UNIVERSIDAD DEL BÍO-BÍO

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

Estructuras de datos compactas para representar coberturas Ráster

Por
Miguel Angel Saavedra Aravena

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

Dirigido por:
Dr. Gilberto Gutiérrez
Universidad del Bío-Bío, Chillán, Chile

Co-Dirigido por:
Dr. Guillermo de Bernardo
Universidade da Coruña, A Coruña, España

2025 - II

A mi madre,

Miriam del Carmen Aravena Figueroa

(1961 – 2021)

Agradecimientos

Quiero expresar mi más profundo agradecimiento a mis padres, Gonzalo y Miriam, por entregarme todo sin esperar nada a cambio y por ser siempre un pilar fundamental en mi vida.

A mis hermanos, Franco y Gonzalo, por su constante apoyo y compañía a lo largo de este camino.

Al profesor Guillermo, por su valiosa guía durante el desarrollo de esta tesis, siempre dispuesto a resolver dudas y a colaborar en cada etapa.

Al profesor Gilberto, por acompañarme desde el inicio de este trabajo y por todos los años de enseñanza y orientación en mi formación profesional.

A todos los profesores de la Universidad del Bío-Bío, quienes con su conocimiento y vocación aportaron de manera significativa a mi crecimiento académico y personal. En especial, agradezco a los profesores Gilberto Gutiérrez, Joel Fuentes y Miguel Romero.

Finalmente, al Proyecto Fondecyt 1230647, por el apoyo brindado para la realización de esta investigación.

Resumen

Los rásteres constituyen un modelo fundamental en sistemas de información geográfica para representar fenómenos espaciales, almacenando valores geoespaciales (como temperatura, elevación o humedad) en una matriz de celdas. Con el avance en tecnologías de observación (sensores remotos, imágenes satelitales y drones), los volúmenes de datos rasterizados han crecido significativamente en resolución y escala, planteando desafíos en almacenamiento y procesamiento eficiente. En este contexto, las estructuras de datos compactas emergen como una solución esencial, permitiendo representar estos conjuntos masivos aprovechando las redundancias espaciales y estadísticas inherentes a los datos geográficos, logrando reducciones significativas en espacio sin comprometer la capacidad de realizar operaciones esenciales.

En este trabajo se presenta el diseño e implementación de una estructura de datos compacta para representar rásteres. Nuestra propuesta se basa en una secuencia de matrices binarias que corresponden a la codificación binaria de los valores (números enteros) del ráster. Estas matrices luego se codifican en una estructura de datos compacta que llamamos k^2-MS , básicamente es una secuencia de k^2-tree. El k^2-tree permite representar de forma compacta matrices binarias dispersas, mientras que el interleaved k^2-tree (ik^2-tree) extiende este enfoque para manejar secuencias de matrices binarias; en este trabajo ambos se adaptan a la representación de rásteres. A partir de estas dos estructuras se definen dos variantes, ik^2-MSv1 e ik^2-MSv2. La primera combina el uso del ik^2-tree con el enfoque de k^2-MS , aprovechando su limitación en la cantidad de matrices que componen la secuencia, la cual depende del número de bits utilizados para representar los valores del ráster (generalmente 32 o 64). Esta restricción permite realizar recorridos de manera eficiente mediante operaciones sobre enteros. La segunda incorpora además el uso de las instrucciones `PDEP` y `PEXT`, que mejoran el tiempo de ejecución de las consultas.

Las estructuras propuestas se evalúan mediante experimentos con distintos conjuntos de datos. Los resultados muestran que nuestro enfoque es competitivo frente al k^2-raster, una de las estructuras más reconocidas en la literatura. Específicamente, ik^2-MSv2 es hasta un 74,54 % más rápida que el k^2-raster al procesar la consulta `Window`. También comparamos con la versión heurística del k^2-raster, que aplana los últimos p niveles del árbol y aplica compresión estadística. En nuestro caso, aplicamos únicamente la estrategia de aplanamiento en ik^2-MSv2. Los resultados indican que ik^2-MSv2 $p = 2$ supera al k^2-raster $p = 4$ hasta en un 47,54 % y un 41,82 % para las consultas `Window` y `Window Range`, respectivamente. En cuanto al almacenamiento, ik^2-MSv2 muestra mejoras promedio cercanas al 40,74 % respecto al k^2-raster en los distintos conjuntos de datos. Asimismo, la variante con aplanamiento ik^2-MSv2 $p = 2$ supera en promedio al k^2-raster $p = 4$ en alrededor de un 36,76 %.

Índice

Índice de Figuras

Índice de Tablas

Índice de Algoritmos

Capítulo 1

Introducción

En este capítulo se presenta el contexto general de la investigación y se establecen las bases sobre las que se desarrolla el trabajo. En primer lugar, se describe la motivación, destacando la relevancia de los modelos ráster en la representación de información geográfica y los desafíos asociados a su almacenamiento y consulta. Luego, se definen la hipótesis y los objetivos. Posteriormente, se muestra la metodología adoptada en la investigación, desde la revisión de literatura hasta la implementación y evaluación experimental de las estructuras propuestas. Finalmente, se detallan las contribuciones más relevantes de este trabajo y se ofrece una visión general de la organización del documento.

1.1. Motivación y Objetivos de la tesis

La representación de información geográfica es un aspecto fundamental en múltiples áreas como la cartografía, la teledetección, la geoinformática y el análisis de fenómenos ambientales. Para describir digitalmente la superficie terrestre y sus características, se emplean distintos modelos de datos espaciales, cada uno con enfoques y aplicaciones específicas. Entre ellos, se distinguen principalmente dos categorías: los modelos vectoriales y los modelos ráster [4]. Los modelos vectoriales representan la información geográfica mediante secuencias finitas de puntos, líneas y polígonos; por ejemplo, una red de calles, la delimitación de parcelas o la ubicación de elementos específicos como postes de luz o estaciones meteorológicas. Los modelos ráster, en cambio, representan la información geográfica particionando el espacio en una cuadrícula finita de celdas, asignando un valor a cada celda; por ejemplo, una imagen satelital donde cada píxel registra un valor de temperatura, elevación, humedad o concentración de contaminantes. Ambos modelos son ampliamente utilizados y resultan complementarios según el tipo de fenómeno que se desea representar. En este trabajo se centra la atención en el modelo ráster, el cual resulta especialmente útil para la representación de fenómenos geoespaciales sobre una superficie.

Los rásteres constituyen una representación ampliamente utilizada para modelar fenómenos geográficos sobre una superficie. Cada ráster R consiste en una matriz de celdas de dimensiones $(n \times m)$, donde cada celda contiene el valor capturado de un fenómeno geoespacial asociado a una ubicación específica. En la Figura 1.1 se observa un ejemplo: a la izquierda, un mapa que representa un área geográfica y, a la derecha, la cuadrícula que discretiza ese espacio en celdas.

Cada celda almacena un valor que describe el fenómeno medido en esa posición, lo que permite realizar análisis científicos sobre el fenómeno representado.

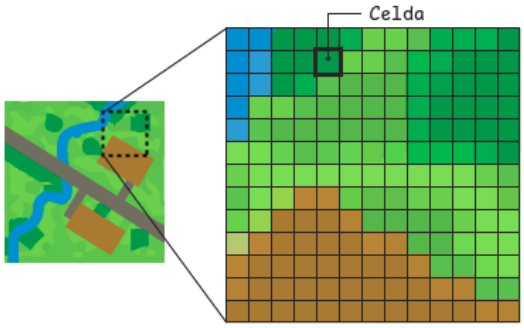

Figura 1.1: Representación de un ráster[1]

Con el avance de las tecnologías de observación y captura de datos, como sensores remotos, imágenes satelitales de alta resolución, drones y estaciones de monitoreo, el volumen y la resolución de los datos ráster han crecido de manera considerable. La Tabla 1.1 muestra una estimación de cómo el tamaño de un ráster incrementa rápidamente con sus dimensiones. Por ejemplo, un sólo ráster de 21.000 × 21.000 ocupa alrededor de 1,64 GB; sin embargo, si se registra una captura cada hora durante un mes, el volumen asciende a más de 1,16 TB, lo que representa un desafío considerable para la capacidad de muchos sistemas convencionales.

Dimensiones	**1 ráster**	**1 mes**	**1 año**	**1 década**	**50 años**
224 × 464	0,40 MB	285,47 MB	3,35 GB	33,45 GB	167,27 GB
1000 × 1000	3,81 MB	2,68 GB	32,19 GB	321,87 GB	1,57 TB
11.000 × 11.000	461,58 MB	324,55 GB	3,80 TB	38,03 TB	190,16 TB
21.000 × 21.000	1,64 GB	1,16 TB	13,86 TB	138,62 TB	693,08 TB
31.000 × 31.000	3,58 GB	2,52 TB	30,21 TB	302,06 TB	1,47 PB
41.000 × 41.000	6,26 GB	4,40 TB	52,84 TB	528,37 TB	2,58 PB
51.000 × 51.000	9,69 GB	6,81 TB	81,75 TB	817,55 TB	3,99 PB
61.000 × 61.000	13,86 GB	9,75 TB	116,96 TB	1,14 PB	5,71 PB
71.000 × 71.000	18,78 GB	13,20 TB	158,45 TB	1,55 PB	7,74 PB
81.000 × 81.000	24,44 GB	17,19 TB	206,23 TB	2,01 PB	10,07 PB
91.000 × 91.000	30,85 GB	21,69 TB	260,29 TB	2,54 PB	12,71 PB

Tabla 1.1: Tamaños de ráster y almacenamiento durante períodos de tiempo

Es fundamental contar con mecanismos que permitan reducir el espacio requerido para almacenar estos datos, pero manteniendo tiempos de acceso y consulta lo más reducidos posible. Aquí es donde cobran relevancia las estructuras de datos compactas, que buscan aprovechar técnicas de

[1]Fuente: `http://localization.veremes.net/FME2022/FME_Transformers/!FME_Geometry/Raster.htm`

compresión y representación eficiente para disminuir el espacio ocupado sin sacrificar el rendimiento de las operaciones. En este contexto, el presente trabajo propone el diseño e implementación de una estructura de datos compacta para rásteres, basada en el ik^2-tree [7], que permite representar esta información de forma eficiente y resulta competitiva tanto en almacenamiento como en tiempo de ejecución en comparación con otras estructuras, como el k^2-raster [9].

1.1.1. Consultas

En esta sección se presentan las consultas principales que pueden ejecutarse sobre la estructura de datos propuesta. Estas operaciones permiten acceder y examinar el contenido del ráster, y constituyen las operaciones básicas necesarias para trabajar con él. A continuación, se describe cada una de ellas y el tipo de información que devuelve.

1. `Access`(r, c): Obtiene el valor de la posición de la matriz $R[r, c]$, donde r es la fila y c la columna.
2. `Window`(r_1, r_2, c_1, c_2): Obtiene todos los valores pertenecientes al rectángulo formado por las esquinas (r_1, c_1) y (r_2, c_2).
3. `Window Range`(r_1, r_2, c_1, c_2, v_{min}, v_{max}): Al igual que la consulta `Window`(r_1, r_2, c_1, c_2), obtiene los valores del rectángulo formado por las esquinas (r_1, c_1) y (r_2, c_2). Sin embargo, esta considera sólo los valores que están dentro del rango [v_{min}, v_{max}].

1.1.2. Hipótesis

Es posible diseñar estructuras de datos compactas para representar rásteres que se beneficien de las características de los datos geográficos, como la ley de Tobler y la repetitividad, entre otras. Estas estructuras deben permitir responder eficientemente a las consultas, mejorando tanto el tiempo de ejecución como la eficiencia en el uso del almacenamiento, en comparación con los enfoques actuales.

1.1.3. Objetivo General

Diseñar estructuras de datos compactas para representar rásteres, utilizando variantes del k^2-tree u otras estructuras afines, con el fin de reducir el uso de almacenamiento y optimizar el tiempo de ejecución de consultas.

1.1.4. Objetivos Específicos

- Desarrollar variantes de estructuras compactas basadas en k^2-tree[3] u otras similares, orientadas a rásteres.
- Implementar dichas estructuras y las consultas asociadas (como `Access`, `Window`, `Window Range`).
- Evaluar experimentalmente el rendimiento de las estructuras propuestas, comparándolas con métodos previos.

1.1.5. Metodología de Trabajo

La investigación sigue una metodología iterativa, en la cual se realizan refinamientos en base a los resultados obtenidos y a enfoques previos de la literatura. Esta metodología comprende las siguientes fases:

1. **Revisión de literatura:** Se analizan trabajos previos relacionados con rásteres y estructuras de datos compactas, con el fin de identificar los fundamentos teóricos y las limitaciones de los métodos existentes.

2. **Diseño de estructuras:** Se proponen y definen enfoques para la representación de rásteres espaciales en base a métodos previos, los cuales posteriormente se refinan a partir de los resultados obtenidos en las fases de implementación y evaluación.

3. **Implementación:** Se desarrolla una implementación en C++ de las estructuras y de las operaciones de consulta, incorporando ajustes derivados del proceso iterativo y de los resultados experimentales.

4. **Evaluación experimental:** Se realizan pruebas con conjuntos de datos reales para medir el espacio de almacenamiento y los tiempos de respuesta a distintas consultas, cuyos resultados permiten retroalimentar y mejorar las fases de diseño e implementación.

1.2. Contribuciones

Este trabajo propone un enfoque para representar rásteres usando ik^2-tree [7]. Se utiliza la instrucción `PDEP` para acelerar las consultas, mejorando significativamente el tiempo de ejecución. Los algoritmos descritos en esta tesis se implementaron en C++ utilizando la librería SDSL [8].

1.3. Descripción de la estructura de la tesis

Este documento se organiza de la siguiente manera: El Capítulo 1 describe los rásteres, presenta las consultas fundamentales (`Access`, `Window` y `Window Range`), los objetivos, hipótesis y la estructura del trabajo. El Capítulo 2 revisa el estado del arte, describiendo estructuras existentes como el k^2-tree acumulado, 3D2D Mapping y el k^2-raster. El Capítulo 3 propone la representación mediante una secuencia de árboles basada en la representación binaria de los valores del dominio del ráster, llamada k^2-MS. El Capítulo 4 presenta la primera versión ik^2-MSv1 que combina k^2-MS y ik^2-tree. El Capítulo 5 introduce una segunda versión, ik^2-MSv2, que además utiliza las instrucciones `PDEP` y `PEXT` del procesador para acelerar los algoritmos. El Capítulo 6 explora la extracción de regiones para reducir el espacio. El Capítulo 7 analiza una variante heurística del k^2-raster con aplanamiento de niveles y compresión estadística. El Capítulo 8 detalla los experimentos realizados, comparando el desempeño en almacenamiento y tiempo de consulta. Finalmente, el Capítulo 9 presenta las conclusiones y el trabajo futuro.

Capítulo 2

Estructuras de Datos Compactas

En este capítulo se presentan las estructuras de datos compactas, cuyo objetivo es representar grandes volúmenes de información empleando la menor cantidad de espacio posible, al mismo tiempo que permiten realizar operaciones de acceso y consulta de manera eficiente. En particular, se describen dos estructuras relevantes para este trabajo: los vectores de bits o *bitmaps*, que incorporan operaciones básicas como *rank* y *select*, y el k^2-tree, una estructura diseñada para la representación compacta de matrices dispersas y utilizada en distintos contextos relacionados con datos espaciales. Estas estructuras son esenciales para representar grandes volúmenes de datos, como los rásteres, aprovechando las redundancias inherentes a los datos geográficos.

2.1. Vectores de bits o Bitmaps

Un bitmap es una representación de un conjunto de bits, almacenados de manera continua en memoria, lo que permite realizar operaciones rápidas sobre ellos. A esta estructura se añaden operaciones para realizar consultas eficientes:

- $rank_1(b, p)$: Devuelve el número de bits 1 en el rango $[0, p-1]$ del bitmap b.
- $select_1(b, k)$: Devuelve la posición del k-ésimo bit 1 en el bitmap b, permitiendo acceder de manera directa a la localización de un determinado elemento sin necesidad de recorrer toda la secuencia.

Estas operaciones constituyen la base para el diseño de algoritmos que requieren contar o acceder a posiciones específicas dentro de secuencias binarias de gran tamaño, manteniendo siempre un uso reducido de espacio y tiempos de consulta. Por esta razón, los bitmaps se consideran un componente fundamental sobre el que se construyen otras estructuras de datos compactas más complejas, como la que se describe en la siguiente sección. En este caso se utiliza la implementación de los vectores de bits, *rank* y *select* de SDSL [8].

2.2. k^2-tree

k^2-tree es una estructura de datos compacta que permite almacenar matrices binarias dispersas de forma eficiente [3]. Inicialmente fue creada con el propósito de almacenar grafos de la Web, pero gracias a su eficiencia, se puede aplicar en diversos campos donde se requiera almacenar grandes cantidades de información que representen relaciones binarias y en general n-arias. Esta estructura se construye de forma jerárquica, dividiendo el espacio de la matriz de forma recursiva. Como se ve en la Figura 2.1 un árbol con $k = 2$, en cada nivel se divide el espacio de la matriz en $k^2 = 4$ cuadrantes, representados por un bit cada uno. Los 1's indican que existe al menos un elemento en el espacio de esa submatriz y los 0's que todos sus elementos son ceros. Estos bits se toman de izquierda a derecha y de arriba a abajo para formar los vectores T (que contiene los niveles internos) y L (para el último nivel).

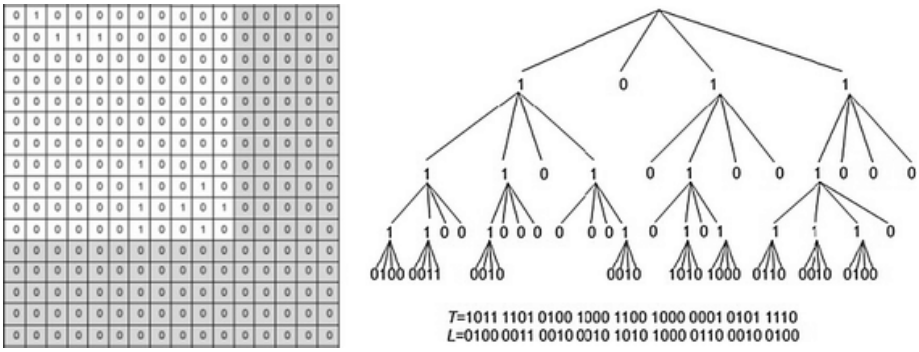

Figura 2.1: Estructura k^2-tree (extraído de [3])

La estructura se representa mediante dos bitmaps: T y L. El vector T almacena información sobre la estructura del árbol, indicando en cada posición si es un nodo interno. El vector L almacena los valores de las hojas, representando las matrices de $k \times k$ en las hojas del árbol.

Además, existe una variante denominada k^2-tree one [1], que permite reducir la cantidad de bits cuando las regiones son uniformes. Se asigna un cero para las regiones homogéneas, y para diferenciarlas se utiliza un nuevo vector de bits T' alineado a los ceros en T. Si $T'[i] = 0$, la región está completamente compuesta por 0's, y si $T'[i] = 1$, la región sólo contiene 1's.

2.3. ik^2-tree

El Interleaved k^2-tree o ik^2-tree [7] es una evolución del k^2-tree, ya que permite almacenar no sólo una matriz binaria sino también una secuencia de matrices binarias, donde la tercera dimensión se denomina dimensión de partición, y el número total de matrices binarias se denota como $|Y|$.

Conceptualmente, cada matriz binaria en la secuencia se representa mediante un k^2-tree independiente. Luego, los bits equivalentes en cada nodo, es decir, aquellos que ocupan la misma posición en los diferentes árboles, se agrupan para formar un único nodo en el árbol. En la raíz,

cada uno de los cuadrantes k^2 contiene inicialmente $|Y|$ bits; en consecuencia, la raíz almacena $|Y| \times k^2$ bits. En los niveles posteriores, cada cuadrante hijo almacena m bits, donde m representa el número de unos presentes en el nodo padre. La Figura 2.2 muestra tanto la estructura conceptual como su representación real utilizando los vectores de bits $T : L$.

Para navegar por el árbol, la posición del primer hijo de un cuadrante se calcula utilizando la expresión $pos_{child} = \text{rank}_1(T, pos) \times k^2 + adjust$, donde el término $adjust = |Y| \times k^2$ se añade porque los primeros $|Y|$ bits no se almacenan en la raíz del árbol. Al mismo tiempo, se debe calcular otro rango para determinar el número de bits activos correspondientes al nodo hijo. Este valor, denotado como m_{child}, se obtiene de la diferencia $\text{rank}_1(T, pos+m) - \text{rank}_1(T, pos)$, donde el segundo rango es el mismo que se usó previamente al calcular la posición del primer hijo.

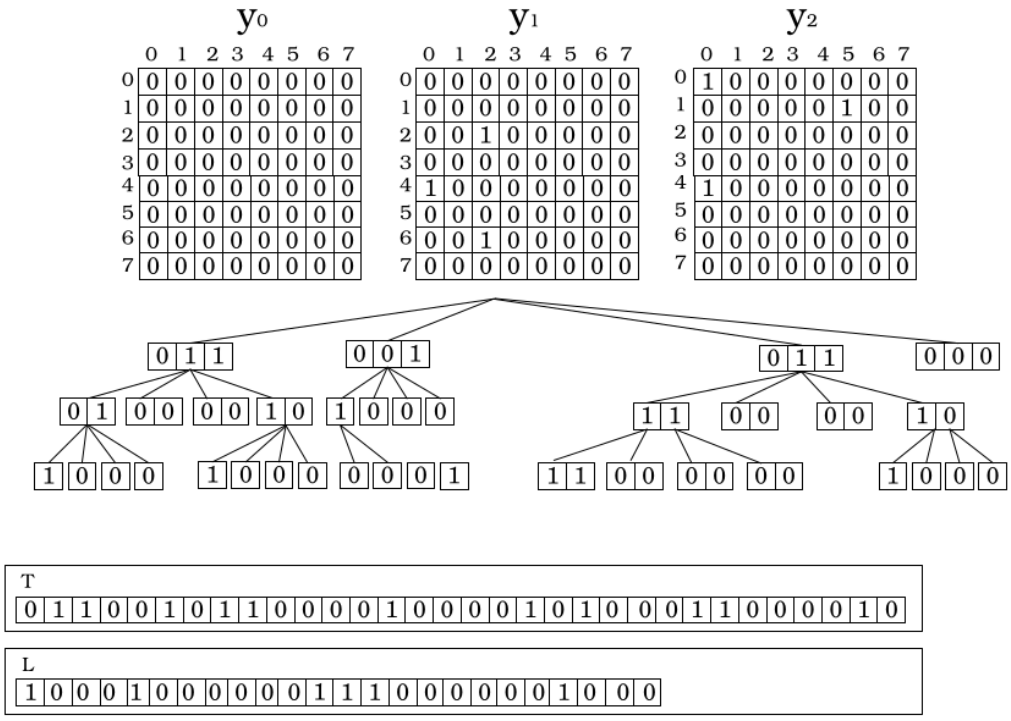

Figura 2.2: Estructura del ik^2-tree (tomada de [7])

Capítulo 3

Estructuras de datos para Coberturas Ráster

En este capítulo se describen distintas estructuras de datos diseñadas para la representación compacta de rásteres a partir del k^2-tree. Se presentan enfoques que adaptan o extienden esta estructura para manejar valores numéricos y espaciales de manera conjunta, con el objetivo de optimizar tanto el espacio de almacenamiento como la eficiencia en las consultas. En particular, se revisan tres variantes: el k^2-acc, que emplea matrices acumuladas; 3D2D Mapping, que transforma el ráster en una matriz binaria de dos dimensiones; y el k^2-raster, que incorpora directamente valores numéricos en la estructura. Cada una de ellas se analiza en las secciones siguientes, mostrando su representación, funcionamiento y limitaciones.

3.1. k^2-trees acumulados

El k^2-tree acumulado o k^2-acc [1] [2] descompone el ráster original R en un conjunto de k matrices binarias $\{B_1, B_2, \ldots, B_k\}$, donde k representa la cantidad de valores únicos presentes en R. A cada valor distinto $v_i \in V$ (siendo V el conjunto de valores únicos de R), se asocia una matriz binaria B_i definida como:

$$B_i(x, y) = \begin{cases} 1, & \text{si } R(x, y) \leq v_i \\ 0, & \text{en otro caso} \end{cases}$$

De esta forma, cada matriz B_i indica las posiciones de los valores menores o iguales a v_i dentro del ráster R. En la Figura 3.1 se muestran las matrices binarias $\{0, 8, 9, 10, 11, 12, 14\} \in B_i$ que indican las celdas acumuladas cuyo valor es menor o igual a sí mismo. Para acceder a una celda (i, j) se recorren las matrices binarias hasta encontrar la primera matriz b_i de izquierda a derecha que contiene un 1 en la celda (i, j). Para ello se realiza una búsqueda binaria entre la secuencia de matrices hasta llegar a la matriz resultado. El principal inconveniente de este enfoque radica en que, al depender de la cantidad de valores distintos, un aumento excesivo en la cantidad de valores diferentes provoca un incremento en el almacenamiento, superando al ráster original.

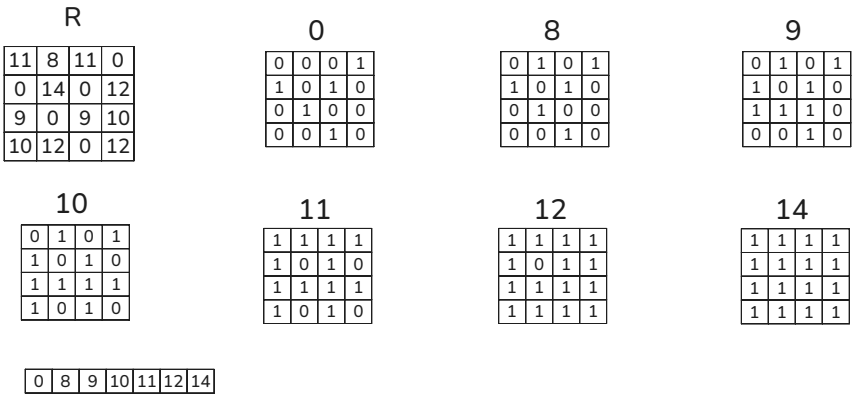

Figura 3.1: k^2-tree acumulado

3.2. 3D2D Mapping

3D2D Mapping [12] consiste en transformar un ráster en una matriz binaria de dos dimensiones. Para realizar este proceso, se utiliza una matriz $B_{i,j} \in \{0,1\}$ de tamaño $(N \times M)$, donde la primera dimensión N representa los valores distintos y M codifica las dimensiones espaciales mediante la curva de llenado Z order [11]. El resultado de esta operación genera una matriz binaria que se almacena en un k^2-tree para permitir consultas eficientes. Para acceder a una celda, se calcula el Z order de (i, j) que indica la columna en B. El Z order se ilusta en la Figura 3.2b. Para encontrar el valor, primero se calculan los vecinos directos, que en este caso corresponde a un único valor, ya que la matriz B contiene un 1 por cada columna. El vecino directo encontrado corresponde al índice del valor en el arreglo que contiene los valores diferentes. Para obtener el resultado, se accede a este arreglo con $tree.directNeigh(zOrder(i, j))$. Dado que esta estructura depende de los valores distintos, presenta la misma limitación que k^2-acc.

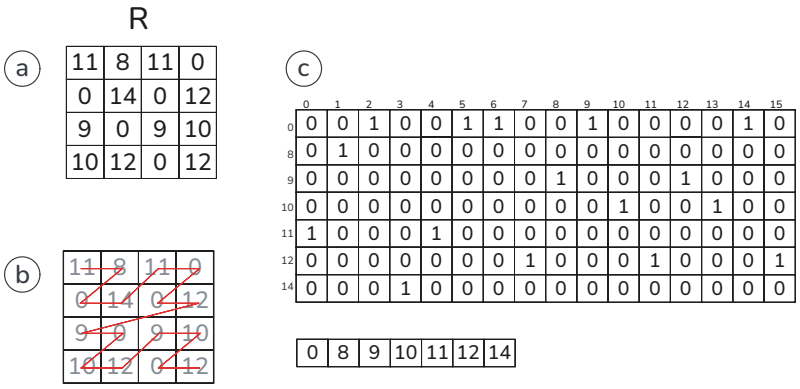

Figura 3.2: 3D2D Mapping

3.3. k^2-raster

Esta estructura de datos constituye una modificación del k^2-tree que permite almacenar matrices Esta estructura de datos constituye una modificación del k^2-tree que permite almacenar matrices numéricas en lugar de matrices binarias [9]. Conserva la misma estructura que el k^2-tree original, pero incorpora en cada nodo i el rango de valores mínimo y máximo ($l_{\text{mín}}$ y $l_{\text{máx}}$) correspondientes a cada uno de los k^2 cuadrantes. La estructura continúa con su descomposición mientras $l_{\text{mín}}$ y $l_{\text{máx}}$ sean diferentes, lo que permite indexar tanto la dimensión espacial como la de valores. En la Figura 3.3 se muestra la representación conceptual, donde en cada paso se seleccionan los valores mínimo y máximo: las celdas en gris oscuro representan el valor máximo, mientras que las de gris claro corresponden al mínimo.

Como se observa en la Figura 3.4, para almacenar estos valores se aplica una codificación diferencial, en la cual los nodos hijos almacenan la diferencia absoluta entre su valor y el de su nodo padre. Este enfoque guarda únicamente el cambio de $l_{\text{mín}}$ y $l_{\text{máx}}$ con respecto a su padre, optimizando así el espacio de almacenamiento. En este proceso, se cumple que $l_{\text{máx},padre} \geq l_{\text{máx},hijo}$ y $l_{\text{mín},padre} \leq l_{\text{mín},hijo}$. Además, cuando una región es completamente homogénea, se mantiene únicamente el valor máximo. Los niveles internos T del k^2-tree se almacenan en un vector de bits denominado *Tree*, que codifica la topología del árbol, mientras que los valores $l_{\text{mín}}$ y $l_{\text{máx}}$ se guardan utilizando DACs [5], una estructura eficiente para el almacenamiento de enteros de longitud variable. Además, se emplea una estrategia en la cual se aplican diferentes valores de k a distintos niveles del árbol: específicamente, k_1 para los primeros n_1 niveles y k_2 para los restantes.

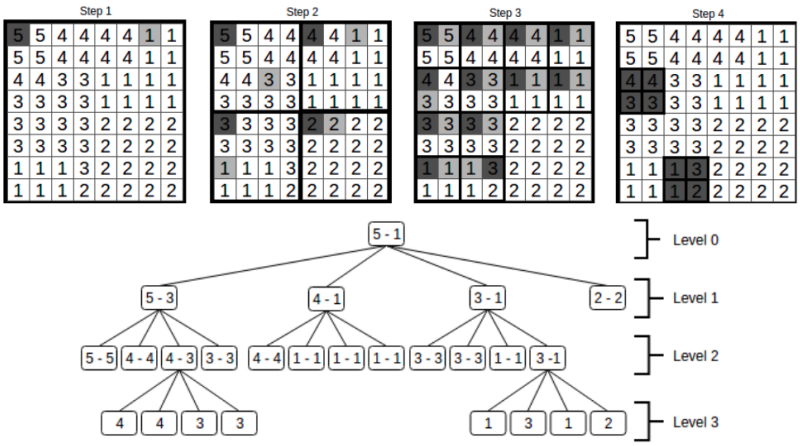

Figura 3.3: Representación conceptual de k^2-raster (extraído de [9])

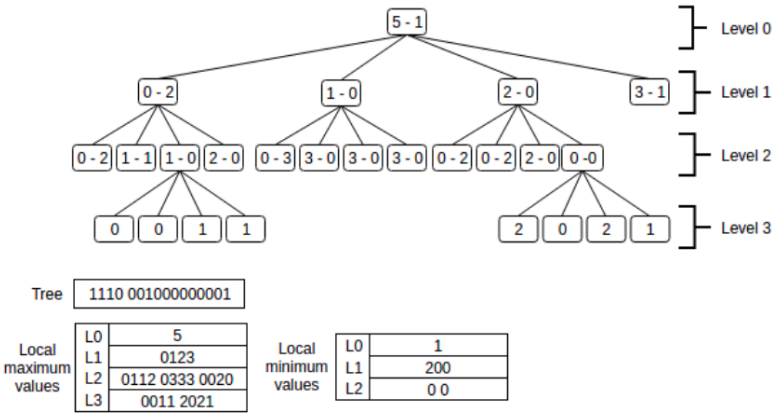

Figura 3.4: Representación real de k^2-raster usando codificación diferencial (extraído de [9])

Capítulo 4

Nuestra Propuesta: Secuencia de árboles

Este capítulo describe una primera aproximación de nuestra propuesta para representar los elementos de un ráster R. A este enfoque se le denomina k^2-MS, donde MS hace referencia a *Matrix Sequence*. En la Tabla 4.1 se describen las variables utilizadas en la propuesta. Este enfoque se basa en la representación binaria de los valores de R. Considerando una matriz de enteros, se descompone en w matrices binarias, cada una correspondiente a bits específicos de la codificación binaria de los valores de las celdas de R. Por ejemplo, si se usan $w = 4$ bits para representar cada valor v de una celda (i, j), este se codifica como una secuencia binaria $R_{i,j} = b_0b_1b_2b_3$, donde b_0 es el bit más significativo y b_3 el menos significativo. Esta codificación se distribuye entre las matrices de modo que la matriz $B^k_{i,j}$ contiene el bit b_k de $R_{i,j}$; es decir, $B^k_{i,j} = b_k \in R_{i,j}$, con $0 \leq k < w$ y dimensiones $0 \leq i < n$, $0 \leq j < m$. Así, B^0 almacena los bits más significativos, mientras que B^{w-1} contiene los menos significativos.

Nombre	Descripción
R	Ráster de tamaño $n \times m$
v	Variable entera de 32 bits representada en el ráster R
$R_{i,j}$	Celda i, j de R, con $0 \leq i < n$ y $0 \leq j < m$
v_{max}	Valor máximo de la variable v almacenado en R
w	$w = \lceil \log_2 v_{max} \rceil$, número de bits usados para representar un valor $R_{i,j}$ de R
b_k	k-ésimo bit ($0 \leq k < w$) de la representación binaria de un valor de la variable v
B	Matriz binaria ($B_{i,j} = 0$ o $B_{i,j} = 1$)

Tabla 4.1: Variables utilizadas en la descripción del enfoque propuesto

En la Figura 4.1 se muestra la representación del ráster R mediante 4 matrices binarias. Por ejemplo, el valor $R_{1,3} = 12$ ($b = 1100$) se representa como $B^0_{1,3} = 1$, $B^1_{1,3} = 1$, $B^2_{1,3} = 0$ y $B^3_{1,3} = 0$. En general, el número de matrices binarias depende de la cantidad de bits (w) necesarios para representar el valor máximo de v en R. Si los valores son enteros de 32 bits, se requieren 32 matrices binarias. Esta estructura de datos consiste básicamente en w k^2-trees, uno

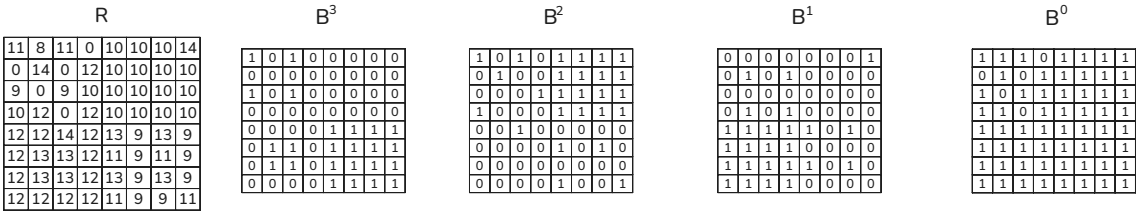

Figura 4.1: Matriz original y su descomposición binaria.

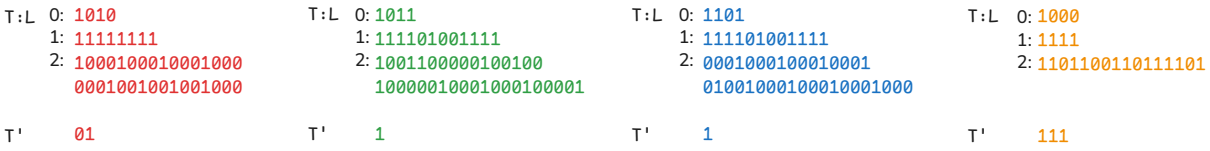

Figura 4.2: Secuencia de k^2-trees.

por cada matriz binaria. En concreto, se utiliza la variante k^2-tree one [1], que comprime regiones homogéneas de una matriz binaria, es decir, áreas completamente llenas de unos o completamente llenas de ceros. En la Figura 4.2, se muestra la secuencia de k^2 trees que representan cada una de las matrices B^k. Los bits de T' se alinean con los ceros en T, pero se almacenan sólo hasta la última posición con un uno. En las consultas, se comprueban estos límites (véase, por ejemplo, la línea 6 del Algoritmo 5.1.2).

4.1. Consultas

La consulta `Access` recupera una celda (r, c) y se basa en recorrer de forma independiente cada uno de los k^2-trees que conforman la secuencia. Usando los k^2-tree de cada una de las matrices B^k, con $0 \leq k < w$, se recuperan los bits en la posición (r, c), de modo que $B^k_{r,c} = b_k$. Estos bits se concatenan como $b_0 b_1 b_2 \dots b_{w-1}$ para formar el valor original almacenado en $R_{r,c}$. En el caso de la consulta `Window`, el procedimiento es similar, pero se recuperan los bits de todas las celdas dentro de la ventana especificada, reconstruyendo en cada una el valor correspondiente a partir de los distintos árboles.

Estas consultas resultan costosas, ya que se deben recorrer los w k^2-trees para obtener el valor de las celdas. Además, con este esquema no se dispone de un método directo para resolver consultas de ventana por rango de valor (`Window Range`) de forma eficiente. En el siguiente capítulo se aborda esta situación y se presentan algoritmos para resolver las consultas `Access`, `Window` y `Window Range`.

Capítulo 5

Variante ik^2-MSv1

En este capítulo se introduce una mejora sobre la aproximación presentada en el capítulo anterior. El objetivo es abordar la principal limitación de la secuencia de k^2-trees, donde es necesario recorrer múltiples estructuras para recuperar el valor de una celda. Para ello se propone el uso de Interleaved k^2-tree o ik^2-tree [7], que permite representar una secuencia de matrices binarias en un único árbol. Siguiendo la misma notación utilizada en el capítulo anterior, a este enfoque se le denominará ik^2-MSv1. La dimensión que representa la secuencia de matrices se denomina *dimensión de partición* y, en este caso, está limitada a un máximo de 32 o 64. De este modo, para realizar recorridos sólo se necesita recorrer una única estructura. Para llevar a cabo este proceso, se requiere información auxiliar para determinar qué bits contienen hijos en cada instante; a estos bits se les llama bits activos y se representan mediante un entero denominado *mask*.

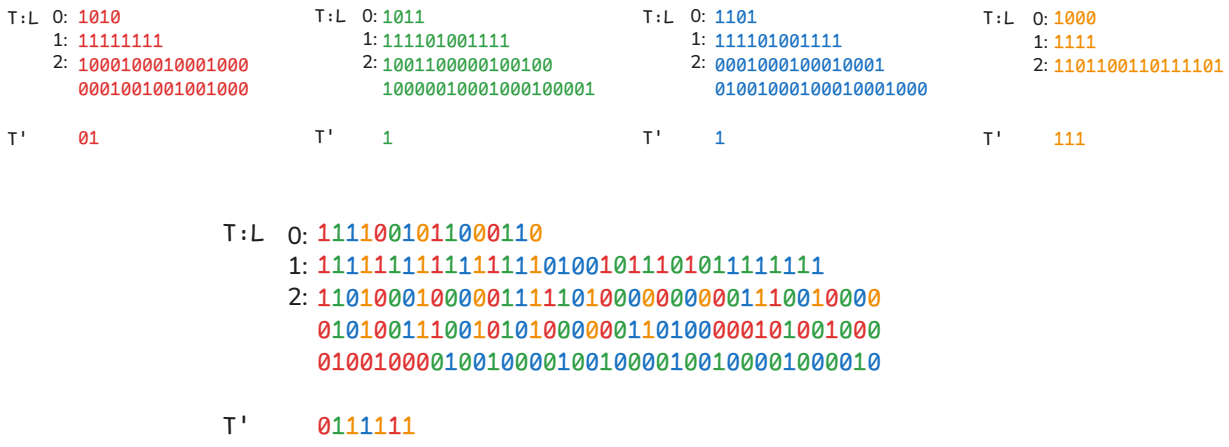

Figura 5.1: Secuencia de k^2-trees fusionados en un ik^2-tree.

Para construir esta estructura, cada matriz binaria se representa mediante un k^2-tree. Posteriormente, los bits se fusionan en un sólo árbol, lo que permite recorrer una única estructura en lugar de manejar una secuencia de k^2-trees independientes (como se describe en el Capítulo 4).

En la Figura 5.1 se muestra la secuencia de árboles del capítulo anterior, los cuales se fusionan generando una sola estructura. En cada cuadrante del árbol fusionado se intercalan los bits activos de las matrices con descomposición de cada nodo. En el nivel 0, todos los bits están activos; por lo tanto, hay $w \times k^2$ bits.

5.1. Access

La consulta `Access` es similar al k^2-tree, pero considera una cantidad variable de bits por nodo. Cada nodo contiene $m \times k^2$ bits, donde m son los bits que tiene cada uno de los k^2 cuadrantes y se calcula contando la cantidad de unos del padre. Inicialmente, se considera $m = w$. Para obtener la posición del primer hijo, se calcula $rank_1(T, pos) \times k^2 + adjust$, donde $rank_1(T, pos)$ cuenta los 1s en $T[0..pos-1]$ y $adjust$ es $w \times k^2$. En cada nodo, se procesan los bits del cuadrante para actualizar la máscara de bits activos ($mask$) y el resultado si existen regiones en T' (Línea 16 del Algoritmo 5.1.1). Finalmente, se actualiza $mask$ con L por última vez y se agrega este valor al resultado (Línea 23 del Algoritmo 5.1.1).

Algoritmo 5.1.1 Access(r, c)

```
 1: if T.empty() and L.empty() then
 2:     return 0
 3: end if
 4: n ← k^(height−1)
 5: row ← ⌊r/n⌋, col ← ⌊c/n⌋
 6: pos ← (k · row + col) × w
 7: r ← r mód n, c ← c mód n
 8: mask ← (1 ≪ w) − 1
 9: result ← min_value
10: while pos < |T| do
11:     ones ← popcount(mask)
12:     if ones = 0 then
13:         return result
14:     end if
15:     rank ← rank_1(T, pos)
16:     mask←updateMask(pos, mask, rank, ones)
17:     n ← n/k
18:     row ← ⌊r/n⌋, col ← ⌊c/n⌋
19:     r ← r mód n, c ← c mód n
20:     pos ← rank · k^2 + w · k^2 + (k · row + col) · ones
21: end while
22: pos ← pos − |T|
23: mask ← updateMaskLeaf(pos, mask)
24: return result + mask
```

Algoritmo 5.1.2 updateMask$(pos,\ mask,\ rank,\ ones)$

```
 1: new_mask ← mask
 2: for each active bit b in mask do
 3:     if T[pos] = 0 then
 4:         clearBit(new_mask, b)
 5:         zr ← pos − rank
 6:         if zr < |T'| ∧ T'[zr] = 1 then
 7:             result ← result + (1 ≪ b)
 8:         end if
 9:     end if
10:     pos ← pos + 1
11: end for
12: return new_mask
```

Algoritmo 5.1.3 updateMaskLeaf$(pos,\ mask)$

```
1: new_mask ← mask
2: for each active bit b in mask do
3:     if L[pos] = 0 then
4:         clearBit(new_mask, b)
5:     end if
6:     pos ← pos + 1
7: end for
8: return new_mask
```

5.2. Window

Esta consulta es similar a la anterior, pero ahora considera una ventana delimitada por las esquinas (r_1, c_1) y (r_2, c_2). El recorrido se realiza en profundidad, descartando cada cuadrante que no intersecta con la ventana (ver Línea 11 del Algoritmo 5.2.1). Durante el recorrido se utiliza una variable *acc* para almacenar el valor acumulado, que representa la información de regiones homogéneas en T' detectadas en niveles superiores. El procedimiento `updateMask()` procesa cada cuadrante y devuelve la máscara $mask'$ junto con acc'. Cuando $mask' = 0$, la región es homogénea; además, si $acc' \neq 0$, este valor se asigna a todas las celdas que intersectan la ventana (Línea 22). Finalmente, al llegar a una hoja, se revisa si cada celda intersecta la ventana y se calcula la máscara mediante `updateMaskLeaf()`, para luego actualizar el resultado sumando $acc + mask'$ (Línea 33 del Algoritmo 5.2.1).

Algoritmo 5.2.1
QueryWindow($result,\ r_1,\ c_1,\ r_2,\ c_2$)

```
 1: width ← c2 − c1 + 1
 2: initialize empty stack
 3: push (0, 0, 0, 0, (1 ≪ w) − 1, w, size, min_value)
 4: while stack not empty do
 5:     pop (lvl, pos, r0, c0, mask, m, n, acc)
 6:     if lvl < height − 1 then
 7:         n′ ← (n/k)
 8:         for c = 0 to k² − 1 do
 9:             r ← r0 + (c ÷ k) · n′
10:             c′ ← c0 + (c mód k) · n′
11:             if region outside window then
12:                 pos ← pos + m
13:                 continue
14:             end if
15:             rk ← rank1(T, pos)
16:             mask′, acc′ ← updateMask(pos, mask, rk, acc)
17:             if mask′ ≠ 0 then
18:                 m′ ← popcount(mask′)
19:                 pos′ ← rk ∗ k² + w ∗ k²
20:                 push (lvl + 1, pos′, r, c′, mask′, m′, n′, acc′)
21:             else
22:                 add acc′ to all cells in intersection
23:             end if
24:             pos ← pos + m
25:         end for
26:     else
27:         pos ← pos − |T|
28:         for each leaf (i, j) do
29:             r ← r0 + i, c ← c0 + j
30:             if (r, c) inside window then
31:                 mask′ ← updateMaskLeaf(pos, mask)
32:                 idx ← (r − r1) · width + (c − c1)
33:                 result[idx] ← acc + mask′
34:             end if
35:             pos ← pos + 1
36:         end for
37:     end if
38: end while
```

Algoritmo 5.2.2
updateMask($pos,\ mask,\ rank,\ acc$)

```
 1: new_mask ← mask
 2: new_acc ← acc
 3: for each active bit b in mask do
 4:     if T[pos] = 0 then
 5:         clearBit(new_mask, b)
 6:         zr ← pos − rank
 7:         if zr < |T′| ∧ T′[zr] = 1 then
 8:             new_acc ← new_acc + (1 ≪ b)
 9:         end if
10:     end if
11:     pos ← pos + 1
12: end for
13: return (new_mask, new_acc)
```

Algoritmo 5.2.3
updateMaskLeaf($pos,\ mask$)

```
1: new_mask ← mask
2: for each active bit b in mask do
3:     if L[pos] = 0 then
4:         clearBit(new_mask, b)
5:     end if
6:     pos ← pos + 1
7: end for
8: return new_mask
```

5.2.1. Ejemplo

En la Figura 5.2 se observa un ejemplo de la consulta `Window` entre $(4, 0)$ y $(7, 3)$. El recorrido comienza en la posición 0, con $mask = 1111$ y $m = 4$. Sólo se procesa el tercer cuadrante que intersecta la ventana (ver p_1 en Figura 5.2). Se procesa bit a bit y si hay un cero en pos, se verifica si $T'[rank_0(T, pos)] = 1$; de ser así, se agrega ese valor a new_acc. En este caso, se añade a new_acc el valor 0011, ya que estas regiones están completamente llenas de unos.

Los primeros $m = 2$ bits de p_2 (10) indican que sus hijos sólo tienen 1 bit activo ($mask =$ 1000), que corresponde a los primeros 4 bits en rojo (0001) en p_3. Los siguientes 2 bits de p_2 (11) muestran que los bits activos son $mask = 1100$ y sus hijos corresponden a los siguientes 8 bits en p_3 (01001000). Los 2 pares restantes de p_2 (10, 10), con $mask = 1000$, generan como hijos, respectivamente, los siguientes 4 bits en p_2 (0100) y los últimos bits en p_3 (1000). En las hojas (p_3), se agrega a la matriz resultado el valor acumulado junto con los bits activos en las hojas.

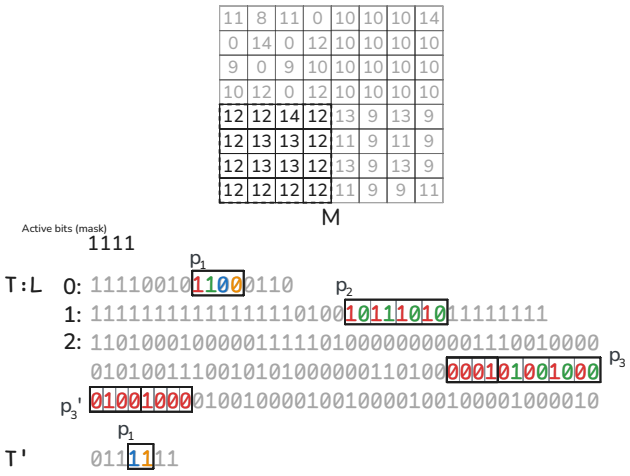

Figura 5.2: Ejemplo de consulta `Window`

5.3. Window Range

Esta consulta permite recuperar una ventana con coordenadas (r_1, c_1) y (r_2, c_2) cuyos valores se encuentran entre (v_1, v_2). A diferencia de la consulta `Window`, en este caso se realizan podas de cuadrantes. Durante el recorrido se mantiene un valor acumulado acc (que representa los bits ya obtenidos en niveles superiores con T'), junto con una máscara $mask$ (que indica los bits activos, es decir, aquellos que aún faltan por calcular o que corresponden a la descomposición pendiente). Con esta información se calculan los posibles valores de cada cuadrante: el valor base es acc, que se toma como límite inferior lb porque, en el peor de los casos, todos los bits activos restantes son 0. En cambio, el límite superior ub se obtiene sumando $mask$ a acc, lo que equivale a asumir que todos los bits restantes toman el valor 1. De esta forma, el intervalo de valores posibles en cada cuadrante es $[lb, ub]$. Si este intervalo no intersecta con el rango buscado $[v_1, v_2]$, se descarta el subárbol completo, evitando exploraciones innecesarias (ver Línea 17 del Algoritmo 5.3.1).

Finalmente, al llegar a una región homogénea o a una hoja, las celdas dentro de la ventana cuyos valores pertenecen a $[v_1, v_2]$ se almacenan en la matriz de salida (Línea 38 del Algoritmo 5.3.1).

Algoritmo 5.3.1
QueryRange($result,\ v_1,\ v_2,\ r_1,\ c_1,\ r_2,\ c_2$)

```
 1: width ← c2 − c1 + 1
 2: initialize empty stack
 3: push (0, 0, 0, 0, (1 ≪ w) − 1, w, size, min_value)
 4: while stack not empty do
 5:     pop (lvl, pos, r0, c0, mask, m, n, acc)
 6:     if lvl < height − 1 then
 7:         n′ ← n/k
 8:         for c = 0 to k^2 − 1 do
 9:             compute (pos, r, c′, n′) as in QueryWindow
10:             if region outside window then
11:                 pos ← pos + m
12:                 continue
13:             end if
14:             rank ← rank_1(T, pos)
15:             mask′, acc′ ← updateMask(pos, mask, rank, acc)
16:             lb ← acc′, ub ← acc′ + mask′
17:             if ub < v1 ∨ lb > v2 then
18:                 pos ← pos + m
19:                 continue
20:             end if
21:             if mask′ ≠ 0 then
22:                 m′ ← popcount(mask′)
23:                 pos′ ← rank ∗ k^2 + w ∗ k^2
24:                 push (lvl + 1, pos′, r, c′, mask′, m′, n′, acc′)
25:             else
26:                 acc′ ∈ [v1, v2] add to all cells in intersection
27:             end if
28:         end for
29:     else
30:         pos ← pos − |T|
31:         for each leaf (i, j) do
32:             r ← r0 + i, c ← c0 + j
33:             if (r, c) inside window then
34:                 mask′ ← updateMaskLeaf(pos, mask)
35:                 val ← acc + mask′
36:                 if v1 ≤ val ≤ v2 then
37:                     idx ← (r − r1) · width + (c − c1)
38:                     result[idx] ← val
39:                 end if
40:             end if
41:             pos ← pos + 1
42:         end for
43:     end if
44: end while
```

Algoritmo 5.3.2
updateMask$(pos, mask, r, acc)$

```
1: new_mask ← mask
2: new_acc ← acc
3: for each active bit b in mask do
4:     if T[pos] = 0 then
5:         clearBit(new_mask, b)
6:         zr ← pos − r
7:         if zr < |T′| ∧ T′[zr] = 1 then
8:             new_acc ← new_acc + (1 ≪ b)
9:         end if
10:    end if
11:    pos ← pos + 1
12: end for
13: return new_mask, new_acc
```

Algoritmo 5.3.3
updateMaskLeaf$(pos,\ mask)$

```
1: new_mask ← mask
2: for each active bit b in mask do
3:     if L[pos] = 0 then
4:         clearBit(new_mask, b)
5:     end if
6:     pos ← pos + 1
7: end for
8: return new_mask
```

Capítulo 6

Variante ik^2-MSv2

En este capítulo se presenta una segunda versión denominada ik^2-MSv2, donde MS nuevamente hace referencia a *Matrix Sequence* y v2 indica que se incorporan las instrucciones `PDEP` y `PEXT` para mejorar el rendimiento. Estas operaciones permiten procesar máscaras de bits de forma más eficiente, reduciendo el tiempo de construcción y acelerando las principales consultas sobre el árbol. Al usar la estructura presentada anteriormente, todavía es necesario recorrer secuencialmente cada bit activo en el cuadrante para obtener la nueva máscara de nodos activos, como ocurre en el Algoritmo `Access` 5.1.1, específicamente en las funciones `updateMask()` y `updateMaskLeaf()`, que contienen un bucle en la Línea 2 en ambos casos. Esto provoca un deterioro en el tiempo de ejecución. Para abordar esta limitación, se emplean las instrucciones Parallel Deposit (`PDEP`) y Parallel Extract (`PEXT`), disponibles en arquitecturas `x86` con soporte para el conjunto de instrucciones `BMI2`.

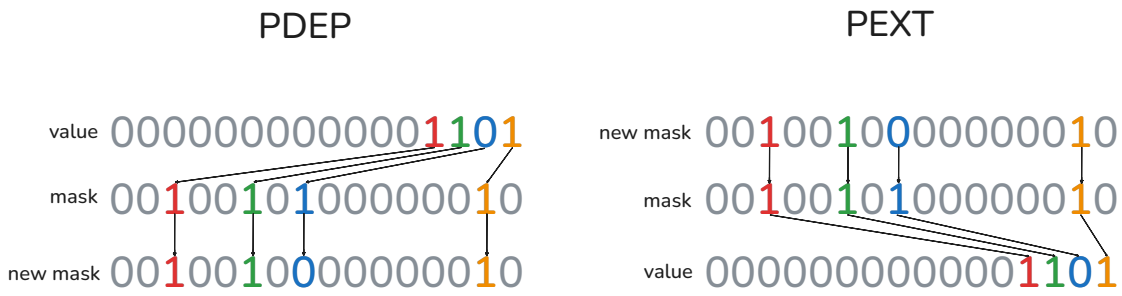

Figura 6.1: Instrucciones `PDEP` y `PEXT`

Estas operaciones permiten procesar en paralelo los bits de cada nodo. En la Figura 6.1 se observa un ejemplo de ambas instrucciones. Por un lado, `PDEP` toma los primeros t bits de *value* (de derecha a izquierda) y los distribuye según *mask*, donde t corresponde al número de unos en *mask*, almacenando el resultado *new_mask*. En este caso se usa *value* = 0000000000001101

y $mask = 0010010100000010$; como $mask$ tiene $t = 4$ bits encendidos, se toman los primeros 4 bits de $value$ (de derecha a izquierda), que son $1, 1, 0, 1$, y se depositan en las posiciones activas de $mask$ (bits $13, 10, 8, 1$ en el mismo orden), obteniendo $new\_mask = 0010010000000010$. Por otro lado, `PEXT` realiza la operación inversa. Toma new_mask y $mask$, y extrae de new_mask los bits que corresponden a las posiciones activas de $mask$, compactándolos consecutivamente a la derecha para generar $value$, manteniendo el orden de los bits seleccionados. En este ejemplo se usa $new\_mask = 0010010000000010$ y $mask = 0010010100000010$; como $mask$ tiene $t = 4$ bits encendidos, `PEXT` toma los bits en las posiciones activas de $mask$ (bits $13, 10, 8, 1$), obtiene la secuencia $1, 1, 0, 1$ extraída de $value$ y la compacta consecutivamente a la derecha para reconstruir $value = 0000000000001101$, preservando su orden.

6.1. Access

En esta versión de la consulta `Access`, se emplea la instrucción `PDEP` para acelerar el procesamiento de máscaras de bits durante la navegación por el árbol. A diferencia del enfoque anterior, donde cada bit activo se procesa secuencialmente mediante un bucle (funciones `updateMask()` y `updateMaskLeaf()` del Algoritmo 5.1.1), ahora es posible procesar múltiples bits simultáneamente. Se mantiene el mismo algoritmo `Access` 5.1.1, pero se modifican las funciones `updateMask()` y `updateMaskLeaf()`. La diferencia principal consiste en el uso de `PDEP`, que permite procesar simultáneamente los bits de cada cuadrante para actualizar la máscara (Línea 2 de los Algoritmos 6.1.1 y 6.1.2) y el valor de las regiones en T' (Línea 9 del Algoritmo 6.1.1).

Algoritmo 6.1.1
updateMask($pos,\ mask,\ m, rank,\ ones$)

```
1: bits_child ← extract_bits(T, pos, m)
2: new_mask ← pdep(bits_child, mask)
3: changed ← new_mask ⊕ mask
4: if changed ≠ 0 then
5:     zr = pos − rank
6:     zeros ← m − ones
7:     bits_t' ← extract_bits(T', zr, zeros)
8:     if bits_t' then
9:         result ← result + pdep(bits_t', changed)
10:    end if
11: end if
12: return new_mask
```

Algoritmo 6.1.2
updateMaskLeaf($pos,\ mask$)

```
1: bits_child ← extract_bits(L, pos, m)
2: new_mask ← pdep(bits_child, mask)
3: return new_mask
```

6.2. Window

La consulta `Window` aplica `PDEP` en cada nodo interno para calcular simultáneamente la nueva máscara de bits activos, evitando así la necesidad de bucles sobre bits individuales (Línea 2 de los Algoritmos 6.2.1 y 6.2.2). Además, se actualiza el valor acumulado (`acc`) durante el recorrido con los valores de las regiones homogéneas encontradas en T'. Para obtener este valor, se utiliza una operación `PDEP` (ver Línea 10 del Algoritmo 6.2.1). El resto del procedimiento se mantiene sin cambios y es idéntico al descrito en el Algoritmo 5.2.1.

Algoritmo 6.2.1
updateMask($pos,\ mask,\ m,\ rank,\ acc$)

```
1: bits_child ← extract_bits(T, pos, m)
2: new_mask ← pdep(bits_child, mask)
3: changed ← new_mask ⊕ mask
4: new_acc ← acc
5: if changed ≠ 0 then
6:     zr ← pos − rank
7:     zeros ← popcount(changed)
8:     bits_t' ← extract_bits(T', zr, zeros)
9:     if bits_t' then
10:        new_acc ← new_acc + pdep(bits_t', changed)
11:    end if
12: end if
13: return (new_mask, new_acc)
```

Algoritmo 6.2.2
updateMaskLeaf($pos,\ mask$)

```
1: bits_child ← extract_bits(L, pos, popcount(mask))
2: new_mask ← pdep(bits_child, mask)
3: return new_mask
```

6.3. Window Range

La consulta `Window Range` extiende el enfoque de `Window`, incorporando un filtrado por valor, de modo que sólo se recuperan las celdas dentro de un área rectangular cuyos valores se encuentran entre los límites v_1 y v_2. En este caso, se utilizan las mismas funciones auxiliares `updateMask()` 6.2.1 y `updateMaskLeaf()` 6.2.2, que se emplean en la consulta `Window`, mientras que la función principal mantiene la misma lógica presentada en el Algoritmo `Window Range` 5.3.1.

6.4. Construcción directa con PEXT

Mediante el uso de la instrucción `PEXT` es posible realizar la construcción del ik^2-tree de forma directa, evitando la necesidad de generar k^2-trees como paso intermedio. En este enfoque, la construcción se ejecuta en profundidad, avanzando nivel por nivel y recorriendo de izquierda a derecha para obtener los vectores T, T' y L. Para calcular qué bits se deben insertar en cada nodo se utiliza `PEXT`, ya que entrega los bits empaquetados hacia la derecha, lo que simplifica su asignación. El procedimiento principal se implementa en la función `buildTreeFromMatrix()`, cuya tarea consiste en construir los vectores de bits T, T' y L nivel a nivel. Para ello se emplean máscaras de enteros como $mask$, all_ones, all_zeros, $mask\_child$, entre otras. Gracias a este mecanismo es posible construir de manera simultánea los w árboles. En el momento de insertar los valores en T, T' y L, se aplica la instrucción `PEXT`, que proporciona directamente los bits empaquetados listos para agregarse (véanse las Líneas 16, 48 y 50 del Algoritmo 6.4.1). Este método ofrece una mejora considerable en el tiempo total de construcción.

Algoritmo 6.4.1 **BuildTreeFromMatrix**$(M, n, height, width, r_0, c_0, \ell)$

if $n = k$ **then**
  $all\_ones \leftarrow FULL\_ONES$, $all\_zeros \leftarrow FULL\_ONES$
  for $q \leftarrow 0$ to $k^2 - 1$ **do**
    $r \leftarrow r_0 + \lfloor q/k \rfloor$, $c \leftarrow c_0 + (q \bmod k)$
    $v \leftarrow (r < height \wedge c < width)\ ?\ M[r, c] : 0$
    $all\_ones \leftarrow all\_ones \wedge v$
    $all\_zeros \leftarrow all\_zeros \wedge \neg v$
  end for
  $mask \leftarrow \neg(all\_ones \vee all\_zeros)$
  if $mask = 0$ **then**
    return $(all\_ones, all\_zeros, 0)$
  end if
  for $q \leftarrow 0$ to $k^2 - 1$ **do**
    $r \leftarrow r_0 + \lfloor q/k \rfloor$, $c \leftarrow c_0 + (q \bmod k)$
    $val \leftarrow (r < height \wedge c < width)\ ?\ (M[r, c] \wedge mask) : 0$
    $bits \leftarrow \texttt{pext}(val, mask)$
    añadir $bits$ a $L[\ell]$
  end for
  return $(all\_ones, all\_zeros, mask)$
end if
$mid \leftarrow n/k$
$all\_t1 \leftarrow FULL\_ONES$, $all\_t0 \leftarrow FULL\_ONES$
for $c \leftarrow 0$ hasta $k^2 - 1$ **do**
  $r_s \leftarrow r_0 + \lfloor c/k \rfloor \cdot mid$
  $c_s \leftarrow c_0 + (c \bmod k) \cdot mid$
  if $r_s \geq height \vee c_s \geq width$ **then**
    $child[c] \leftarrow (0, 1, 0)$
  else
    $child[c] \leftarrow BuildTreeFromMatrix(M, mid, height, width, r_s, c_s, \ell + 1)$
  end if
  $all\_t1 \leftarrow all\_t1 \wedge child[c].all\_ones$
  $all\_t0 \leftarrow all\_t0 \wedge child[c].all\_zeros$
end for
$bits \leftarrow \neg(all\_t1 \vee all\_t0)$
if $\ell = 0$ **then**
  $bits \leftarrow FULL\_ONES$
end if
if $bits = 0$ y $\ell > 0$ **then**
  return $(all\_t1, all\_t0, 0)$
end if
for cada hijo c **do**
  $t0 \leftarrow child[c].all\_zeros \wedge bits$
  $t1 \leftarrow child[c].all\_ones \wedge bits$
  $mask\_zero \leftarrow t0 \vee t1$
  $mask\_child \leftarrow bits \wedge \neg mask\_zero$
  añadir $\texttt{pext}(mask\_child, bits)$ a $T[\ell]$
  if $mask\_zero \neq 0$ **then**
    añadir $\texttt{pext}(t1, mask\_zero)$ a $T'[\ell]$
  end if
end for
return $(all\_t1, all\_t0, bits)$

Capítulo 7

Extracción de regiones de k^2-trees

En este capítulo se presenta una extensión del k^2-tree que incorpora un nuevo mecanismo de compresión basado en la extracción de regiones. La motivación surge de observar que, en ciertos casos, la descomposición recursiva de un cuadrante puede resultar más costosa que almacenar dicho cuadrante de manera explícita. Para abordar esta situación, se introduce un vector adicional denominado T'', alineado con los ceros de T, que permite almacenar estas regiones de forma directa y recuperarlas eficientemente. A lo largo del capítulo se detalla el procedimiento de construcción de T'' y su integración en consultas como `Access`, `Window` y `Window Range`.

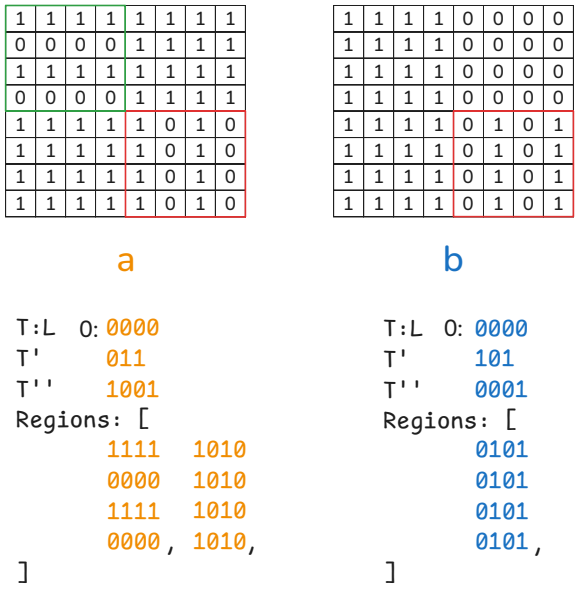

Figura 7.1: k^2-tree con regiones extraídas

En lugar de comprimir únicamente las regiones homogéneas, se observa que existen áreas dentro del k^2-tree cuya descomposición resulta más costosa que el cuadrante original. Como se aprecia en la Figura 7.1, hay zonas rojas y verdes que contienen una descomposición completa del subárbol. Para optimizar el espacio, se extraen estas regiones costosas en vectores de bits y se incorpora un nuevo bitmap T'', alineado con los ceros (al igual que T'), junto con un arreglo de bitmaps que representa las regiones extraídas. Esto permite recuperarlas de forma eficiente. El cálculo del coste de cada nodo se realiza mediante la operación: $cost(\text{node}) = k^2 + \sum_{h \in \text{hijos}} cost(h)$. Esta función considera los k^2 bits correspondientes al nodo, más el coste de los hijos con descomposición. Para decidir qué región extraer, se verifica si $cost(\text{node}) > (n_{\text{node}} \times n_{\text{node}})$, donde n_{node} corresponde al lado de la submatriz que representa cada nodo, de modo que $(n_{\text{node}} \times n_{\text{node}})$ indica el número total de celdas cubiertas. Este enfoque permite analizar y extraer cualquier cuadrante del árbol, optimizando el almacenamiento en presencia de regiones con estas características.

7.1. Construcción

Para construir los vectores de bits T, L, T' y T'', se realiza una construcción en profundidad: se llenan los bits en un vector por cada nivel, procesándolos de izquierda a derecha. Este enfoque permite eliminar regiones durante la construcción. Una vez completada la construcción por niveles, se fusionan los vectores para generar T, L, T' y T'', que conforman la estructura final. La profundidad de T'' corresponde al nivel máximo permitido para extraer regiones. Si la profundidad llega al final, el almacenamiento aumenta, ya que T'' crece considerablemente en el último nivel. Según los resultados experimentales, una profundidad equivalente a 3/4 del máximo ofrece el mejor equilibrio en almacenamiento. A continuación se detalla el uso de T'' en la versión 2 del ik^2-tree.

7.2. Intercalado de regiones extraídas

Al fusionar los k^2-trees individuales, se intercalan las regiones extraídas que pertenecen al mismo cuadrante del árbol, agrupándolas en la primera ocurrencia de T'' correspondiente al cuadrante. En la Figura 7.2 se muestran dos regiones (en rojo) extraídas de un mismo cuadrante, las cuales se intercalan para generar un nuevo vector de bits de tamaño $(n_c \times 2n_c)$, donde $(n_c \times n_c)$ representa las dimensiones del cuadrante. Para recuperar los valores, se extraen los dos bits contiguos en la región almacenada del nodo. Estos se ubican en la posición del primer bit activo de T'' y se procesan mediante `PDEP` para obtener el valor real. En las siguientes secciones se detalla el proceso de consulta.

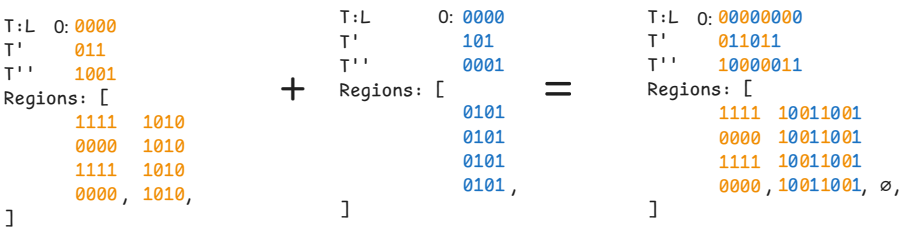

Figura 7.2: ik^2-tree con regiones extraídas e intercaladas

7.3. Access

En esta versión del algoritmo de acceso, al igual que ik^2-MSv2, se modifican únicamente las funciones `updateMask()` y `updateMaskLeaf()`. Cuando se detectan regiones extraídas, primero se extraen los bits pertenecientes a T'' (Línea 14 del Algoritmo 7.3.1). Luego, con `PDEP`, se obtiene una máscara de las regiones extraídas ($mask\_t''$). Para calcular el tamaño de los bits en T'' se utiliza $len\_mask = popcount(mask\_t'')$, y con $rank_1(T'', rank_0(T, pos))$ se determina el primer vector de bits que contiene las regiones intercaladas del nodo. A continuación, con `PDEP`, se calcula el valor real de la región intercalada ubicada entre $init$ y $init + len\_mask$ (Línea 22). El resto del algoritmo es similar al de ik^2-MSv2.

Algoritmo 7.3.1
updateMask(pos, $mask$, n, m, $rank$, $ones$)

```
1: bits_child ← extract_bits(T, pos, m)
2: new_mask ← pdep(bits_child, mask)
3: changed ← new_mask ⊕ mask
4: if changed ≠ 0 then
5:     zr = pos − rank;
6:     zeros ← m − ones
7:     bits_t' ← extract_bits(T', zr, zeros)
8:     if bits_t' then
9:         result ← result + pdep(bits_t', changed)
10:    end if
11:    bits_t'' ← extract_bits(T'', zr, zeros)
12:    if bits_t'' then
13:        mask_t'' ← pdep(bits_t'', changed)
14:        len_mask ← popcount(mask_t'')
15:        pos_t'' ← rank_1(T'', zr)
16:        bv ← extracted_regions[pos_t'']
17:        init ← (r ∗ n + c) ∗ len_mask
18:        bits_t'' ← extract_bits(bv, init, len_mask)
19:        result ← result + pdep(bits_t'', mask_t'')
20:    end if
21: end if
22: return new_mask
```

Algoritmo 7.3.2
updateMaskLeaf(pos, $mask$)

```
1: bits_child ← extract_bits(L, pos, popcount(mask))
2: new_mask ← pdep(bits_child, mask)
3: return new_mask
```

7.4. Window

En la consulta `Window`, el procedimiento para calcular $mask\_t''$, len_mask, pos_t'' e $init$ es idéntico al empleado en la consulta `Access`. Si se detectan regiones extraídas en T'', estas se decodifican mediante una máscara y la instrucción `PDEP` (Líneas 12–22 del Algoritmo 7.4.1). En este caso, es necesario recorrer todas las celdas que intersectan la ventana. Para obtener el valor real, se extraen los bits de la región extraída y luego se aplica `PDEP` para recuperar el valor original. Dado que no es posible agregar directamente el valor a acc, los resultados se asignan directamente a la matriz de salida (Línea 22 del Algoritmo 7.4.1).

Algoritmo 7.4.1
updateMask(pos, $mask$, n, m, r, $ones$, acc)

```
1: bits_child ← extract_bits(T, pos, m)
2: new_mask ← pdep(bits_child, mask)
3: new_acc ← acc
4: changed ← new_mask ⊕ mask
5: if changed ≠ 0 then
6:     zr = pos − r;
7:     zeros ← popcount(changed)
8:     bits_t' ← extract_bits(T', zr, zeros)
9:     if bits_t' then
10:        new_acc ← new_acc + pdep(bits_t', changed)
11:    end if
12:    bits_t'' ← extract_bits(T'', zr, zeros)
13:    if bits_t'' then
14:        mask_t'' ← pdep(bits_t'', changed)
15:        len_mask ← popcount(mask_t'')
16:        pos_t'' ← rank_1(T'', zr)
17:        bv ← extracted_regions[pos_t'']
18:        for (r, c) inside window do
19:            init ← (r ∗ n + c) ∗ len_mask
20:            bits_t'' ← extract_bits(bv, init, len_mask)
21:            p ← (r − r1) ∗ width + (c − c1)
22:            result[p]←result[p] + pdep(bits_t'', mask_t'')
23:        end for
24:    end if
25: end if
26: return new_mask, new_acc
```

Algoritmo 7.4.2
updateMaskLeaf(pos, $mask$)

```
1: bits_child ← extract_bits(L, pos, popcount(mask))
2: new_mask ← pdep(bits_child, mask)
3: return new_mask
```

7.5. Window Range

Las regiones extraídas en esta consulta requieren un tratamiento especial: no se agregan de forma inmediata como en el caso de `Window`, ya que es necesario verificar si sus celdas individuales cumplen con la condición de pertenecer al rango (v_1, v_2). Para ello, el algoritmo mantiene una lista de referencias a estas regiones y sus respectivas máscaras, las cuales se procesan en etapas posteriores, ya sea al descender completamente o al llegar a nodos hoja (función `updateResult`, Algoritmo 7.5.3). Finalmente, para cada celda que se encuentra dentro de la ventana espacial y cuyo valor acumulado, combinado con los valores de las regiones extraídas (T''), pertenece al rango solicitado, se almacena el resultado en la posición correspondiente de la matriz de salida.

Algoritmo 7.5.1
QueryRange($result,\ v_1,\ v_2,\ r_1,\ c_1,\ r_2,\ c_2$)

```
 1: width ← c2 − c1 + 1
 2: initialize empty stack
 3: push (0, 0, 0, 0, (1 ≪ w) − 1, w, 0, [], size, 0)
 4: while stack not empty do
 5:    pop (lvl, pos, r0, c0, msk, m, msk3, off3, n, acc)
 6:    if lvl < height − 1 then
 7:       for c = 0 to k^2 − 1 do
 8:          compute (pos, r, c′, n′) as in QueryWindow
 9:          if region outside window then
10:             continue
11:          end if
12:          off3′←off3.copy()
13:          rank ← rank1(T, pos)
14:          msk′, msk3′, acc′←updateMask(pos, msk,
                                          msk3, n′, acc, off3′)
15:          lb ← acc′, ub ← acc′ + msk′ + msk3′
16:          if ub < v1 ∨ lb > v2 then
17:             continue
18:          end if
19:          if msk′ ≠ 0 then
20:             m′ ← popcount(msk′)
21:             pos′ ← rank ∗ k^2 + w ∗ k^2
22:             push (lvl + 1, pos′, r, c′,
                      msk′, m′, msk3′, off3′, n′, acc′)
23:          else
24:             for each (r, c) in window do
25:                updateResult(off3′, r, c, acc′)
26:             end for
27:          end if
28:       end for
29:    else
30:       for each leaf-cell (i, j) do
31:          r ← r0 + i, c ← c0 + j
32:          if (r, c) in window then
33:             bits_child ← extract_bits(L, pos, m)
34:             value ← pdep(bits_child, mask)
35:             acc′ ← acc + value
36:             updateResult(off3, r, c, acc′)
37:          end if
38:          pos ← pos + m
39:       end for
40:    end if
41: end while
```

Algoritmo 7.5.2
updateMask($pos, mask, mask3, n, acc,$ `off3`)

```
 1: bits_child ← extract_bits(T, pos, m)
 2: new_mask ← pdep(bits_child, mask)
 3: new_mask3 ← mask3
 4: new_acc ← acc
 5: changed ← new_mask ⊕ mask
 6: if changed ≠ 0 then
 7:    zr = pos − rank;
 8:    zeros ← popcount(changed)
 9:    bits_t′ ← extract_bits(T′, zr, zeros)
10:    if bits_t′ then
11:       new_acc ← new_acc + pdep(bits_t′, changed)
12:    end if
13:    bits_t′′ ← extract_bits(T′′, zr, zeros)
14:    if bits_t′′ then
15:       mask_t′′ ← pdep(bits_t′′, changed)
16:       new_mask3 ← new_mask3 | mask_t′′
17:       len_mask ← popcount(mask_t′′)
18:       p ← rank1(T′′, zr)
19:       // r,c base (line 8 QueryRange)
20:       add to off3(mask_t′′, len_mask, p, r, c, n)
21:    end if
22: end if
23: return new_mask, new_mask3, new_acc
```

Algoritmo 7.5.3
updateResult(`offsets_t3, r, c, acc`)

```
 1: value ← acc
 2: for (mask, len, pos, rc, cc, nc) ∈ offsets_t3 do
 3:    bv ← extracted_regions[pos]
 4:    bits_bv ← extract_bits(bv, (r ∗ n + c) ∗ len, len)
 5:    val ← pdep(bits_bv, mask)
 6:    value ← value + val
 7: end for
 8: if v1 ≤ value ≤ v2 then
 9:    idx ← (r − r1) ∗ width + (c − c1)
10:    result[idx] ← value
11: end if
```

Nota. Aunque se desarrollan e implementan estas variantes con el vector T'', finalmente no se incluyen en la evaluación experimental. En pruebas preliminares se observa que, si bien se obtiene una mejora en almacenamiento, se incrementa el coste de ejecución: en la consulta `Window`, debido a múltiples asignaciones por celda (Líneas 12–24 del Algoritmo 7.4.1), y en `Window Range`, por la necesidad de propagar referencias de regiones hasta los niveles hoja (Línea 20 del Algoritmo 7.5.2) y procesarlas con la función `updateResult` (Algoritmo 7.5.3). Por estas razones, en los experimentos posteriores se trabaja únicamente con ik^2-MSv1 e ik^2-MSv2, dejando la mejora de esta estrategia como trabajo futuro. Una posible alternativa consiste en extraer regiones completamente, considerando una función de coste que evalúe simultáneamente todos los bits.

Capítulo 8

Aplanamiento de Niveles y Heurística de Compresión

El k^2-raster cuenta con una variante denominada k^2-raster Heurístico [10], la cual reduce el almacenamiento y mejora los tiempos de las consultas. En este capítulo se presentan las ideas del k^2-raster Heurístico y, posteriormente, se muestra cómo se incorporan en la estructura de datos ik^2-MSv2.

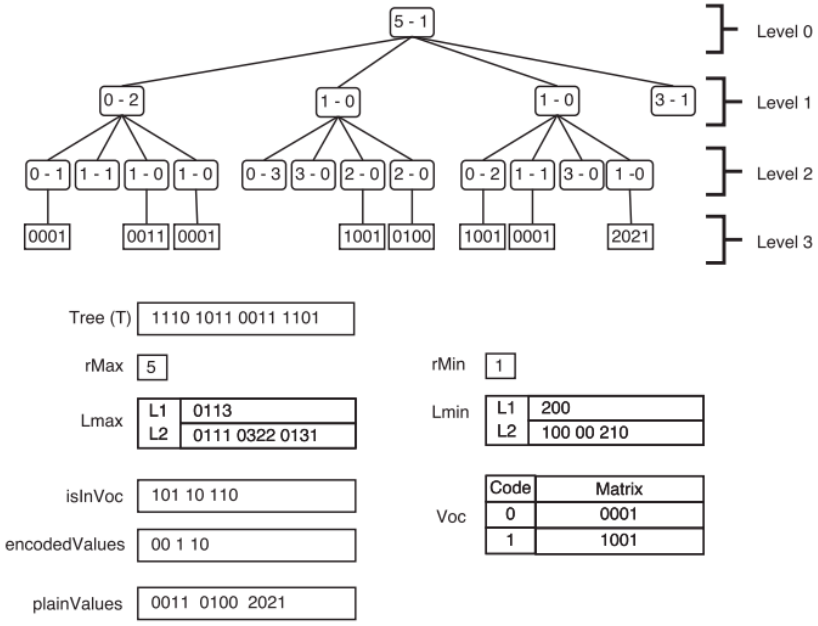

Figura 8.1: Representación de k^2-raster Heurístico (extraído de [10])

En general, el k^2-raster Heurístico aplica dos ajustes: (i) aplanar los niveles finales del árbol para almacenar en las hojas bloques ($k_p \times k_p$) y (ii) reutilizar los bloques más frecuentes mediante un vocabulario. La primera mejora consiste en aplanar los niveles inferiores del árbol, almacenando

en el último nivel matrices de tamaño $(k_p \times k_p)$. Cuando el árbol tiene $k = 2$ y no se aplica aplanamiento adicional ($p = 0$), entonces $k_p = 2$ y el árbol mantiene la misma estructura que el k^2-raster original, como se muestra en la Figura 8.1, con hojas que contienen matrices de (2×2).

Si $p = 1$, el árbol se extiende sólo hasta el segundo nivel, almacenando en sus hojas matrices de (4×4), y así sucesivamente para otros valores de p. Se denomina k_p al tamaño de las matrices en las hojas. Inicialmente, estos valores se almacenan en el último nivel de `Lmax`, conservando la estructura del k^2-raster clásico.

La segunda optimización aplica un análisis estadístico a las matrices del último nivel con el fin de comprimir aquellas que se repiten. El vector `Voc` (ver Figura 8.1) almacena las matrices $(k_p \times k_p)$ sin compresión, permitiendo un acceso directo y más eficiente. Un vector de bits, `isInVoc`, indica qué matrices utilizan el vocabulario y cuáles se almacenan directamente en `plainValues` con DACs. La posición de cada matriz en `Voc` se obtiene mediante `encodedValues`.

Estas optimizaciones permiten una reducción significativa en el almacenamiento, gracias al análisis estadístico, y una mejora en el tiempo de ejecución, ya que las matrices en `Voc` se almacenan de forma directa, evitando el acceso mediante DACs. En consecuencia, a mayor compresión, menor tiempo de acceso, dado que las celdas frecuentes residen en `Voc` en lugar de `plainValues`.

8.1. Optimizaciones aplicadas a ik^2-MSv2

En la implementación de ik^2-MSv2 se aplica la misma estrategia de aplanamiento de los últimos niveles del árbol. En este caso, los bits del nivel aplanado se almacenan en L. No se implementa la compresión estadística de las matrices del último nivel, debido a que no se dispone de un método directo para identificar la posición de cada nodo en el ik^2-MSv2 durante el recorrido del árbol.

Además, la aplicación de dicha compresión en este contexto únicamente proporciona beneficios en términos de almacenamiento, pero no en tiempo de ejecución, ya que aún es necesario aplicar la instrucción `PDEP` sobre cada celda individual. Sin embargo obstante, no se descarta la existencia de una forma eficiente de combinar ambas estrategias, lo cual constituye una posible línea de investigación para trabajos futuros.

Capítulo 9

Evaluación experimental

En este capítulo se describe una serie de experimentos para evaluar el rendimiento de la estructura de datos propuesta. Se detalla el entorno de ejecución, los conjuntos de datos, las métricas y un análisis de resultados. Los datasets fueron obtenidos de fuentes públicas [13] [6], con datos mensuales mundiales de temperatura mínima, media y máxima, lluvia, viento y presión de vapor de agua. También incluyen variables bioclimáticas que resumen tendencias climáticas y precipitación anual, útiles en estudios ecológicos y ambientales. En este caso, los rásteres corresponden a series temporales, aunque aquí se utilizan únicamente para el enfoque espacial. En la Tabla 9.1 se presentan algunos de estos rásteres. Los resultados son similares para todos los conjuntos, pero aquí se muestra un extracto para mantener claridad. Se incluyen los valores únicos mínimos y máximos; por ejemplo, en `wc2.1_2.5m_bio` se registran rásteres con 501 y 8.373.905. La Tabla 9.2 describe las estructuras utilizadas, con parámetros k_1, k_2 y n_1, donde k_1 corresponde a los valores de k en los primeros n_1 niveles, mientras que en los niveles posteriores se utiliza $k = k_2$. Los algoritmos se implementaron en C++ y se prueban en Ubuntu 24.04 LTS (64 bits), con procesador Intel Core i7-1165G7 (4 núcleos, 8 hilos) y 8 GB de RAM. Se mide almacenamiento y tiempo de ejecución con la librería estándar de C++ `std::chrono`. En los experimentos se utiliza la implementación k^2-raster de [9]. Todas las estructuras se compilan con optimización -O3.

Dataset	**Rásteres**	**Valores únicos**	
		Máx	**Mín**
Dimensiones: 4.320 × 8.640			
wc2.1_2.5m_bio	19	8.373.905	501
wc2.1_2.5m_prec	12	2.076	655
wc2.1_2.5m_tavg	12	101.586	91.255
Dimensiones: 2.160 × 4.320			
wc2.1_5m_bio	19	2.833.565	470
wc2.1_5m_tavg	12	195.772	169.718
wc2.1_5m_wind	12	62.753	56.557
Dimensiones: 1.080 × 2.160			
wc2.1_10m_bio	19	783.433	404
wc2.1_10m_vapr	12	33.158	30.619

Tabla 9.1: Descripción de los datasets

Nombre	**Descripción**	$\boldsymbol{k_1, k_2, n_1}$
ik^2-MSv2	ik^2-MSv2 descrito en Capítulo 6	4, 2, 4
ik^2-MSv2 p=2	ik^2-MSv2 con 2 niveles planos	4, 2, 4
k^2-raster	k^2-raster descrito en Sección 3.3	4, 2, 4
k^2-raster $p = 2$	k^2-raster Heurístico descrito en Capítulo 8 con 2 niveles planos	4, 2, 4
k^2-raster p=4	k^2-raster Heurístico con 4 niveles planos	4, 2, 4

Tabla 9.2: Descripción de las estructuras

A continuación se presentan los experimentos diseñados para evaluar el comportamiento de las estructuras. En primer lugar, se realiza una comparación entre las versiones ik^2-MSv1 y ik^2-MSv2, cuyo objetivo es mostrar las mejoras que la segunda versión presenta respecto de la primera. Luego se presentan los resultados de la estructura ik^2-MSv2 en conjunto con el k^2-raster, considerando sus diferentes configuraciones (aplanamiento de niveles). Finalmente, se muestran los tiempos de consulta para `Access`, `Window` y `Window Range`, lo que permite analizar el rendimiento de las estructuras en distintos tipos de consultas.

9.1. Comparación entre ik^2-MSv1 e ik^2-MSv2

En esta sección se muestra una comparación entre las versiones ik^2-MSv1 y ik^2-MSv2. Los resultados se centran en dos métricas principales: el tiempo de construcción y el tiempo de acceso. Para el primer caso, se calcula el promedio de cinco ejecuciones, mientras que para el acceso se toman promedios de 10.000 ejecuciones aleatorias. El dataset utilizado en esta evaluación corresponde a `wc2.1_2.5m_bio`, específicamente se escogen los tres con mayor cantidad de valores diferentes (1, 2 y 3) y los tres con menor cantidad de valores diferentes (4, 5 y 6). Con este experimento se busca comparar ambas versiones y mostrar las mejoras en rendimiento que logra la versión 2.

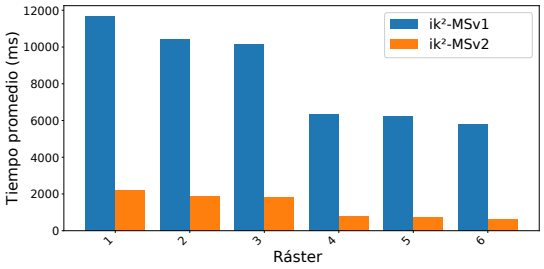

Figura 9.1: Tiempo construcción para wc2.1_2.5m_bio(ms)

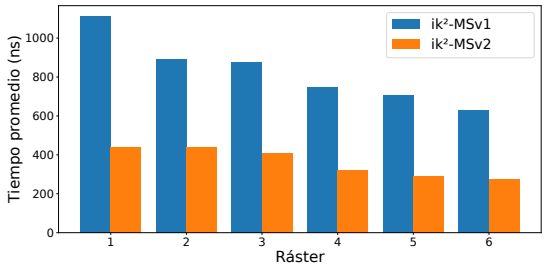

Figura 9.2: Tiempo `Access` para wc2.1_2.5m_bio(ns)

Los resultados evidencian que la versión 2 obtiene un mejor desempeño en ambas métricas. En particular, para el ráster 1, el tiempo de construcción se reduce de $11.667,2$ ms en la versión v1 a sólo $2.230,8$ ms en la versión v2, lo que representa una mejora superior al 80 %. Por otro lado, el tiempo de acceso pasa de $1.111,196$ ms en la versión v1 a $441,706$ ms en la versión 2, mostrando también una mejora significativa en tiempo de acceso. Esto se repite en los demás casos, tanto en los rásteres con más valores diferentes como en aquellos con menos. De este modo, se confirma que la segunda versión no sólo reduce de manera considerable los tiempos de construcción, sino que también mejora el acceso a los datos, independientemente de los valores diferentes del ráster.

9.2. Resultados de almacenamiento y tiempos de construcción

En esta sección se presentan los resultados para la construcción. Se usa un total de 6 rásteres en cada conjunto de datos; los tres primeros corresponden a aquellos con una mayor cantidad de valores diferentes, mientras que los tres restantes se seleccionan entre los que presentan una

menor cantidad de valores. De este modo, se consideran tanto rásteres con mayor variabilidad como otros con menor variación en los datos. El tiempo de ejecución se calcula como el promedio de cinco ejecuciones. Todas las mediciones se realizan en milisegundos. En estos experimentos se consideran, además de las estructuras base, sus variantes con diferentes niveles de aplanamiento. El parámetro p indica la cantidad de niveles aplanados en la estructura. En particular, se evalúan las versiones heurísticas del k^2-raster con p=2 y p=4, y la variante de ik^2-MSv2 con p=2. En nuestro caso, no se aplica compresión estadística adicional.

9.2.1. Almacenamiento

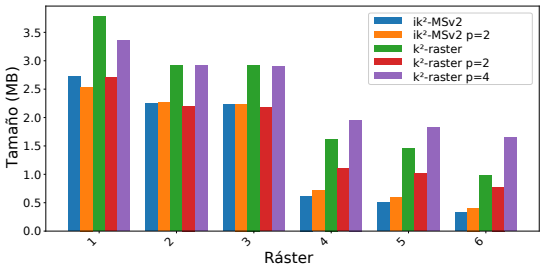

Figura 9.3: Almacenamiento para wc2.1_10m_bio (MB)

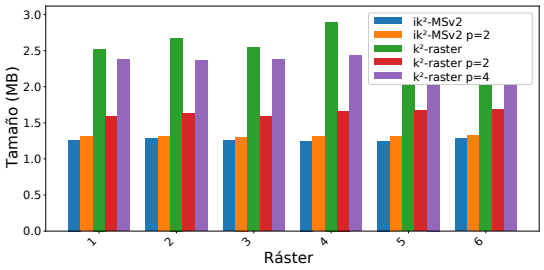

Figura 9.4: Almacenamiento para wc2.1_10m_vapr (MB)

Ráster	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
1	2,724	**2,538**	3,774	2,703	3,357
2	2,251	2,268	2,920	**2,201**	2,919
3	2,225	2,239	2,924	**2,178**	2,903
4	**0,618**	0,720	1,624	1,108	1,956
5	**0,509**	0,589	1,456	1,010	1,829
6	**0,325**	0,408	0,982	0,779	1,649

Tabla 9.3: Almacenamiento para wc2.1_10m_bio (MB)

Ráster	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
1	**1,261**	1,306	2,521	1,592	2,375
2	**1,282**	1,312	2,676	1,625	2,368
3	**1,254**	1,305	2,548	1,591	2,375
4	**1,247**	1,309	2,897	1,664	2,428
5	**1,250**	1,313	2,822	1,672	2,427
6	**1,280**	1,328	2,969	1,687	2,430

Tabla 9.4: Almacenamiento para wc2.1_10m_vapr (MB)

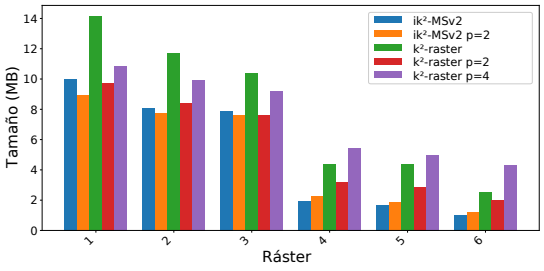

Figura 9.5: Almacenamiento para wc2.1_5m_bio (MB)

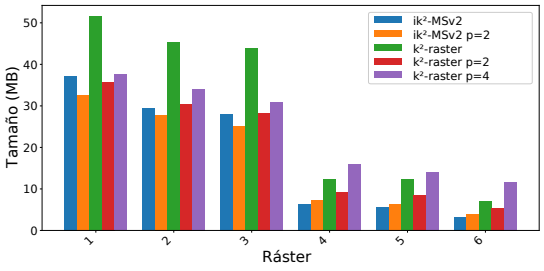

Figura 9.6: Almacenamiento para wc2.1_2.5m_bio (MB)

Ráster	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
1	9,976	**8,972**	14,155	9,710	10,838
2	8,080	**7,766**	11,707	8,390	9,946
3	7,882	**7,592**	10,399	7,593	9,181
4	**1,957**	2,292	4,396	3,167	5,449
5	**1,650**	1,885	4,368	2,850	4,976
6	**1,001**	1,251	2,555	1,986	4,303

Tabla 9.5: Almacenamiento para wc2.1_5m_bio (MB)

Ráster	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
1	37,208	**32,695**	51,620	35,749	37,564
2	29,376	**27,873**	45,368	30,371	34,042
3	27,975	**25,237**	43,837	28,381	30,871
4	**6,336**	7,442	12,422	9,211	15,916
5	**5,525**	6,253	12,399	8,507	14,172
6	**3,197**	3,986	6,990	5,354	11,673

Tabla 9.6: Almacenamiento para wc2.1_2.5m_bio (MB)

Al analizar los resultados del ráster 1 de la Tabla 9.6, el ik^2-MSv2 $p=2$ presenta un mejor almacenamiento (32,6 MB), seguido por el k^2-raster $p=2$ (35,7 MB) y el ik^2-MSv2 (37,2 MB).

Las estructuras con mayor uso de almacenamiento son el k^2-raster $p = 4$ (37,5 MB) y el k^2-raster (51,5 MB). Cabe destacar que el ik^2-MSv2 $p = 2$, sin compresión, mantiene un tamaño similar al del k^2-raster $p = 2$. En el caso del k^2-raster $p = 4$, aunque el almacenamiento aumenta, se considera su uso porque mejora el tiempo de ejecución de las consultas posteriores.

9.2.2. Tiempo de construcción

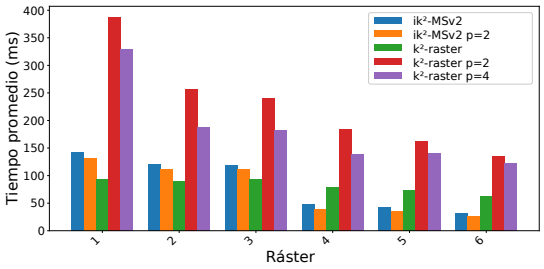

Figura 9.7: Tiempo construcción para wc2.1_10m_bio (ms)

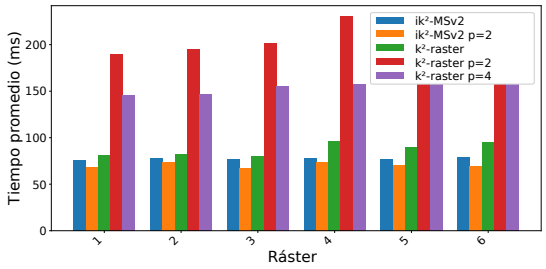

Figura 9.8: Tiempo construcción para wc2.1_10m_vapr (ms)

Ráster	ik^2-MSv2	ik^2-MSv2 $p = 2$	k^2-raster	k^2-raster $p = 2$	k^2-raster $p = 4$
1	142,800	131,400	**93,200**	387,800	329,800
2	119,800	112,200	**89,400**	256,000	187,400
3	119,000	112,000	**94,000**	240,400	182,800
4	48,000	**39,200**	79,000	183,200	139,600
5	42,000	**35,000**	73,800	161,600	140,200
6	32,000	**25,400**	62,400	134,600	122,800

Tabla 9.7: Tiempo construcción para wc2.1_10m_bio (ms)

Ráster	ik^2-MSv2	ik^2-MSv2 $p = 2$	k^2-raster	k^2-raster $p = 2$	k^2-raster $p = 4$
1	75,800	**68,000**	81,400	189,200	145,200
2	77,800	**73,600**	82,200	194,800	147,200
3	76,600	**67,600**	80,400	201,000	155,600
4	78,000	**73,400**	96,000	230,400	157,000
5	76,800	**70,000**	90,000	222,000	157,000
6	78,600	**69,200**	95,200	227,200	159,000

Tabla 9.8: Tiempo construcción para wc2.1_10m_vapr (ms)

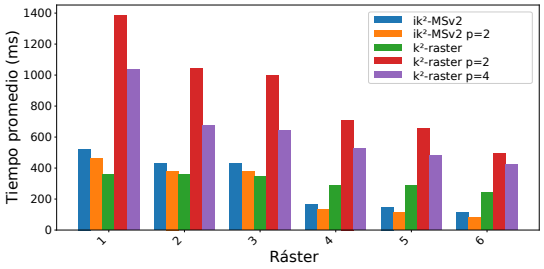

Figura 9.9: Tiempo construcción para wc2.1_5m_bio (ms)

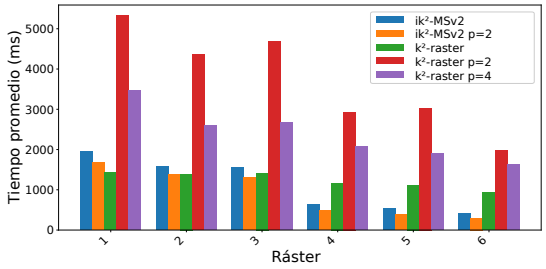

Figura 9.10: Tiempo construcción para wc2.1_2.5m_bio (ms)

Ráster	ik^2-MSv2	ik^2-MSv2 $p = 2$	k^2-raster	k^2-raster $p = 2$	k^2-raster $p = 4$
1	522,600	460,400	**359,400**	1.383,200	1.035,400
2	431,800	381,200	**358,600**	1.043,000	676,000
3	427,000	379,200	**344,000**	998,000	642,800
4	164,400	**131,800**	289,800	710,800	528,200
5	148,400	**114,400**	286,000	655,800	480,200
6	114,400	**82,600**	244,600	494,200	421,200

Tabla 9.9: Tiempo construcción para wc2.1_5m_bio (ms)

Ráster	ik^2-MSv2	ik^2-MSv2 $p = 2$	k^2-raster	k^2-raster $p = 2$	k^2-raster $p = 4$
1	1.946,800	1.672,800	**1.438,400**	5.324,200	3.464,800
2	1.578,000	**1.371,800**	1.393,800	4.357,200	2.595,000
3	1.544,800	**1.304,600**	1.418,200	4.693,200	2.674,400
4	635,000	**490,600**	1.167,200	2.934,200	2.077,000
5	544,800	**395,600**	1.120,800	3.031,000	1.896,200
6	419,200	**299,000**	934,000	1.985,000	1.636,800

Tabla 9.10: Tiempo construcción para wc2.1_2.5m_bio (ms)

En cuanto al tiempo de ejecución, el k^2-raster obtiene mejor resultado. Cuando la cantidad de valores diferentes es pequeña, el ik^2-MSv2 presenta un mejor tiempo de construcción. Los que presentan el peor resultado son el k^2-raster $p = 2$ y $p = 4$ (5.324 ms y 3.464 ms). Estos aumentan considerablemente su tiempo (ver Tabla 9.10) debido al análisis adicional que se requiere para comprimir las matrices en las hojas. La estructura k^2-raster es ligeramente superior

al ik^2-MSv2 $p = 2$ (1.438 ms frente a 1.672 ms) cuando el ráster contiene más valores diferentes, pero cuando la cantidad de valores diferentes es reducida (como en los últimos tres rásteres), la estructura con mejor tiempo de ejecución es el ik^2-MSv2 $p = 2$.

9.3. Tiempo de consultas

En esta sección se evalúa el tiempo de consulta de las estructuras usando los tres tipos de consultas: `Access`, `Window` y `Window Range`. Cada subsección detalla la configuración experimental y los resultados obtenidos.

9.3.1. Access

En la consulta de acceso, se miden las diferentes estructuras considerando un promedio de un total de 30.000 ejecuciones aleatorias; el tiempo se mide en nanosegundos.

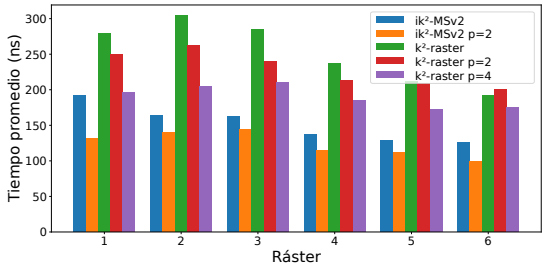

Figura 9.11: Consulta `Access` para wc2.1_10m_bio (ns)

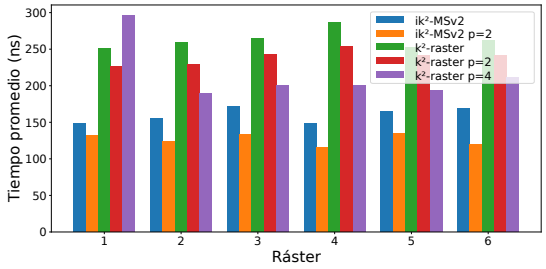

Figura 9.12: Consulta `Access` para wc2.1_10m_vapr (ns)

Ráster	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
1	191,875	**131,526**	196,304	249,277	279,403
2	164,393	**140,129**	205,060	262,383	304,057
3	162,793	**144,725**	210,274	239,794	284,791
4	137,604	**114,890**	184,262	213,122	237,427
5	128,041	**111,989**	171,865	208,300	211,769
6	125,454	**99,443**	174,529	192,036	199,899

Tabla 9.11: Consulta `Access` para wc2.1_10m_bio (ns)

Ráster	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
1	147,693	**132,167**	250,533	225,970	295,692
2	155,840	**123,850**	259,570	229,543	189,312
3	171,000	**133,907**	265,081	242,974	200,288
4	148,837	**115,894**	287,078	254,192	200,086
5	164,949	**134,092**	251,692	242,044	193,867
6	168,528	**119,898**	261,270	241,818	211,782

Tabla 9.12: Consulta `Access` para wc2.1_10m_vapr (ns)

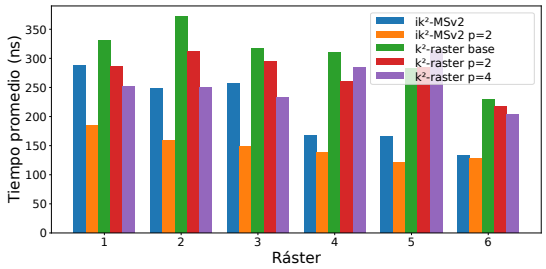

Figura 9.13: Consulta `Access` para 2.1_5m_bio (ns)

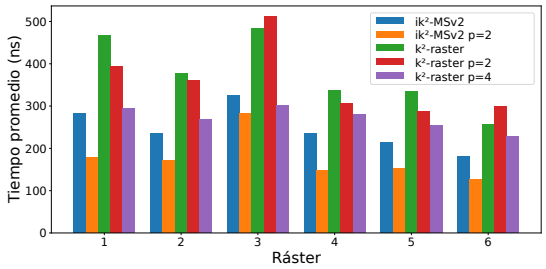

Figura 9.14: Consulta `Access` para 2.1_2.5m_bio (ns)

Ráster	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
1	287,400	**185,458**	330,681	286,385	251,882
2	247,914	**158,478**	371,501	311,945	250,002
3	257,109	**148,593**	317,329	294,605	233,178
4	167,030	**138,175**	309,994	259,531	284,390
5	164,994	**121,846**	282,820	284,142	313,449
6	133,926	**127,868**	230,107	216,677	203,192

Tabla 9.13: Consulta `Access` para wc2.1_5m_bio (ns)

Ráster	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
1	283,229	**179,587**	465,945	392,833	294,311
2	236,165	**171,767**	377,402	360,494	267,962
3	325,613	282,325	484,646	511,247	**302,502**
4	234,815	**148,373**	336,218	306,307	279,377
5	213,142	**151,917**	334,724	287,329	254,347
6	179,893	**126,739**	256,701	298,190	228,476

Tabla 9.14: Consulta `Access` para wc2.1_2.5m_bio (ns)

El tiempo de acceso muestra que la estructura con mejor desempeño es el ik^2-MSv2 $p=2$, mientras que el k^2-raster presenta el peor tiempo de ejecución. Asimismo, se observa que el k^2-raster $p=4$ tiene un rendimiento ligeramente inferior al ik^2-MSv2. En el caso del ráster 1 del conjunto de datos `wc2.1_2.5m_bio` (Tabla 9.14), el mejor resultado corresponde al ik^2-MSv2 $p=2$ con un tiempo de 179,58 ns, seguido por el ik^2-MSv2 (283,22 ns), el k^2-raster $p=4$ (294,31 ns), el k^2-raster $p=2$ (392,83 ns) y, finalmente, el k^2-raster (465,94 ns), que presenta el mayor tiempo de ejecución.

9.3.2. Window

Para la consulta de ventana, se realiza un total de 150 ejecuciones, de las cuales se toma el promedio en milisegundos. Se utilizan distintos tamaños de ventanas cuadradas, considerando porcentajes de área desde el 0,005 % hasta el 100 %, donde el 100 % corresponde a un cuadrado de lado igual a mín($rows$, $cols$). Para el último punto (Full), se toma el tamaño de la ventana completa. Por cada dataset, se escoge el ráster que tiene una mayor cantidad de números diferentes.

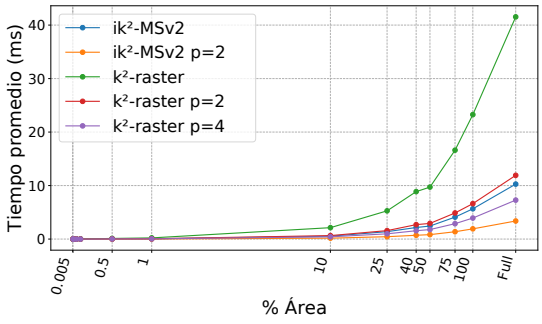

Figura 9.15: Consulta `Window` para wc2.1_10m_bio (ms)

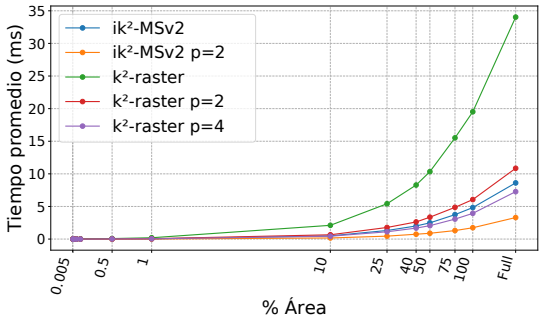

Figura 9.16: Consulta `Window` para wc2.1_10m_vapr (ms)

% Área	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
0,005	0,00	0,00	0,00	0,00	0,00
0,01	0,00	0,00	0,00	0,00	0,00
0,05	0,00	0,00	0,00	0,00	0,00
0,1	0,00	0,00	0,02	0,00	0,00
0,5	0,02	**0,00**	0,12	0,03	0,02
1	0,05	**0,02**	0,22	0,06	0,04
10	0,53	**0,17**	2,13	0,65	0,42
25	1,39	**0,45**	5,28	1,58	0,99
40	2,16	**0,71**	8,87	2,70	1,56
50	2,44	**0,83**	9,72	2,93	1,78
75	4,11	**1,37**	16,62	4,88	2,88
100	5,66	**1,91**	23,27	6,62	3,93
Full	10,27	**3,38**	41,53	11,91	7,28

Tabla 9.15: Consulta `Window` para wc2.1_10m_bio (ms)

% Área	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
0,005	0,00	0,00	0,00	0,00	0,00
0,01	0,00	0,00	0,00	0,00	0,00
0,05	0,00	0,00	0,00	0,00	0,00
0,1	0,00	0,00	0,02	0,00	0,00
0,5	0,02	**0,00**	0,08	0,02	0,02
1	0,04	**0,01**	0,21	0,06	0,04
10	0,49	**0,17**	2,11	0,66	0,44
25	1,32	**0,45**	5,42	1,78	1,12
40	2,01	**0,75**	8,29	2,63	1,69
50	2,49	**0,89**	10,34	3,36	2,06
75	3,75	**1,31**	15,52	4,87	3,07
100	4,83	**1,74**	19,52	6,06	3,95
Full	8,61	**3,30**	34,03	10,85	7,27

Tabla 9.16: Consulta `Window` para wc2.1_10m_vapr (ms)

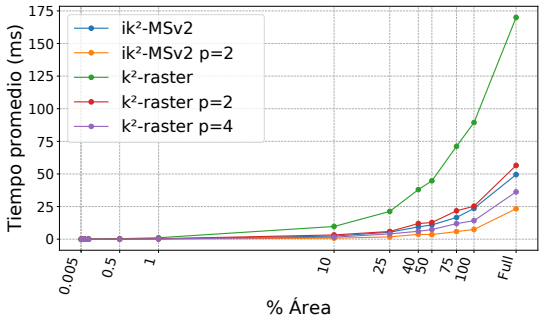

Figura 9.17: Consulta `Window` para wc2.1_5m_bio (ms)

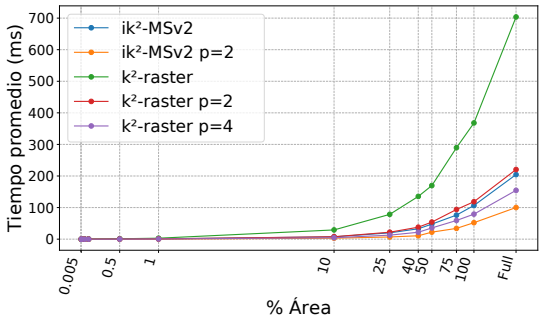

Figura 9.18: Consulta `Window` para wc2.1_2.5m_bio (ms)

% Área	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
0,005	0,00	0,00	0,00	0,00	0,00
0,01	0,00	0,00	0,00	0,00	0,00
0,05	0,00	0,00	0,04	0,00	0,00
0,1	0,02	**0,00**	0,09	0,04	0,01
0,5	0,09	**0,03**	0,35	0,10	0,06
1	0,27	**0,08**	1,11	0,34	0,21
10	2,35	**0,75**	9,74	3,26	1,54
25	5,33	**1,77**	21,31	5,90	4,03
40	9,42	**3,70**	37,99	11,99	6,07
50	10,93	**3,53**	44,72	12,78	7,41
75	16,63	**5,83**	71,18	21,75	11,91
100	23,58	**7,46**	89,41	25,31	14,22
Full	49,49	**23,31**	170,01	56,55	36,25

Tabla 9.17: Consulta `Window` para wc2.1_5m_bio (ms)

% Área	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
0,005	0,00	0,00	0,02	0,00	0,00
0,01	0,00	0,00	0,02	0,00	0,00
0,05	0,04	**0,00**	0,16	0,04	0,02
0,1	0,09	**0,02**	0,30	0,09	0,04
0,5	0,35	**0,12**	1,29	0,36	0,19
1	0,77	**0,25**	3,02	0,88	0,51
10	7,37	**3,13**	29,29	7,70	4,51
25	20,02	**6,95**	78,55	22,04	12,72
40	32,89	**10,76**	135,37	37,95	21,89
50	47,71	**22,13**	169,75	54,13	35,59
75	76,12	**34,06**	289,64	93,96	59,12
100	106,62	**52,14**	368,02	118,60	79,24
Full	204,47	**100,18**	703,66	220,47	154,57

Tabla 9.18: Consulta `Window` para wc2.1_2.5m_bio (ms)

En la consulta `Window` se observa que, bajo el 10 %, todas las estructuras presentan tiempos similares. Luego, el k^2-raster incrementa considerablemente el tiempo de ejecución. El ik^2-MSv2 y el k^2-raster $p = 2$ mantienen un rendimiento parecido en todos los tamaños de ventana, siendo el ik^2-MSv2 ligeramente mejor. Las estructuras con mejor tiempo son el k^2-raster $p = 4$ y el ik^2-MSv2 $p = 2$; este último resulta competitivo en todos los tamaños y con el mejor resultado. Por ejemplo, al obtener el ráster completo (Full) de la Tabla 9.14 correspondiente a `wc2.1_2.5m_bio`, el ik^2-MSv2 $p = 2$ logra 100,18 ms, seguido por el k^2-raster $p = 4$ (154,57 ms), el ik^2-MSv2 (204,47 ms), el k^2-raster $p = 2$ (220,47 ms) y finalmente el k^2-raster (703,66 ms).

9.3.3. Window Range

Esta consulta es similar a la de ventana, ya que se realizan 150 ejecuciones y se analiza el resultado variando distintos porcentajes de ventana. Además, se agrega la componente de valor, donde se consideran porcentajes de los valores no nulos comprendidos entre el valor mínimo y el máximo del ráster. Por ejemplo, si se aplica un rango de valor del 10 % y el ráster tiene un valor mínimo de 20 y un máximo de 120, el intervalo total es de 100. Un rango del 10 % implica seleccionar un segmento de 10 unidades dentro de ese intervalo, el cual se selecciona aleatoriamente entre 20 y 110. De esta forma, cada ventana queda definida tanto por un porcentaje espacial como por un porcentaje de valor.

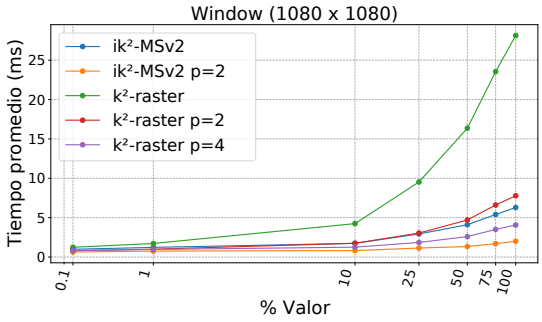

Figura 9.19: Consulta `Window Range` para wc2.1_10m_bio (ms)

Value %	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
0,1	0,987	**0,640**	1,240	0,813	0,813
1	1,227	**0,753**	1,720	1,000	0,967
10	1,740	**0,820**	4,247	1,747	1,253
25	2,933	**1,140**	9,533	3,040	1,853
50	4,100	**1,333**	16,367	4,707	2,593
75	5,400	**1,693**	23,547	6,607	3,507
100	6,293	**2,007**	28,140	7,780	4,067

Tabla 9.19: Consulta `Window Range` para wc2.1_10m_bio (ms)

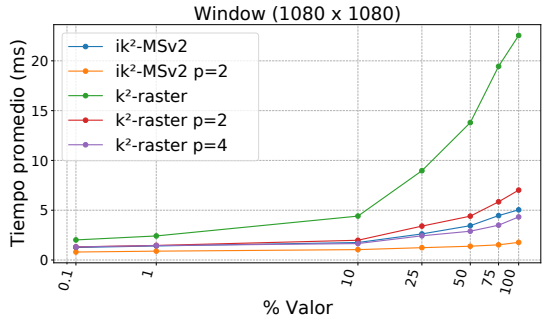

Figura 9.20: Consulta `Window Range` para wc2.1_10m_vapr (ms)

Value %	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
0,1	1,260	**0,807**	2,020	1,320	1,313
1	1,413	**0,893**	2,420	1,467	1,453
10	1,760	**1,047**	4,413	1,987	1,653
25	2,620	**1,240**	8,967	3,407	2,433
50	3,447	**1,387**	13,800	4,407	2,893
75	4,460	**1,527**	19,447	5,847	3,500
100	5,040	**1,773**	22,547	7,020	4,327

Tabla 9.20: Consulta `Window Range` para wc2.1_10m_vapr (ms)

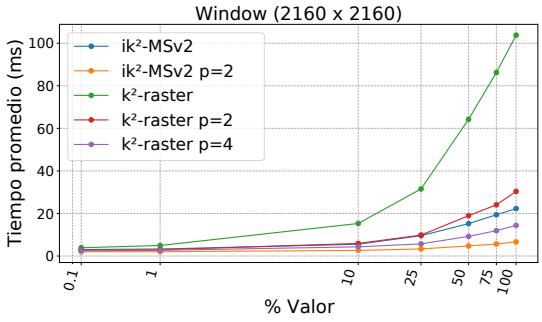

Figura 9.21: Consulta `Window Range` para wc2.1_5m_bio (ms)

Value %	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
0,1	3,067	**2,033**	3,907	2,720	2,520
1	3,300	**2,047**	4,960	2,947	2,693
10	5,547	**2,633**	15,307	5,927	4,327
25	9,573	**3,347**	31,547	9,847	5,720
50	15,233	**4,740**	64,273	18,973	9,227
75	19,387	**5,660**	86,267	24,080	11,907
100	22,313	**6,660**	103,787	30,353	14,400

Tabla 9.21: Consulta `Window Range` para wc2.1_5m_bio (ms)

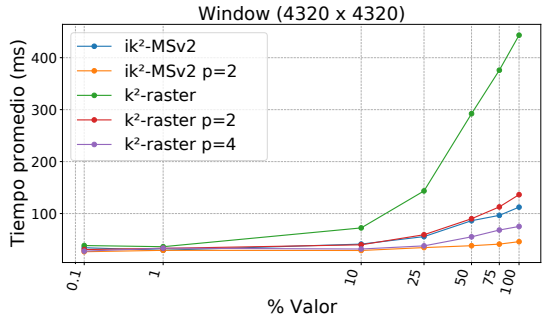

Figura 9.22: Consulta `Window Range` wc2.1_2.5m_bio (ms)

Value %	ik^2-MSv2	ik^2-MSv2 $p=2$	k^2-raster	k^2-raster $p=2$	k^2-raster $p=4$
0,1	34,260	**26,573**	38,440	30,713	27,693
1	29,400	**29,260**	36,207	33,013	33,400
10	41,220	**28,967**	72,387	39,887	31,773
25	55,653	**34,533**	143,580	59,227	37,833
50	86,220	**38,133**	292,200	90,020	55,227
75	96,520	**41,180**	375,940	112,760	68,287
100	112,307	**45,887**	443,327	136,373	75,167

Tabla 9.22: Consulta `Window Range` para wc2.1_2.5m_bio (ms)

Los resultados para esta consulta son similares a los de la consulta `Window`. Debido a que los resultados son iguales para diferentes porcentajes de ventana, sólo se muestran ventanas grandes. El k^2-raster presenta el peor tiempo de ejecución, mientras que las estructuras k^2-raster $p=2$ e ik^2-MSv2 muestran tiempos similares. ik^2-MSv2 $p=2$ obtiene el mejor rendimiento en todos los porcentajes de valor; específicamente, en la Tabla 9.22, con un 100 %, registra 45,88 ms (mejor), seguido por k^2-raster $p=4$ con 75,16 ms, y luego k^2-raster $p=2$ (136,37 ms), ik^2-MSv2 (112,3 ms) y finalmente k^2-raster (443,32 ms).

9.4. Discusión de resultados

Para el análisis de resultados, en cada resultado (Almacenamiento, Tiempo de construcción, `Access`, `Window` y `Window Range`) se calcula el promedio por dataset de cada estructura. A partir de esos valores se obtiene el porcentaje de mejora en dos casos: (Caso I) comparando ik^2-MSv2 con k^2-raster, y (Caso II) comparando ik^2-MSv2 $p = 2$ con k^2-raster $p = 4$. La fórmula que se utiliza para este cálculo es: % mejora $= \frac{Valor\ ref - Valor\ de\ ik^2\text{-}MSv2}{Valor\ ref} \times 100$, donde el *Valor ref* corresponde a la estructura de referencia (k^2-raster o k^2-raster $p = 4$, según el caso). Un valor positivo indica que ik^2-MSv2 requiere menos espacio o menos tiempo que su respectiva referencia (más eficiente). En `Window`, los resultados de la Tabla 9.26 muestran que ik^2-MSv2 obtiene mejoras claras respecto de k^2-raster, superándolo en todos los conjuntos evaluados. Algo similar ocurre en `Window Range`, como se aprecia en la Tabla 9.27, donde la diferencia a favor de ik^2-MSv2 se mantiene tanto frente al caso del k^2-raster como frente a la variante con $p = 4$. En `Access`, mostrado en la Tabla 9.25, la ventaja también se observa de forma estable. En cuanto al almacenamiento, en la Tabla 9.23, ik^2-MSv2 utiliza en promedio menos espacio que k^2-raster. Finalmente, en el tiempo de construcción, presentado en la Tabla 9.24, se aprecia que k^2-raster presenta tiempos similares, pero al comparar ik^2-MSv2 $p = 2$ con k^2-raster $p = 4$ la reducción en tiempo de construcción es evidente; esto se debe al análisis extra que debe realizar el k^2-raster $p = 4$ para comprimir las hojas. En conjunto, los resultados demuestran que ik^2-MSv2, y especialmente la variante con $p = 2$, ofrece un mejor rendimiento en la mayoría de los aspectos evaluados, destacando los tiempos de ejecución en consultas `Window` y `Window Range`.

Dataset	Caso I (%)	Caso II (%)
wc2.1_2.5m_bio	36.50	28.25
wc2.1_5m_bio	35.80	33.42
wc2.1_10m_bio	36.75	40.04
wc2.1_10m_vapr	53.91	45.34

Tabla 9.23: Almacenamiento

Dataset	Caso I (%)	Caso II (%)
wc2.1_2.5m_bio	10.76	61.42
wc2.1_5m_bio	3.92	59.05
wc2.1_10m_bio	-2.40	58.72
wc2.1_10m_vapr	11.73	54.20

Tabla 9.24: Tiempo de Construcción

Dataset	Caso I (%)	Caso II (%)
wc2.1_2.5m_bio	34.70	34.80
wc2.1_5m_bio	31.70	42.68
wc2.1_10m_bio	39.70	34.98
wc2.1_10m_vapr	39.26	41.15

Tabla 9.25: Consulta `Access`

Dataset	Caso I (%)	Caso II (%)
wc2.1_2.5m_bio	72.10	37.64
wc2.1_5m_bio	73.51	43.14
wc2.1_10m_bio	75.29	53.23
wc2.1_10m_vapr	75.34	56.15

Tabla 9.26: Consulta `Window`

Dataset	Caso I (%)	Caso II (%)
wc2.1_2.5m_bio	67.51	25.76
wc2.1_5m_bio	74.71	46.61
wc2.1_10m_bio	73.25	44.29
wc2.1_10m_vapr	72.83	50.64

Tabla 9.27: Consulta `Window Range`

Capítulo 10

Conclusiones y Trabajo Futuro

La propuesta presentada en este trabajo introduce una nueva estructura de datos compacta para la representación eficiente de rásteres, basada en la representación binaria de los valores y en el uso del ik^2-tree, denominada ik^2-MSv2. Esta estructura aprovecha la representación en matrices binarias, técnicas como el intercalado de árboles gracias al ik^2-tree y el uso de instrucciones del procesador (`PDEP`/`PEXT`), para mejorar tanto el espacio de almacenamiento como el rendimiento en las consultas.

Los experimentos realizados demuestran que ik^2-MSv2, y especialmente su variante con aplanamiento de niveles (ik^2-MSv2 $p = 2$), ofrece un rendimiento competitivo y, en muchos casos, superior frente al k^2-raster y sus variantes heurísticas. En particular, ik^2-MSv2 $p = 2$ logra una mejora promedio del 36,76 % en espacio de almacenamiento en comparación con k^2-raster $p = 4$, y una reducción del 58,34 % en el tiempo de construcción.

Las mayores ventajas se observan en consultas espaciales, donde ik^2-MSv2 $p = 2$ demuestra mejoras frente al k^2-raster $p = 4$. En consultas `Window`, la propuesta alcanza una mejora promedio del 47,54 % frente al k^2-raster $p = 4$ en todos los conjuntos de datos evaluados. Asimismo, en consultas `Window Range`, que combinan ventanas espaciales y por valor, ik^2-MSv2 $p = 2$ supera al k^2-raster $p = 4$ en un 41,82 %.

Estos resultados confirman que el enfoque basado en la fusión de k^2-trees mediante ik^2-tree, junto con optimizaciones como el aplanamiento de niveles y el uso de `PDEP`/`PEXT`, permite un equilibrio favorable entre compresión y eficiencia en el acceso a datos.

Como trabajo futuro, se plantea finalizar y aplicar la estrategia de extracción de regiones (presentada en el Capítulo 7) directamente sobre ik^2-MSv2, con el fin de reducir más el espacio de almacenamiento. Otra línea de mejora consiste en incorporar la compresión estadística utilizada en el k^2-raster heurístico, lo que requiere el desarrollo de métodos eficientes para acceder a las matrices aplanadas en los nodos hoja. Adicionalmente, se considera extender esta estructura para soportar rásteres temporales, aprovechando la compresión a lo largo de las dimensiones espaciales y temporales.

Bibliografía

[1] de Bernardo, G., Álvarez-García, S., Brisaboa, N., Navarro, G., Pedreira, O.: Compact querieable representations of raster data. In: Proc. 14th Intl. Symp. on String Processing and Information Retrieval. pp. 96–108 (2013)

[2] Brisaboa, N., Cerdeira-Pena, A., de Bernardo, G., Navarro, G., Pedreira, O.: Extending general compact querieable representations to GIS applications. Information Sciences 506, 196–216 (2020)

[3] Brisaboa, N., Ladra, S., Navarro, G.: K2-trees for compact web graph representation. In: Proc. 16th Intl. Symp. on String Processing and Information Retrieval. pp. 18–30 (2009)

[4] Brisaboa, N.R., De Bernardo, G., Gutiérrez, G., Luaces, M.R., Paramá, J.R.: Efficiently querying vector and raster data. Computer Journal 60(9), 1395–1413 (2017)

[5] Brisaboa, N.R., Ladra, S., Navarro, G.: Dacs: Bringing direct access to variable-length codes. Inf. Process. Manage. 49(1), 392–404 (Jan 2013), https://doi.org/10.1016/j.ipm.2012.08.003

[6] Fick, S.E., Hijmans, R.J.: Worldclim 2: new 1-km spatial resolution climate surfaces for global land areas. International Journal of Climatology 37(12), 4302–4315 (2017), https://rmets.onlinelibrary.wiley.com/doi/abs/10.1002/joc.5086

[7] García, S., Brisaboa, N.R., Bernardo, G.d., Navarro, G.: Interleaved k2-tree: Indexing and navigating ternary relations. In: 2014 Data Compression Conference. pp. 342–351 (2014)

[8] Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: 13th Intl. Symp. on Experimental Algorithms, (SEA 2014). pp. 326–337 (2014)

[9] Ladra, S., Paramá, J., Silva-Coira, F.: Compact and queryable representation of raster datasets. In: Proc. 28th Intl. Conf. on Scientific and Statistical Database Management. pp. 15:1–15:12 (2016)

[10] Ladra, S., Paramá, J.R., Silva-Coira, F.: Scalable and queryable compressed storage structure for raster data. Information Systems 72, 179–204 (2017), https://www.sciencedirect.com/science/article/pii/S0306437916306214

[11] Morton, G.M.: A computer oriented geodetic data base and a new technique in file sequencing. Tech. rep., IBM Ltd., Ottawa, Ontario, Canada (1966)

[12] Pinto, A., Seco, D., Gutiérrez, G.: Improved queryable representations of rasters. In: 2017 Data Compression Conference (DCC). pp. 320–329 (2017)

[13] WorldClim: Worldclim version 2.1 climate data for 1970-2000. `https://www.worldclim.org/data/worldclim21.html` (2020)