres tenga la palabra), sino que ocupará tantos nodos como caracteres no estén incluidos en el prefijo más largo que esté contenido en la palabra a insertar, como se muestra en la **Figura 2.2**.

Figura 2.2: Solo un nodo debe ser agregado ya que 0011 tiene prefijo común 001 con 0010

La **Figura 2.3** muestra el trie resultante al insertar las palabras restantes 0100 y 1000.

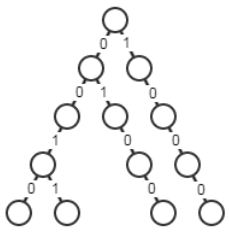


Figura 2.3: Trie con las cuatro palabras insertadas

2.3 Quadtree

El quadtree es una estructura de datos que representa una partición del espacio en dos dimensiones descomponiendo la región en cuatro cuadrantes iguales. Estos cuadrantes se dividen en subcuadrantes y así sucesivamente hasta tener regiones que contienen un único elemento [7]. Cada nodo en el árbol tiene exactamente cuatro hijos o ninguno en el caso de las hojas. Como veremos más adelante, el quadtree es un caso particular de trie.

Un quadtree con una profundidad de n puede ser usado para representar un arreglo de $2^n \times 2^n$. Si este arreglo está compuesto de solo 0's (1's), entonces el quadtree consiste en una raíz (representando todo el arreglo) y, si alguna región (cuadrante) no está completamente llena con 1's o 0's, entonces se crea una subdivisión.

Entre los usos que se le pueden dar al quadtree están la representación de imágenes, información como la temperatura en una región del espacio o representar un conjunto de puntos (como la latitud y longitud de un conjunto de ciudades).

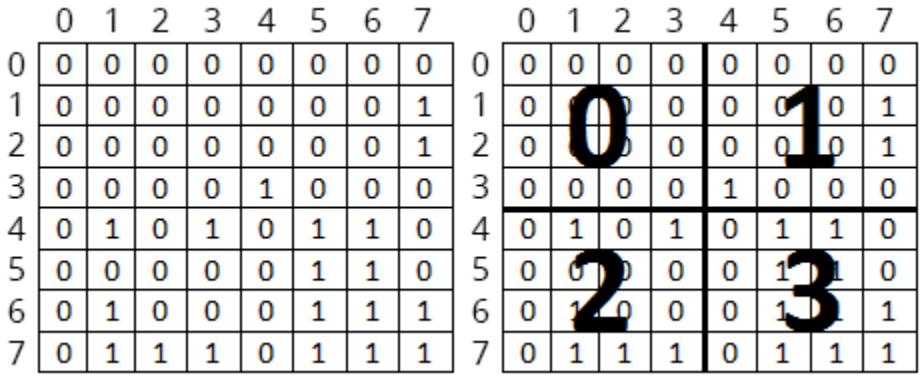
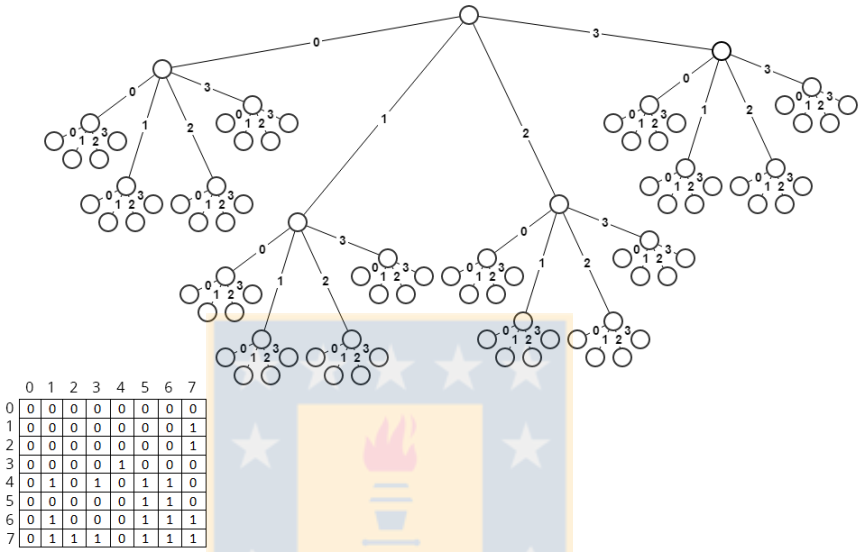


Figura 2.4: Arreglo de dos dimensiones donde cada casilla toma los valores 1 o 0. La imagen de la derecha muestra cómo se dividen los cuadrantes. Esta división es útil para hacer el seguimiento de cada cuadrante en el quadtree [3].

A continuación se muestra un ejemplo de un quadtree correspondiente a una matriz de $2^3 \times 2^3$ y cómo se encuentra un elemento de la matriz sabiendo los cuadrantes que hay que usar. En el ejemplo, se busca el punto (7, 2) y el recorrido es {1 3 1}.



	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	1
3	0	0	0	0	1	0	0	0
4	0	1	0	1	0	1	1	0
5	0	0	0	0	0	1	1	0
6	0	1	0	0	0	1	1	1
7	0	1	1	1	0	1	1	1

Figura 2.5: Representación de una matriz de 8×8 en un quadtree. El hijo de más a la izquierda de la raíz no es necesario puesto que toda esa región está compuesta solo de 0's

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	1
3	0	0	0	0	1	0	0	0
4	0	1	0	1	0	1	1	0
5	0	0	0	0	0	1	1	0
6	0	1	0	0	0	1	1	1
7	0	1	1	1	0	1	1	1

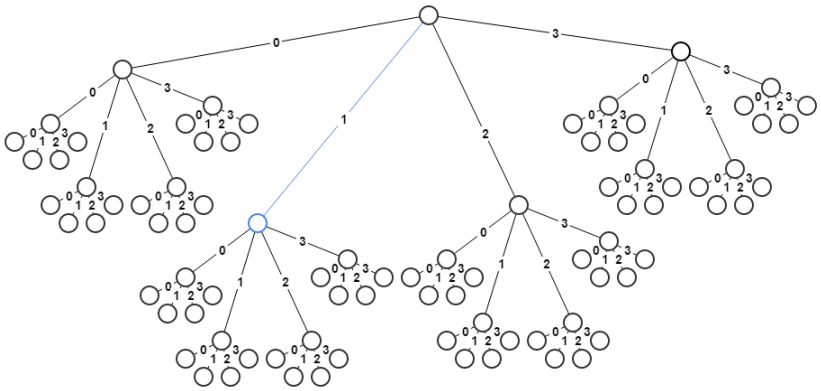
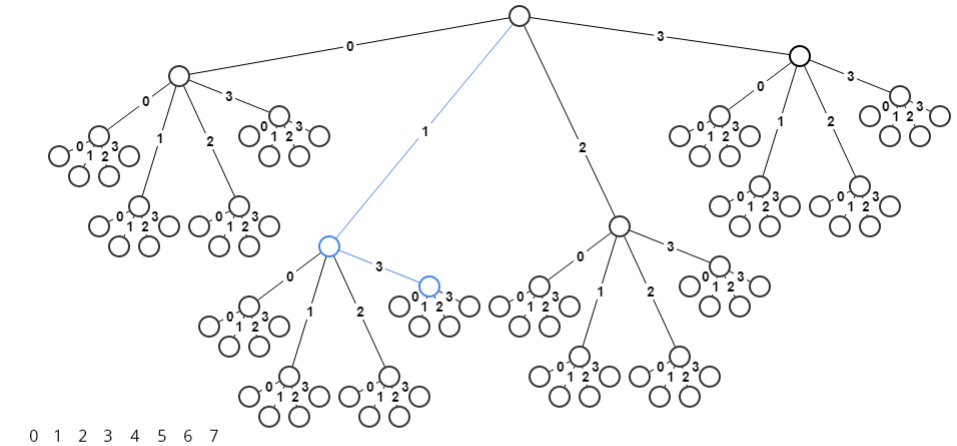
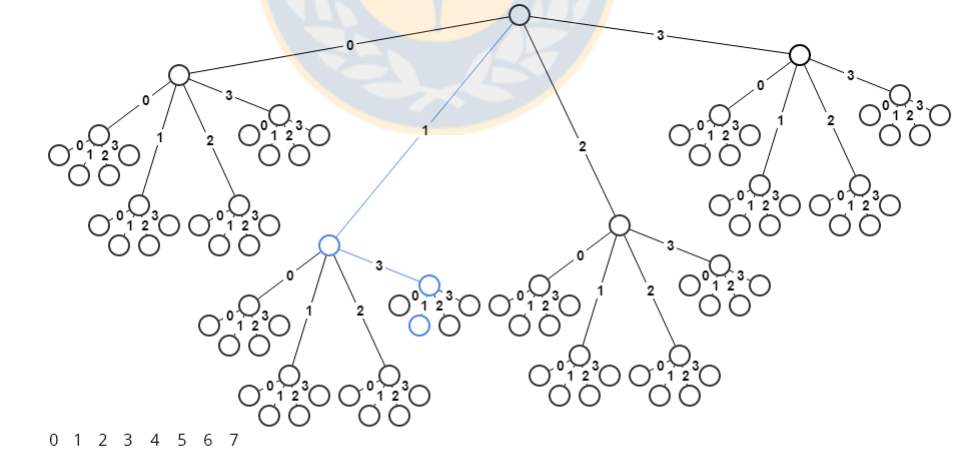


Figura 2.6: Descender un nivel en el quadtree limita la búsqueda a una matriz de 4×4



	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	1
3	0	0	0	0	1	0	0	0
4	0	1	0	1	0	1	1	0
5	0	0	0	0	0	1	1	0
6	0	1	0	0	0	1	1	1
7	0	1	1	1	0	1	1	1

Figura 2.7: Descender otro nivel limita la búsqueda a una matriz de 2×2



	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	1
3	0	0	0	0	1	0	0	0
4	0	1	0	1	0	1	1	0
5	0	0	0	0	0	1	1	0
6	0	1	0	0	0	1	1	1
7	0	1	1	1	0	1	1	1

Figura 2.8: En tres iteraciones se llega a una hoja, en este caso es el punto (7, 2)

2.3.1 Linear quadtree

Como se describe en [14], un quadtree puede ser representado sin punteros (implementación clásica de un árbol). En un **linear quadtree**, dada una matriz binaria de $2^m \times 2^m$ donde el valor de cada celda es 0 o 1, cada punto es codificado como las sucesivas subdivisiones de cuadrantes necesarios para localizar dicho punto (ver sección 2.3).

El **linear quadtree** presenta las siguientes características:

1. Solo los 1's son almacenados.
2. La codificación usada tiene propiedades de adyacencia (puntos en un mismo cuadrante tienen similitudes).
3. La representación implícitamente codifica la ruta desde la raíz al nodo.
4. La complejidad y el espacio dependen del número de 1's.
5. Los punteros no son necesarios.

Cada punto (celda con valor 1) es codificado usando las divisiones antes mencionadas (ver sección 2.5). Luego, los códigos son almacenados en una lista y son ordenados.

Buscar un punto en la estructura implica realizar una **búsqueda binaria** sobre la lista.

2.4 K^2-tree

2.4.1 Definición

Estructura de datos pensada para comprimir el grafo de la web [1]. La matriz de adyacencia de un grafo de la web de n páginas es una matriz cuadrada $\{a_{ij}\}$ de $n \times n$ donde cada columna y cada fila representa una página de la web. Una celda a_{ij} es 1

si hay un hipervínculo en la página i hacia la página j.

Está diseñada para obtener mejores resultados en matrices esparsas y con clústers ya que comprime áreas de 0's con muy pocos bits. Se puede ver como una versión eficiente en espacio de un quadtree.

La matriz de adyacencia se representa por un árbol de altura $h = \lceil \log_{k^2} n \rceil$. Cada nodo contiene un bit de información: 1 para nodos internos y 0 para las hojas, excepto para las del último nivel. Los cuales almacenan valores de la matriz.

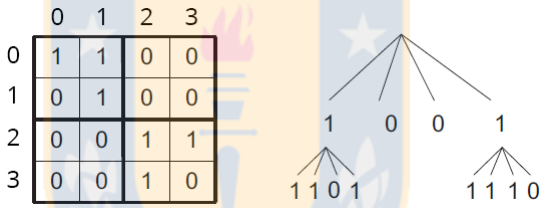


Figura 2.9: Representación de la matriz de adyacencia de un grafo de la web

La **Figura 2.9** muestra un ejemplo. Cada nodo interno tiene cuatro hijos o ninguno. En el ejemplo, los cuatro hijos de la raíz corresponden a cada cuadrante de la matriz de adyacencia. El hijo de más a la izquierda tiene un bit en 1 porque el cuadrante superior izquierdo contiene valores distintos de 0. En cambio, el segundo hijo de la raíz (de izquierda a derecha) contiene un bit en 0 porque el cuadrante superior derecho no contiene 1's. Notar que los cuadrantes siguen el siguiente orden: superior izquierdo, superior derecho, inferior izquierdo, inferior derecho. Si un nodo interno contiene un 1, éste debe tener cuatro hijos, correspondientes a los subcuadrantes de dicho cuadrante. Por ejemplo, los cuatro hijos del hijo de más a la izquierda de la raíz (los cuales son 1 1 0 1) corresponden a subdividir el cuadrante superior izquierdo (en cuatro subcuadrantes). Como esta división resulta en cuadrantes compuestos de

solo una celda de la matriz de adyacencia, ya no se sigue con la división porque se llegó a un valor de la matriz (una hoja).

Para el caso particular de la **Figura 2.9** cada nodo interno contiene cuatro hijos (o ninguno) pero en general éste depende de una variable k definida por el $K^2 - tree$. En la **Figura 2.9**, $k = 2$.

2.4.2 Navegar por el $K^2 - tree$

Para obtener las páginas a las que apunta una página p (vecinos directos de p), es necesario encontrar los 1's en la fila p de la matriz.

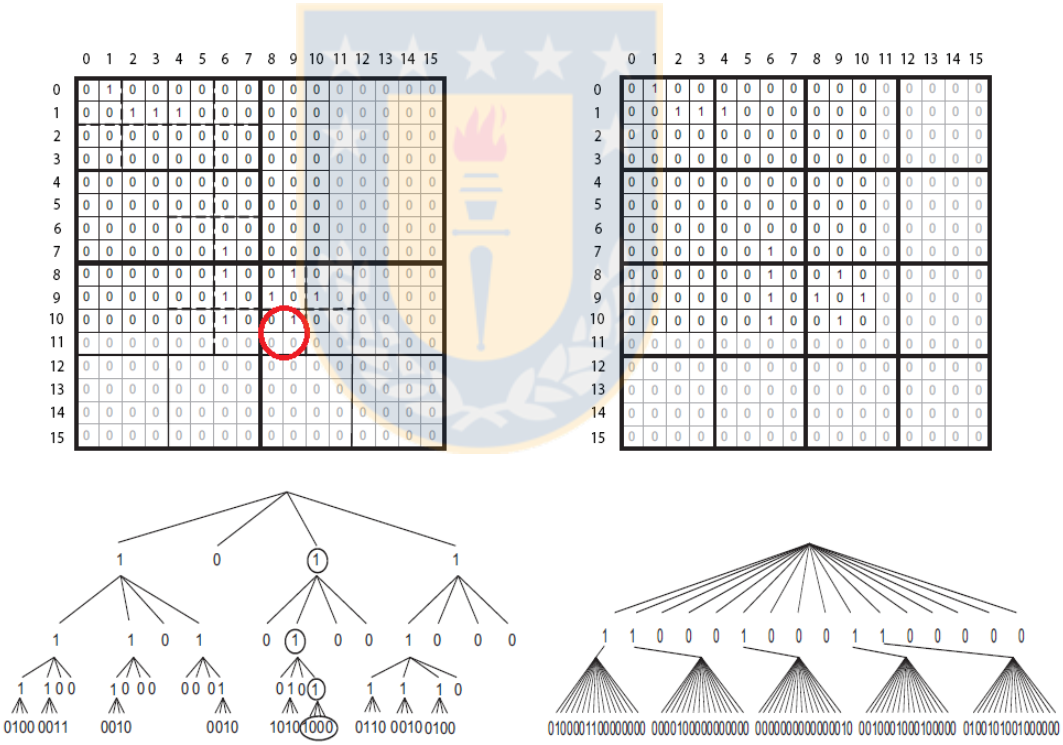


Figura 2.10: Ejemplos de $K^2 - tree$ para $k = 2$(izquierda) y $k = 4$(derecha). Notar que para $k = 4$ hay 16 divisiones. En la imagen de la izquierda se muestra cómo se desciende en el árbol. El circulo rojo muestra las casillas que representan las hojas a las que se llegaron.

Ejemplo: Encontrar las páginas a las que apunta la primera página en el ejemplo

de la **Figura 2.9** (encontrar los 1's en la primera fila de la matriz).

1. Se comienza en la raíz y se baja por los hijos que se solapen con la primera fila (cuadrantes 0 y 1), los cuales corresponden a los dos primeros hijos.

2. El primer hijo es un 1, así que tiene hijos. Para saber cuáles de estos hijos son necesarios se repite el mismo proceso. Los dos primeros hijos solapan con la primera fila y son hojas. Los valores de estas hojas son 1 y 1.

3. El segundo hijo es un 0, así que ese cuadrante está lleno de 0's.

4. Por lo tanto, la página 0 apunta a sí misma y a la página 1.

La implementación de la estructura se basa en almacenar el árbol como dos bitmaps (uno para los nodos internos y otro para las hojas) y realizar operaciones de bit rank/select (que se verán en la sección 2.9) para resolver las consultas.

2.5 Quadcode

Representación binaria de un punto en un quadtree (ver sección 2.3). La **Figura 2.11** muestra un quadtree, cada arista está etiquetada con un número del 0 al 3 que indica uno de los cuatro cuadrantes (o subcuadrantes). El orden de los cuadrantes es como sigue: 0 (superior izquierdo), 1 (superior derecho), 2 (inferior izquierdo), 3 (inferior derecho). La ruta coloreada en celeste representa al punto (1, 2). Una forma de transformar esta ruta [2, 1] a una representación binaria es usando directamente cada etiqueta de la ruta como código binario. De este modo, el quadcode para el punto (1, 2) es 1001.

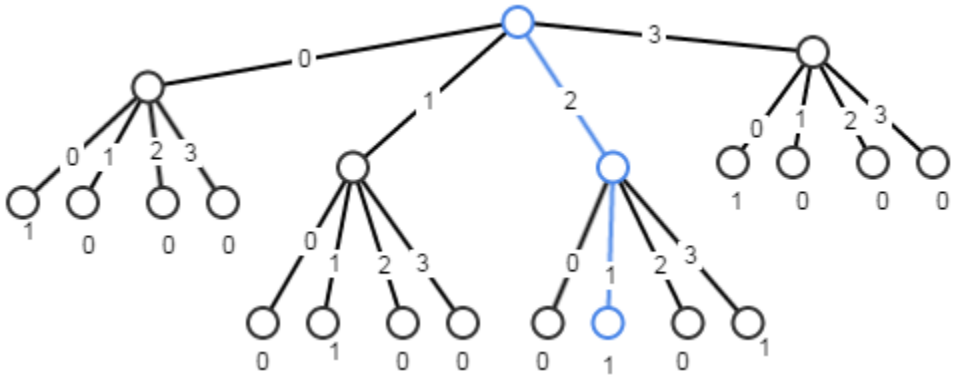


Figura 2.11: .
Esto significa ir al cuadrante 2 (valores entre 2 y 3 para el eje Y, 0 y 1 para el eje X). Luego, al subcuadrante 1 (valor 1 para X, 2 para Y). Por lo tanto, la ruta representa al punto (1, 2)]Quadtree donde cada arista está etiquetada con un número del 0 al 3 indicando el cuadrante al cual pertenece. Dada una matriz de 4x4, la ruta en celeste es [2, 1]. Esto significa ir al cuadrante 2 (valores entre 2 y 3 para el eje Y, 0 y 1 para el eje X). Luego, al subcuadrante 1 (valor 1 para X, 2 para Y). Por lo tanto, la ruta representa al punto (1, 2)

El mismo resultado se puede obtener a partir del punto (1, 2). Se toma la representación en binario de cada coordenada, en este caso (01, 10). Luego, se usa **entrelazado de bits** entre las coordenadas [3]. El **entrelazado de bits** es un método para transformar dos o más bitmaps en uno solo. El método consiste en ir intercalando cada bit de cada bitmap siguiendo algún orden, como se muestra en la **Figura 2.12**.

x	y	Entrelazado (pasos)
01	10	
01	10	1
01	10	10
01	10	100
01	10	1001

Figura 2.12: Entrelazado de bits. A partir de 01 y 10 se obtiene 1001. En cada paso, el bit coloreado en rojo se agrega al final del bitmap resultante.

La ventaja que tienen los quadcodes es que esta representación hace que puntos que estén cerca (en el mismo cuadrante) compartan un prefijo común. Por ejemplo el quadcode de (1, 2) es 1001 y el de (1, 3) es 1011, tienen prefijo común 01.

2.6 Morton encoding/decoding y lookup table

El código de Morton es una función que mapea datos multidimensionales a una dimensión preservando la localidad de los puntos. El código de Morton de un punto es el resultado de **entrelazar los bits** (ver sección 2.5) de la representación binaria de cada punto. El orden del código de Morton de un punto es equivalente al orden que se obtiene al realizar un recorrido en profundidad en un quadtree.
El código de Morton de un punto puede ser calculado en tiempo constante usando tablas precomputadas.

Método Lookup Table

Método de **dividir y conquistar**. Se precomputan todos los códigos de Morton para los primeros 2^8 enteros, que son todos los números que pueden ser almacenados en 1 byte. Luego, para conseguir el código de Morton de un punto (x, y), cada coordenada se divide en cuatro (asumiendo que cada coordenada es un entero de 4 bytes) y se usa la tabla precomputada para obtener los códigos de cada byte. El código de Morton es el resultado de unir lo obtenido en x e y mediante una operación de bits OR. El costo de tener una de estas tablas en memoria es de $256 * 32bits = 8kb$, lo cual es despreciable en la práctica. A continuación se muestra la tabla con algunos valores precomputados (en hexadecimal).

```
MortonTable256[256] =
{
  0x0000, 0x0001, 0x0004, 0x0005, 0x0010, 0x0011, 0x0014, 0x0015,
  0x0040, 0x0041, 0x0044, 0x0045, 0x0050, 0x0051, 0x0054, 0x0055,
  0x0100, 0x0101, 0x0104, 0x0105, 0x0110, 0x0111, 0x0114, 0x0115,
  0x0140, 0x0141, 0x0144, ...
```

```
}
```

Por ejemplo, si se quiere obtener el código de Morton del punto (1, 2). Se sabe que la representación binaria de 1 es 01 y la de 2 es 10. En la **Figura 2.13** se da una idea de cómo funciona la tabla precomputada.

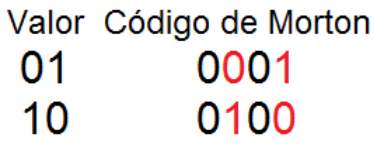


Figura 2.13: Cada valor en la tabla precomputada representa el byte original, pero con 0's agregados entre bits.

Una vez se tienen los valores precomputados de cada byte basta con un par de operaciones sobre bits para obtener el código de Morton. Como se muestra en la **Figura 2.14**.

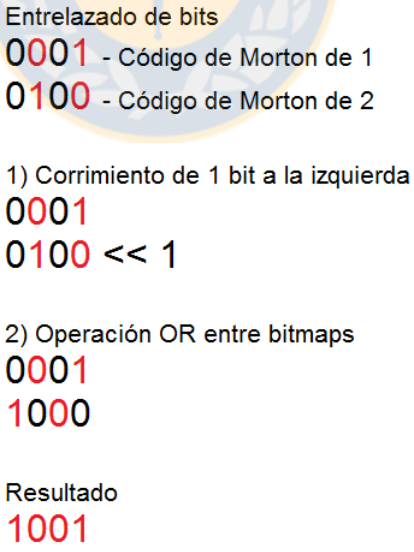


Figura 2.14: Colocando 0's entre los bits de cada coordenada y luego agregando un 0 a la derecha de una de las coordenadas permite generar el código de Morton al utilizar la operación OR entre ambas.

2.7 Heavy path decomposition

Técnica para descomponer un árbol con raíz en un conjunto de **heavy paths**. Un **heavy path** es una secuencia de **heavy edges**. Un **heavy edge** es la arista cuyo subárbol tiene la mayor cantidad de hojas [4]. El resultado del algoritmo es un **Path tree**.

Path tree

Árbol donde cada nodo representa un **heavy path**. Si p es un **heavy path**, entonces el padre de p es el **heavy path** que contiene al padre del primer elemento de p.

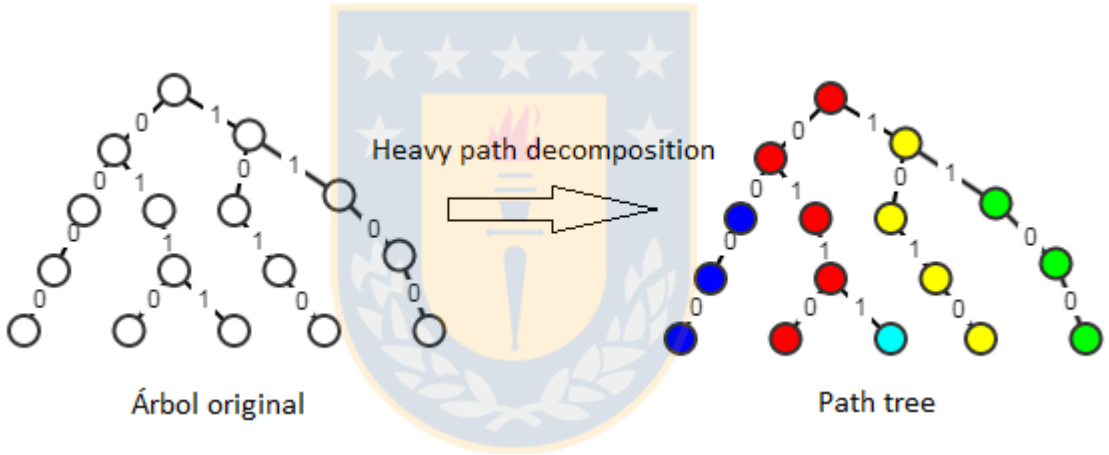


Figura 2.15: Nodos del mismo color representan un mismo heavy path.

Como se muestra en la **Figura 2.15**. El path tree está compuesto de heavy paths. Cada heavy path puede ser visto como un nodo que contiene todos los elementos que conforman dicho heavy path. Por ejemplo, el heavy path en rojo es un nodo conteniendo {0, 1, 1, 0}. Cada nodo puede contener 0 o más hijos. Un nodo tiene un hijo u si hay una arista entre el heavy path representando dicho nodo y el heavy path de u en el árbol original. El heavy path más largo es el nodo raíz del path tree.

La navegación en el path tree es como sigue: suponer que se quiere buscar la secuencia {0, 1, 1, 1}. Empezando de la raíz, se tiene que el nodo raíz es {0, 1, 1,

0}. La secuencia es igual al nodo raíz salvo por el último elemento, el 0. Pero, en este punto hay una arista hacia el heavy path celeste. Así que se comparan los elementos restantes de la secuencia, que son {1} con el heavy path celeste ({1}). Ambas secuencias son idénticas, por lo tanto la secuencia {0, 1, 1, 1} está en la estructura.

Como se mostró en el ejemplo anterior, encontrar una secuencia en el path tree es igual que hacerlo en el árbol original. Sin embargo, si esa secuencia es un bitmap entonces se pueden realizar operaciones de bits como XOR. Estas operaciones permiten bajar más rápido en el path tree.

2.8 Paralelismo de bits

Técnica que permite realizar operaciones eficientes en tiempo al aprovechar las operaciones de bits. En estructuras de datos sucintas es común representar estructuras de árbol como bitmaps. Usando operaciones de bits como XOR es posible saber si un punto está en la estructura sin tener que navegar el árbol nodo por nodo (como el caso del árbol con punteros). Esto porque el punto a buscar se representa como un bitmap, al igual que la estructura. Por lo tanto, buscar el punto se puede reducir a realizar operaciones de bits.

En la sección 2.7, se hizo referencia a la búsqueda de una secuencia en el path tree. Para encontrar la secuencia {0, 1, 1, 1} se puede usar el siguiente algoritmo:

```
Secuencia := {0, 1, 1, 1}
FOR i in {0, Largo(Secuencia) - 1}:
    IF Secuencia[i] != HeavyPath[i]:
        RETURN i
```

Un algoritmo lineal en el tamaño de la secuencia. Esta secuencia esta formada de 0's y 1's por tanto se puede representar como un bitmap. Ahora el problema se transforma en encontrar el bit en el que el heavy path 0110 es diferente del bitmap 0111. Entonces se puede emplear el siguiente algoritmo:

```
RETURN PosiciónPrimerBitConValor1(0110 XOR 0111)
```

Ambos algoritmos dan como resultado 3, pero el segundo (usando bitmaps) es $O(1)$. Realizar la operación de bits entre la secuencia y el heavy path es un ejemplo de **paralelismo de bits**.

2.9 Estructuras de datos sucintas

Una estructura de datos sucinta es una estructura de datos que usa una cantidad de espacio cercana al límite inferior propuesto por la **Teoría de la información**. Además, estas estructuras no tienen que ser descomprimidas para recuperar la información contenida en ellas [10].

La motivación detrás de las estructuras que se enfocan en utilizar poco espacio viene de la jerarquía de memoria de los computadores actuales. Recuperar información desde disco es órdenes de magnitud más costoso que desde memoria principal, y éste a su vez es más costoso que operar en caché. Por esto, tener estructuras que puedan ser almacenadas más cerca de la CPU aumenta la velocidad.

La mayoría de estas estructuras se reducen a operaciones sobre bitmaps, que se presentan a continuación.

2.9.1 Operaciones sobre bitmaps

Rank

Cuando se tiene un bitmap comprimido, una de las operaciones disponibles es **rank**. La operación **rank** se define como sigue:

$Rank_q(T, x)$ = Número de elementos iguales a q hasta la posición x en la secuencia T.

Por ejemplo, suponer que $T = 01101$ y las posiciones empiezan por 0. Entonces:

$Rank_1(T, 2) = 2$
$Rank_0(T, 1) = 1$

Select

Select retorna la posición del i-ésimo valor p. Se define como:

$Select_p(T, i)$ = Posición del i-ésimo valor p en T.

Por ejemplo, suponer que $T = 01101$ y las posiciones empiezan por 0. Entonces:
$Select_1(T, 1) = 1$
$Select_0(T, 2) = 3$

Access

Access retorna el valor del bit en una posición. Se define como:

$Acess(T, x)$ = Valor del bit en la posición x

Por ejemplo, suponer que $T = 01101$ y las posiciones empiezan por 0. Entonces:
$Acess(T, 3) = 0$

Existen implementaciones de estas operaciones en tiempo $O(1)$ y espacio $n + o(n)$ bits [16], [17].

2.9.2 Bitmaps comprimidos

Raman, Raman and Rao

Representación comprimida de secuencias binarias [10]. La secuencia se divide en un conjunto de bloques de igual tamaño, cada uno representado por la cantidad de 1's que contienen y un identificador (como se muestra en la **Figura 2.16**).

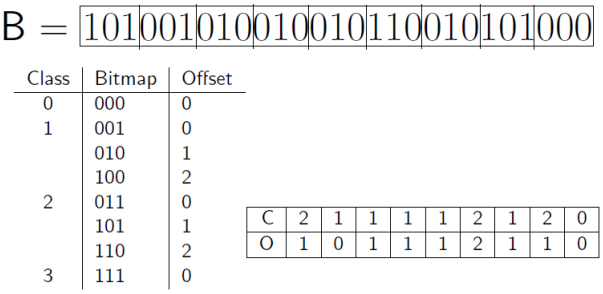


Figura 2.16: La secuencia binaria B se divide en bloques de tamaño 3. Bloques con igual número de 1's pertenecen a la misma clase. Imagen extraída de [10]

Permite operaciones rank, select y access en tiempo constante utilizando $nH_0(B)+o(n)$ bits con n el número de bits y H_0 la entropía de orden 0 (medida de compresión de una secuencia que depende de la frecuencia de los elementos [19]).

Sadakane's SDArray

Representación comprimida de secuencias binarias que funciona mejor que **Raman, Raman and Rao** para bitmaps muy esparsos [10]. Se basa en almacenar $S[i] = select(B, i)$ y resolver la operación rank usando búsqueda binaria. Soporta select en tiempo constante, pero rank y access en $O(logn/m)$. Ocupa $O(m + mlogn/m)$ bits. Con n el número de bits y m un factor usado para subdividir el bitmap, que proporciona un trade-off espacio tiempo.

2.10 Librería LIBCDS

Librería que contiene implementaciones de estructuras de datos sucintas como bitmaps, secuencias, permutaciones, entre otras. Disponible en [6], desarrollado usando en lenguaje de programación C++. Existen otras librerías alternativas como SDSL-Lite [18], también desarrollado en C++.

De esta librería se utilizan las estructuras para bitmaps comprimidos y la operación rank sobre los mismos.

Capítulo 3

Solución propuesta

En esta sección se detallan dos versiones de la estructura propuesta, de forma conceptual. La primera alternativa fue la que se diseñó e implementó inicialmente. Dado que la evaluación experimental no fue satisfactoria (ver sección 4), se buscó una forma más eficiente de realizar las consultas sin utilizar operaciones select (ambas versiones tienen igual complejidad teórica, pero la operación select es costosa en la práctica). Por esto, se diseñó e implementó una segunda versión con la mejora.

3.1 Primera versión de la estructura

La estructura presenta las siguientes características:

1. Cada punto es representado como un **quadcode**.
2. En el proceso de construcción, los quadcodes son almacenados en un **trie**. Se ahorra espacio porque algunos quadcodes comparten prefijos.
3. Se usa la técnica de **Heavy path decomposition** sobre el trie. Los heavy paths se representan como bitmaps.
4. Mediante **operaciones XOR** es posible navegar varios niveles a la vez en el path tree.

Se considera la siguiente matriz como ejemplo, se hace un seguimiento de la construcción de la estructura y luego se consulta por la existencia de dos puntos, uno

que está en el conjunto y otro que no.

	0	1	2	3	4
0	1	0	0	1	0
1	0	0	0	0	0
2	0	1	1	0	0
3	0	1	0	0	0
4	1	0	0	1	1

Figura 3.1: Matriz de 5x5 con 8 elementos.

La matriz de la **Figura 3.1** tiene un universo $u = 5$ (puede contener hasta 25 elementos) y 8 elementos, los cuales descritos por sus índices son: (0, 0); (3, 0); (1, 2); (2, 2); (1, 3); (0, 4); (3, 4); (4, 4).

Representando cada coordenada como un quadcode, se obtiene lo siguiente (ver sección 2.5 para detalles del entrelazado de bits):

Coordenada	Quadcode
(0, 0)	000000
(3, 0)	001010
(1, 2)	000110
(2, 2)	001100
(1, 3)	000111
(0, 4)	010000
(3, 4)	011010
(4, 4)	110000

Figura 3.2: Notar que el punto (4, 4) es el de mayor valor numérico que puede estar en la matriz, por lo que se necesitan 6 bits por cada quadcode (3 bits para representar el valor 4).

El algoritmo de construcción de un quadcode dada una coordenada tiene complejidad $O(1)$ y consiste en tener una tabla con valores precomputados que nos permitan construir el quadcode. Dada una coordenada (x, y), se divide cada coordenada en subconjuntos de 8 bits. Los subconjuntos de 8 bits están precomputados y se pueden conseguir de la tabla. Luego, se unen ambos bitmaps (resultado de las coordenadas x e y) mediante la operación OR. El algoritmo es como sigue (Morton encoding/decoding lookup table y operaciones de bits de C++, ver detalles en sección 2.6):

```
z1 = MortonTable256[x1 >> 8]   << 17 |
     MortonTable256[y1 >> 8]   << 16 |
     MortonTable256[x1 & 0xFF] <<  1 |
     MortonTable256[y1 & 0xFF];

z2 = MortonTable256[x2 >> 8]   << 17 |
     MortonTable256[y2 >> 8]   << 16 |
     MortonTable256[x2 & 0xFF] <<  1 |
     MortonTable256[y2 & 0xFF];

z  = z1 & ~(~0 << 32) | (z2 << 32); //Quadcode resultante
```

Figura 3.3: Cálculo de un quadcode. $x1$ e $y1$ representan los primeros 32 bits de las coordenadas x e y. $x2$ e $y2$ los últimos 32 bits.

Los quadcodes se almacenan en un trie (ver sección 2.2), ahorrando espacio aprovechando que algunos quadcodes tienen prefijos similares entre sí. El trie, para el ejemplo de la **Figura 3.2**, queda como sigue:

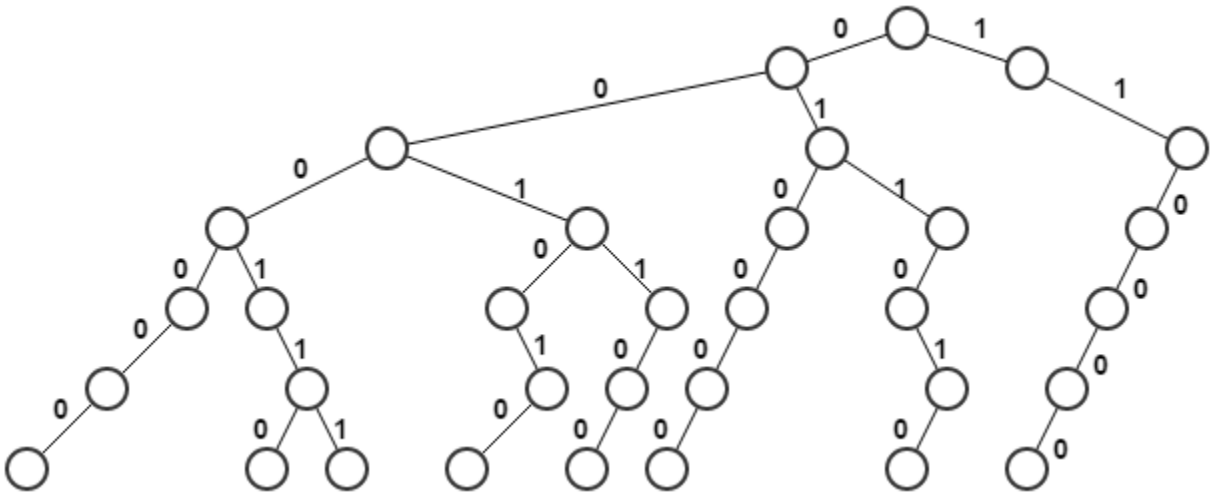


Figura 3.4: Los ocho quadcodes forman parte del árbol, el cual tiene altura $O(\log u)$, en este ejemplo 6

La representación como árbol con punteros ocuparía mucho espacio y las consultas tomarían tiempo $O(\log u)$, por lo que se propone una representación alternativa que tiene las siguientes características:

1. La altura del árbol se reduce de $O(\log u)$ a $O(\log n)$ mediante la técnica de Heavy path decomposition.

2. El path tree se representará como una serie de bitmaps. Los bitmaps se pueden comprimir, lo cual mejora el espacio.

3. Se emplearán operaciones de paralelismo de bits para responder consultas (búsqueda de puntos, por ejemplo).

A continuación se presenta la técnica de Heavy path decomposition, que toma el trie de la **Figura 3.4** y lo transforma en un **path tree**. El trie debe ser preprocesado antes de aplicar la técnica. En cada nodo del trie, es necesario saber cuántas hojas contiene el subárbol con raíz en dicho nodo. Esta información se usa para que el algoritmo de heavy path decomposition decida cuál es el **heavy edge** en cada iteración. El siguiente pseudocódigo calcula el número de hojas para cada nodo.

```
CalculateNumLeafOfEachNode:
    Input: Trie T

    NumSubtreeLeafs(Root(T)) = NumLeafs(LeftChild(Root(T)) +
                               NumLeafs(RightChild(Root(T))

NumLeafs:
    Input: Node n
    Output: Number of leafs in the subtree

    If Not Exists(LeftChild(n)) And Not Exists(RightChild(n)):
        Return 1

    If Not Exists(LeftChild(n)):
        Return NumLeafs(RightChild(n))
    Elseif Not Exists(RightChild(n)):
        Return NumLeafs(LeftChild(n))
    Else
        Return NumLeafs(LeftChild(n)) + NumLeafs(RightChild(n))
```

Figura 3.5: Calcula el número de hijos que tiene el subárbol con raíz en un nodo cualquiera del árbol. Realiza este calcula para todos los nodos del árbol.

El algoritmo se basa en que el número de hijos de un nodo es igual a la suma del número de hijos de los hijos de dicho nodo. A su vez, se puede aplicar recursivamente el mismo criterio para los hijos. La **Figura 3.6** muestra un ejemplo.

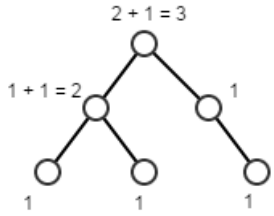


Figura 3.6: Todas las hojas tienen un valor de 1, en cada nodo se muestra el número de hojas como resultado de la suma del número de hojas de sus hijos.

3.1.1 Construcción de Heavy path decomposition

En esta sección se explica cómo funciona la técnica de Heavy path decomposition con un ejemplo simple. Al final de la sección se presenta el pseudocódigo aplicado al ejemplo de la **Figura 3.4**.

Inicialmente se tiene el quadtree de la **Figura 3.7** y una cola auxiliar **Queue**. Cada nodo del quadtree es etiquetado con un identificador único y un número que representa la cantidad de hojas del subárbol con raíz en dicho nodo. Cada arista tiene valor 0 o 1.

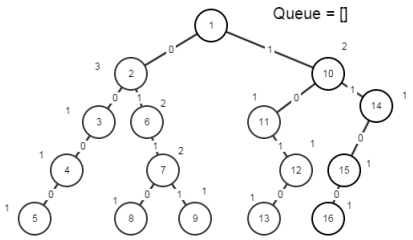


Figura 3.7: Quadtree al que se le aplica la técnica de Heavy path decomposition. Los números en el interior de cada nodo etiquetan a estos de forma única. Los números sobre las aristas son los bits que forman cada quadcode. Los números que acompañan a cada nodo indican la cantidad de hojas del subárbol con raíz en dicho nodo.

A continuación se describe en detalle cómo funciona, de forma conceptual, el algoritmo de Heavy path decomposition.

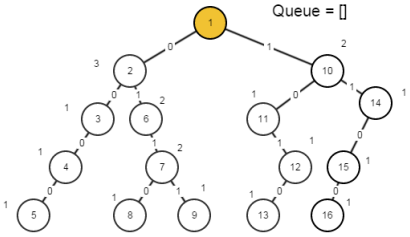


Figura 3.8: Empezando desde la raíz (nodo seleccionado coloreado con amarillo) y con una cola (Queue) vacía.

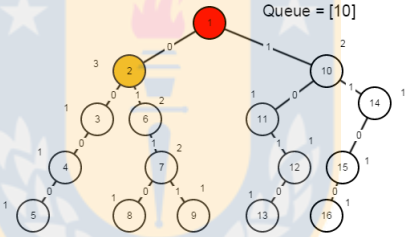


Figura 3.9: El hijo de la izquierda tiene un valor de 3, superior al hijo derecho cuyo valor es 2. En rojo se colorean los nodos que forman el primer Heavy path. El nodo 10, que no fue seleccionado, se añade a la Queue.

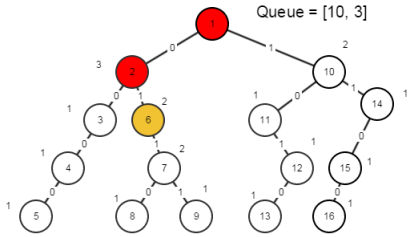


Figura 3.10: Se repite el proceso. El nodo seleccionado es el 6 y el nodo 3 se añade a la cola.

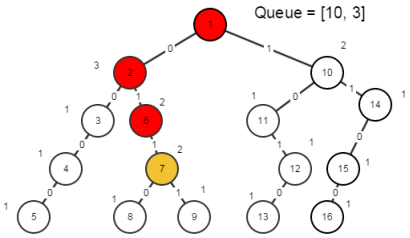


Figura 3.11: Se elige el nodo 7 porque no hay más hijos.

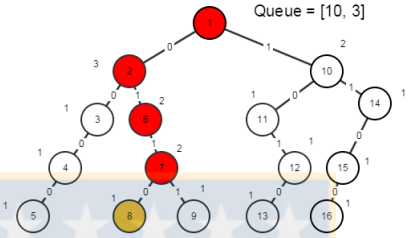


Figura 3.12: En caso de empate, arbitrariamente se elige el hijo de la izquierda. El primer heavy path es 1-2-6-7-8

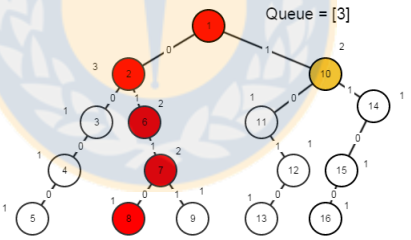


Figura 3.13: Al comienzo de la cola Queue está el nodo 10. Este nodo es el primero del segundo Heavy path.

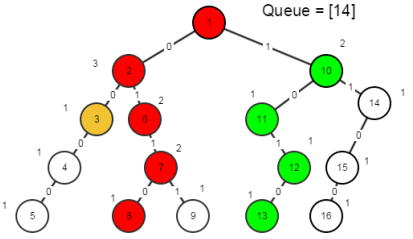


Figura 3.14: El mismo proceso se lleva a cabo con el nodo 3.

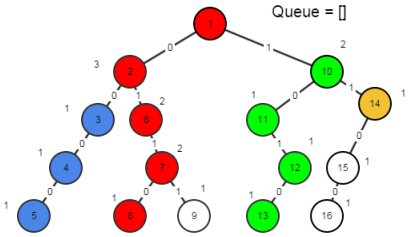


Figura 3.15: Una vez más con el nodo 14.

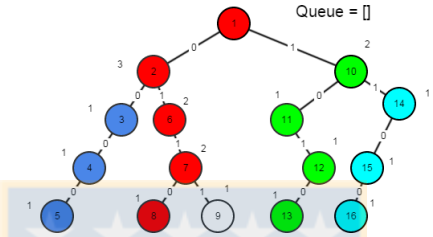


Figura 3.16: Este es el resultado de usar Heavy path decomposition. Notar que el nodo 9 no se considera porque, salvo para el primer heavy path, no se va tomar en cuenta el primer bit de cada heavy path (se puede deducir).

Conceptualmente el resultado del Heavy path decomposition puede ser representado como en la **Figura 3.17**.

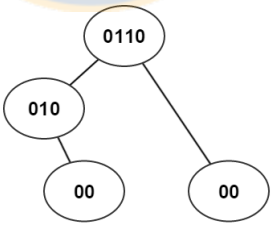


Figura 3.17: Representación conceptual de Heavy path decomposition (path tree). Cada nodo es un Heavy path. Una arista significa que hay un vínculo entre ambos heavy paths.

El nodo raíz es el primer heavy path (coloreado en rojo). Este nodo tiene un arista que lo conecta con el segundo heavy path (coloreado en verde). Esto significa que en algún bit del primer heavy path se puede ir al segundo heavy path. Este concepto se

utiliza para navegar por el árbol. Por ejemplo, si se quiere buscar el quadcode 0110, el resultado es directo ya que está en el nodo raíz. Si se quiere buscar el quadcode 1010, notar que el primer bit es diferente al primer bit del bitmap en la raíz, pero este nodo tiene un vínculo con el nodo 010. De este modo, el quadcode 1010 también está en el árbol.

La altura de este árbol es menor que la del trie, por lo que navegar por él es más rápido (sin considerar que hay que realizar un trabajar adicional en cada nodo). A continuación se muestra el pseudocódigo del algoritmo de Heavy Path Decomposition:

```
BuildHeavyPathDecomposition:
Input: Trie T
Output: Path Bitmap, Length Bitmap, Next Bitmap

Queue Q
Q.push(Root(T))
While NotEmpty(Q):
    curr = Q.front
    Q.pop
    While Exists(curr):
        If NumChild(LeftChild(curr)) >= NumChild(RightChild(curr)):
            curr = LeftChild(curr)
            Q.push(RightChild(curr)) //Agrega si RightChild(curr)
                                       existe
        Else
            curr = RightChild(curr)
            Q.push(LeftChild(curr)) //Agrega si LeftChild(curr)
                                      existe
```

Figura 3.18: Heavy path decomposition

La **Figura 3.19** muestra una iteración de un nodo interno del trie, mientras que la **Figura 3.20** es de un nodo terminal. La **Figura 3.21** muestra los heavy paths resultantes, cada uno con un color distinto. Los bitmaps **Path**, **Next** y **Length** se

explican en la sección siguiente.

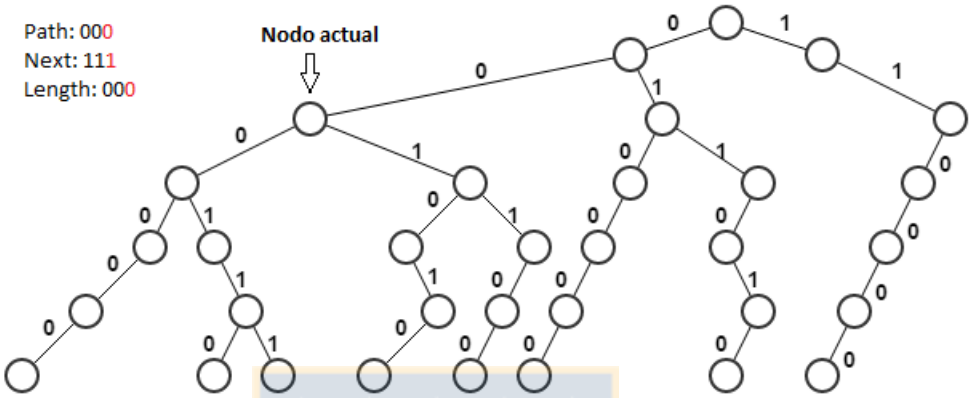


Figura 3.19: Cuando un nodo tiene dos hijos, se elige el que tiene un subárbol con mayor número de hojas (por Heavy Path Decomposition) mientras que el otro nodo se guarda en una cola. En next bitmap se añade un 1, indicando que existe una ramificación desde este punto.

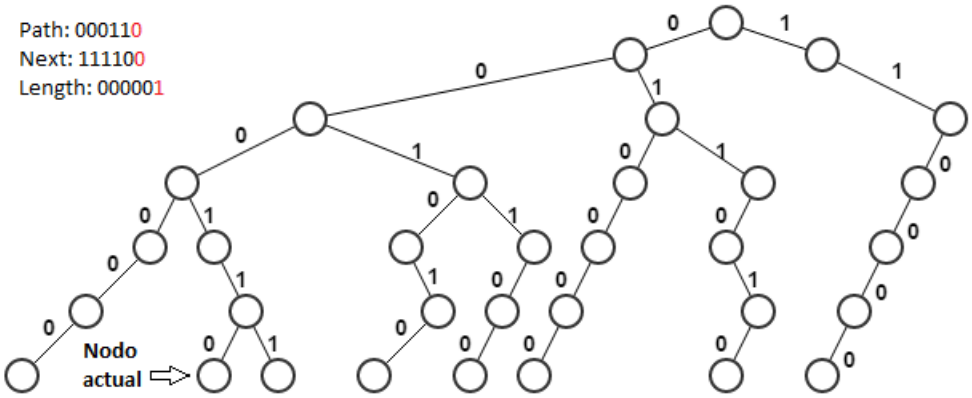


Figura 3.20: Cada vez que se llega a una hoja, se marca un 1 en el Length path.

3.1.2 Representación de la estructura

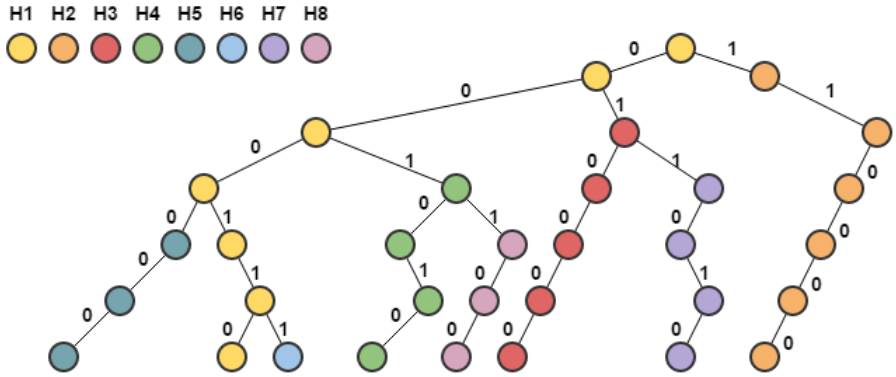


Figura 3.21: Cada path tiene un color distinto.

El resultado del algoritmo es el siguiente:

Path bitmap	000110|10000|0000|010|00|010|00
Next bitmap	111101|00000|1000|100|00|000|00
Length bitmap	000001|00001|0001|001|01|001|01

A los bitmaps Next y Length se le añaden estructuras rank/select para poder navegar los mismos. (ver sección 2.9).

- Path bitmap: Concatenación de cada **heavy path** en el mismo orden en el que van siendo construidos durante el heavy path decomposition.
- Next bitmap: Indica si un **heavy path** tiene una continuación hacia otro **heavy path**. Por ejemplo, desde el segundo bit del Path bitmap se puede navegar a otro **heavy path**. Esto porque el segundo bit del Next bitmap es un 1.
- Length bitmap: Indica el final de cada **heavy path**. El final se indica con un bit 1. Usando la operación select se puede saber la posición de cada **heavy path** en el Path bitmap.

3.1.3 Optimización de espacio en heavy paths

En la **Figura 3.21**, el **Path bitmap** tiene múltiples colores. Bits del mismo color representan un mismo **heavy path**. Por ejemplo, 1010 corresponde al **heavy path** coloreado en verde. Notar el caso del **heavy path** coloreado en rojo, 10000. En el **Path bitmap** este **heavy path** está representado como 0000, omitiendo el primer bit. Esta optimización es posible porque cada nodo del trie puede tener a lo más dos hijos, y estos pueden tener únicamente los valores 0 y 1. Por ejemplo, suponer que se está buscando el quadcode 100110. Se compara el quadcode 100110 con el primer **heavy path** 000110. Ambos bitmaps difieren en el primer bit, pero a partir de ese bit hay una continuación porque Next bitmap[0] = 1. Esto quiere decir que el primer bit del siguiente **heavy path** debe ser 1. Por esto, salvo el primer **heavy path** todos los demás **heavy paths** se almacenan omitiendo el primer bit. **Heavy paths** de largo uno no se guardan.

3.1.4 Navegación de la estructura (Paralelismo de bits)

Las consultas sobre la estructura se basan en paralelismo de bits. En esta sección se explica lo que significa este concepto.

Se tiene el trie de la **Figura 3.22**, y se quiere consultar por la existencia del punto (2,2) cuyo quadcode es 1100.

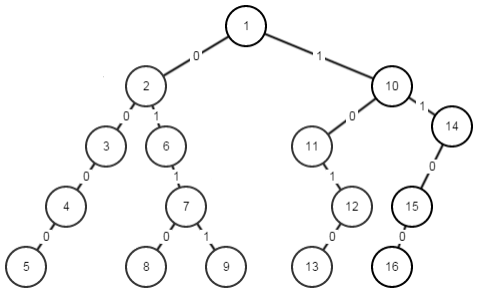


Figura 3.22: Trie que contiene los puntos (0, 0), (3, 0), (1, 2), (2, 2), (1, 3)

Una forma de hacerlo es ir iterativamente bajando en el árbol comparando cada bit del quadcode con los valores de los nodos. Por ejemplo, el siguiente pseudocódigo busca un punto en el árbol.

```
BUSCAR:
    Input: Quadcode QC, Tree T
    Output: Boolean

    //Asumir que los valores de los bits están en
    //los nodos y no en las aristas.

    Node n = Root(T)
    Integer index = 0;
    WHILE TRUE:
        Integer bit = QC.BitAt(index)
        IF Exists(LeftChild(n))
            AND IsEqual(LeftChild(n).value,  bit):
            n = LeftChild(n)
        ELSE IF Exists(RightChild(n))
            AND IsEqual(RightChild(n).value, bit):
            n = RightChild(n)
        ELSE:
            return FALSE

        index = index + 1

    return TRUE
```

Figura 3.23: Buscar un punto en el árbol T.

Con este método, el punto (2, 2) se encuentra en cuatro pasos.

Al estar comparando bitmaps se puede bajar por el árbol en un menor número de pasos. Usando la técnica de Heavy path decomposition (ver sección 2.7) resulta el árbol de la **Figura 3.24**.

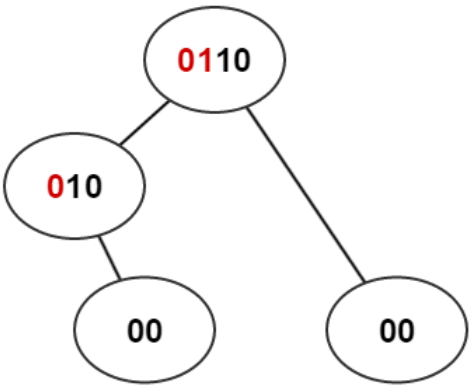


Figura 3.24: Representación conceptual de usar la técnica de Heavy path decomposition. Los bits en rojo indican que desde ese bit hay una arista al siguiente nodo. El primer bit del bitmap de la raíz apunta al nodo 010 y el segundo bit al nodo 00. Lo mismo para el nodo 010, el primer bit apunta al nodo 00.

En vez de tener un bit por nodo (lo que obliga a tener que moverse de un nodo a otro por cada bit del quadcode a comparar), se tienen nodos representando bitmaps. De este modo, usando operaciones sobre bits es posible comparar bitmaps en $O(1)$, en lugar de $O(m)$ (con m el tamaño del bitmap más pequeño).

El problema de búsqueda de un punto (o quadcode) en el árbol se reduce a saber, mediante operaciones de bits, si dos bitmaps son iguales. En caso contrario, saber la posición donde los bits son diferentes. Esto se logra usando la operación XOR (ver apéndice C.1). XOR retorna 0 si los bitmaps son iguales (se encontró el punto), si no retorna un bitmap donde el primer 1 de izquierda a derecha indica la posición donde los primeros bits son diferentes. Las **Figuras 3.25, 3.26 y 3.27** muestran cómo se encuentra el punto (2, 2).

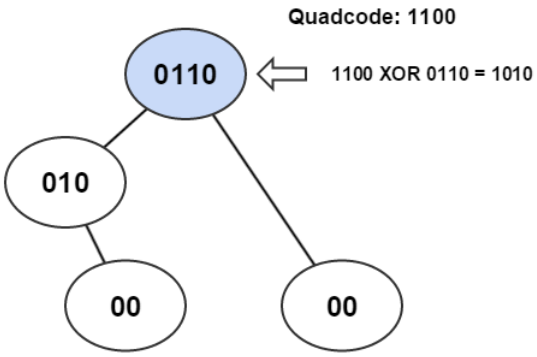


Figura 3.25: Empezando de la raíz, se compara el bitmap 1100 con el valor del nodo. La operación XOR nos dice qué el primer bit es diferente. Desde ese bit se puede ir al nodo izquierdo.

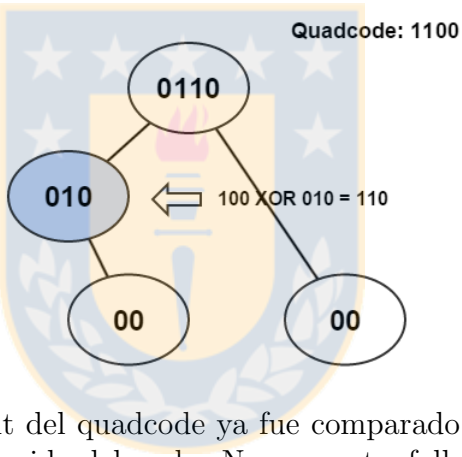


Figura 3.26: El primer bit del quadcode ya fue comparado, así que se compara solo el bitmap 100 con el contenido del nodo. Nuevamente, falla en la primera posición.

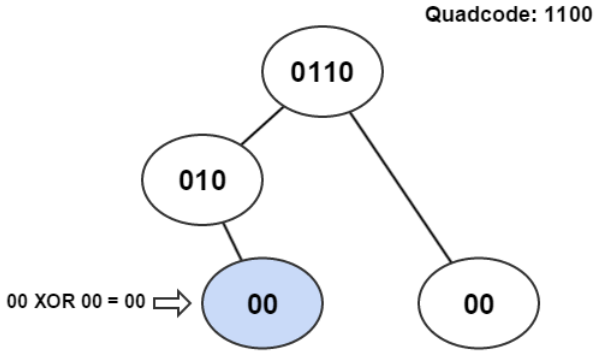


Figura 3.27: Se repite el proceso, esta vez con el bitmap 00. Ambos bitmaps son iguales por lo tanto el quadcode 1100 está en el árbol.

Con tres pasos el punto (2, 2) es encontrado (un paso menos que la navegación por el árbol). Este es el peor caso, pues se tuvo que llegar a una hoja. Ver si el punto (1, 2) cuyo quadcode es 0110 toma un paso. El proceso que se lleva a cabo en cada nodo se denomina **paralelismo de bits**.

3.1.5 Membership queries

A continuación se presenta el pseudocódigo de la operación CheckPoint. Esta operación indica si una coordenada está en el conjunto o no. Además se muestran dos ejemplos paso a paso de la ejecución del algoritmo. La idea del algoritmo es buscar el quadcode navegando a través de los heavy paths. Por cada heavy path una operación XOR nos dice en qué bit el quadcode y el heavy path son diferentes. Con esta información, es posible saber si se puede continuar navegando hacia otro heavy path o no. En caso en que no sea posible, se puede concluir que el quadcode no se encuentra en la estructura.

```
CheckPoint:
    Input: Bitmap bmp, Structure S (estructura propuesta)
    Output: Boolean b (true si existe, false en otro caso)

    current_pos := 0

    While True:
        position := S.PathBitmap.XOR(bmp, current_pos)

        If position == -1: //bmp coincide con S.PathBitmap
                           completamente
            Return True
        Else //bmp coincide hasta la position "position"
            Bit := S.PathBitmap.BitAt(position)
            If Bit == 0 //No hay continuación, el quadcode no
                        está en la estructura
                Return False
            Else
                numOnes := S.NextBitmap.Rank1(position)
                current_pos = S.LengthBitmap.Select1(numOnes) + 1
```

Figura 3.28: Operación de membership

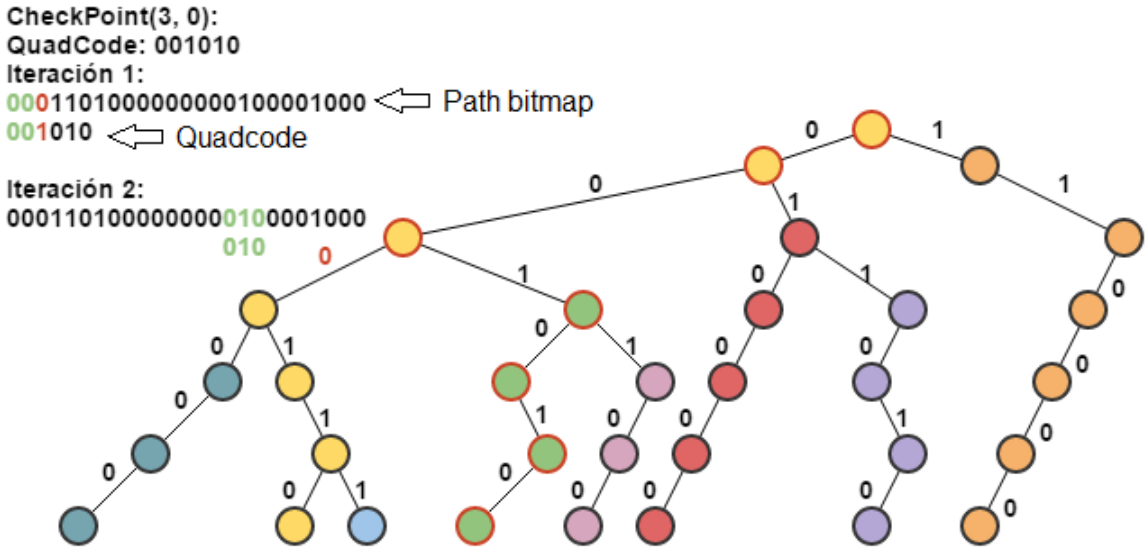


Figura 3.29: XOR retorna la posición 2, que es donde ya no coincide el quadcode con el Path Bitmap. Next Bitmap en la posición 2 contiene un 1, lo que quiere decir que hay un path por donde podemos continuar. Para saber cuál es, se sabe que hasta la posición 2 hay tres 1's, lo que significa que el siguiente path que sirve es el tercero que se encoló durante el algoritmo. Para saber la posición se usa Length Bitmap.

```
Ejemplo 1: CheckPoint(3, 0)
    Iteración 1:
        Current_pos = 0;
        Position = XOR(PathBitmap, Current_Pos, Binario(001010)) = 2
        BitAt(NextBitmap, 2) retorna 1
        NumOnes = Rank1(NexBitmap, 2) = 3
        Current_pos = Select1(LengthBitmap, 3) + 1 = 15
    Iteración 2:
        Position = XOR(PathBitmap, 15, Binario(010)) = -1
        CheckPoint retorna True
```

Figura 3.30: El punto (3, 0) está en la estructura.

```
Ejemplo 2: CheckPoint(1, 1)
    Iteración 1:
        Current_pos = 0
        Position = XOR(PathBitmap, 0, Binario(000011)) = 3
        BitAt(NextBitmap, 3) retorna 1
        NumOnes = Rank1(NextBitmap, 3) = 4
        Current_pos = Select1(LengthBitmap, 4) + 1 = 18
    Iteración 2:
        Position = XOR(PathBitmap, 18, Binario(11)) = 18
        BitAt(NextBitmap, 18) return 0
        CheckPoint retorna False
```

Figura 3.31: El punto (1, 1) no está en la estructura.

3.2 Segunda versión de la estructura

Como se mostrará en el capítulo 4, la versión anterior no es capaz de superar en tiempo al $k^2 - tree$ (consulta Membership), pero sí lo supera en menor número de operaciones. El problema es que la operación select (la cual se usa en consultas membership) es costosa. A continuación se muestra el cambio que se realizó y luego se explican en detalle dos ejemplos de Membership.

La construcción de la estructura es igual que en la primera versión salvo por dos cambios: los heavy paths están ordenados por altura decreciente (aristas que no son seleccionadas se almacenan en una cola de prioridad donde la prioridad es la altura en la que se encuentra la arista), por lo que heavy paths más largos aparecen antes que los más cortos. El next bitmap ahora es un arreglo de bitmaps de tamaño igual a la altura del **path tree**. En esta versión no existe el length bitmap, en su lugar se guardan los índices correspondientes al primer bit en el path bitmap de cada nivel del **path tree**. Del trie de la **Figura 3.4** se obtiene lo siguiente:

Path bitmap: 000110|10000|0000|010|010|00|00

Sean $L_0, ..., L_5$ los next bitmaps:
L_0: 1
L_1: 10
L_2: 101
L_3: 10010
L_4: 0000000
L_5: 1000000

Length vector: [0, 6, 11, 15, 18, 21, 23]

- Path bitmap: Concatenación de los **heavy paths**. El orden es altura (decreciente).

- Next bitmap: Una lista L por cada nivel del **path tree**. Cada bitmap L contiene tantos bits como nodos en dicho nivel. Por ejemplo, L_2 tiene tres bits porque a esa altura hay 3 nodos (ver **Figura 3.21**). El primer y tercer **heavy path** de ese nivel tienen continuación a otros **heavy paths** porque sus respectivos bits en L_2 están en 1.

- Length bitmap: Posición del primer bit de cada nivel del **path tree** en el Path bitmap.

Para realizar la consulta CheckPoint(3, 0) lo que se hace es lo siguiente: (3, 0) es 001010 en su representación como quadcode. Se ve que al hacer un XOR con el Path bitmap el tercer bit es diferente, lo que significa que el heavy path falla en el tercer nodo (altura 3 del trie de la **Figura 3.21**). El heavy path actual es el 0 (primero) por lo que para saber si desde ese punto podemos continuar por otro heavy path se tiene que revisar el primer índice de L_2, el cual es 1. Además se sabe que el próximo heavy path está en un nivel más profundo en el trie y que es el primer heavy path de ese nivel (según el ordenamiento al crear la estructura) así que se tiene que revisar la posición 15 (LenVector[3] = 15). Al comparar lo que resta del quadcode (010) con el bitmap en la posición 15 se tiene que el punto (3, 0) se encuentra en el conjunto. Para ver más ejemplos detallados de Membership y las funciones auxiliares usadas en el pseudocódigo ver apéndice B.

```
CheckPoint:
    Input: Bitmap bmp, Structure S (estructura propuesta)
    Output: Boolean b (true si existe, false en otro caso)

    current_pos := 0
    current_heavypath := 0
    height := Length(bmp) - 1

    While True:
        position := S.PathBitmap.XOR(bmp, current_pos)

        If position == -1: //bmp coincide con S.PathBitmap
                           completamente
            Return True
        Else //bmp coincide hasta la posición "position"
            L := (height - 1) - (Length(bmp) -
                 (position - current_pos) - 1) + 1;
            Bit := bitSequence[L]->access(current_heavypath)
            If Bit == 0 //No hay continuación, el quadcode no
                        está en la estructura
                Return False
            Else
                rank := bitSequence[L].Rank1(current_heavypath)
                EliminateFirstBits(bmp, position - current_pos)
                current_pos = S.lenVec[L + 1] + (rank - 1) *
                              Length(bmp)
                current_heavypath = Length(bitSequence[L]) +
                                    rank - 1
```

Figura 3.32: Operación CheckPoint.

La operación select es innecesaria en esta versión de la estructura, lo cual permite obtener mejores resultados en la sección de experimentos.

3.2.1 Range reporting queries

Range reporting es una operación que dado un rectángulo de la forma $(x_l, y_l); (x_u, y_u)$, donde (x_l, y_l) es el punto superior izquierdo del rectángulo y (x_u, y_u) el punto inferior derecho, entrega los puntos que están dentro de ese rectángulo.

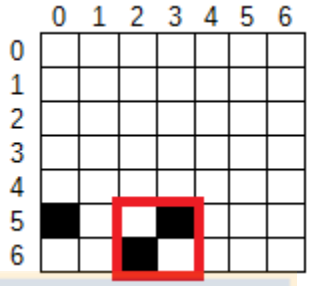


Figura 3.33: Los puntos $(3, 5)$ y $(2, 6)$ están contenidos en el rectángulo $(2, 5); (3, 6)$.

Como se muestra en la **figura 3.33**, el rectángulo en rojo corresponde a $(2, 5); (3, 6)$. La información que se quiere obtener es: 1) ¿Existe al menos un punto en esa región? Respuesta: Sí, 2) ¿Qué puntos están en esa región? Respuesta: $(3, 5)$ y $(2, 6)$.

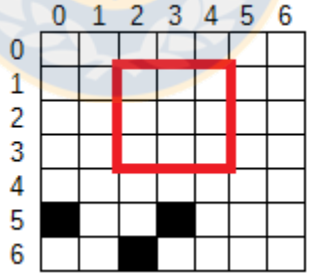


Figura 3.34: No hay puntos contenidos en el rectángulo $(2, 1); (4, 3)$.

En el ejemplo de la figura **figura 3.34**, no hay puntos contenidos en el rectángulo $(2, 1); (4, 3)$. Así que la respuesta de Range reporting debe ser un conjunto vacío.

La operación **Range reporting** se formula como sigue (Rect es un rectángulo, Point un punto en dos dimensiones):

Sea R := Rect(Point(x_l, y_l),Point(x_u, y_u)) y U el conjunto de todos los puntos de la matriz.

1. RR_Report(R) $\rightarrow R \cap U$
2. RR_Exist(R) $\rightarrow$ True o False (solo es necesario saber si esa región contiene al menos un punto)

La implementación de la operación se basa en la misma idea que sigue el $k^2 - tree$ para este tipo de consultas, usando un algoritmo de dividir y conquistar. Dado el trie formado por los quadcodes, encontrar los puntos en una región pasa por encontrar todos los cuadrantes que contienen a estos puntos y por ende, bajar por varias ramas del árbol y unir las soluciones parciales (las que se obtienen en cada rama por separado).

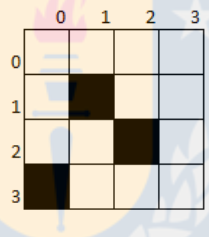


Figura 3.35: Matriz binaria que contiene los puntos (0, 3); (1, 1) y (2, 2). Marcados como casillas de color negro.

La estructura HP para el ejemplo de la figura 3.35 es la siguiente:

Path bitmap: 00111100101

Next bitmap list:

L_0 : 1

L_1 : 10

L_2 : 000

L_3 : 000

Len Vector: $[0, 5, 9, 0, 0]$

A continuación se muestra como funciona el algoritmo de Range Reporting para dos

casos distintos: La región es un cuadrante de la matriz y la región no es un cuadrante de la región.

3.2.2 La región es un cuadrante de la matriz

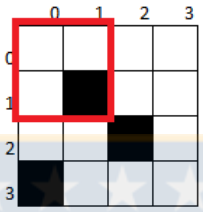


Figura 3.36: Consulta por la región cuyo coordenada superior izquierda es (0, 0) y su coordenada inferior derecha es (1, 1).

Suponer que se quiere consultar por la región cuyo coordenada superior izquierda es (0,0) y su coordenada inferior derecha es (1,1). Como se muestra en la figura 3.36. El algoritmo funciona en dos etapas: Descomponer la región solicitad en quadboxes y realizar la operación Membership en cada quadbox.

Descomponer la región solicitada

Para la descomposición en quadboxes maximales se usa el algoritmo propuesto en [25]. Un quadbox maximal es el mayor subárbol o subárboles en un quadtree que pueden representar la región solicitada. Como entrada se requiere un punto (x, y): La esquina superior izquierda de la región y un par (n_1, n_2): el largo y ancho de la región.

```
k = 0;
while true do
    if x mod 2^k ≠ 0 then
        n = n1 / 2^(k-1);
        for i ← 0 to n do
            MaximalQuadbox(x, y + 2^(k-1) * i, 2^(k-1));
        end
        x = x + 2^(k-1);
        n2 = n2 − 2^(k-1);
    end
    if y mod 2^k ≠ 0 then
        n = n2 / 2^(k-1);
        for i ← 0 to n do
            MaximalQuadbox(x + 2^(k-1) * i, y, 2^(k-1));
        end
        y = y + 2^(k-1);
        n1 = n1 − 2^(k-1);
    end
    if (x + n2) mod 2^k ≠ 0 then
        n = n1 / 2^(k-1);
        for i ← 0 to n do
            MaximalQuadbox(x + n2 − 2^(k-1), y + 2^(k-1) * i, 2^(k-1));
        end
        n2 = n2 − 2^(k-1);
    end
    if (y + n1) mod 2^k ≠ 0 then
        n = n1 / 2^(k-1);
        for i ← 0 to n do
            MaximalQuadbox(x + 2^(k-1) * i, y + n1 − 2^(k-1), 2^(k-1));
        end
        n1 = n1 − 2^(k-1);
    end
    k = k + 1;
end
```

Algoritmo 1: Decomposición en quadboxes maximales

Las iteraciones del algoritmo para el ejemplo se muestran en la tabla 3.1. El resultado es el quadbox que empieza en el punto (0,0) con dimensión 2 y con el es posible contener toda la región solicitada.

Tabla 3.1: Ejecución del algoritmo de descomposición

k	window	F()	Maximal Quadboxes
1	w($x = 0$,$y = 0$,$n_1 = 2$,$n_2 = 2$)	$[x = 0] \bmod 2^1 = 0$	
	w($x = 0$,$y = 0$,$n_1 = 2$,$n_2 = 2$)	$[y = 0] \bmod 2^1 = 0$	
	w($x = 0$,$y = 0$,$n_1 = 2$,$n_2 = 2$)	$[x + n_2 = 2] \bmod 2^1 = 0$	
	w($x = 0$,$y = 0$,$n_1 = 2$,$n_2 = 2$)	$[y + n_1 = 2] \bmod 2^1 = 0$	
2	w($x = 0$,$y = 0$,$n_1 = 2$,$n_2 = 2$)	$[x = 0] \bmod 2^2 = 0$	
	w($x = 0$,$y = 0$,$n_1 = 2$,$n_2 = 2$)	$[y = 0] \bmod 2^2 = 0$	
	w($x = 0$,$y = 0$,$n_1 = 2$,$n_2 = 2$)	$[x + n_2 = 2] \bmod 2^2 = 2$	MB(0,0,2)

Operación Membership

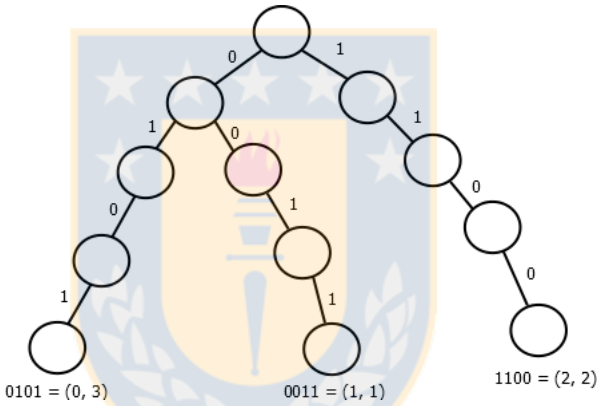


Figura 3.37: Representación conceptual en forma de árbol de la estructura HP.

El objetivo de descomponer la región en quadboxes maximales es poder encontrar los puntos usando la operación Membership. Como se muestra en la figura 3.37, cada punto en la estructura HP es representado por un camino que es una subdivisión de la matriz en quadboxes hasta llegar al punto en específico. Así por ejemplo los puntos (0,3) y (1,1) están contenidos en la mitad izquierda de la matriz. Esto es posible de deducir porque tanto el (0,3) como el (1,1) empiezan con un 0 en su forma de quadcode. En el caso del ejemplo en estudio, el único quadbox máximal que se debe analizar es el que comienza en el punto (0,0) y termina en el (1,1). Si se representan ambos extremos del quadbox en su forma de quadcode, entonces se tienen los puntos 0000 y 0011. Estos puntos tienen como prefijo común el 00, lo que significa que se

debe bajar en el árbol de la figura 3.37 por el hijo con valor 0 y luego por el siguiente hijo con valor 0. Una vez en ese nodo, todas las hojas del subárbol con raíz en dicho nodo son parte del resultado. Recuperar esos hijos es simplemente recorrer el subárbol nodo por nodo reconstruyendo los quadcodes. Conceptualmente, el proceso se muestra en la figura 3.38.

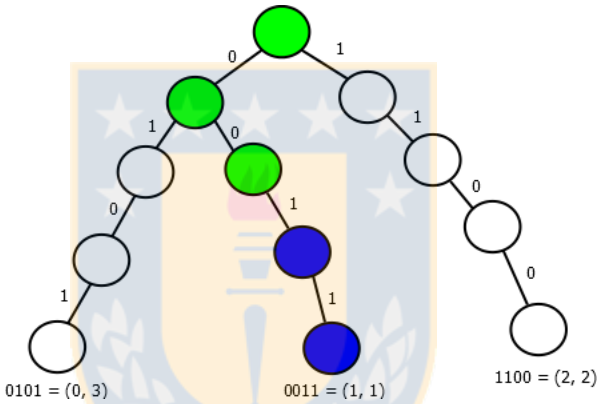


Figura 3.38: Representación conceptual de la búsqueda de los puntos dentro de la región. En verde lo que se recorre usando la operación Membership, y en azul el subárbol que contiene parte de la solución.

El ejemplo mostrado en esta sección tiene dos propósitos: el primero es explicar como funciona el algoritmo de Range Reporting y el segundo es hacer notar que cuando la región se puede construir con pocos quadboxes maximales, obtener los puntos es muy eficiente.

Descomponer la región solicitada

3.2.3 La región no es un cuadrante de la matriz

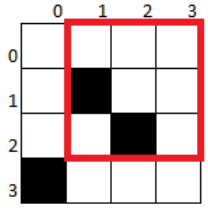


Figura 3.39: Representación conceptual de la búsqueda de los puntos dentro de la región. En verde lo que se recorre usando la operación Membership, y en azul el subárbol que contiene parte de la solución.

Uno de los inconvenientes del algoritmo es cuando la región contiene quadboxes maximales de tamaño 1. En el ejemplo de la figura 3.39, la región solicitad es la que va desde el punto (1, 0) al (3, 2).

Descomponer la región solicitada

Tabla 3.2: Ejecución del algoritmo de descomposición

k	window	F()	Maximal Quadboxes
1	w(1, 0, 3, 3)	$[x = 1] \bmod 2^1 \neq 0$	MB(1, 0, 1); MB(1, 1, 1); MB(1, 2, 1)
	w(2, 0, 3, 2)	$[y = 0] \bmod 2^1 = 0$	
	w(2, 0, 3, 2)	$[x + n_2 = 4] \bmod 2^1 = 0$	
	w(2, 0, 3, 2)	$[y + n_1 = 3] \bmod 2^1 \neq 0$	MB(2, 2, 1); MB(3, 2, 1)
2	w(2, 0, 2, 2)	$[x = 2] \bmod 2^4 \neq 0$	MB(2, 0, 2)

El resultado de la descomposición, como se muestra en la tabla 3.2, se compone de 5 quadboxes de tamaño 1 y uno de tamaño 4.

Operación Membership

Notar que cinco de los seis quadboxes son de tamaño uno, lo que significa que solo se esta revisando si hay o no un punto en esa casilla. Esto es ineficiente ya que no se

aprovecha el hecho de que se pueda llegar a cierto punto en el árbol de la estructura HP y desde ahi recuperar todos los puntos de la subregión. Salvo los quadboxes MB(1, 1, 1) y MB(2, 2, 1) los demas estan vacíos, por lo que el resultado son los puntos (1, 1) y (2, 2).

Capítulo 4

Evaluación experimental

La estructuras de las secciones 3.2 y 3.1 (resultados de esta estructura se muestran en la **Figura 4.2**) se implementaron y compararon experimentalmente con el k^2-tree. Se compara solo con el k^2-tree por las conclusiones obtenidas en [11].

Los experimentos fueron realizados en un Intel Core i7-3820@3.60GHz, 32GB RAM, bajo el sistema operativo Ubuntu server (kernel 3.13.0-35). Se compiló con gnu/g++ versión 4.6.3 usando la directiva -O3.

La implementación usa los bitmaps disponibles en [6]. Dos variantes de la solución propuesta son usadas. (1)HP^p, con bitmaps planos (sin compresión). (2)HP^c, con bitmaps comprimidos. Se usaron dos tipos de bitmaps comprimidos dependiendo de quién use menos espacio para un determinado dataset, Raman, Raman and Rao (RRR) o Sadakane's SDArray.

Para el $k^2 - tree$, se usaron dos variantes. $k^2 - tree^b$, versión con $k = 2$ para todos los niveles del árbol. $k^2 - tree^h$, versión que usa distintos valores de k para cada nivel del árbol (versión óptima según [1]).

Para la evaluación experimental, datasets de diferentes dominios fueron usados: Geographic Information Systems (GIS), social networks (SN), web graphs (WEB) y RDF

(RDF).

1. GIS: Coordenadas (Latitud, Longitud) de lugares geográficos sacados de Geonames [20]. Se convirtieron los datos a coordenadas (x, y) con distinta granularidad para producir tres datasets diferentes: Geo-sparse, Geo-med y Geo-dense (a mayor la granularidad, más esparsa la matriz), obtenidos de [20].

2. SN: Matriz de adyacencia asociada a dos redes sociales (dblp-2011, enwiki-2013) obtenidos de [12].

3. WEB: Matriz de adyacencia asociada a dos grafos de la web (indochina-2014, uk-2002) obtenidos de [12].

4. RDF: Triples (S, P, O). Cada tripleta indica **sujetos** relacionados a **objetos** por un **predicado** específico. Tres datasets se crearon a partir de los datasets obtenidos de [13]: triples-sparse, triples-med, triples-dense (seleccionado predicados con diferente número de objetos relacionados).

4.1 Espacio

File	Type	Grid(u)	Points(n)	k^2-tree^b	k^2-tree^h	HP^p	HP^c
Geo-dense	GIS	524,288	9,188,290	16.68	**13.27**	18.50	15.34
Geo-med	GIS	4,194,304	9,328,003	30.27	24.97	31.77	**21.84**
Geo-sparse	GIS	67,108,864	9,335,371	44.19	39.67	45.36	**28.55**
dblp-2011	SN	986,324	6,707,236	10.76	**9.84**	12.62	10.69
enwiki-2013	SN	4,206,785	101,355,853	16.96	**14.66**	18.56	15.33
indochina-2004	WEB	7,414,866	194,109,311	2.57	**1.22**	4.29	4.09
uk-2002	WEB	18,520,486	298,113,762	3.30	**2.04**	5.04	4.94
triples-dense	RDF	66,973,084	98,714,022	31.61	26.95	32.94	**23.26**
triples-med	RDF	66,973,084	7,936,138	9.80	**6.93**	12.19	10.10
triples-sparse	RDF	66,973,084	138,303	45.69	46.98	45.96	**29.97**

Figura 4.1: Comparación de espacio entre las cuatro variantes. Espacio medido en bits por punto. Los valores en negrita son los mejores resultados para cada dataset.

La **Figura 4.1** muestra una tabla con todos los datasets. u es el tamaño de la matriz (máxima dimensión de una coordenada). n es el número de puntos (casillas

con el valor 1) en la matriz. **bpp** es la medida usada (bits per point). El **bpp** se obtiene dividiendo el espacio total de la estructura por el número de puntos en la matriz.

HP^p obtiene la peor compresión entre todas las alternativas. Sin embargo, HP^c ocupa menos espacio que el $k^2 - tree$ para matrices esparsas.

4.2 Membership

El tiempo es medido como tiempo promedio por consulta en nanosegundos, donde cada consulta son 100.000 puntos. Tres tipos de consultas de membership son evaluadas:

1. Empty cell: Una celda con valor 0 en la matriz.
2. Filled cell: Una celda con valor 1 en la matriz.
3. Isolated filled cell: Una celda con valor 1 en la matriz y que esté rodeada de celdas con valor 0.

La versión de la sección 3.1 no se considera en los experimentos porque los tiempos de consulta son muy altos en comparación con el $k^2 - tree$, como se muestra en la **Figura 4.2**.

Estructura	Empty cell	Filled cell	Isolated filled cell
HP con select	113	413	71
$k^2 - tree$	3	30	30

Figura 4.2: La operación select es muy costosa. Debido a esto la primera versión de la estructura no supera en ninguna consulta al $k^2 - tree$. Tiempo medido en nanosegundos usando el dataset indochina-2004

Se espera que el $k^2 - tree$ obtenga mejores resultados para las consultas del tipo empty cells. Esto debido a que para encontrar un punto que no está en la estructura, no es necesario descender todos los niveles del quadtree. En cambio, la estructura propuesta tendrá que descender hasta las hojas en algunos casos para saber si un

punto está o no en la estructura. La estructura propuesta debería tener mejores resultados para filled cells. En ambas estructuras encontrar un punto significa descender hasta las hojas del árbol, pero el path tree tiene menor altura que el quadtree del $k^2 - tree$. Para isolated filled cell la estructura propuesta debería ser mejor porque estos puntos tienen la característica de que están en los heavy paths más largos. Por lo tanto, requieren pocas iteraciones para ser encontrados.

En esta sección se muestran los resultados para Geographic Information Systems (GIS). El resto de los resultados siguen la misma tendencia y se incluyen en el apéndice A.

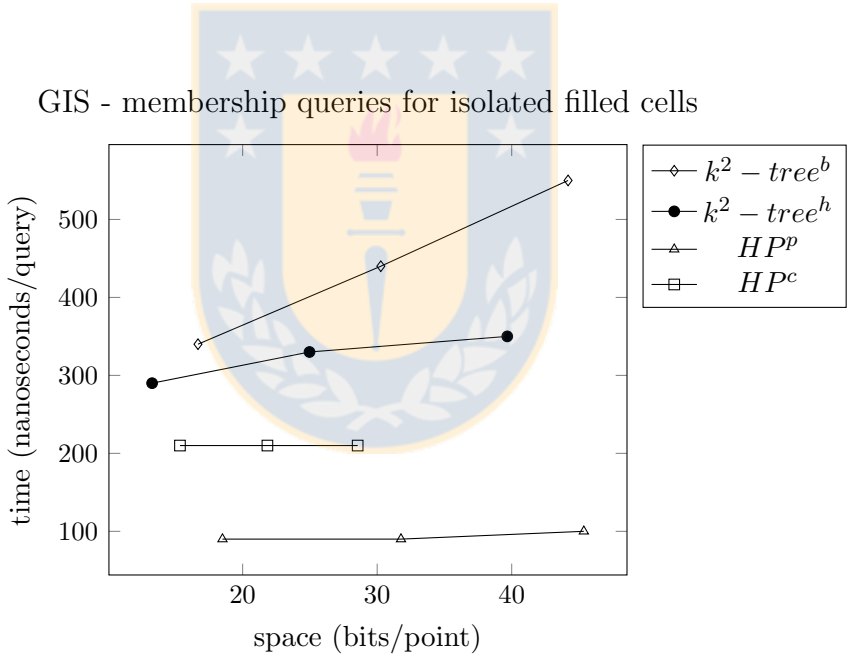


Figura 4.3: De izquierda a derecha cada punto representa los datasets: Geo-dense, Geo-med y Geo-sparse

Los resultados presentan comportamientos similares en todos los datasets. $k^2 - tree$ siempre es mejor para **Empty cells**. En el caso de **Filled cells** HP^p es la mejor. Ambas variantes de HP obtienen mejores resultados en las consultas sobre **Isolated Filled cells**. Los resultados son similares a los esperados según un análisis teórico de

las estructuras. El caso de isolated filled cells es el más favorable para la estructura propuesta porque estos puntos se encuentran en los primeros heavy paths del path tree. Esto significa que encontrar el punto implica bajar solo un par de niveles en el path tree. El caso mas desfavorable para la estructura propuesta es para puntos que no se encuentran en la estructura. La forma en que el $k^2 - tree$ está construido permite saber si un punto no está en la estructura rápidamente. Esto porque si un punto no se encuentra en la estructura el $k^2 - tree$ no tiene que bajar hasta las hojas del quadtree.

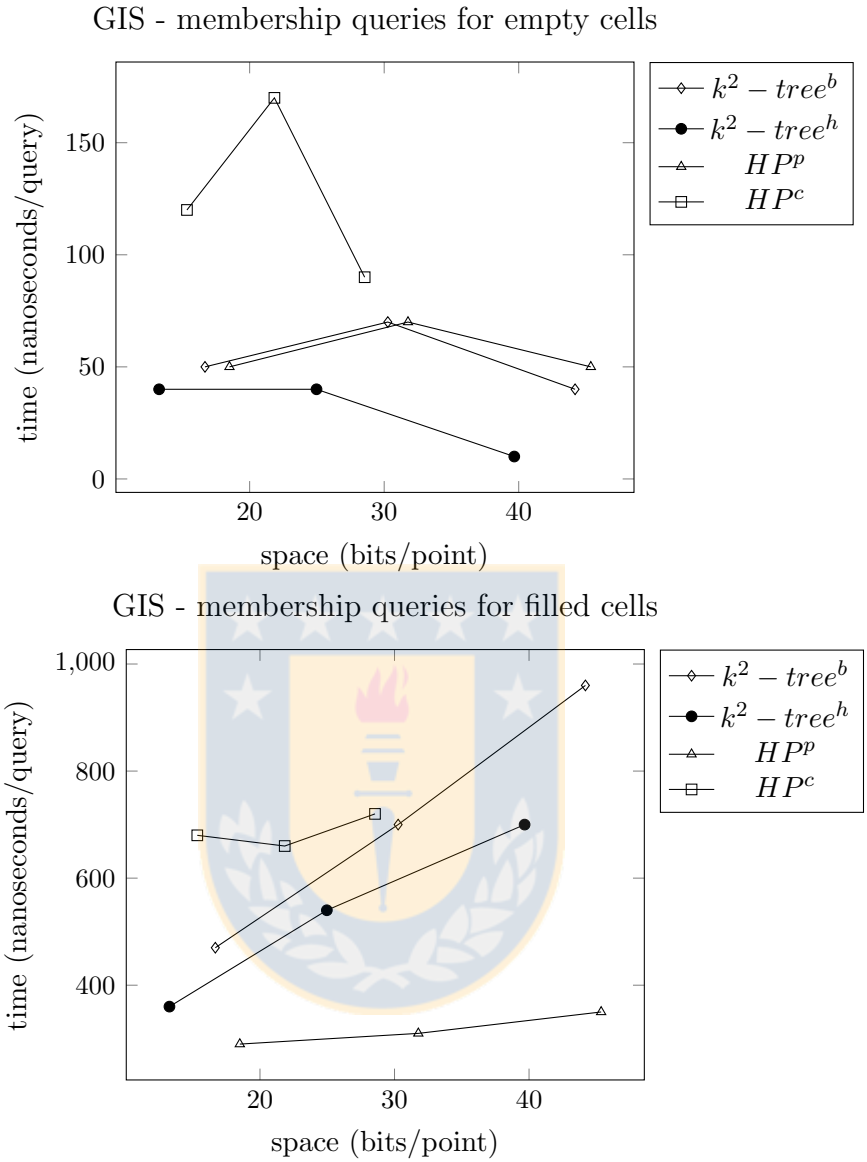


Figura 4.4: De izquierda a derecha cada punto representa los datasets: Geo-dense, Geo-med y Geo-sparse

4.2.1 Range Reporting

Se comparan los tiempos de consulta con el $k^2 - tree$ (versión básica), en dos tipos de consulta diferentes: Filled (región al azar que contiene al menos un punto)

y OneQuad (región que puede ser representada con solo un quadbox). Las consultas están hechas en base a los datasets UK-2002 (grafo de la web) y EN-WIKI 2013 (red social). Los experimentos para la estructura HP consideran ademas tiempos parciales, correspondientes a distintas partes del algoritmo. Estas fases son las siguientes:

- Fase 1: Descomposición en quadboxes maximales.
- Fase 2: Cada quadbox maximal es de la forma (x, y, longitud), donde x e y forman la coordenada superior izquierda.
- Fase 3: El quadbox representa un prefijo del quadcode, se revisa si ese prefijo está en la estructura. Si está, quiere decir que hay puntos en ese quadbox.
- Fase 4: Backtracking sobre la estructura a partir de cierto punto, donde se recuperan todos los puntos que tienen igual prefijo.

Tabla 4.1: HP para quadboxes aleatorios con al menos un punto (en segundos).

Dataset	Tamaño	Fase 1	Fase 2	Fase 3	Fase 4	Total
UK-2002	3x3	0.04	0.03	0.12	0.07	0.26
UK-2002	5x5	0.06	0.05	0.3	0.16	0.57
UK-2002	10x10	0.14	0.13	0.65	0.52	1.44
UK-2002	25x25	0.35	0.36	2.16	2.48	5.35
ENWIKI-2013	3x3	0.03	0.04	0.16	0.01	0.24
ENWIKI-2013	5x5	0.07	0.06	0.3	0.04	0.47
ENWIKI-2013	10x10	0.15	0.16	0.81	0.06	1.18
ENWIKI-2013	25x25	0.36	0.37	2.63	0.12	3.48

Tabla 4.2: HP para matrices representadas por un solo quadbox (en segundos).

Dataset	Tamaño	Fase 1	Fase 2	Fase 3	Fase 4	Total
UK-2002	4x4	0.03	0.00	0.04	0.07	0.14
UK-2002	8x8	0.03	0.01	0.05	0.24	0.33
UK-2002	16x16	0.02	0.01	0.06	0.83	0.92
UK-2002	32x32	0.01	0.01	0.04	2.76	2.82
ENWIKI-2013	4x4	0.02	0.01	0.03	0.02	0.08
ENWIKI-2013	8x8	0.02	0.01	0.04	0.05	0.12
ENWIKI-2013	16x16	0.01	0.0	0.05	0.12	0.18
ENWIKI-2013	32x32	0.02	0.01	0.06	0.26	0.35

Tabla 4.3: Comparativa con $k^2 - tree$ para quadboxes aleatorios (en segundos).

Dataset	Tamaño	$k^2 - tree$	HP
UK-2002	3x3	0.18	0.26
UK-2002	5x5	0.25	0.57
UK-2002	10x10	0.42	1.44
UK-2002	25x25	1.32	5.35
ENWIKI-2013	3x3	0.16	0.24
ENWIKI-2013	5x5	0.19	0.47
ENWIKI-2013	10x10	0.21	1.18
ENWIKI-2013	25x25	0.34	3.48

Tabla 4.4: Comparativa con $k^2 - tree$ para un solo quadbox (en segundos) (en segundos).

Dataset	Tamaño	$k^2 - tree$	HP
UK-2002	4x4	0.13	0.14
UK-2002	8x8	0.21	0.33
UK-2002	16x16	0.42	0.92
UK-2002	32x32	1.12	2.82
ENWIKI-2013	4x4	0.12	0.08
ENWIKI-2013	8x8	0.12	0.12
ENWIKI-2013	16x16	0.2	0.18
ENWIKI-2013	32x32	0.37	0.35

Como se muestra en las tablas 4.3 y 4.4, los tiempos del $k^2 - tree$ son inferiores a los HP. Pero hay algunas consideraciones a notar:

- La fase 4 (recuperación de puntos) tiene una implementación basada en backtracking básica, puede ser mejorada y los tiempos reducidos.
- Si bien para regiones arbitrarias el $k^2 - tree$ parece superior, para regiones que se puedan construir en base a pocos quadboxes HP tiene tiempos competitivos (mejores, si se considera que la fase 4 puede reducirse en tiempo).

Capítulo 5

Híbrido entre $k^2 - tree$ y HP

En esta sección se explica la construcción y las operaciones de Membership y Range Reporting de una estructura que combina lo mejor del $k^2 - tree$ y de HP. La idea general es tomar como base el $k^2 - tree$ y a cierta profundidad cambiar esos subárboles por estructuras HP. Esta contribución se propone sólo a nivel teórico y no ha sido implementada.

5.1 Estructura

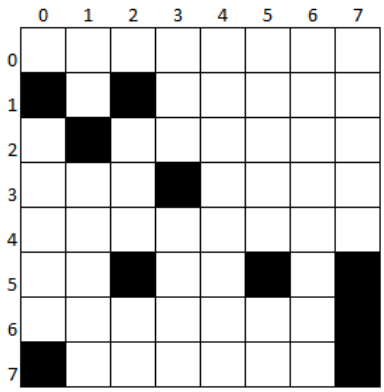


Figura 5.1: Matriz de ejemplo para la explicación de la estructura híbrida. La matriz contiene los puntos (0, 1); (2, 1); (1, 2); (3 ,3); (2, 5); (5, 5); (7, 5); (7, 6); (7, 7)

Se toma como base $k^2 - tree$ y a cierta altura se reemplazan los subárboles por estructuras HP por las siguientes razones:

1. Encontrar regiones sin puntos es mucho mas eficiente en $k^2 - tree$, esto porque cuando no hay puntos no es necesario llegar a una hoja. En cambio en HP el número de iteraciones es independiente de si existen puntos o no.

2. HP funciona de forma más eficiente en regiones que tengan que ser dividades en muchos quadboxes maximales (referirse a la sección 4.2.1), lo que es más probable mientras más pequeña la matriz a analizar.

Usando el ejemplo de la figura 5.1 se explicará en que consiste la estructura híbrida y como se realizan las consultas de Membership y Range Reporting.

5.1.1 Estructura híbrida usando factor de profundidad M

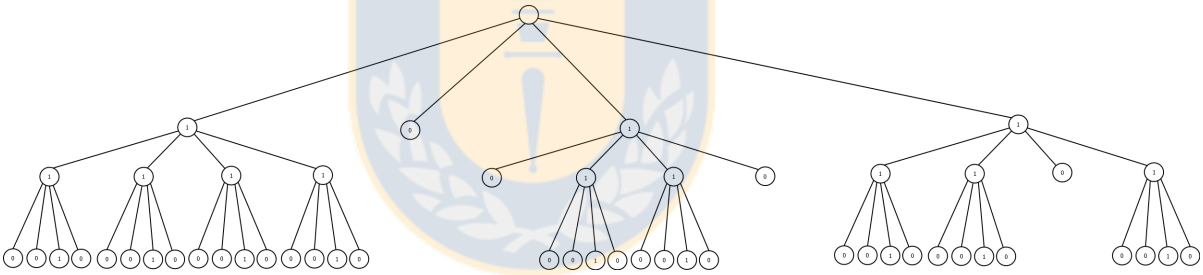


Figura 5.2: Matriz de ejemplo para la explicación de la estructura híbrida. La matriz contiene los puntos (0, 1); (2, 1); (1, 2); (3 ,3); (2, 5); (5, 5); (7, 5); (7, 6); (7, 7)

Suponer un $k^2 - tree$ como el de la figura 5.2. Esta estructura se representa por dos bitmaps, los cuales son $N := 1011111101101101$ para los nodos internos y $L := 0010001001000001001000001000010101$ para las hojas. Suponer, además, que se quiere llegar al resultado de la figura 5.3 donde $HP1$, $HP2$ y $HP3$ son estructuras HP representando los subárboles respectivos de la figura 5.2. El algoritmo de construcción crea un $k^2 - tree$ hasta un nivel de profundidad M y luego construye un HP por cada subárbol (La construcción de $k^2 - tree$ se puede revisar en [1] y la de HP en el capítulo 3).

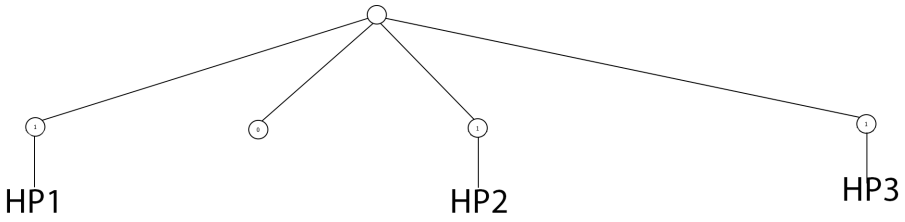


Figura 5.3: El primer nivel es una estructura $k^2 - tree$ y los niveles inferiores son estructuras HP.

Al ser solo la primera parte de la estructura un $k^2 - tree$, entonces las hojas se guardan usando HP y por ello el bitmap L no es necesario. El bitmap N es necesario parcialmente, hasta el nivel donde ya no existan nodos que sean parte del $k^2 - tree$. Para el ejemplo de la figura 5.3, $N = 1011$ (los cuatro nodos del primer nivel). Cada bit en N que tenga valor 1 apunta a un HP que representa una región particular de la matriz.

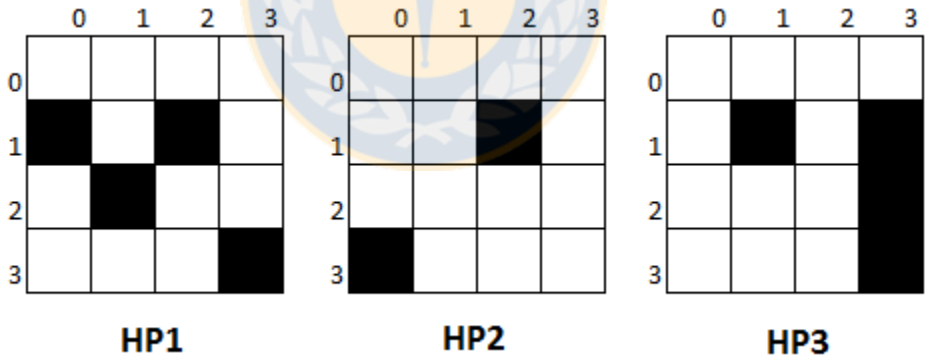


Figura 5.4: HP1 contiene los puntos (0, 1); (1, 2); (1, 2) y (3, 3). HP2 contiene los puntos (2, 5) y (0, 7). HP3 contiene los puntos (5, 5); (7, 5); (7, 6); (7, 7)

Cuando se llega a una profundidad M, la estructura se reduce a almacenar submatrices. Donde cada submatriz es una estructura HP distinta. Como se ve en la figura 5.4.

Membership

Suponer que se quiere buscar el punto (6, 0). Se sabe que el punto está en el segundo cuadrante por lo que se verifica el bitmap N en la posición 1, $N[1] = 0$ por lo que el punto no se encuentra en la estructura. Este ejemplo demuestra que se mantiene la característica que permite al $k^2 - tree$ reportar que un punto no se encuentra en la estructura de manera mas eficiente que HP.

Suponer que se quiere buscar el punto (2, 5). Se sabe que el punto está en el tercer cuadrante por lo que se verifica el bitmap N en la posición 2, $N[2] = 1$ por lo que hay que seguir iterando el algoritmo. Como la profundidad es 1, se sabe que a partir de este punto hay solo estructuras HP y la estructura a revisar es $HP2$. La entrada para este algoritmo no es el punto (2, 5), si no que hay que convertir este punto a uno que este en una matriz cuyo punto inicial (0, 0) sea el punto (0, 4) (inicio del tercer cuadrante). Luego el punto a buscar en $HP2$ es el punto (2, 1). Para más información de cómo buscar un punto en una estructura HP, ver el capítulo 3.

Range Reporting

Suponer que se quieren buscar los puntos en el rango (2, 4); (5,5). Este rango está en los cuadrantes 3 y 4, formando los subrangos (2,4);(3,5) y (4,4);(5,5) respectivamente. Nuevamente, como se ha alcanzado el nivel de profundidad donde se empieza a usar la estructura HP, se usa Range Reporting en HP: (2,0);(3,1) en $HP2$ y (0,0);(1,1) en $HP3$. Para más información sobre Range Reporting en HP ver sección 3.2.1.

5.1.2 Estructura híbrida usando nodos internos M

La estructura se puede generalizar para permitir que no solo existan estructuras HP a partir de cierta profundidad, sino que a partir de cualquier nodo interno del $k^2 - tree$. En el ejemplo de la figura 5.5, se toma como base un $k^2 - tree$. Luego, en cada nodo del $k^2 - tree$ se evalúa una función $F(n := nodo) = \{Verdadero, Falso\}$. Esta función retorna $\{Verdadero\}$ si el subárbol con raíz en n debe ser reemplazado por

un HP, $\{Falso\}$ en otro caso. Para el ejemplo, la función es la siguiente: Verdadero si la cantidad de puntos en dicha región es menor al 25% del tamaño total o si la región es de tamaño $2x2$. Con esto se busca que las regiones que tengan muy pocos puntos se guarden en estructuras HP.

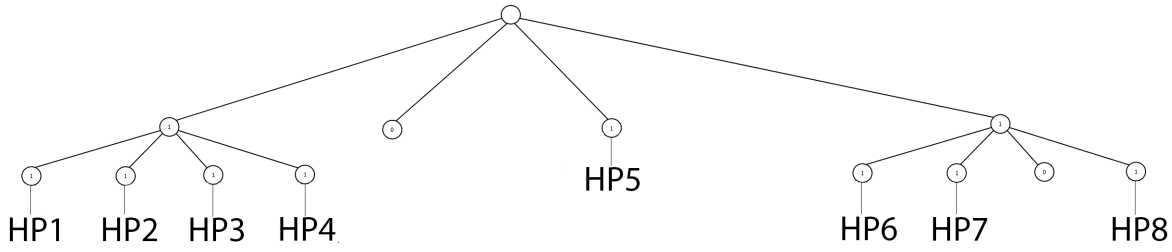


Figura 5.5: Como base es un $k^2 - tree$ y en nodos internos que cumplan alguna condición en especifico, se utiliza una estructura HP

.

El bitmap que representa el $k^2 - tree$ es $N := 100100000000$. Notar que los nodos que apuntan una estructura HP están marcados con un 0, esto genera el problema de que no es posible determinar si ese nodo apunta a una estructura HP o es un nodo terminal del $k^2 - tree$. Por ello se construye un bitmap adicional, el cual indicará si es un nodo terminal (0) o es una estructura HP (1). Para el ejemplo, el bitmap $N_{hp} := 001011111101$.

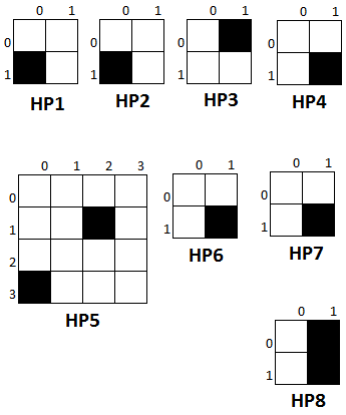


Figura 5.6: Cada HP puede representar submatrices de distinto tamaño.

Membership

Suponer que se quiere buscar el punto (5, 6). (5, 6) está en el cuarto cuadrante, entonces $N[3] = 1$. Luego $rank_1(3) = 8$ y la siguiente iteración es determinar en qué cuadrante de la región (4,7);(4,7) está el punto (5,6). Está en el cuadrante 3, así que $N[8 + 2] = 0$. Como $N[10] = 0$, se verifica $N_{hp}[10] = 0$. Se concluye que el punto (5, 6) no está en la estructura.

Suponer que se quiere buscar el punto (2, 5). El punto (2, 5) está en el tercer cuadrante, entonces $N[2] = 0$ y $N_{hp}[2] = 1$ por lo que hay una estructura HP donde seguir. Para saber qué estructura es, se determina con $rank_1(2) = 1$. Luego queda buscar el punto (2, 1) en $HP5$.

Range Reporting

Suponer que se quiere buscar la región $(1, 2); (2, 5)$. Esta región se subdivide en las regiones $(1, 2); (2, 3)$ en el primer cuadrante y $(1, 4); (2, 5)$ en el cuadrante 3. $N[0] = 1$, así que se debe seguir iterando. $N[2] = 0$ y $N_{hp}[2] = 1$ por lo que se busca la región $(1, 0); (2, 1)$ en $HP5$ y se reporta el punto $(2, 5)$. Para la región $(1, 2); (2, 3)$, ésta se subdivide en las regiones $(1, 2); (1, 3)$ y $(2, 2); (2, 3)$ que corresponden a las estructuras $HP3$ y $HP4$, respectivamente. Luego, se debe buscar la región $(1, 0); (1, 1)$ en $HP3$ y $(0, 0); (0, 1)$ en $HP4$, reportando el punto $(1, 2)$.

Capítulo 6

Conclusiones y trabajo futuro

6.1 Conclusiones

Se presentó una representación de quadtrees rápida y eficiente en espacio (comparado con el $k^2 - tree$, que en la práctica es la representación de quadtrees con mejores resultados). La estructura presenta las siguientes características:

1. En matrices esparsas, ocupa menos espacio que el $k^2 - tree$. Esto se debe a que el $k^2 - tree$ representa los puntos en relación a su posición en la matriz y, por tanto, gasta espacio representando cuadrantes con pocos puntos. En cambio, la representación propuesta almacena solo los puntos, por lo que no tiene problemas con regiones con pocos puntos.

2. Navegar por la estructura es $O(\log n)$ en comparación con el $O(\log u)$ del $k^2 - tree$. Para matrices esparsas donde $n << u$, las consultas se responden más rápido.

3. El espacio usado es similar a otras representaciones de quadtrees eficientes en espacio.

4. La estructura es mucho mejor que el estado del arte manejando **Isolated Filled cells**. Esto se debe a que estos puntos requieren muy pocas iteraciones del algoritmo de Membership para ser encontrados (pocos heavy paths que recorrer).

5. La operación Range Reporting tiene el inconveniente de que cuando la región no está representada por cuadrantes de la estructura, la doscomposición en quadboxes maximales es ineficiente. Pero en caso contrario, se obtienen mejores tiempos que $k^2 - tree$.

Además se presentó una propuesta donde se combinan $k^2 - tree$ y HP, con el objetivo de mejorar los tiempos de la consulta Range Reporting. Al usar HP en regiones relativamente pequeñas aumenta la probabilidad de que sea un cuadrante de la estructura, obteniendo mejores resultados que el $k^2 - tree$ por si solo.

6.2 Trabajo futuro

Se diseñó e implementó la estructura en base a datasets bidimensionales. Sin embargo, existen escenarios reales en donde más dimensiones son requeridas (coordenadas espaciales, coordenadas bidimensionales y el tiempo como tercera dimensión, etc.). Extender a más dimensiones es posible solo añadiendo más bits a los quadcodes, por lo que puede ser posible tener mejores resultados que el estado del arte.

Suponer que tenemos un punto tridimensional (x, y, z). Por ejemplo $(3, 1, 2)$: Transformando cada coordenada a su representación binaria $(3 = 11, 1 = 01, 2 = 10)$ y con **entrelazado de bits** el quadcode es 101110. Por lo tanto, más dimensiones implica un pequeño aumento en la altura del path tree. El resto de los algoritmos se mantiene igual. En cambio en el $k^2 - tree$ cada nuevo nodo implica k^2 bits, lo cual es muy costoso cuando la matriz es esparsa (habitual en más de dos dimensiones).

Bibliografía

[1] Nieves R. Brisaboa, Susana Ladra and Gonzalo Navarro. Compact Representation of Web Graphs with Extended Functionality. 2009.

[2] Gonzalo Navarro and Eliana Providel. Fast, Small, Simple Rank/Select on Bitmaps.

[3] Morton. A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing, Technical Report, Ottawa, Canada: IBM Ltd. 1966.

[4] Harel, Dov; Tarjan, Robert E., "Fast algorithms for finding nearest common ancestors", SIAM Journal on Computing. 1984.

[5] Rene de la Briandais, "File searching using variable length keys". In Proceedings of the Western Joint Computer Conference, pages 295-298, 1959.

[6] http://libcds.recoded.cl/. Succint data structures library.

[7] G. de Bernardo, S. Alvarez-Garcia, N. R. Brisaboa, G. Navarro, and O. Pedreira. Compact querieable representations of raster data. In Proc. SPIRE, pages 96–108, 2013.

[8] Sandra Álvarez García. Compact and Efficient Representations of Graphs

[9] Susana Ladra González. Algorithms and Compressed Data Structures for Information Retrieval

[10] Francisco Claude, Gonzalo Navarro. Space-Efficient Data Structures, manual for LIBCDS library usage.

[11] Guillermo de Bernardo, Sandra Álvarez-García, Nieves R. Brisaboa, Gonzalo Navarro, Oscar Pedreira. Compact Querieable Representations of Raster Data.

[12] http://law.dsi.unimi.it. Laboratory for Web algorithmics.

[13] http://wiki.dbpedia.org/Downloads351. dbpedia dataset.

[14] Irene Gargantini. An effective way to represent quadtrees.

[15] P. Venkat and d D. M. Mount. A succinct, dynamic data structure for proximity queries on point sets. In Proc. CCCG, 2014.

[16] Clark. Compact Pat Trees. Ph.D. thesis, University of Waterloo (1996)

[17] Jacobson. Space-efficient static trees and graphs. In: Proc. FOCS. pp. 549–554 (1989)

[18] Simon Gog. https://github.com/simongog/sdsl-lite

[19] Gonzalo Navarro and Veli Mäkinen. Compressed Full-Text Indexes.

[20] Geonames. http://download.geonames.org/export/dump/

[21] Antonin Guttman. R-trees. A dynamic index structure for spatial searching.

[22] Nieves R. Brisaboa, Miguel R. Luacesa, Gonzalo Navarro, Diego Seco. Space-Efficient Representations of Rectangle Datasets Supporting Orthogonal Range Querying.

[23] Roberto Grossi, Jeffrey Scott Vitter, Bojian Xu. Wavelet Trees: from Theory to Practice.

[24] Travis Gagie, Javier González, Susana Ladra, Gonzalo Navarro, Diego Seco. Faster Compressed Quadtrees. DCC '15

[25] Yao-Hong Tsai, Kuo-Liang Chung, Wan-Yu Chen. A Strip-Splitting-Based Optimal Algorithm for Decomposing a Query Window into Maximal Quadtree Blocks

Apéndice A

Experimientos de la consulta membership

A.1 Social networks (SN)

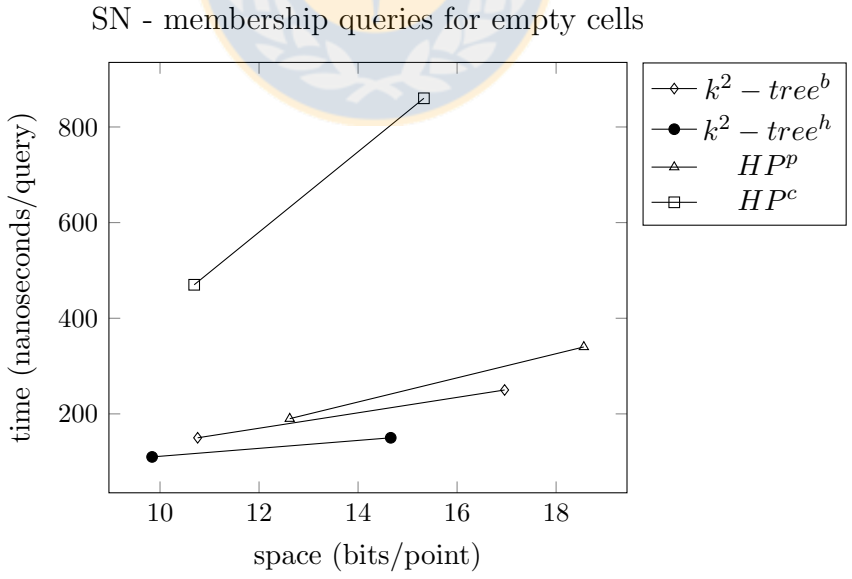


Figura A.1: De izquierda a derecha cada punto representa los datasets: dblp-2011 y enwiki-2013

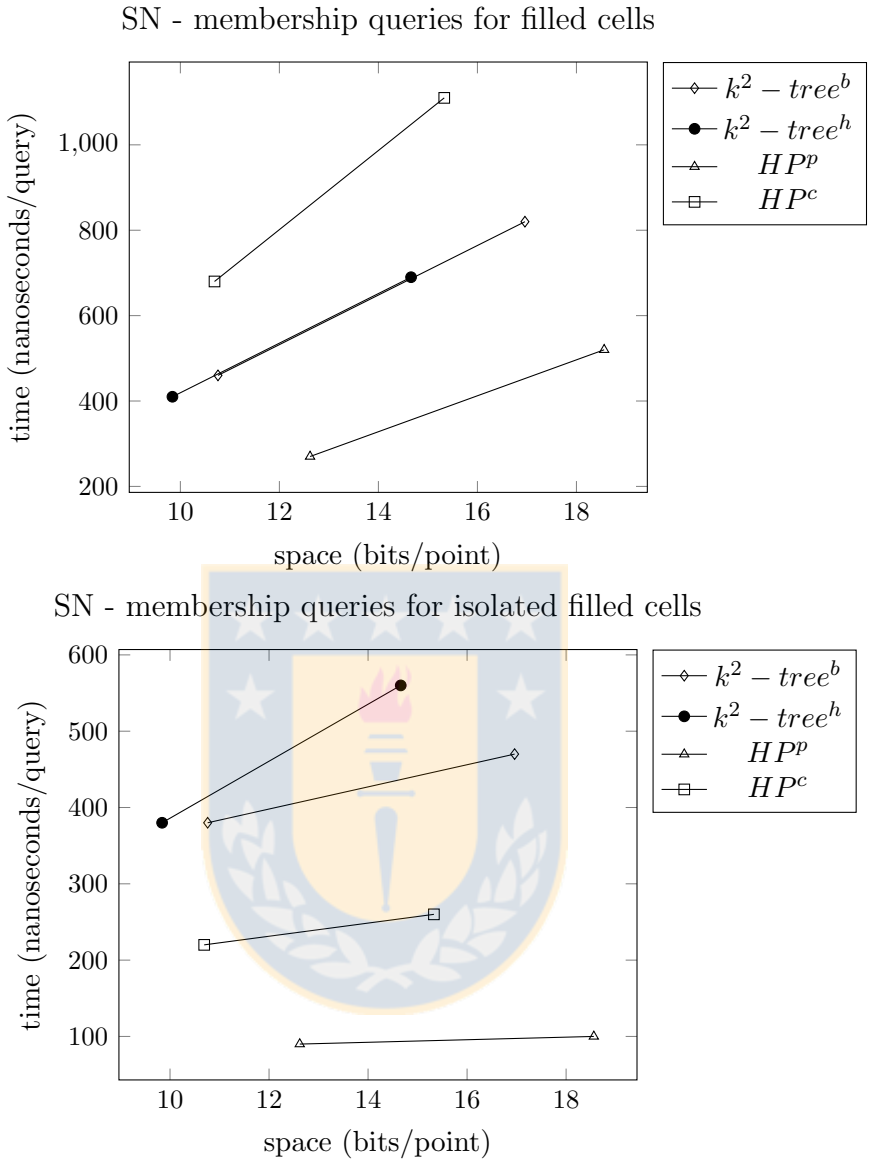


Figura A.2: De izquierda a derecha cada punto representa los datasets: dblp-2011 y enwiki-2013

A.2 WEB (WEB)

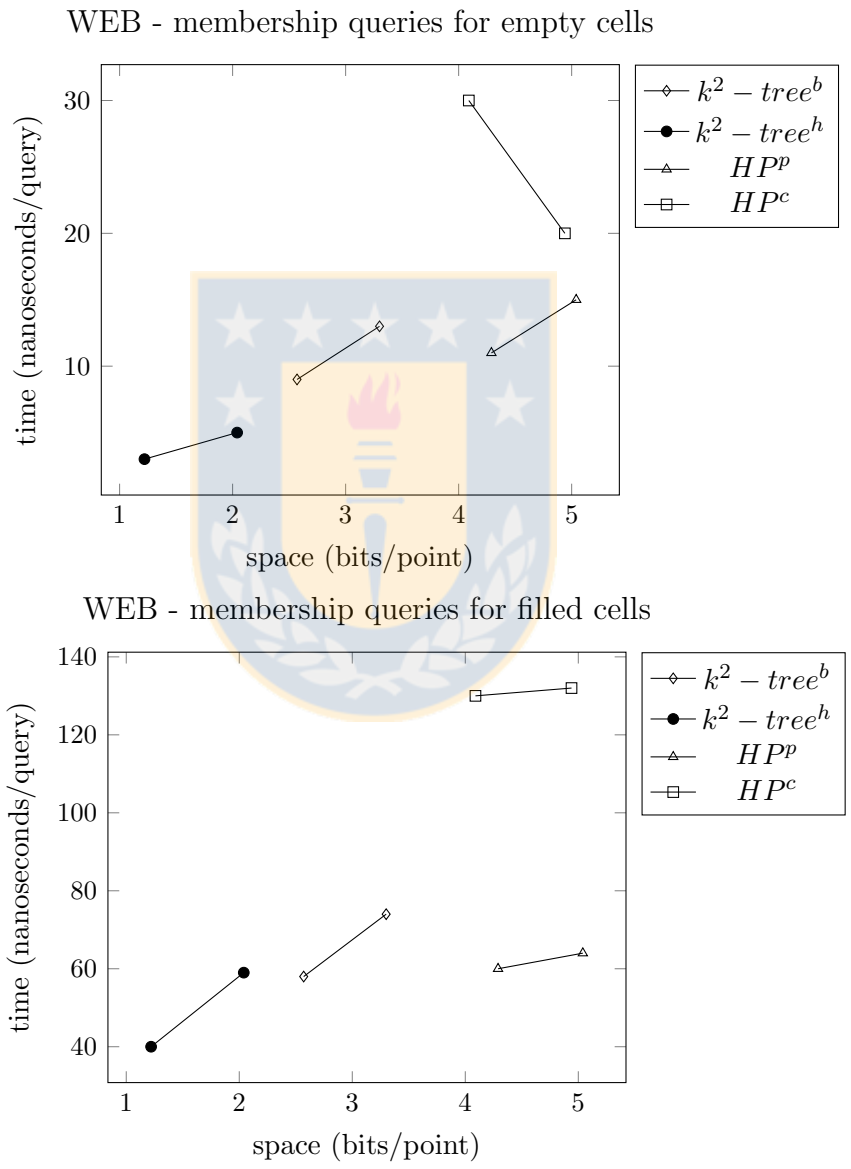


Figura A.3: De izquierda a derecha cada punto representa los datasets: indochina-2004 y uk-2002

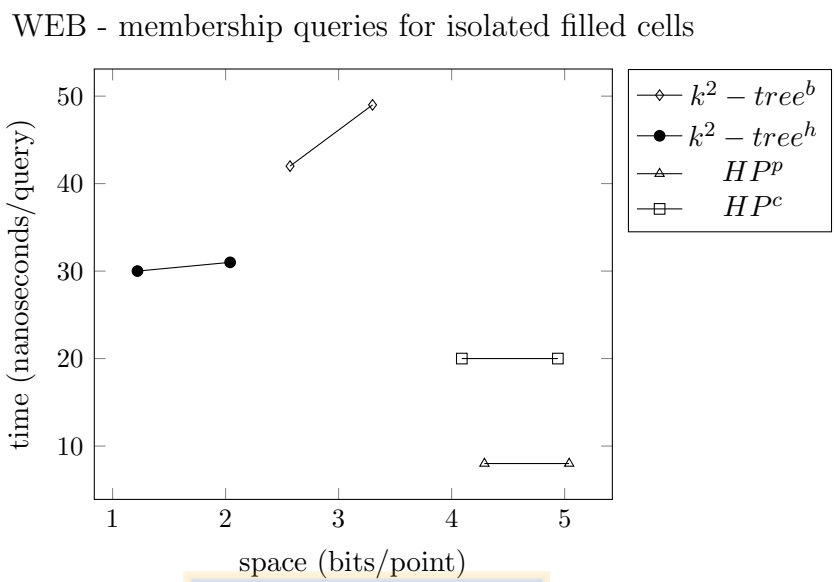


Figura A.4: De izquierda a derecha cada punto representa los datasets: indochina-2004 y uk-2002

A.3 RDF (RDF)

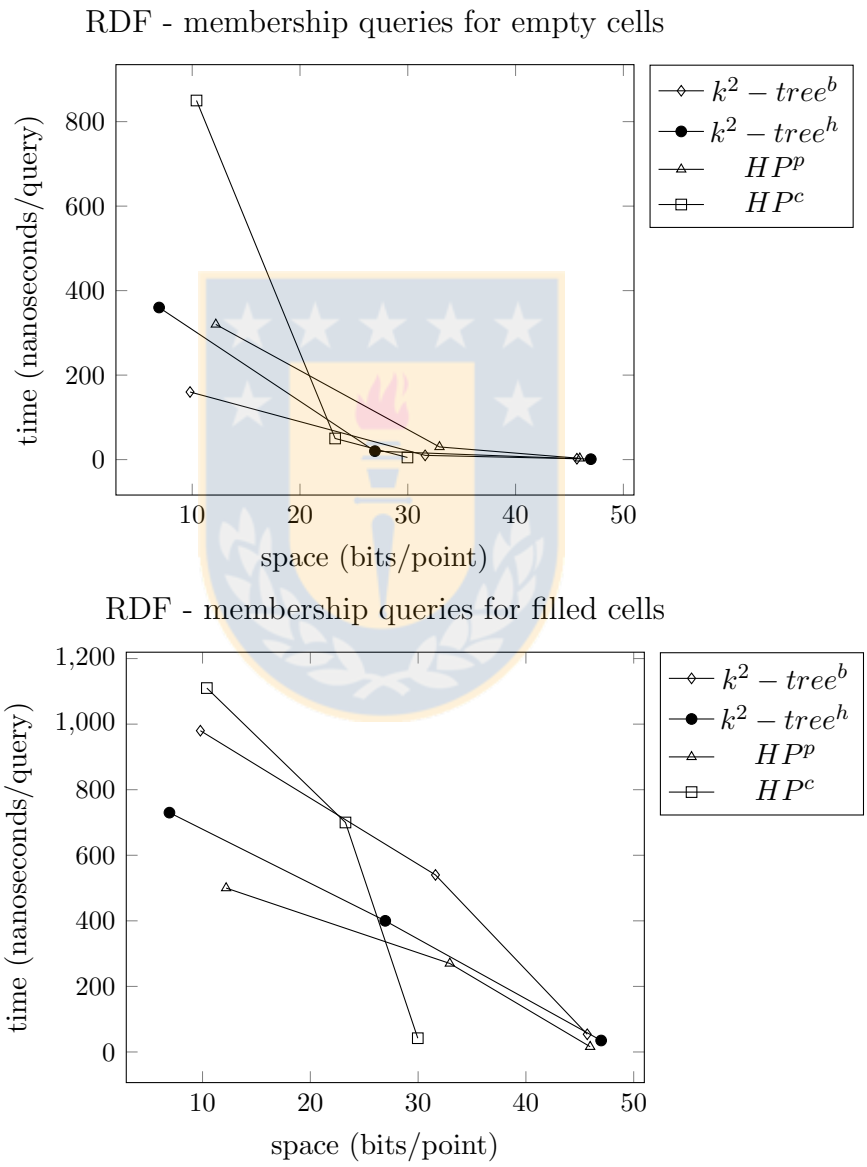


Figura A.5: De izquierda a derecha cada punto representa los datasets: triples-dense, triples-med y triples-sparse

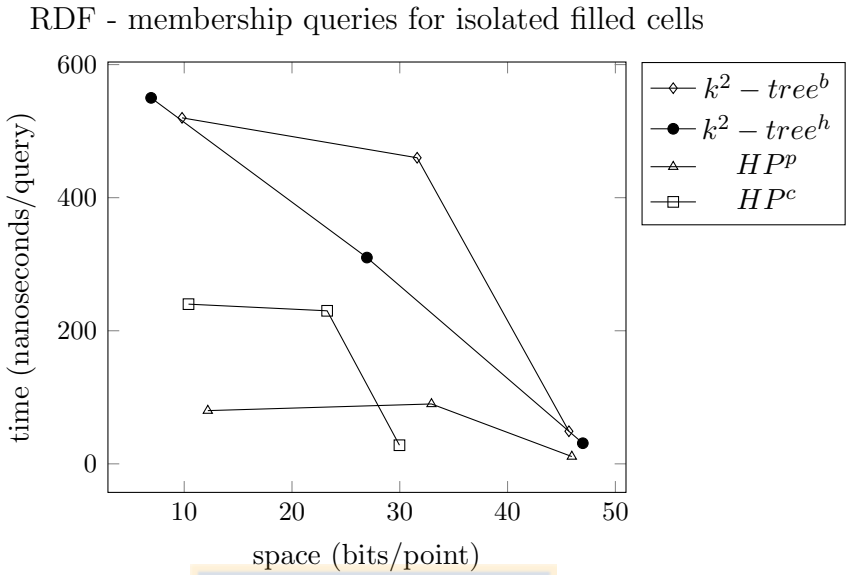


Figura A.6: De izquierda a derecha cada punto representa los datasets: triples-dense, triples-med y triples-sparse

Apéndice B

Ejemplo consulta Membership

En esta sección se explican en profundidad tres ejemplos de membership usando la versión de la estructura de la sección 3.2.

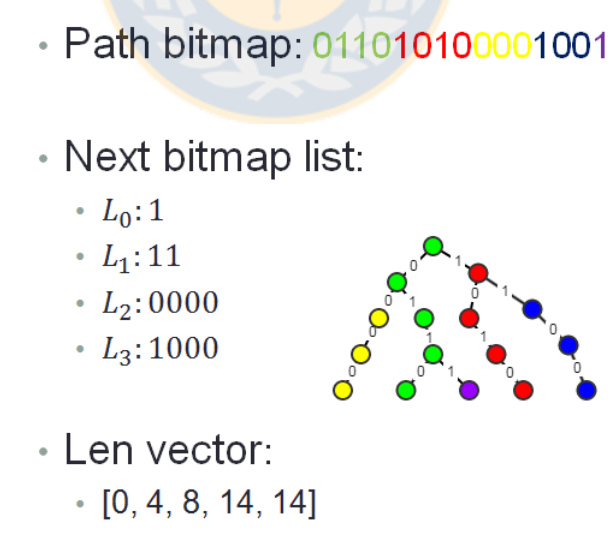


Figura B.1: El árbol es una representación conceptual del resultado de aplicar Heavy path decomposition sobre un trie que almacena quadcodes. Nodos del mismo color representan un mismo heavy path.

B.1 Membership: 0111

En dos iteraciones es posible encontrar el quadcode 0111.

Se definen las siguientes variables:

- POS := Posición en el **Path bitmap**
- HP := Heavy path en el que debería estar el bit en la pos_c en el **path tree**.
- H := Altura del **path tree**

P, L y Len son los bitmaps Path, Next y Len respectivamente. Q es el quadcode a buscar.
Operaciones:

- XOR(Path bitmap, Quadcode, p) := Realiza una operación XOR entre Quadcode y un subconjunto P_p del Path bitmap. P_p es un bitmap cuyo primer bit es el que está en la posición p del Path bitmap y comparte todos los bits desde ese punto en adelante con el Path bitmap. Por ejemplo, $P_3 = 00100000$. Retorna la posición del primer bit en el que P_p y Quadcode difieren. -1 si ambos bitmaps son idénticos.
- Length(bitmap) := Retorna la cantidad de bits que tiene bitmap.
- Rank(bmp, p) := Cantidad de 1's en el bitmap **bmp** hasta la posición **p**.
- EliminateFirstBits(bmp, b) := Eliminar los primeros p bits del bitmap bmp.

Variables auxiliares:

- *position* :=Posición en P del primer bit en el que P y Q difieren.
- *height* :=Altura en la que debería estar el nodo donde P y Q difieren en el **heavy tree**.

- $hasNext$:=Indica si existe una continuación o no. Si hay una $hasNext = 1$. $hasNext = 0$ en caso contrario.
- $rank$:=Valor de la operación Rank.

Por cada paso del algoritmo se muestra cómo funciona la implementación. Además, se muestra cómo funciona **conceptualmente** mediante el uso del **path tree**.

Inicialización:

$Q = 0111$

$POS = 0$

$HP = 0$

$H = Length(0111) - 1 = 3$

Iteración 1:

$position := XOR(P = 01101010000100, Q = 0111, POS = 0) = 3$

$height := H - (Length(0111) - (position - POS) - 1) = 3 - (4 - (3 - 0) - 1) = 3$

$hasNext := L_{height=3}[HP = 0] = 1$

$rank := Rank(L_{height=0} = 1, HP = 0) = 1$

$EliminateFirstBits(Q, position - POS)$

$POS = Len[height + 1] + (rank - 1) * Length(Q) = 4 + (1 - 1) * 1 = 14$

$HP = Length(L_{height}) + rank - 1 = 4 + 1 - 1 = 4$

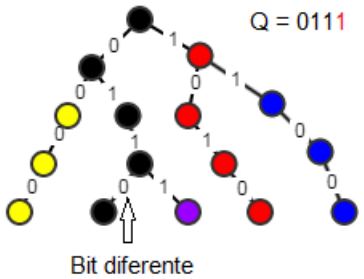


Figura B.2: En negro están marcados los nodos por los que navega el algoritmo. El último bit del quadcode no coincide con el del primer heavy path.

Iteración 2:

$position := XOR(P = 01101010000100, Q = 0, POS = 14) = -1$

Por lo tanto, Q está en la estructura.

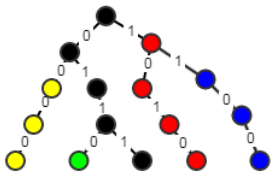


Figura B.3: Se encontró una ruta que contiene todos los bits del quadcode.

B.2 Membership: 1100

En tres iteraciones es posible encontrar el quadcode 1100.

Inicialización:

$Q = 1100$

$POS = 0$

$HP = 0$

$H = Length(0111) - 1 = 3$

Iteración 1:

$position := XOR(P = 01101010000100, Q = 1100, POS = 0) = 0$

$height := H - (Length(1100) - (position - POS) - 1) = 3 - (4 - (0 - 0) - 1) = 0$

$hasNext := L_{height=0}[HP = 0] = 1$

$rank := Rank(L_{height=0} = 1, HP = 0) = 1$

$POS = Len[height + 1] + (rank - 1) * Length(Q) = 4 + (1 - 1) * 4 = 4$

$HP = Length(L_{height}) + rank - 1 = 1 + 1 - 1 = 1$

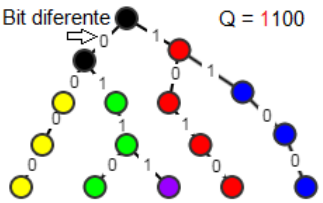


Figura B.4: El primer bit del primer heavy path difiere con el primer bit del quadcode. Continuar con el segundo heavy path es posible.

Iteración 2:

$position := XOR(P = 01101010000100, Q = 1100, POS = 4) = 5$

$height := H - (Length(1100) - (position - POS) - 1) = 4 - (3 - (5 - 4) - 1) = 1$

$hasNext := L_{height=1}[HP = 1] = 1$

$rank := Rank(L_{height=1} = 11, HP = 1) = 2$

$EliminateFirstBits(Q, position - POS)$

$POS = Len[height + 1] + (rank - 1) * Length(Q = 100) = 8 + (2 - 1) * 3 = 11$

$HP = Length(L_{height}) + rank - 1 = 2 + 2 - 1 = 3$

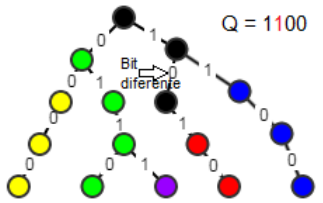


Figura B.5: El segundo bit del segundo heavy path difiere del segundo del quadcode. Continuar con el cuarto heavy path es posible.

Iteración 3:

$position := XOR(P = 01101010000100, Q = 100, POS = 11) = -1$

Por lo tanto, Q está en la estructura.

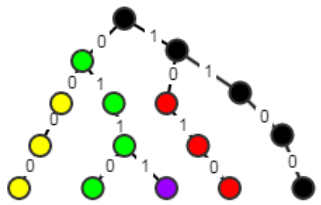


Figura B.6: Se encontró una ruta que contiene todos los bits del quadcode.

B.3 Membership: 0100

El quadcode no se encuentra en la estructura.

Inicialización:

$Q = 0100$

$POS = 0$

$HP = 0$

$H = Length(0111) - 1 = 3$

Iteración 1:

$position := XOR(P = 011001000001, Q = 0100, POS = 0) = 2$

$height := H - (Length(0100) - (position - POS) - 1) = 3 - (4 - (2 - 0) - 1) = 2$

$hasNext := L_{height=2}[HP = 0] = 0$

Por lo tanto, Q no está en la estructura.

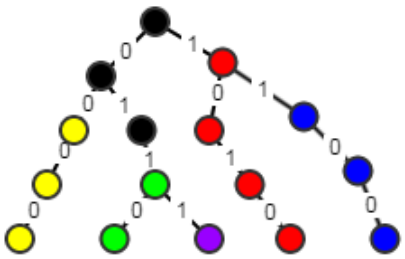


Figura B.7: No es posible navegar a otro heavy path. Por lo tanto, el quadcode no está en la estructura.

Apéndice C

Misceláneo

C.1 Operación XOR

XOR u OR exclusivo es una operación entre bits. La **Figura C.1** muestra la tabla de verdad de la operación XOR.

XOR Truth Table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Figura C.1: Tabla de verdad de la operación XOR.

Esta operación permite saber si dos bitmaps son iguales o no. En caso de no serlos, también es posible saber en qué bits son diferentes. Por ejemplo, 0110 XOR 0110 = 0000. En este ejemplo XOR retorna 0, lo que indica que ambos bitmaps son

iguales. Otro ejemplo, 0110 XOR 0010 = 0100. El resultado es distinto de 0 por lo que los bitmaps son diferentes. Además, se sabe que el segundo bit es diferente porque el bitmap resultante contiene un 1 en la segunda posición.

C.2 Funciones y estructuras auxiliares

1. Queue: Cola, estructura de datos con las siguientes funciones:
   - push(x) := Inserta x a la cola.
   - front := Retorna el elemento que primero se insertó en la cola, de los que están almacenados.
   - pop := Elimina el elemento que primero se insertó en la cola, de los que están almacenados.
2. Exists(n) := Retorna **True** si n es no nulo. **False** de otro modo.
3. NumChild(n) := Retorna el número de hojas que tiene el subárbol con raíz en n.
4. LeftChild(n) := Retorna el hijo izquierdo de n. Nulo si no existe.
5. RightChild(n) := Retorna el hijo derecho de n. Nulo si no existe.
6. Root(T) := Retorna el nodo raíz de T.
7. IsEqual(a, b) := Retorna **True** si a y b son iguales. **False** en otro caso.
8. Quadcode: bitmap que representa un punto en un quadtree.
   - BitAt(i) := valor del bit en la posición i en el quadcode.
9. Bitmap
   - XOR(quadcode, i) := Posición en Bitmap donde el primer bit de quadcode difiere con el de Bitmap. La comparación empieza desde la posición i del Bitmap.