UNIVERSIDAD DEL BÍO-BÍO

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

Ampliación de las Capacidades de Estructuras de Datos Compactas

---

Por
Carlos Felipe Quijada Fuentes

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:
Dra. Susana Ladra
Universidade Da Coruña, A Coruña, España

2017 - I

Estudios de Postgrado financiados por CONICYT-PCHA/MagisterNacional/2015 - 22150776.

a mi hija Antonia Estefanía

Agradecimientos

Los principales agradecimientos de este trabajo estan dirigidos a mis padres, quienes siempre han entregado su apoyo (a mis hermanos y a mi) en nuestro desarrollo académico-profesional. Así también a mi numerosa familia, que siempre estuvo al pendiente en el desarrollo de mis estudios (Quijada, Fuentes, Guzmán y Solis).

Agradecer especialmente a la profesora M. Angélica Caro, gracias a quien inicié este desafío debido a las reiteradas ocasiones en las que me habló del magíster y las ventajas de realizarlo.

Agrego también en este apartado a los profesores guía de esta tesis: profesor Gilberto Gutiérrez y profesora Susana Ladra. Así también el profesor Miguel R. Penabad. Fue una grata experiencia el trabajar con ellos en el desarrollo de la tesis.

Al profesor Alfonso Rodríguez quien, en su labor de director, estaba al pendiente del avance de los tesistas, y quien nos traspasó diferentes experiencias de sus labores de investigador en el pasillo, sala de estudio y en las ocasiones en que pudimos disfrutar de la hospitalidad de su casa.

A mis compañeros de magíster, integrantes del grupo de base de datos de la Universidade da Coruña y amigos que preguntaron por el avance de la tesis, y estaban al pendiente de las actividades, a quienes no mencionaré debido a que mi memoria fallaría y más de alguno faltaría en la lista.

A los docentes y personal administrativo de la Facultad de Ciencias Empresariales de la sede Chillán, quienes también entregaron su afecto y preocupación.

A todos quienes dieron alguna palabra de apoyo en esta experiencia, muchas gracias.

Resumen

Actualmente las personas generan enormes cantidades de datos día a día, ya sea por los sitios web que visitan, por las redes sociales en las que se registran, por las búsquedas que hacen en Internet, etc.. Así también existen muchas instituciones que se dedican a procesar y extraer información desde los datos generados día a día, donde toma real importancia el tamaño del conjunto de datos que se procesa, y el tiempo que tardan en extraer la información.

Existen estructuras de datos que pretenden abordar esta problemática almacenando los datos en el menor espacio, con la posibilidad de realizar consultas de interés sin la necesidad de descomprimir la información almacenada. Estas estructuras son llamadas Estructuras de Datos Compactas, y se utilizan en distintos ámbitos de la computación, como por ejemplo: representación de grafos, almacenamiento de texto, sumas parciales, geometría computacional, autoindexación, representación de relaciones binarias, etc...

Una problemática importante que se presenta en el uso de Estructuras de Datos Compactas radica en la carencia de algunas funciones que podrían resultar útiles para el procesamiento de la información que se almacena en ellas. Considerando lo anterior, si se tiene información almacenada en una Estructura de Datos Compacta, y esta no dispone de una determinada consulta necesaria para el procesamiento de dichos datos, será necesario descompactar la información que contiene la estructura a una representación ingenua en la que sí existe tal consulta, y posteriormente, para mantener la información en un espacio de memoria reducido, se debería compactar el resultado. La situación anterior resulta totalmente ineficiente, tanto por el tiempo que toma el pasar desde una estructura a otra, y también por el espacio de memoria requerido durante todo el proceso.

Una de las aplicaciones para estructuras de datos compactas, como ya se mencionó, es representar relaciones binaras. Las relaciones binarias pueden ser utilizadas para representar las relaciones que existen entre usuarios de redes sociales, grafos web, diccionarios de palabras, etc. Algunos de los ejemplos anteriores pueden llegar a contener mucha información, por lo que almacenarla de modo compacto y disponer de una estructura de datos compacta para manipular dicha información resulta de gran interés.

En esta tesis se diseñan, implementan y evalúan algoritmos para resolver operaciones de conjunto sobre dos estructuras de datos compactas utilizadas para la representación de relaciones binarias. Estas estructuras de datos compactas son el k^2-tree y el wavelet tree de relaciones binarias.

Mediante una serie de experimentos que consideraron datos reales, se evaluaron los algoritmos en términos de tiempo de ejecución y de memoria necesaria para procesar las operaciones de conjuntos de relaciones binarias. Los resultados experimentales muestran que tanto en

tiempo y almacenamiento nuestras implementaciones superan a aquellas que primero descompactan la estructura de datos, realizan la operación y luego compactan el resultado. De esta forma, nuestros algoritmos permiten ampliar las capacidades de las estructuras de datos k^2-tree y wavelet tree de relaciones binarias.

Índice

Parte I

Motivación

Capítulo 1

Introducción

1.1. Motivación

Las Estructuras de Datos Compactas (EDC) son estructuras que buscan almacenar la mayor cantidad de información en el menor espacio posible soportando consultas de interés sin descomprimir la información almacenada para entregar una respuesta [18, 24].

La compactación trae algunos beneficios como: permite reducir el número de lecturas que se realizan desde el disco cuando la información no cabe en memoria principal, se puede mantener una representación muy grande que en su formato original no cabe en memoria RAM pero al estar compactada es tratable en memoria principal, mejora el rendimiento de la memoria caché permitiendo más cantidad de información en ella. Sin embargo, lo anterior sólo es útil si la EDC permite la ejecución eficiente de consultas sobre la información que contiene [2].

Las EDC se utilizan para representar imágenes [22], relaciones en redes sociales [3], datos ráster (Mosaicos de datos) [7, 16], relaciones binarias, permutaciones de texto [21], diccionarios, sumas parciales, grafos web [7, 14, 13], secuencias de ADN [8], árboles binarios de decisión [17, 23, 25], filtros bloom [15], entre otras aplicaciones.

En general, las EDC evolucionan en el tiempo pudiendo haber algunos cambios en su estructura general, lo que produce variantes de la EDC, o también pueden existir extensiones a la EDC, lo que incorpora nuevas funcionalidades para la información almacenada en la EDC; en ambos casos se pretende ampliar los usos de la EDC respecto de las propuestas originales.

Entre las EDC más conocidas podemos mencionar el k^2-tree propuesto por Brisaboa *et al.* [10] y el wavelet tree, los cuales se utilizan en esta tesis de magíster.

Anteriormente Brisaboa *et al.* [7] diseñaron algoritmos para resolver las operaciones de Unión, Intersección, Diferencia y Complemento sobre el k^2-tree, con el fin de generar una representación de relaciones binarias funcional más completa [7].

En esta tesis se implementan y evalúan los algoritmos propuestas por Brisaboa *et al.* para resolver las operaciones de Unión, Intersección, Diferencia y Complemento sobre el k^2-tree. Además se realiza el diseño del algoritmo para la operación de Diferencia Simétrica, su implementación y evaluación sobre la misma estructura. También sobre el wavelet tree se diseñan algoritmos para las operaciones de Unión, Diferencia, Intersección, Diferencia Simétrica y Complemento, así como

su implementación y evaluación. Para el k^2-tree se compara el rendimiento de la implementación contra el tiempo que toma descomprimir la información desde la EDC, aplicar las operaciones sobre la representación no compacta y comprimir el resultado, pensando que la EDC no posee una implementación para resolver estas operaciones de manera directa. En el caso del wavelet tree se realiza una comparación contra la implementación del k^2-tree.

1.2. Hipótesis y Objetivos

Hipótesis

Dadas dos relaciones binarias sobre el mismo conjunto de objetos representados por la misma EDC es posible diseñar algoritmos que realicen las operaciones de Unión, Diferencia, Intersección, Diferencia Simétrica y Complemento de manera más eficiente que una implementación de EDC que no posee las operaciones, la cual requiere descomprimir los dos conjuntos, operar sobre éstos y posteriormente compactar el resultado de la operación.

Objetivo General

El objetivo general es desarrollar y evaluar algoritmos eficientes para implementar las operaciones de Unión, Intersección, Diferencia, Diferencia Simétrica y Complemento de modo que tomen ventajas de las propiedades de las EDC manipuladas en memoria principal.

Objetivos Específicos

- Implementar los algoritmos de Unión, Intersección, Diferencia y Complemento propuestos por Brisaboa *et al.* [7] sobre el k^2-tree.
- Diseñar e implementar el algoritmo de Diferencia Simétrica sobre el k^2-tree.
- Diseñar e implementar los algoritmos de Unión, Diferencia, Intersección, Diferencia Simétrica y Complemento sobre el wavelet tree de Relaciones Binarias propuesto por Barbay *et al.* [1].
- Evaluar los algoritmos implementados con datos de prueba reales. En el caso del k^2-tree, evaluar el desempeño contra ejecutar las operaciones sobre representaciones compactas en las que se deben descompactar, operar y compactar el resultado. En el caso del wavelet tree, evaluar contra el comportamiento del k^2-tree.

1.3. Alcance de la Investigación

Este trabajo contempla la implementación de operaciones sobre conjuntos en el k^2-tree y el wavelet tree de Relaciones Binarias utilizando el lenguaje de programación C.

Esta implementación será la primera en realizar operaciones de conjuntos sobre EDC pensadas en la representación de relaciones binarias entre objetos.

1.4. Metodología de Trabajo

La metodología de trabajo consta de las siguientes etapas:

1. Ampliar la revisión de la literatura.
2. Implementar las operaciones de Unión, Diferencia, Intersección y Complemento sobre el k^2-tree.
3. Diseñar el algoritmo de Diferencia Simétrica para el k^2-tree.
4. Implementar la operación de Diferencia Simétrica sobre el k^2-tree.
5. Diseñar algoritmos para las operaciones de Unión, Intersección, Diferencia, Diferencia Simétrica y Complemento para el wavelet tree de Relaciones Binarias.
6. Implementar las operaciones de Unión, Intersección, Diferencia, Diferencia Simétrica y Complemento en el wavelet tree de Relaciones Binarias.
7. Implementar, para la etapa de pruebas, las operaciones de Unión, Intersección, Diferencia, Diferencia Simétrica y Complemento para el k^2-tree con el proceso de descompactación de las entradas, cálculo del resultado como lista de adyacencia, y compactación del resultado a k^2-tree.
8. Realizar el análisis experimental con datos reales.

1.5. Contribución

En el desarrollo del presente trabajo se tiene las siguientes contribuciones:

- Se implementan los algoritmos propuestos para operaciones de conjunto sobre la EDC k^2-tree propuestas por Brisaboa *et al.* [7].
- Se propone un algoritmo para calcular la Diferencia Simétrica en el k^2-tree.
- Se realizan pruebas del comportamiento de la implementación de los puntos anteriores con datos de prueba reales.
- Se proponen algoritmos para las mismas operaciones (Unión, Diferencia, Intersección, Diferencia Simétrica y Complemento) para el wavelet-tree de relaciones binarias.
- Se realizan pruebas del comportamiento con datos reales sobre una implementación básica del wavelet-tree.

De lo anterior se ha logrado la difusión de la investigación por medio de la participación en:

- Quijada-Fuentes, C., Gutiérrez G., Ladra S.: Implementación y Experimentación de Operaciones de Conjunto de Relaciones Binarias representados mediante k^2-tree. V Encuentro de Investigación de Estudiantes de Postgrado. Universidad del Bío-Bío (2016).

Así también se ha enviado el artículo a la revista Information Systems[1] de la editorial Elsevier:

- Quijada-Fuentes, C., Penabad, M. R., Ladra S., Gutiérrez G.: Set Operations over Compressed Binary Relations. Information Systems (2017).

1.6. Contenidos

Este documento de tesis se organiza en siete capítulos. El primero incluye la introducción, motivación y objetivos planteados para el trabajo que se presenta.

En el Capítulo 2, producto de la revisión de la literatura relativa a las estructuras de datos compactas, se mencionan algunas aplicaciones de las operaciones de conjunto.

El Capítulo 3 describe los algoritmos que plantean Brisaboa *et al.* [7] para realizar operaciones de conjunto sobre relaciones binarias almacenadas en k^2-tree. Así también se describe un algoritmo adicional incorporado al desarrollo de esta tesis de magíster para complementar la propuesta anterior en relaciones de conjunto sobre EDC.

El Capítulo 4 describe la implementación y la experimentación sobre el k^2-tree, incluyendo la descripción de los datos utilizados, las *líneas de comparación* definidas y los criterios evaluados. En este capítulo se incluyen gráficas de los resultados obtenidos para los tiempos de ejecución sobre el k^2-tree.

Los detalles de implementación, experimentación y resultados obtenidos sobre el wavelet tree son desarrollados en el Capítulo 5. En este capítulo también se incluye una sección con resultados de la medición del espacio de memoria requerido por las implementaciones para ejecutar las operaciones de conjunto sobre las EDC.

Se finaliza este trabajo con el Capítulo 6 donde se entregan las conclusiones y la descripción de futuras líneas de investigación posibles a abordar.

[1] `https://www.journals.elsevier.com/information-systems/`

Parte II

Trabajo Relacionado

Capítulo 2

Estructuras de Datos Compactas

Una EDC es una estructura de datos que almacena información en un espacio de menor al espacio utilizado por una representación común, permitiendo realizar consultas de interés sobre esta información de manera eficiente.

En general se dice que una EDC usa $O(n)$ bits de espacio donde n es el número óptimo de bits necesarios para almacenar los datos según la teoría de la información propuesta por Shannon [29].

2.1. Algunas EDC

A continuación se describen brevemente dos EDC ampliamente conocidas, las cuales se utilizan para compactar distintos tipos de información.

2.1.1. Bitmap

Los bitmaps son utilizados ampliamente por otras EDC, formando parte importante de éstas. Básicamente, las diferentes EDC definen la forma en que se almacenará la información en los bitmaps e implementan nuevas consultas de interés para determinados escenarios o contextos.

El bitmap corresponde a un arreglo unidimensional de bits (o secuencia de bits) propuesto por Jacobson [20], el cual posee estructuras adicionales para responder consultas de rank y select, donde la primera llega a ser en tiempo constante. Las consultas de *rank* y *select* permiten responder cuántos bits con valor 1 existen desde el inicio hasta la i-ésima posición y cuál es la posición del i-ésimo bit con valor 1 respectivamente. Adicionalmente se debe disponer de la consulta *access* que entrega el valor del i-ésimo bit. En la Figura 2.1 se puede ver un ejemplo de un bitmap y de sus operaciones.

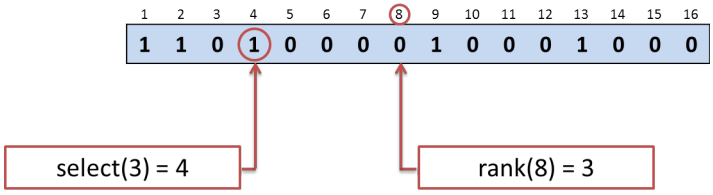


Figura 2.1: Ejemplo de bitmap y sus operaciones.

Para lograr responder estas consultas de *rank* y *select*, esta estructura segmenta el arreglo de bits en trozos de tamaño constante, y por cada trozo se almacena la suma de bits en 1 que contiene hasta su última casilla. Por ejemplo, si para el bitmap de la Figura 2.1 se divide en segmentos de 4 elementos, se requerirá de un arreglo adicional que almacene 4 valores, los cuales serían 3, 3, 4 y 5. De esta manera se responde la operación rank de manera constante, donde en el peor de los casos se deben revisar 4 casillas del bitmap.

2.1.2. Wavelet Tree

El wavelet tree es una EDC propuesta por Grossi, Gupta y Vitter [19] para la representación de secuencias de texto.

Dada una cadena de texto S del alfabeto Σ, el wavelet tree corresponde a un árbol en el que cada nodo posee dos hijos, donde cada nodo representa un subconjunto del alfabeto. Un nodo se compone de un bitmap, donde el nodo raíz tiene tantos bits como caracteres tiene S. En dicho bitmap el bit b_i es 0 si el i-ésimo elemento de S pertenece a la primera mitad de Σ (lexicográficamente ordenada), y 1 si el elemento pertenece a la segunda mitad. Luego, el hijo izquierdo del nodo raíz considera como S aquellos elementos de S_i donde $b_i = 0$ en el nodo padre y considera como alfabeto la primera mitad del alfabeto del padre. El hijo derecho corresponde a los elementos S_i tal que $b_i = 1$ en el padre y su alfabeto es la segunda mitad del alfabeto. La construcción del árbol genera nodos hasta que el alfabeto sólo posee un elemento. Un ejemplo de esto se puede ver en la Figura 2.2 que representa un wavelet tree para la secuencia de texto S ={ÑS_YRDOD_RD_FOGOVOÑ_FR_GOMSÑOXSR!_:).} y el alfabeto Σ = {,_,:,!,),.,D,F,G,M,Ñ,O,R,S,V,X,Y,}.

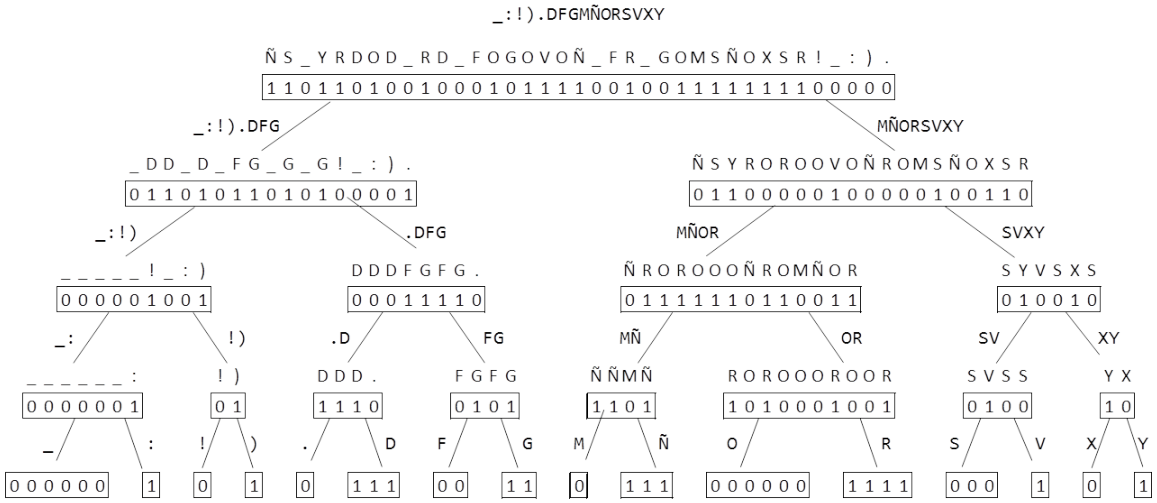


Figura 2.2: Ejemplo de wavelet tree.

2.2. EDC para la Representación de Relaciones Binarias

Dada una relación $R = \{(a, b) \in A \times B\}$ esta se puede representar mediante una matriz de adyacencia o también por medio de una lista de adyacencia, ambas ejemplificadas en la Figura 2.3 donde se da que $A = B$.

Estas estructuras son las que tradicionalmente se utilizan para su representación y permiten consultar la existencia de una relación en orden lineal para una lista de adyacencia y en orden constante para una matriz de adyacencia..

Una matriz de adyacencia de una relación binaria sobre objetos del mismo conjunto con n objetos se representa mediante una matriz cuadrada de $n \times n$ celdas que corresponde a un subconjunto del producto cartesiano entre los objetos. Si la celda c_{ij} es 1 significa que el objeto i posee una relación hacia el nodo j, cuando la celda c_{ij} es 0 no existe relación desde i hacia j [11]. De esta forma, los pares de R están representados por aquellas celdas $c_{ij} = 1$ que indican que el objeto a_i se relaciona con a_j, cuando la relación es entre objetos del mismo conjunto.

A continuación se describen dos EDC utilizadas para la representación de relaciones binarias, que en el desarrollo de esta tesis se utilizarán para representar relaciones sobre objetos del mismo conjunto.

2.2.1. k^2-tree

El k^2-tree es un árbol que toma ventaja de las grandes áreas que contienen 0s en la matriz de adyacencia de la relación binaria. Esta EDC fue propuesta originalmente para la representación compacta de grafos web pero posteriormente se ha aplicado, entre otros, a:

- Representación de relaciones binarias, como las que se dan en las redes sociales [11], presentando buenos resultados.

	1	2	3	4	5	6	7	8
A	.	.	1	.	.	.	.	.
B	.	.	.	.	.	1	1	.
C	.	.	.	1	.	1	.	1
D	.	1	.	.	.	.	.	.
E	1	.	.	1	1	.	.	.
F	.	.	.	.	.	.	.	.
G	.	.	.	.	1	.	1	.
H	1	1	.	.	.	.	.	.

A: 3
B: 6, 7
C: 4
D: 2
E: 1, 4, 5
F:
G: 5, 7
H: 1, 2

Figura 2.3: Ejemplo de Matriz de Adyacencia (izquierda), modificada de Barbay *et al.* [1] y su respectiva Lista de Adyacencia (derecha).

- Datos ráster [16] (o Raster Data) donde se representa información de una superficie mediante una matriz en la que cada celda corresponde a una porción de la superficie original.
- Conjuntos de datos RDFs [9] (Resource Description Framework).

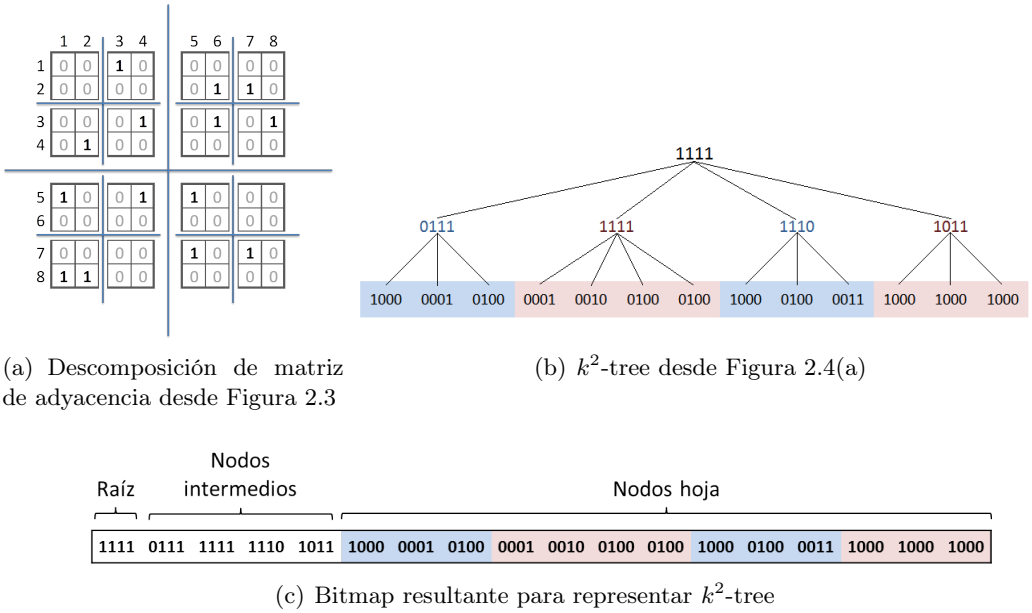
En la Figura 2.4(b) se muestra la estructura del árbol compuesto de nodos que poseen k^2 bits, donde los $1s$ representan un nodo hijo, y los $0s$ una rama vacía, exceptuando los nodos hoja donde 1 indica la existencia de una relación entre dos objetos, y el valor 0 lo contrario. El árbol se construye de manera recursiva dividiendo la matriz de adyacencia en k^2 cuadrantes cada vez, hasta que cada cuadrante contiene k^2 celdas. Cada nodo representa un cuadrante de la matriz de adyacencia donde las dimensiones del cuadrante representado dependen del nivel al que se encuentre el nodo. En el nodo raíz cada bit representa n^2/k^2 celdas, donde n es el número de objetos del conjunto y, por tanto, n^2 es el número de celdas de la matriz de adyacencia. De esta manera se continúa dividiendo la matriz de adyacencia hasta obtener cuadrantes de k^2 celdas.

En cada nodo, los cuadrantes se representan como sigue; si posee únicamente $0s$ el bit que lo representa en el nodo del árbol es 0 y no se continúa realizando la división de ese cuadrante. En caso contrario, cuando el cuadrante posee $1s$ ó $1s$ y $0s$, el bit que lo representa en el nodo es 1 y la división recursiva continua realizándose hasta que se obtiene un cuadrante de k^2 celdas o uno que sólo posea $0s$. En la Figura 2.4(b) se muestra el k^2-tree resultante para la matriz de adyacencia de la Figura 2.4(a).

Cabe destacar que si una matriz de adyacencia no posee exactamente k^i nodos, con $i \in \mathbb{N}$, el k^2-tree resultante representará una matriz de adyacencia tal que $i = \lceil log_k n \rceil$, que corresponde a la siguiente potencia de k, donde el contenido de las celdas adicionales se define como $0s$. En este caso la estructura compacta aquellos cuadrantes generados, ya que sólo poseen $0s$. En cualquier caso la altura del árbol resultante corresponde a $\lceil log_k n \rceil$.

Una vez construida la estructura no es necesario almacenar el árbol resultante con todos sus punteros, la estructura corresponde a un bitmap con la concatenación de todos los niveles como se muestra en el ejemplo de la Figura 2.4(c). Este bitmap se puede recorrer mediante operaciones de rank y select.

El k^2-tree es capaz de responder consultas como: la existencia de una relación entre dos objetos, los objetos que están relacionados con un objeto dado, los objetos con los que se relaciona



(a) Descomposición de matriz de adyacencia desde Figura 2.3

(b) k^2-tree desde Figura 2.4(a)

(c) Bitmap resultante para representar k^2-tree

Figura 2.4: Descomposición de matriz de adyacencia para generar k^2-tree con $k = 2$.

un objeto dado y las relaciones existentes dado un rango de objetos de origen y un rango de objetos de destino para las relaciones.

Brisaboa *et al.* [9] proponen una estructura dinámica para la representación de relaciones binarias que soporta las mismas consultas antes mencionadas (existencia de una relación, consultas por rango, vecinos directos y vecinos reversos). La estructura se denomina dk^2-tree y demuestra un buen comportamiento en la inserción/eliminación de vínculos de la relación binaria y añadiendo/quitando objetos desde los conjuntos A o B. Adicionalmente la estructura mantiene un espacio cercano al de su símil estático, el k^2-tree.

Posteriormente existe un nuevo aporte de Brisaboa *et al.* [7] donde se proponen algoritmos de operaciones de conjunto (Unión, Intersección, Diferencia y Complemento) sobre el k^2-tree con el objetivo de dar mayor soporte a la representación de relaciones binarias. Estos algoritmos corresponden al punto de partida para este trabajo de tesis de magíster en el que se evalúa el comportamiento de los algoritmos.

2.2.2. Wavelet Tree de Relaciones Binarias

Luego de la propuesta original, Barbay, Claude y Navarro [1] describen una variante del wavelet tree para la representación de relaciones binarias a la que llaman Binary Relations Wavelet Tree (BRWT).

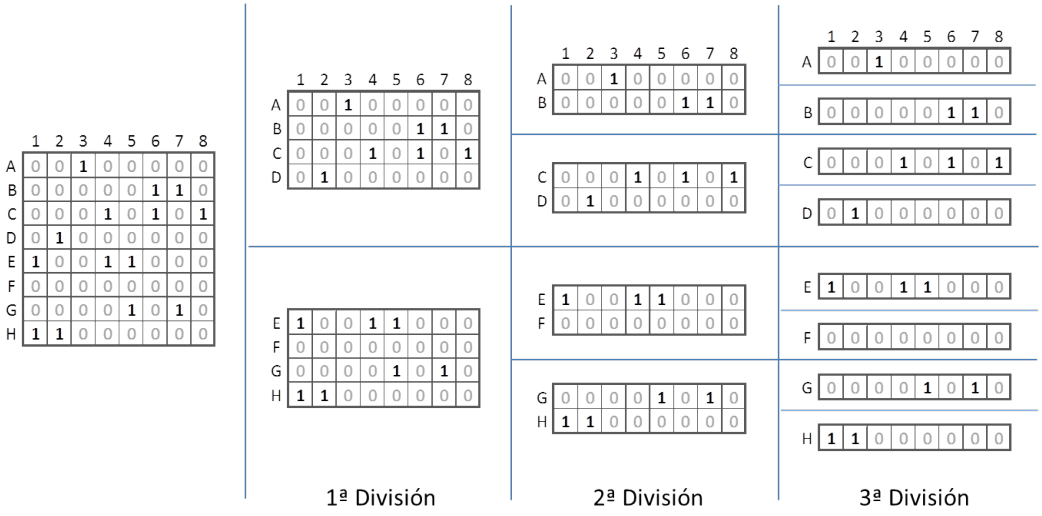


Figura 2.5: Descomposición de matriz de adyacencia (desde Figura 2.3) para formar un wavelet tree de relaciones binarias.

El BRWT es una estructura que se construye recursivamente a partir de una matriz de adyacencia como se ve en la Figura 2.5. Cada nodo posee dos bitmaps, los cuales en el primer nivel representan cada mitad de las filas de la representación. Luego el hijo izquierdo define la primera mitad de las filas, y el hijo derecho la segunda mitad de las filas dividiéndolas de la misma manera. Los bitmaps contienen 1 siempre que exista al menos una relación desde el conjunto de filas representado hacia la columna correspondiente. En el primer nivel se definen todas las columnas, independiente si existen o no vínculos desde las filas, en cambio en los demás niveles, sólo se definen las columnas que contienen al menos un vínculo, omitiendo aquellas columnas que no presentan relaciones. Esta situación se puede apreciar en la Figura 2.6 en el segundo nivel, donde el nodo de la izquierda no define la columna 1 dado que no posee relaciones desde el conjunto de filas que le corresponde a ese nodo. Esta figura muestra la representación de un BRWT donde las filas son etiquetas desde A a H, y las columnas son objetos numerados de 1 al 8, y representan el mismo conjunto de relaciones que el de la Figura 2.3. Se debe señalar que estos conjuntos podrían ser el mismo donde las etiquetas de las filas y las columnas coincidirían.

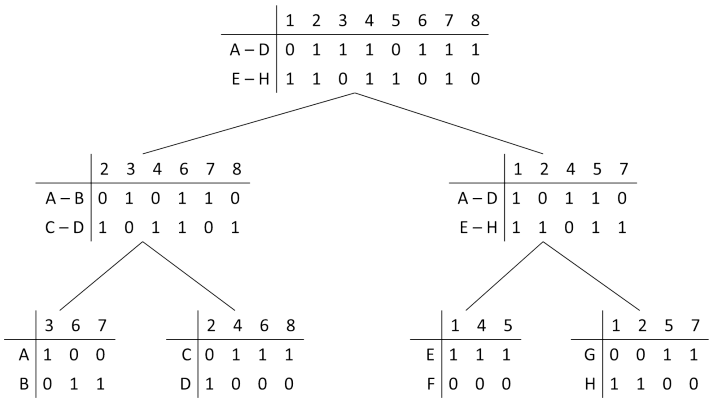


Figura 2.6: Representación de BRWT desde Barbay *et al.* [1] desde Figura 2.5.

2.3. Operaciones de Conjunto en EDC

2.3.1. Operaciones de Conjunto

En esta tesis de magíster se hace uso de cinco operaciones de conjunto; Unión, Diferencia, Intersección, Diferencia Simétrica y Complemento, donde las cuatro primeras son operaciones que se realizan sobre dos conjuntos, en cambio el Complemento se aplica sobre un conjunto. Recordar que una relación binaria corresponde a un conjunto de pares, y que los elementos sobre los que se aplican las operaciones de conjunto que aquí se describen son sobre relaciones binarias.

Unión

La Unión de dos conjuntos A y B se denota por $A \cup B$ y corresponde a todos los elementos del conjunto A y B (Ver Figura 2.7(a)). Formalmente se describe como $A \cup B = \{x | x \in A$ o bien $x \in B\}$

Por lo anterior también se cumple que A y B son subconjuntos de $A \cup B$. Así también se da que $A \cup B = B \cup A$.

Si tomamos como ejemplo las relaciones entre los usuarios de una red social, y se representan dos instantes de tiempos distintos (podría ser un año de diferencia por ejemplo) el aplicar la Unión sobre estas dos representaciones se obtendría como resultado todas las relaciones entre usuarios que existen en cualquiera de esos dos instantes de tiempo en la red social.

Diferencia

La Diferencia se denota como $A - B$ y corresponde a todos aquellos elementos de A que no pertenecen a B (Ver Figura 2.7(b)), vale decir $A - B = \{x | x \in A$ y $x \notin B\}$

En la operación de Diferencia se cumple que $A - B \subseteq A$ y que $A - B \neq B - A$.

Considerando las relaciones que existen entre usuarios de una red social en dos instantes de tiempo distinto, al realizar la operación de Diferencia del instante más reciente menos el instante

más antiguo, se obtiene como resultado una representación de aquellas nuevas relaciones generadas antes del instante de tiempo más reciente que no estaban presentes en el instante antiguo.

Intersección

La Intersección de los conjuntos A y B se denota por $A \cap B$ y se compone por aquellos objetos que pertenecen a A y también pertenecen a B, formalmente $A \cap B = \{x | x \in A \text{ y } x \in B\}$ (Ver Figura 2.7(c)).

Al igual que la Unión, la Intersección cumple que $A \cap B = B \cap A$. Además, todos los elementos de la intersección están tanto en A como en B, vale decir $A \cap B \subseteq A$ y $A \cap B \subseteq B$.

Retomando el ejemplo de las relaciones entre los usuarios de una red social, la operación de Intersección sobre dos representaciones de instantes de tiempo distinto generaría como resultado una representación de aquellas relaciones que se han mantenido durante el transcurso del tiempo que existe entre ellas.

Complemento

La operación de Complemento se aplica sobre un conjunto de objetos, y es necesario comprender lo que se denomina **universo** el cual se compone por todos los posibles objetos a ser contenidos en un conjunto. Por tanto, para una relación binaria sobre objetos de un conjunto, el conjunto *universo* corresponde a todas relaciones posibles entre los objetos del conjunto. El *universo* se denota por U, y según lo anterior $U = A \times A$.

El Complemento del conjunto A corresponde a los elementos que pertenecen a U y que no están en A (Ver Figura 2.7(e)), denotado por $\overline{A} = U - A$.

Por tanto se tiene que $\overline{A} = \{x | x \in U \text{ y } x \notin A\}$, y se cumple que $\overline{\overline{A}} = A$, vale decir que al aplicar la operación de complemento sobre el complemento de un conjunto A, se obtiene por resultado el conjunto A.

Si se aplica el complemento sobre una representación de relaciones entre usuarios de una red social, el resultado contendría aquellas relaciones que no existen en la red social.

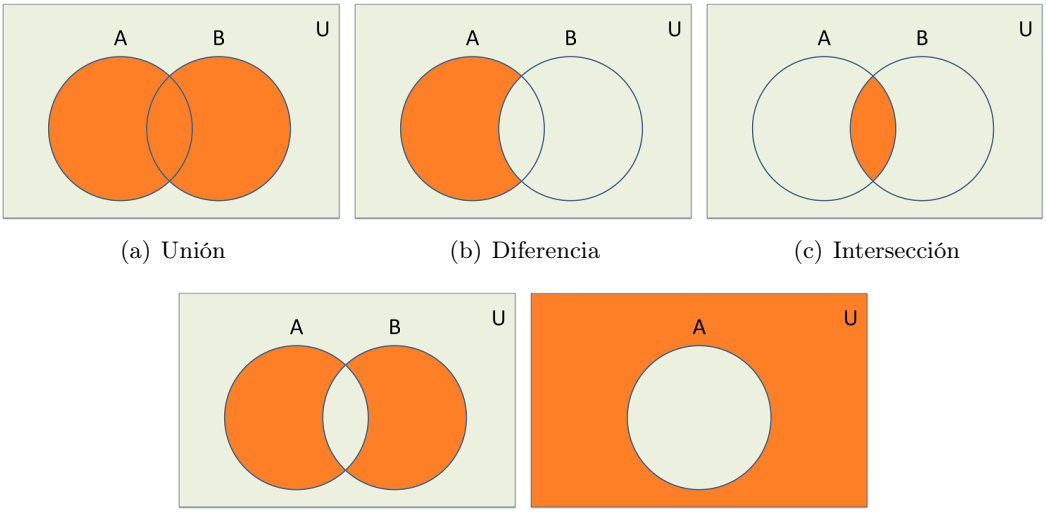
Diferencia Simétrica

En este trabajo se ha decidido incluir una operación de conjunto adicional a las comunes, la cual corresponde a la Diferencia Simétrica, la que se aplica sobre dos conjuntos y corresponde a todos los elementos que están en uno de los conjuntos, pero no en el otro (Ver Figura 2.7(d)). Se denota por $A \triangle B = \{A - B\} \cup \{B - A\}$, o lo que también es equivalente $A \triangle B = \{A \cup B\} - \{B \cap A\}$.

Más formalmente se define $A \triangle B = \{x | x \in A \text{ y } x \notin A \text{ o bien } x \in B \text{ y } x \notin B\}$.

Así también se cumple que $A \triangle B = B \triangle A$.

Un ejemplo utilizando relaciones entre usuarios de una red social, la operación de Diferencia Simétrica, al operar con relaciones que representan dos instantes de tiempo diferentes, entrega como resultado una representación que contendría aquellas nuevas relaciones (presentes en el instante de tiempo reciente pero no en el antiguo) y aquellas relaciones antiguas (aquellas que sólo están en la representación antigua pero no en la reciente).



(a) Unión (b) Diferencia (c) Intersección

(d) Diferencia Simétrica (e) Complemento

Figura 2.7: Diagramas de Venn de las operaciones de conjunto que se implementarán.

En la Figura 2.7 se representa mediante diagramas de Venn las cinco operaciones antes descritas.

2.3.2. Aplicación de Operaciones de Conjunto en EDC

El uso de Operaciones de conjunto con relaciones binarias almacenadas en EDC no ha sido altamente explotado según la revisión de la literatura que se ha realizado. Las referencias encontradas de operaciones de conjunto implementadas sobre una EDC corresponde al quadtree y se describen a continuación.

Samet [27], en una revisión de la literatura, menciona algunas investigaciones que hacen referencia a la utilidad que entrega el quadtree para realizar operaciones de conjunto sobre imágenes, y se mencionan las operaciones de Unión e Intersección de capas.

Más tarde, Lin [22] propone más operaciones de conjuntos para una adaptación del quadtree denominada Constant Bit-Length Linear Quadtree (CBLLQT), donde tales operaciones están pensadas para realizar comparaciones entre imágenes. Las operaciones son definidas mediante tablas de acción y corresponden al Complemento, la Unión, la Intersección, el Complemento Relativo y la Diferencia Simétrica. Cabe destacar que las implementaciones logran operar en tiempo lineal sobre el quadtree.

Por otra parte, el uso que se registra de operaciones de conjunto en el ámbito de las Ciencias de la Computación es muy variado. A continuación se describen algunos ejemplos:

- En la Geometría computacional, específicamente en el Diseño Dirigido por Computadora (CAD por sus siglas en inglés), se hace un uso intensivo de las operaciones de Unión, Intersección o Diferencia al momento de manipular las capas que son diseñadas en el software [26].

- Casey y Shelmire [12] hacen uso de las Operaciones de Conjunto para determinar las coincidencias en los segmentos de código de malware perteneciente a las familias Stuxnet y Duqu. Estas coincidencias se reconocen mediante el uso de estructuras de datos sucintas y estructuras de datos de sufijos, aplicando operaciones de Unión, Intersección y Complemento sobre los códigos maliciosos.
- Por otra parte, es posible determinar coincidencias en imágenes representadas en quadtrees y OcTrees como lo menciona Samet [28], donde también discuten la ventaja que presenta las estructuras de datos jerárquicas para aplicar este tipo de operaciones [27].

Finalmente, queda de manifiesto la importancia de las operaciones sobre conjuntos dentro de las Ciencias de la Computación, pero al momento sólo el CBLLQT plantea la implementación de estas operaciones y su uso no es sobre relaciones binarias, ya que el quadtree se utiliza para la representación de imágenes y su manipulación.

Capítulo 3

Operaciones de Conjunto sobre el k^2-tree

En este Capítulo se describen los algoritmos propuestos por Brisaboa *et al.* [7] para realizar las operaciones de Unión, Diferencia, Intersección y Complemento sobre la EDC k^2-tree. Adicionalmente, en esta tesis, se propone un algoritmo para el cálculo de la Diferencia Simétrica sobre la misma EDC.

3.1. Los Algoritmos

A continuación se describen algunos elementos generales que facilitarán la comprensión al momento de analizar los algoritmos presentado más adelante.

- A y B corresponde a la concatenación de los bitmaps con que se representa la relación binaria almacenada según la estructura del k^2-tree.
- C se usa para almacenar el resultado de la operación en los algoritmos de operaciones de conjunto. En la operación de Unión corresponde a un bitmap, mientras que en las demás operaciones se considera a C como un arreglo de n bitmaps, donde n corresponde a la cantidad de niveles que posee el k^2-tree.
- pA y pB son arreglos que, al inicio de cada algoritmo, almacenan las posiciones donde comienza cada nivel del k^2-tree en los bitmaps A y B respectivamente. Por lo tanto el tamaño del arreglo corresponde a la cantidad de niveles del k^2-tree, o lo que es lo mismo; corresponde a la altura del árbol que representa.
- l es un entero que indica el nivel del k^2-tree que se está procesando.
- t es un arreglo de tamaño k^2 que almacena el resultado parcial del nodo que se procesa. El contenido de t será añadido (de corresponder) al bitmap C en el nivel l.
- *writesomething* es una variable utilizada en las operaciones recursivas (Diferencia, Intersección, Diferencia Simétrica y Complemento) para determinar si corresponde añadir el

resultado parcial al nivel l del bitmap C. Se podría dar el caso que en una exploración, al llegar a las hojas (o antes) se determine como resultado parcial sólo 0s (ceros), los cuales no deben ser insertados en el k^2-tree.

Algunos de los algoritmos recursivos requieren de las funciones adicionales: *Copy*, *SkipNodes* y *FillIn* que serán descritos más adelante.

Para apoyar la descripción de los algoritmos se hace uso de ejemplos de k^2-tree que están representados en las figuras 3.1 y 3.2. Ambas figuras presentan la matriz de adyacencia y la estructura del k^2-tree correspondiente.

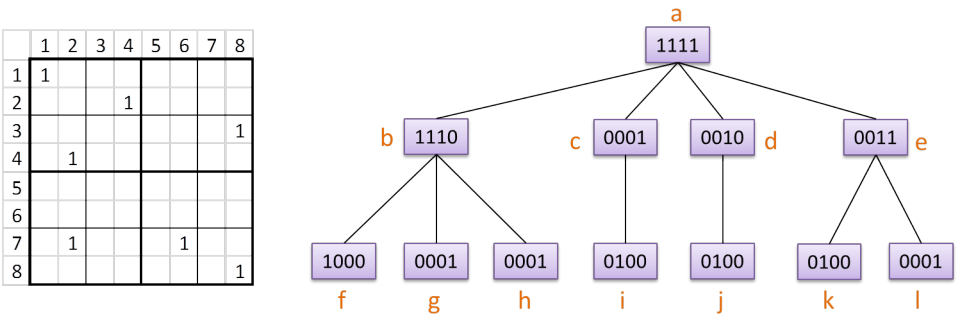


Figura 3.1: Ejemplo: k^2-tree A.

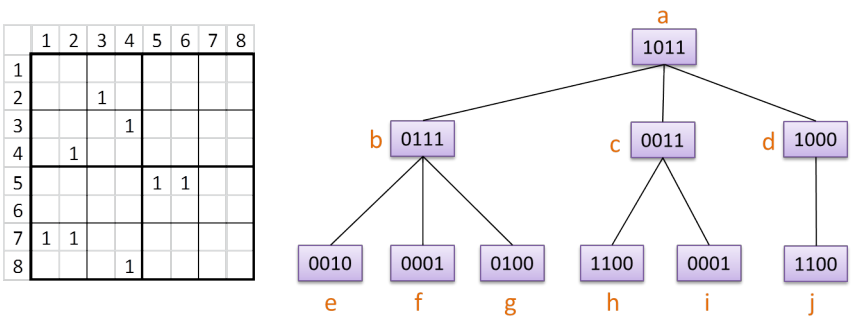


Figura 3.2: Ejemplo: k^2-tree B.

3.1.1. Unión

El Algoritmo 3.1.1 se propuso para el cálculo de la operación de Unión entre dos relaciones binarias representadas mediante k^2-tree. Este algoritmo recibe como parámetros dos k^2-tree representados por los bitmaps A y B, y realiza un recorrido sincronizado de estos bitmaps utilizando como estrategia de recorrido una exploración en anchura de los árboles a la vez que registra el resultado en el bitmap C. Este recorrido es posible ya que, a diferencia de las demás operaciones, la Unión no requiere llegar hasta los nodos hojas para determinar el resultado en un nodo padre, es suficiente saber que existe un elemento en el sector representado por un nodo (de al menos una de

las representaciones de entrada) para determinar que el resultado también tendrá algún elemento en el mismo sector, independiente de la posición exacta de el/los elemento/s referenciado/s.

Algorithm 3.1.1 Union(A, B)

```
1: Q.Insert(⟨0, 1, 1⟩)
2: pA ← 0, pB ← 0
3: while not Q.Empty() do
4:     ⟨l, rA, rB⟩ ← Q.Delete()
5:     for i ← 0 . . . k^2 − 1 do
6:         bA ← 0, bB ← 0
7:         if rA = 1 then
8:             bA ← A[pA], pA ← pA + 1
9:         end if
10:        if rB = 1 then
11:            bB ← B[pB], pB ← pB + 1
12:        end if
13:        C ← C||(bA ∨ bB)
14:        if (l < H) ∧ (bA ∨ bB) then
15:            Q.Insert(⟨l + 1, bA, bB⟩)
16:        end if
17:    end for
18: end while
19: return C
```

Para mantener la sincronización del recorrido en ambas estructuras se considera que algunas ramas de un árbol pueden no estar definidas en el otro, por lo que los bitmaps serán de una extensión diferente. Por lo anterior es que el algoritmo de Unión hace uso de Q: una cola que almacena tripletas conformados por: el nivel que se explora (l), el bit del padre en A (rA) y el bit del padre en B (rB). Con los elementos de esta cola se sabe si, por ejemplo, el elemento que está en la posición pA del bitmap A corresponde al mismo elemento indicado por la posición pB en el bitmap B. Las situaciones posibles son: *(a)* rA y rB contienen 1, entonces los elementos de $A[pA]$ y $B[pB]$ son correspondientes, *(b)* rA contiene 1 y rB contiene 0, entonces sólo se considera $A[pA]$ para determinar el resultado en C, y $B[pB]$ no se utiliza, *(c)* rA es 0 y rB es 1, entonces no se considera $A[pA]$ y el contenido de $B[pB]$ determina el resultado en C, d si rA y rB son 0 entonces se inserta 0 en el resultado.

En el algoritmo de Unión cada tripleta presente en la cola ayuda a determinar el resultado de un nodo en el k^2-tree resultante representado en C. Para iniciar el algoritmo se inserta un tripleta con los valores $\langle 0, 1, 1\rangle$ para procesar la raíz del árbol ($l = 0, rA = 1, rB = 1$) (Línea 4). El algoritmo calcula el valor que debe ir en el resultado, lo inserta y antes de insertar un tripleta en la cola para los hijos, se debe corroborar que el nodo no esté en el último nivel (no sea una hoja) y que la unión del bit analizado sea 1 (Línea 14 del Algoritmo 3.1.1).

La Figura 3.3 muestra la matriz de adyacencia y el k^2-tree resultado de unir las relaciones binarias de las figuras 3.1 y 3.2. El orden en el que se obtiene cada nodo corresponde al recorrido $a, b, c, d, e, f, g, h, i, j, k, l, m, n, o$ se calcula en base al contenido de los k^2-tree de entrada, que también se recorren en anchura. En el ejemplo la Figura 3.1 (k^2-tree A) tiene cuatro nodos en el segundo nivel, pero el 3.2 (k^2-tree B) sólo posee tres nodos, donde los nodos b, d y e del (k^2-tree A) son correspondientes con los nodos b, c y d en el (k^2-tree B). Para esto se utiliza Q que se inicia con un tripleta de valores $\langle 0, 1, 1\rangle$ para procesar el nodo raíz, indicando el nivel que se procesa (0), que el k^2-tree A si tiene elementos en el nodo que se procesa (1), y que el k^2-tree B también

posee elementos en el nodo que se procesa (1). Luego, por cada bit del nodo en proceso se inserta un nuevo tripleta en la cola dependiendo de sus valores. Un tripleta de Q procesa k^2 bits del par de nodos, en este caso se procesan los 4 primeros bits de cada representación. El primer bit del k^2-tree A ($A[1]$) tiene 1, y el del k^2-tree B ($B[1]$) también, por lo tanto se inserta 1 en C y luego se inserta un nuevo tripleta en Q con los valores $\langle 1, 1, 1\rangle$ (siguiente nivel, A posee hijo, B posee hijo). Luego, en el siguiente bit de ambas representaciones se tiene que $A[2]$ es 1 pero $B[2]$ es 0, se añade 1 a C pero en Q se inserta $\langle 1, 1, 0\rangle$ (siguiente nivel, A posee hijo, B **no** posee hijo), los bits 3 y 4 se procede de manera análoga. Luego se extrae un nuevo tripleta desde Q para procesar el nodo b de los k^2-tree A y k^2-tree B (posiciones 5 a 8 incluyéndolas), y luego se extrae un nuevo tripleta desde Q que contiene $\langle 1, 1, 0\rangle$ donde; el nivel es 1, el k^2-tree A si posee el nodo que se analiza, y el k^2-tree B **no** posee el nodo. En este caso el resultado de C sólo se determina en base al k^2-tree A utilizando las posiciones desde la 9 a la 12 (incluyéndolas), y sin considerar el k^2-tree B. Por lo tanto, para el siguiente tripleta que se extraiga de Q se utilizarán las posiciones 13 a 16 en el k^2-tree A (posiciones que representan su nodo d) y k^2-tree A se utilizarán las posiciones 9 a 12 (que representan su nodo c). De este modo el recorrido para cualquier topografía de un k^2-tree se realiza de manera sincronizada. Cabe destacar que cuando las dos representaciones de entrada no definen un nodo, no es necesario insertar en Q el tripleta para procesarlo, por lo tanto Q nunca insertará un tripleta con los valores $\langle l, 0, 0\rangle$, con $0 \leq l < H$, siendo H el nivel máximo del árbol.

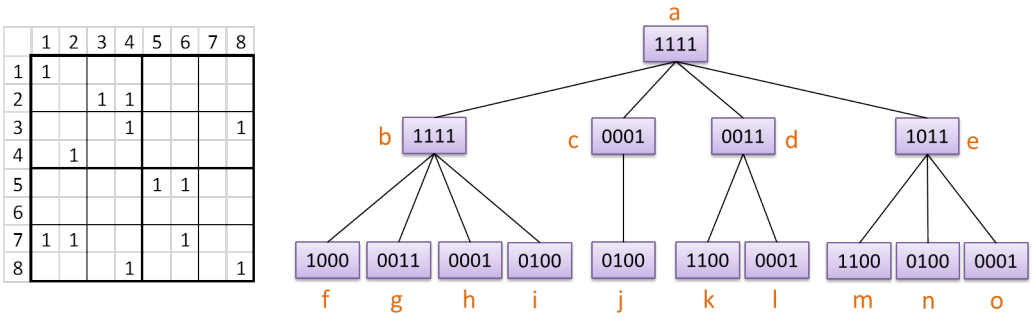


Figura 3.3: k^2-tree del resultado de la Operación de Unión sobre los k^2-tree A (Figura 3.1) y B (Figura 3.2).

3.1.2. Complemento

El Algoritmo 3.1.2 muestra el procedimiento para calcular el Complemento, la única operación unaria que se implementa, por lo que recibe como parámetro de entrada sólo una representación de k^2-tree.

Para determinar el Complemento, el Algoritmo 3.1.2 explora la representación en profundidad considerando los siguientes casos para cada bit que analiza; *Caso 1*: El bit analizado está en 1 y no es una hoja, por lo que se debe continuar la exploración con una nueva llamada recursiva (Línea 6), *Caso 2*: el bit es 1 y se está en una hoja, por lo que se cambia su valor para el resultado (Línea 8), y *Caso 3*: el valor del bit es 0 y no es una hoja, por lo que se deben generar los bits de esa rama sólo con valores 1 para lo que se utiliza la función ***FillIn*** (Línea 11). Nótese que cuando

Algorithm 3.1.2 Complement(A, l, pA)

```
1: writesomething ← 0
2: for i ← 0 ... k^2 − 1 do
3:     t[i] ← 0
4:     if (A[pA[l]] = 1) then
5:         if l < H then
6:             t[i] ← Complement(l + 1, pA) {internal nodes}
7:         else
8:             t[i] ← ∼ A[pA[l]]
9:         end if
10:    else if (l < H) then
11:        FillIn(l + 1)  {Fill in the subtree with 1 values}
12:    end if
13:    writesomething ← writesomething ∨ t[i]
14:    pA[l] ← pA[l] + 1
15: end for
16: if (writesomething = 1) then
17:    C[l] ← C[l]||t
18: end if
19: return writesomething
```

un bit es hoja y su valor es 1 no hay un caso definido explícitamente, pero se asume cubierto en la Línea 3 donde el resultado temporal t se inicia con el valor 0.

La función *FillIn* se utiliza para generar una rama (que contiene únicamente valores 1 en sus hojas) en el resultado C desde el nivel siguiente al nodo que se procesa.

El algoritmo sólo registra el resultado parcial en C si *writesomething* lo indica, retornando el mismo valor como resultado de su llamada recursiva.

La Figura 3.4 muestra la matriz de adyacencia y el k^2-tree resultado de realizar la operación de complemento sobre la relación binaria de la Figura 3.1. Para obtener este resultado el algoritmo recorre los nodos en profundidad en el orden: $a, b, f, g, h, c, i, d, j, e, k, l$ (del k^2-tree de la Figura 3.1). El algoritmo recorre los nodos, y por cada bit en un nodo considera los tres casos ya descritos. En este ejemplo, comienza revisando el primer bit del nodo a, que se encuentra en la posición 1 del bitmap A ($A[1]$), dado que el valor del bit es 1 se procede según el *Caso 1*, por lo que se explora el contenido de su hijo, el nodo b. En el nodo b se analiza el primer bit ubicado en $A[5]$ y se procede según el mismo caso. Al explorar los bits del nodo f se insertan valores $0, 1, 1, 1$ en el último nivel del bitmap C, que corresponde al nodo f de la Figura 3.4. Luego, el algoritmo retorna 1 según su línea 19 y, al volver al nodo b del k^2-tree A procesa el segundo bit, tal como lo hizo con el primero y como se hará con el tercero. Para el cuarto bit del nodo b se utiliza la función ***FillIn*** según lo indicado para el *Caso 3*, esto genera nodos completos con valores 1 en los niveles inferiores a donde se realiza la llamada, para este caso se genera el nodo i de la Figura 3.4. De este modo se genera el resultado para la operación de Complemento sobre el k^2-tree A. Nótese que el cambio de procesamiento entre los niveles que sigue el algoritmo es apoyado por el arreglo de posiciones pA, el cual es actualizado en la medida que se procesa un bit; por ejemplo al procesar el nodo f del k^2-tree A se utiliza el valor en $pA[3]$, el que en ese momento es 21, luego que se procesa el primer bit el valor cambia a 22 y así con cada bit de ese nodo según la Línea 14 del Algoritmo 3.1.2. Una vez que se procesan los k^2 bits el valor de $pA[3]$ es 24, justamente el primer bit del nodo g que será el siguiente en ser procesado. De esta forma se da que para cada nivel del k^2-tree las posiciones se almacenan en pA y son actualizadas a medida que se procesa

cada nodo.

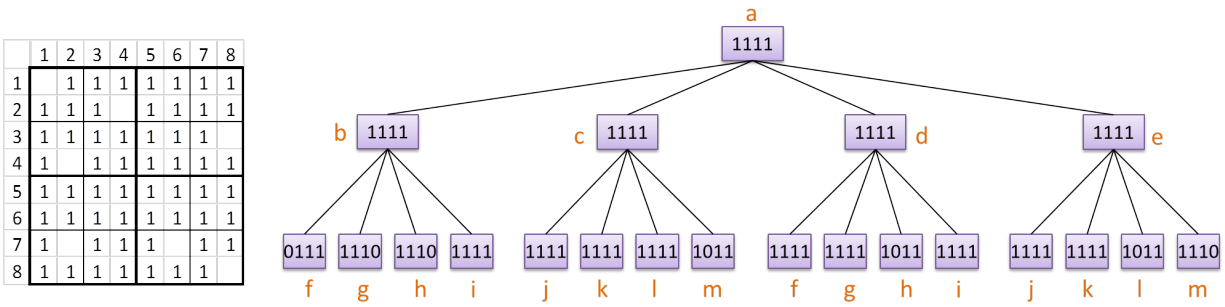


Figura 3.4: k^2-tree del resultado de la operación de Complemento sobre el k^2-tree de la Figura 3.1.

3.1.3. Diferencia

El algoritmo de Diferencia 3.1.3 se ejecuta de manera recursiva, analizando en cada recursión un nodo, y por cada nodo se analizan k^2 bits. En cada par de bits que analiza para determinar el resultado considera los siguientes casos; *Caso 1*: los bits de ambas representaciones son 1 y no pertenecen a un nodo hoja, entonces se realiza una llamada recursiva que explora la posible coincidencia en los hijos de A y B (Línea 6), *Caso 2*: ambos bits tienen valor 1 y pertenecen a un nodo hoja, entonces se añade 0 al resultado temporal (Línea 8), *Caso 3*: el bit de A es 1, el bit de B es 0 y el nivel no es el último; entonces se utiliza la función *Copy* (Algoritmo 3.1.5) para realizar una copia de la rama de A hacia el resultado C (Línea 12), *Caso 4*: cuando el bit de A es 1, el de B es 0 y el nivel es una hoja se añade el valor 1 al resultado C (Línea 14), *Caso 5*: Cuando el bit de A es 0, y el de B es 1, entonces se utiliza la función *SkipNodes* (Algoritmo 3.1.4) para descartar la rama de B. Esto último se logra actualizando los valores de pB en los niveles siguientes al nodo en proceso (Línea 17).

En la Figura 3.5 se puede ver la matriz de adyacencia y el k^2-tree resultado de la operación de Diferencia sobre las relaciones binarias de la Figura 3.1 como A y la Figura 3.2 como B. En este caso el resultado de la operación corresponde a todos aquellos elementos que están exclusivamente en la relación A. Este ejemplo primero procesa los nodos a de las relaciones A y B, donde al procesar el primer bit de cada uno se encuentra ante el *Caso 1*, por lo que se realiza una llamada recursiva para el nivel siguiente, ahí se analizan los nodos b de ambas relaciones que tienen valores 1 y 0 para A y B respectivamente, por lo que se procede según el *Caso 3*; se utiliza la función *Copy* (Algoritmo 3.1.5) que realiza una copia de la rama compuesta por el nodo f de la relación A, lo que genera el nodo e del resultado (Figura 3.5). Luego se procesa el siguiente bit de los nodos b en ambas relaciones, donde ambas contienen 1 y se procede según el *Caso 1*; ahora se procesa el nodo g de la representación A, y el nodo e de B. Las primeros dos bits de ambas relaciones son 0 por lo que el algoritmo sólo añade 0 al resultado temporal t. Para el tercer par el nodo g de A es 0 y el nodo e de B es 1, donde según el *Caso 5* se descarta la rama de B (En este caso, no existe rama bajo el nodo e de B por lo que la llamada no realiza cambios sobre el arreglos de posiciones pB). Para el último par de bits, el nodo g de A tiene valor 1 y el nodo e

Algorithm 3.1.3 Difference(A, B, l, pA, pB)

```
1: writesomething ← 0
2: for i ← 0...k^2 − 1 do
3:     t[i] ← 0
4:     if (A[pA[l]] ∧ B[pB[l]]) then
5:         if (l < H) then
6:             t[i] ← Difference(l + 1, pA, pB)) {Internal nodes}
7:         else
8:             t[i] ← A[pB[l]]∧ ∼ B[pB[l]]  {Last level}
9:         end if
10:    else if (A[pA[l]]) ∧ (∼ B[pB[l]]) then
11:        if (l < H) then
12:            t[i] = Copy(l + 1,A, pA)){copy subtree}
13:        else
14:            t[i] ← 1
15:        end if
16:    else
17:        skipNodes(B, pB, l + 1, B[pB[l]])
18:    end if
19:    writesomething ← writesomething ∨ t[i]
20:    pA[l] ← pA[l] + 1, pB[l] ← pB[l] + 1
21: end for
22: if writesomething = 1 then
23:    C[l] ← C[l]||t
24: end if
25: return writesomething
```

de B 0, por lo que según el *Caso 4* se añade el valor 1 al resultado C. Se procesan de manera análoga los bits h de A y f de B. Posteriormente, cuando se procesa el último par de los nodos b de ambas relaciones, donde A es 0 y para B es 1, se realiza la llamada a la función *SkipNodes* (Algoritmo 3.1.4) para descartar desde B la rama compuesta por el nodo g. Luego de esto, se continúa con el segundo bit de los nodos a en ambas relaciones, donde A es 1 y B es 0, por lo que según el *Caso 3* se copia la rama desde A al resultado C, donde los nodos c e i desde A generan los nodos c y g de la Figura 3.5. De esta manera se continúa hasta procesar todos los elementos de A y B generando el resultado de la Figura 3.5.

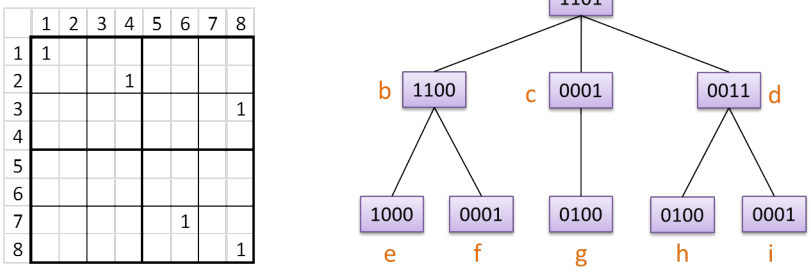


Figura 3.5: k^2-tree del resultado de la operación de Diferencia sobre los k^2-tree A (Figura 3.1) y B (Figura 3.2).

A continuación se presentan los algoritmos de *SkipNodes* (Algoritmo 3.1.4) y *Copy* (Algoritmo

3.1.5) utilizados por los algoritmos de Diferencia (Algoritmo 3.1.3), Intersección (Algoritmo 3.1.6) y Diferencia Simétrica (Algoritmo 3.1.7). Estos algoritmos fueron descritos en la Sección 3.1. La función *Copy* realiza la copia de una rama desde la relación de origen A/B hacia la relación del resultado C. Durante su ejecución actualiza los valores del arreglo pA/pB según corresponda. Por otra parte, la función *SkipNodes* actualiza las posiciones almacenadas en el arreglo pA/pB descartando la rama del árbol A/B desde el nodo siguiente al que se encuentra en proceso.

Algorithm 3.1.4 SkipNodes(X, pX, l, s)

$newpos \leftarrow pX[l] + s * k^2 - 1$
$nOnes \leftarrow rank(X, newpos) - rank(X, pX[l] - 1)$
$pX[l] \leftarrow newpos + 1$
if $(nOnes > 0) \wedge (l < H)$ **then**
  SkipNodes$(X, pX, l + 1, nOnes)$
end if

Algorithm 3.1.5 Copy(X, pX, l, s)

$end \leftarrow pX[l] + s * k^2 - 1$
$nOnes \leftarrow rank(X, end) - rank(X, pX[l] - 1)$
$C[l] \leftarrow C[l]||X[pX[l]..end]$
$pX[l] \leftarrow end + 1$
if $(nOnes > 0) \wedge (l < H)$ **then**
  Copy$(X, pX, l + 1, nOnes)$
end if

3.1.4. Intersección

El Algoritmo 3.1.6 para calcular la Intersección de dos relaciones representadas en k^2-tree realiza un recorrido en profundidad de los nodos, considerando tres casos para determinar el resultado: *Caso 1*: no se está en una hoja y ambos bits son 1, entonces se explora el siguiente nivel de las relaciones (Línea 5), *Caso 2*: no es un nodo hoja y alguno de los bits es 0, entonces se utiliza la función ***SkipNodes*** (Algoritmo 3.1.4) para actualizar las posiciones almacenadas en los arreglos pA y pB (Líneas 8 y 9). *Caso 3*: se analiza un nodo hoja, por lo que se puede calcular el resultado directamente con la operación lógica *AND* sobre los bits (Línea 12).

La Figura 3.6 muestra la matriz de adyacencia y el k^2-tree que representan la relación de resultado luego de aplicar la operación de Intersección sobre las relaciones binarias de la Figuras 3.1 (relación A) y la Figura 3.2 (relación B). En esta operación también se realiza un recorrido en profundidad de los nodos para calcular el resultado donde el ejemplo comienza con el nodo a de ambas relaciones, las que contienen 1 en el primer bit por lo que se realiza una llamada recursiva, como lo indica el *Caso 1*, en la que se continúa con el procesamiento de los nodos b en ambos k^2-tree. El primer bit del nodo b de A es 1, mientras que el primer bit del nodo b de B es 0, por tanto, según *Caso 2*, se descarta la rama en A compuesta por el nodo f mediante la función *SkipNodes* (Algoritmo 3.1.4) y se añade el valor 0 al resultado temporal t. Luego, se continúa con el segundo bit de los nodos b de A y B donde ambos son 1 (*Caso 1*), por lo que se realiza una nueva llamada recursiva para procesar los nodos g y e de A y B respectivamente. Dado que se está en el último nivel de los k^2-tree se calcula el resultado aplicando el *Caso 3*; se aplica la

Algorithm 3.1.6 Intersection(A, B, l, pA, pB)

```
1: writesomething ← 0
2: for i ← 0 ... k^2 − 1 do
3:     if l < H then
4:         if (A[pA[l]] ∧ B[pB[l]]) then
5:             t[i] ← Intersection(l + 1, pA, pB)
6:         else
7:             t[i] ← 0
8:             SkipNodes(A, pA, l + 1, A[pA[l]])
9:             SkipNodes(B, pB, l + 1, B[pB[l]])
10:        end if
11:    else
12:        t[i] ← A[pA[l]] ∧ B[pB[l]]
13:    end if
14:    writesomething ← writesomething ∨ t[i]
15:    pA[l] ← pA[l] + 1, pB[l] ← pB[l] + 1
16: end for
17: if writesomething = 1 then
18:    C[l] ← C[l] || t
19: end if
20: return writesomething
```

operación lógica AND sobre los bits. En este caso, todos los bits generados al realizar la operación AND sobre los bits resulta en valores 0 que se añaden en el resultado temporal t (0000) de esta llamada, donde al finalizar se comprueba si existen relaciones ante de insertar el resultado parcial en C (Línea 17). Dado que no hay elementos que añadir se retorna 0. Actualmente, en el nodo b de ambas relaciones se tiene 00 como resultado parcial en t. Ahora se procesa el tercer bit de cada nodo b, donde ambos son 1 y se procede nuevamente con el (*Caso 1*) donde se evaluarán los nodos h i f de las relaciones A y B respectivamente. Al evaluar estos nodos hoja de las relaciones A y B, el resultado de aplicar el *Caso 3* (Operación lógica AND) entrega como resultado parcial 0001, por lo que el resultado temporal, de esta llamada, en t se añade en el nivel correspondiente de C y se retorna 1. Luego, se procesa el cuarto bit de los nodos b en ambas representaciones y, dado que uno es 0, se procede según el *Caso 2* descartando la rama de la relación B compuesta por el nodo g. Luego el resultado temporal en esta llamada tiene 0010, lo que se añade en el bitmap C y se retorna 1 a la llamada original. Actualmente se han formado los nodos b y d de la Figura 3.6. Se procede como hasta ahora para el resto de los elementos a procesar para obtener el resultado C.

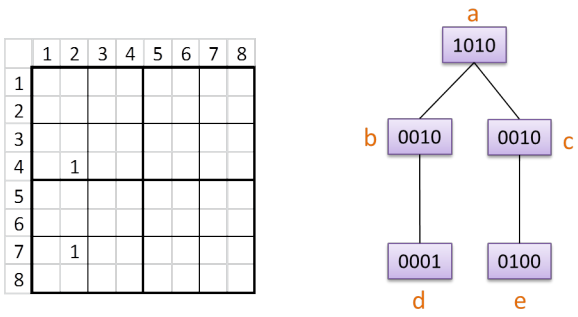


Figura 3.6: k^2-tree del resultado de la operación de Intersección sobre los k^2-tree A (Figura 3.1) y B (Figura 3.2).

3.1.5. Diferencia Simétrica

Los algoritmos propuestos por Brisaboa *et al.* [7] abordan las operaciones de Unión, Intersección, Diferencia y Complemento. Adicionalmente se ha incorporado en esta tesis el algoritmo de Diferencia Simétrica para complementar las operaciones anteriores.

El Algoritmo 3.1.7 de Diferencia Simétrica considera los mismos aspectos generales que las operaciones binarias recursivas expuestas en la sección 3.1 y así también mantiene la nomenclatura de las variables utilizadas.

En este caso se consideran las siguiente situaciones para resolver la operación de Diferencia Simétrica: *Caso 1*: si ambos bits son 1 y no se está en un nodo hoja, se debe explorar el siguiente nivel mediante una llamada recursiva (Línea 6), *Caso 2*: el bit de sólo una representación es 1 y no se está en un nodo hoja, se copia la rama correspondiente con la función ***Copy*** (Algoritmo 3.1.5) (Líneas 10 y 15), *Caso 3*: sólo uno de los bits es 1 y se está en el nivel hoja, se inserta 1 en el resultado parcial (Líneas 12 y 17). El caso en que ambos bits contienen el mismo valor está considerado al comienzo del algoritmo, ya que se inician los valores del resultado parcial con 0 (Línea 3).

La Figura 3.7 muestra la matriz de adyacencia y el k^2-tree resultado de realizar la operación de Diferencia Simétrica sobre las relaciones binarias de las figuras 3.1 (relación A) y 3.2 (relación B). El proceso se inicia con los nodos a de ambas relaciones, donde se analiza el primer bit que en ambos casos es 1, por lo que se procede según el *Caso 1* con una llamada recursiva que continuará con el análisis de los nodos b de las dos relaciones. En el primer bit del nodo b se tiene 1 en A y 0 en B por lo que, según el *Caso 2* se realiza la copia de la rama compuesta por el nodo f de A hacia el resultado en C. Luego se procede con el segundo bit del nodo b donde ambas relaciones tienen valor 1, por lo que se hace una nueva llamada recursiva para procesar los nodos g de A y e de B. En los dos primeros casos, los bits de éstos nodos corresponden a valores 0 por lo que el algoritmo únicamente añade 00 al resultado parcial t, luego en el tercer bit del nodo g en A es 0 y en el nodo e de B es 1, por lo que según el *Caso 3* se inserta 1 en el resultado parcial. Luego, con el cuarto bit de los mismos nodos también se procede según el *Caso 3*, con lo que t almacena 0011, lo que luego forma el nodo g del resultado C en la Figura 3.7. Posteriormente se procesa el tercer bit de los nodos b, por lo que el algoritmo indica que se debe procesar el nodo h

Algorithm 3.1.7 SymmetricDifference(A, B, l, pA, pB)

```
1: writesomething ← 0
2: for i ← 0...k^2 − 1 do
3:     t[i] ← 0
4:     if (A[pA[l]] ∧ B[pB[l]]) then
5:         if l < H then
6:             t[i] ← SymmetricDifference(l + 1, pA, pB)) {Internal nodes}
7:         end if
8:     else if (A[pA[l]]) then
9:         if l < H then
10:            Copy(l + 1,A, pA)  {copy subtree}
11:        end if
12:        t[i] ← 1
13:    else if (B[pB[l]]) then
14:        if l < H then
15:            Copy(l + 1,B, pB)  {copy subtree}
16:        end if
17:        t[i] ← 1
18:    end if
19:    writesomething ← writesomething ∨ t[i]
20:    pA[l] ← pA[l] + 1, pB[l] ← pB[l] + 1
21: end for
22: if (writesomething = 1) then
23:    C[l] ← C[l]||t
24: end if
25: return writesomething
```

de A con el nodo f de B: estos nodos son coincidentes, y según el algoritmo se inserta 0000 en t, ya que todos los bits son coincidentes en estos nodos, esto no genera registros en C y se retorna 0. Al momento de procesar el cuarto bit de los nodos b se procede según el *Caso 2* utilizando la función *Copy* (Algoritmo 15) con lo que el nodo g de B se copia hacia el resultado formando el nodo h en C. Luego de esto el resultado temporal en la llamada que procesa los nodos b contiene 1101, lo que pasa a formar el nodo b, y se procede de la misma forma con el procesamiento del segundo bit de los nodos a, y demás elementos.

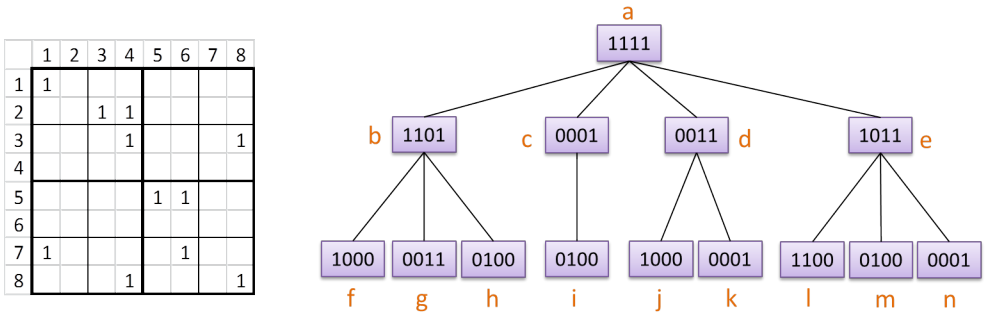


Figura 3.7: k^2-tree del resultado de la operación de Diferencia Simétrica sobre los k^2-tree A (Figura 3.1) y B (Figura 3.2).

Parte III

Extensión de las Capacidades en Estructuras de Datos Compactas

Capítulo 4

Implementación de Operaciones de Conjunto sobre el k^2-tree

En este capítulo se describen aspectos de la implementación y los resultados de la experimentación sobre los algoritmos descritos en el Capítulo 3 de operaciones de conjunto sobre relaciones binarias en la EDC k^2-tree.

4.1. Sobre la Implementación

La implementación de todos los algoritmos se realizó sobre el lenguaje de programación C, utilizando el compilador gcc v4.4.1.

En general existen dos estrategias para resolver las operaciones de conjunto en los algoritmos presentados. La primera es mediante un recorrido en anchura (por niveles) utilizado por el algoritmo que calcula la *Unión*. La segunda estrategia es un recorrido en profundidad utilizado por las operaciones de *Diferencia*, *Intersección*, *Diferencia Simétrica* y *Complemento*, de la cual se detallarán algunos aspectos a continuación.

4.1.1. Sobre el Recorrido en Anchura

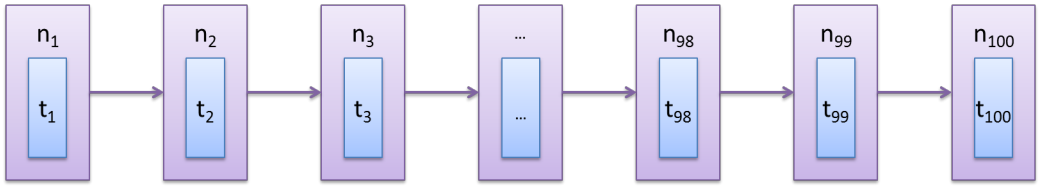
El recorrido en anchura, utilizado por la operación de Unión, requiere del uso de una cola que almacena tripletas de información con la forma $\langle rA, rB, l\rangle$, donde l es el nivel que se explora, rA corresponde al valor del bit padre para el nodo de A que se procesa y rB es el valor del bit padre que representa el nodo de B que se procesa. La forma en que se utiliza esta información se detalla en la Sección 3.1.1.

La cola necesaria para la operación de Unión, almacenará elementos que requieren de poca memoria; los valores rA y rB requieren de 1 bit cada uno, y en la implementación se dejaron seis bits, con lo que es posible representar números en entre el 0 y el 63 incluyéndolos. Nótese que con un k^2-tree de 10 niveles, y $k = 2$ es posible representar una relación binaria de 1.024 objetos, es decir 1.048.576 posibles relaciones. Lo anterior indica que cada tripleta de información será contenida en ocho bits. La desventaja que presenta la cola es que, aún cuando las unidades que

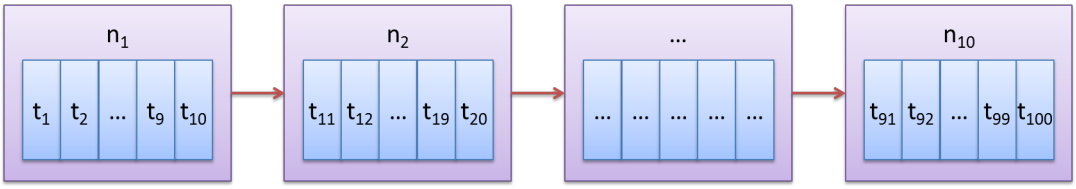
almacenará requieren poco espacio de memoria, la cantidad de elementos que serán contenidos corresponden a un gran volumen. En el caso de una representación que contenga muchos elementos, por ejemplo, para un k^2-tree de 10 niveles podrían existir, en el peor de los casos, 262.144 elementos contenidos en la cola.

Es por lo anterior que la cola debe ser eficiente en términos de espacio. Si en la implementación de esta cola, se considerase que un nodo corresponde a una tripleta de información, la cola podría requerir más espacio de almacenamiento en las referencias de memoria de lo que se necesita para almacenar las tripletas. Para solucionar esto es que se ha diseñado una cola en la que se considera cada nodo como un conjunto de tripletas almacenadas en un arreglo unidimensional. Para la experimentación se almacenaron 100 tripletas en cada nodo de la cola.

Lo anterior se puede ver en la Figura 4.1, donde se muestra una cola que posee una tripleta por cada nodo (Figura 4.1(a)) la que requieren 99 referencias entre los nodos al contener 100 tripletas, y por otra parte la Figura 4.1(b) presenta una cola en la que cada nodo posee 10 tripletas, donde sólo se requieren 9 referencias entre ellos.



(a) Cola con un tripleta por nodo



(b) Cola con 10 tripletas por nodo

Figura 4.1: Colas para almacenar tripletas de información.

4.1.2. Sobre el Recorrido en Profundidad

En las operaciones de *Diferencia*, *Intersección*, *Diferencia Simétrica* y *Complemento* son necesarios los valores contenidos en las hojas para determinar el contenido de los nodos superiores. De hecho, para conocer qué resultado tendrá el primer bit del nodo raíz, en cualquiera de estas operaciones, se requiere conocer los valores que contienen las hojas que descienden de ese nodo.

Para el k^2-tree un recorrido en profundidad realiza repetidas operaciones de *rank* y *select* sobre el bitmap de la EDC, es por lo anterior que se hace uso de un arreglo que contiene las posiciones donde inicia cada nivel de un k^2-tree. Esto implica calcular la posición del primer elemento de cada nivel sólo una vez.

Este arreglo de posiciones permite procesar los nodos de manera organizada realizando un incremento de la posición una vez que ha sido utilizado el contenido del bit al que apunta dicha

posición. Suponiendo que se está en un nivel i la casilla i-ésima apunta la posición que actualmente se procesa en el bitmap, y si esa posición contiene un valor 1, la posición de su hijo izquierdo estará en la casilla $i+1$ del arreglo de posiciones. Para procesar todos los hijos del nodo se deben revisar las k^2 casillas contiguas del nivel $i+1$ lo cual se logra realizando incrementos de la posición almacenada en la casilla $i+1$ de arreglo.

Por lo anterior, una vez que han sido procesados todos los hijos de un nodo x en el nivel i, la casilla $i+1$ del arreglo de posiciones está apuntando al hijo izquierdo del hermano derecho del nodo x, lo que se ve en la Figura 4.2.

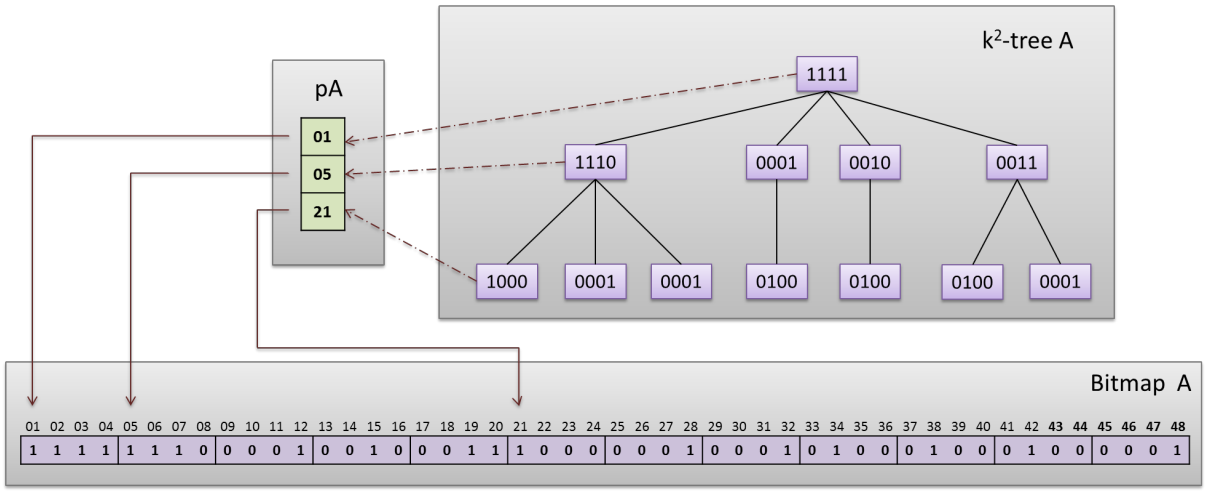


Figura 4.2: Ejemplo de k^2-tree (desde Figura 3.1), su bitmap y el arreglo de posiciones correspondiente.

La estrategia de recorrido en profundidad genera el resultado desde los niveles inferiores, ya que para determinar el resultado de un elemento en el nivel i, es necesario conocer el resultado de sus hijos en el nivel $i+1$. Es por ello que esta estrategia genera de manera progresiva el resultado en todos los niveles, y se hace necesario almacenar estos resultados parciales en un arreglo de bits por cada nivel.

Lo anterior puede ser implementado mediante un arreglo de bitmaps, que solicitan memoria adicional en la medida que lo requieran mientras se calcula el resultado de una operación mediante la estrategia en profundidad. Esta solución tiene un riesgo frente a resultados extensos: suponiendo que una vez finalizada la operación los arreglos utilizan más de la mitad de la memoria que hay disponible para el resultado no se podrá pedir la memoria suficiente para guardar las soluciones de todos los niveles concatenados, lo que implicaría invertir tiempo de procesamiento en operaciones que por limitaciones de memoria no generarán resultados.

Ante tal caso es que se ha decidido implementar una estructura de bitmaps que dispone los resultados de cada nivel de manera contigua considerando espacio suficiente para el peor de los casos en cada nivel. Una vez finalizada la operación los resultados de cada nivel son desplazados hacia la izquierda para formar el bitmap que representa el k^2-tree y liberar la memoria adicional que fue reservada. Este proceso se grafica en la Figura 4.3. Lo anterior permite que una operación

sólo se ejecute si existe memoria suficiente para el peor de sus casos evitando invertir tiempo de ejecución en operaciones que no obtendrán resultados.

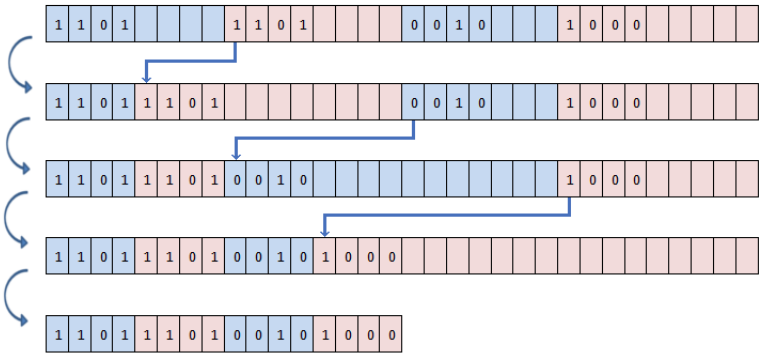


Figura 4.3: Secuencia de concatenación para la implementación del bitmap por niveles.

Para esta estructura de bitmaps fue necesario definir el peor de los casos para cada operación, que se especifica a continuación de la forma *Operación(A, B)*, donde A y B son las representaciones de entrada para realizar *Operación*:

- **Unión(A,B)**: en el peor de los casos la longitud del bitmap resultante será tan grande como la suma de las longitudes de los bitmaps de A y B, siempre que esta suma no sea mayor que una representación completa de k^2-tree. (Una representación completa se refiere a un k^2-tree que no compacta ningún sector de la relación de objetos que representa). A pesar que esta operación no genera los resultados de manera parcial, ya que realiza un recorrido por niveles y no uno en profundidad, también hace uso de esta estructura de bitmap considerando sólo un nivel lo suficientemente extenso para almacenar el peor caso descrito.

- **Diferencia(A,B)**: se debe reservar memoria necesaria para la longitud de A, ya que se podría dar que los elementos de A y B no coincidan, por lo que no se sustraerían elementos desde A en la operación.

- **Intersección(A,B)**: se debe reservar el espacio de la menor entre las longitudes de A y B, porque se puede dar que la menor representación sea un subconjunto de la otra y ante tal caso el resultado sería tan extenso como la representación más pequeña.

- **Diferencia Simétrica(A,B)**: como en la unión, se reserva la suma de las longitudes de A y B, y siempre que no sea mayor que una representación completa. Esto ya que la intersección de A y B podría ser vacía y ambas añaden elementos al resultado.

- **Complemento(A)**: Se reserva el espacio necesario para una representación completa.

Este modo de abordar el problema también presenta una desventaja: se descartan soluciones que podrían ser almacenadas en memoria porque no requieren la memoria que se estima con la

peor situación. Esta situación no se puede identificar de manera temprana ya que no es posible determinar cuándo existe suficiente memoria para almacenar un resultado sin explorar su solución.

4.2. Experimentación

En esta sección se describe la evaluación experimental realizada sobre la implementación de los algoritmos antes descritos en el k^2-tree. A continuación se describen, en primer lugar, los datos utilizados para realizar los experimentos, y luego, los diferentes procedimientos considerados para obtener los datos del tiempo de ejecución que se exponen en la siguiente sección.

4.2.1. Datos Experimentales

Los datos utilizados para realizar las pruebas formales de los algoritmos se dividen en dos conjuntos reales, los cuales son `land-use` y `uk-snaps`.

land-use

El conjunto de datos `land-use` corresponde a una relación binaria de coordenadas geográficas sobre la destinación de suelos en la comuna de Carahue, de la Región de la Araucanía, durante los años 1986 y 2001. El tamaño de la grilla es de 7.665 columnas $\times$ 4.085 filas, donde cada relación representa una resolución de 5,7 m^2. En la Figura 4.4 se pueden ver dos ejemplos de la distribución de relaciones que presenta este conjunto de datos.

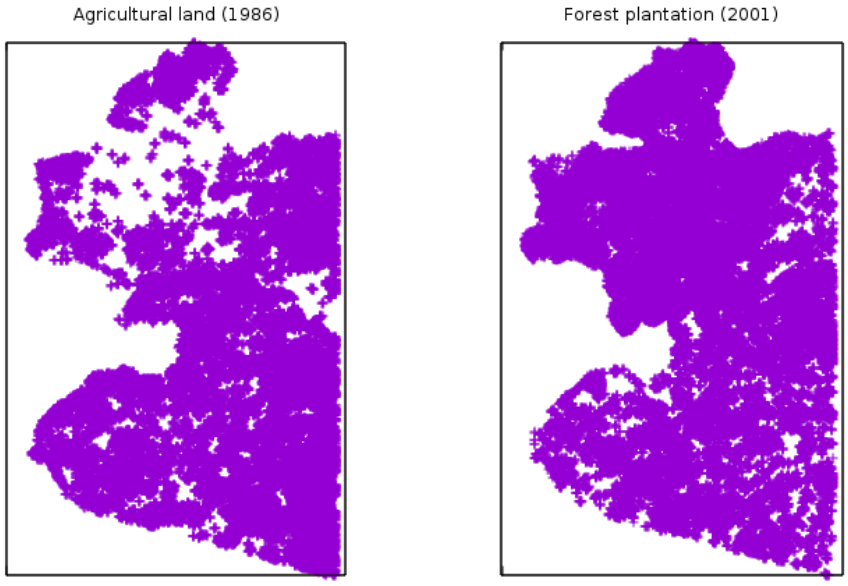
Este conjunto de datos posee 5 categorías por cada año: agricultura, bosque nativo, matorrales, plantación forestal y suelo desnudo. Por cada combinación categoría-año existe un fichero k^2-tree que lo representa, obteniéndose en total 10 ficheros para realizar las pruebas sobre los datos de uso de suelo.

En la Tabla 4.1 se describen algunas de las propiedades que presentan estos ficheros como: la cantidad de relaciones, la densidad de las relaciones, el tamaño del fichero en formato k^2-tree (*kt*) y el tamaño del fichero en formato de lista de adyacencia (*list*). Los nombres de los ficheros tienen un formato de dos letras seguidas por el año, donde las letras corresponden a un indicio de la categoría que representa.

El valor m de la Tabla 4.1 corresponde a la cantidad de relaciones que existen en la representación, en el caso de `land-use` se refiere a los puntos activos[1] en la categoría y año correspondiente. La densidad se define como el porcentaje de puntos activos m sobre la cantidad de puntos posibles en la matriz (n^2). Para el conjunto de datos `land-use` se considera como puntos posibles el valor 31.311.525 (7.665$\times$4.085).

Los tamaños de los ficheros se encuentran registrados en bytes y corresponden al espacio utilizado en memoria secundaria por los ficheros que almacenan la información según el formato que corresponda. El formato de fichero *kt* almacena; el bitmap con la información requerida para realizar operaciones de rank y select sobre el mismo, el nivel máximo del árbol, el tamaño del bitmap, la posición donde inician las hojas del k^2-tree en el árbol y algunos otros. El formato de fichero *list* considera un fichero binario que almacena: el número de objetos a relacionar (en el

[1]Se considera como punto activo una casilla de la grilla/mosaico que presenta la característica de interés.

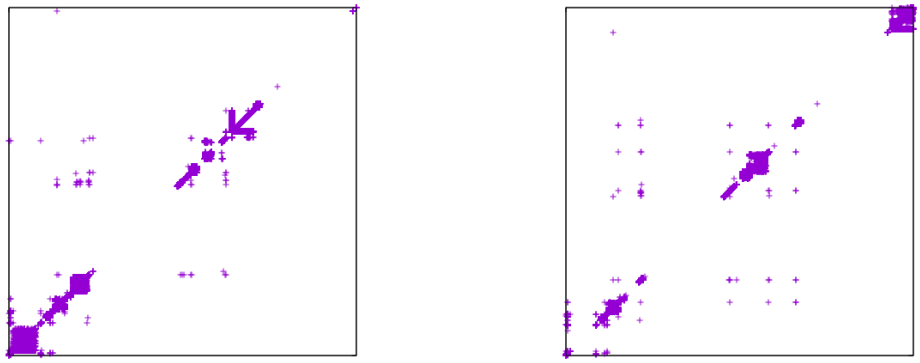


(a) `ag_1986` (b) `fp_2001`

Figura 4.4: Ejemplo de dos distribuciones con el conjunto de datos de `land-use`.

	cantidad m	densidad $m/n^2 \times 100$	list (bytes)	kt (bytes)
`ag_1986`	4.423.600	14,13 %	17.725.072	818.544
`ag_2001`	3.437.825	10,98 %	13.781.972	626.596
`nf_1986`	5.985.825	19,12 %	23.973.972	1.073.284
`nf_2001`	4.026.725	12,86 %	16.137.572	762.280
`sc_1986`	2.640.775	8,43 %	10.593.772	510.160
`sc_2001`	2.861.100	9,14 %	11.475.072	553.516
`fp_1986`	2.602.625	8,31 %	10.441.172	491.984
`fp_2001`	5.683.000	18,15 %	22.762.672	1.044.488
`bg_1986`	1.168.925	3,73 %	4.706.372	226.680
`bg_2001`	813.100	2,60 %	3.283.072	159.712

Tabla 4.1: Propiedades del conjunto de datos `land-use`,



(a) `snap0` (b) `snap11`

Figura 4.5: Ejemplo de dos distribuciones con el conjunto de datos de `uk-snaps`.

caso `land-use` almacena el valor mayor, 7.665) en un entero de 4 bytes, el número de relaciones entre objetos existentes en un entero de 8 bytes, y por cada objeto, el número que lo identifica y el número que identifica cada uno de los objetos con los que se relaciona (en enteros de 4 bytes cada uno).

En el conjunto de datos `land-use` se aprecia que el formato *kt* utiliza entre el 8,75 % al 14,56 % de espacio requerido por el formato *list*.

uk-snaps

El conjunto de datos de `uk-snaps` se compone de 12 capturas del grafo web de United Kingdom recogidas por el Laboratorio para Algoritmos Web[2] [6, 4, 5].

Originalmente cada snap consideraba 133,6 millones de nodos y 5,5 miles de millones de vínculos, ocupando más de 22 Gigabytes de espacio de almacenamiento en su representación como lista de adyacencia (de enteros de 4 bytes). El conjunto de datos que se considera en esta etapa de pruebas contiene 1 millón de objetos por cada uno de los 12 snaps. Esto con el fin de ajustarse a las restricciones de memoria al momento de cargar los datos desde las listas de adyacencia. En la Figura 4.5 se muestra una distribución de dos ejemplares del conjunto de datos `uk-snaps`.

El grafo web representa la presencia de vínculos desde una página a otra, en aquellas páginas con dominio *.uk*. En este caso, dado que páginas de un sub-dominio tenderán a presentar vínculos hacia otras páginas del mismo sub-dominio se puede apreciar en la Figura 4.5 una proximidad entre los elementos existentes.

La Tabla 4.2 presenta información sobre los ficheros que almacenan el conjunto de datos `uk-snaps`.

Cabe destacar que la densidad en este caso se calcula con la cantidad m (cantidad de elementos relacionados en la representación) contra el total posible de puntos, que en este caso es 1 billón (1

[2] http://law.di.unimi.it

	cantidad m	densidad $m/n^2 \times 100$	list (bytes)	kt (bytes)
`snap00`	2.404.620	0,00024 %	13.618.492	1.462.364
`snap01`	1.360.019	0,00014 %	9.440.088	855.900
`snap02`	3.200.913	0,00032 %	16.803.664	2.129.416
`snap03`	3.395.695	0,00034 %	17.582.792	2.559.828
`snap04`	2.350.289	0,00024 %	13.401.168	1.721.128
`snap05`	2.526.547	0,00025 %	14.106.200	1.701.508
`snap06`	1.507.033	0,00015 %	10.028.144	889.680
`snap07`	1.950.981	0,00020 %	11.803.936	1.089.980
`snap08`	1.772.026	0,00018 %	11.088.116	1.027.872
`snap09`	1.767.230	0,00018 %	11.068.932	968.788
`snap10`	2.665.612	0,00027 %	14.662.460	1.504.804
`snap11`	1.989.568	0,00020 %	11.958.284	1.117.508

Tabla 4.2: Propiedades del conjunto de datos `uk-snaps`.

millón de nodos × 1 millón de nodos) de elementos posibles. Lo anterior también involucra que el porcentaje de elementos relacionado sea considerablemente bajo comparado con el conjunto de datos `land-use`.

4.2.2. Líneas de Comparación (Baselines)

A continuación se describen los procesos desde los que se registraron los tiempos de ejecución utilizados en la sección de resultados.

Para la experimentación se define una línea de comparación para las operaciones implementadas: descompactar los datos desde ficheros en formato k^2-tree a listas de adyacencia, obtener el resultado desde las listas de adyacencia y compactar el resultado a un formato k^2-tree. Adicionalmente se incluye en los resultados el tiempo que tarda realizar las operaciones sólo sobre listas de adyacencia, pero esto último no se considera una línea de comparación del trabajo realizado.

kt - list - kt

Para la línea de comparación *kt-list-kt* se asume que los datos están almacenados en un fichero en formato k^2-tree cuya implementación no dispone de las operaciones de conjunto, pero que sí permite descompactar la información a formato de listas de adyacencia, donde se puede obtener el resultado de la operación en formato de lista de adyacencia y compactar este resultado a formato k^2-tree.

Respecto del tiempo considerado para este proceso, se inicia una vez que la estructura k^2-tree es cargada en memoria principal, y se finaliza cuando el proceso ha compactado el resultado de la operación hacia el formato k^2-tree. Vale la pena decir que no se consideran en los resultados los tiempos de carga y escritura sobre la memoria secundaria, sólo se toma el tiempo de la operación de descompactación desde k^2-tree a lista de adyacencia, la operación sobre listas de adyacencia

y la compactación del resultado como lista de adyacencia a k^2-tree.

Esta línea de comparación tiene como propósito probar que la implementación de los algoritmos propuestos son mejor opción que realizar el proceso de descompactación y compactación antes descrito para obtener resultados de operaciones de conjunto sobre información representada por un k^2-tree.

list

Este ítem incluido en las gráficas de resultados considera realizar la operación de conjunto directamente desde y hacia listas de adyacencia, sin utilizar EDCs.

Cabe destacar que este ítem no está incluido para establecer una comparación sobre la implementación del k^2-tree, sino que se ha generado como una referencia para apreciar el tiempo que toma el proceso de compactación y descompactación de la línea de comparación *kt-list-kt*. Esto debido a que se asume que el tiempo que tarda la operación será significativamente menor que el tiempo requerido para ir desde k^2-tree a lista de adyacencia y desde lista de adyacencia a k^2-tree.

4.2.3. Entorno de Pruebas

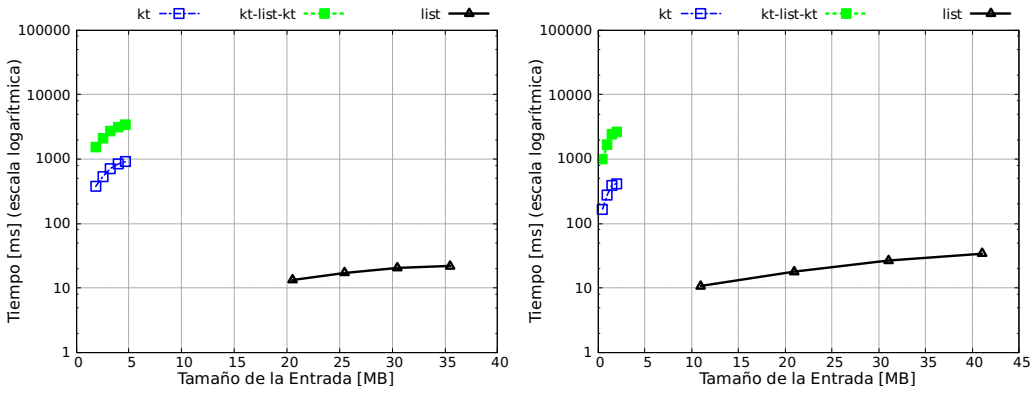
Las pruebas de los algoritmos implementados en este capítulo fueron ejecutadas en un equipo con procesador Intel Xeon E5520 de 2,27GHz, con 8MB de memoria caché. La RAM fue de tipo DDR3 con una velocidad de 800MHz y 72GB de capacidad. El servidor posee una instalación de Ubuntu 9.10 (Karmic) de 64 bits. Los algoritmos fueron implementados sobre el lenguaje de programación C y compilados con gcc 4.4.1.

En la toma de tiempos, como ya se mencionó antes, se midió cada operación sin considerar los tiempos de lectura/escritura en memoria secundaria, sólo se tomó el tiempo relativo a la transformación desde/hacia *list*/*kt* y el tiempo de operación una vez que las representaciones estuvieron cargadas en memoria principal. Más específicamente los tiempos que se muestran en la sección de resultados consideran tiempo de CPU utilizando el atributo *tms_utime* de la estructura *tms* utilizado por la función *times() POSIX*, que entrega como respuesta el tiempo de procesos del usuario medido en ticks de CPU, los que fueron transformados a milisegundos.

Por cada operación realizada, el tiempo registrado corresponde al promedio de 20 repeticiones de la misma operación.

4.3. Resultados

A continuación se presentan, mediante gráficas, los resultados obtenidos al realizar los procesos descritos en las líneas de comparación haciendo uso de los conjuntos de datos `land-use` y `uk-snaps`. Para esta sección se consideran diferentes criterios al momento de organizar los resultados: *Tamaño de la Entrada* que considera el espacio de almacenamiento utilizado por el formato que representa las relaciones binarias, *Densidad de Entrada* que corresponde al número de relaciones que poseen las relaciones de entrada, *Densidad de Salida* que definimos como el números de relaciones del resultado y *Compresibilidad de Entrada* donde se considera un factor para evaluar la compresión de las relaciones.



(a) Unión en conjunto de datos `uk-snaps` (b) Unión en conjunto de datos `land-use`

Figura 4.6: Gráficas del tiempo para la operación de Unión según Tamaño de la Entrada. Se incluyen las listas.

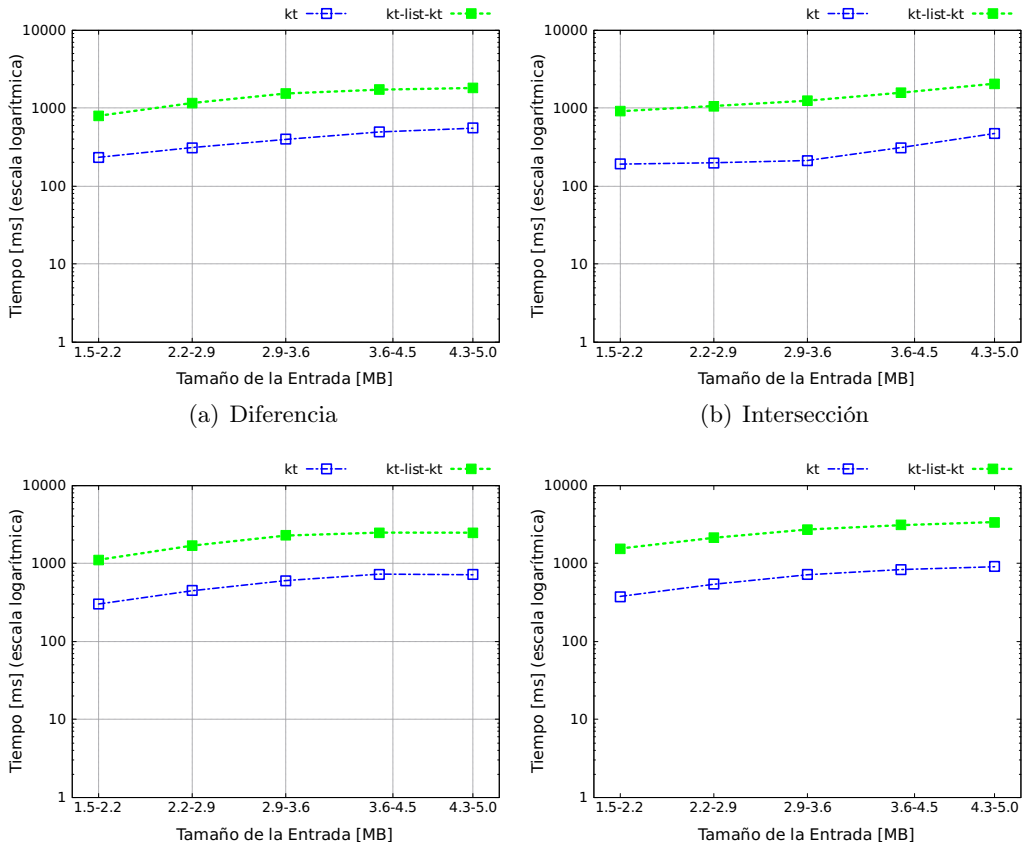
En el conjunto de datos `uk-snaps` se presentan resultados para las operaciones de *Diferencia*, *Intersección*, *Diferencia Simétrica* y *Unión*. Para la operación de *Complemento* no existen resultados debido a que en el conjunto de datos `uk-snaps` la operación requiere de mucha memoria para realizar la operación. Este conjunto de datos posee una gran cantidad de objetos y poca densidad en las relación como indica la información de la Tabla 4.2, por lo que la operación de *Complemento* genera un resultado con una alta densidad en el número de relaciones. Es por lo anterior que, para el conjunto de datos `uk-snaps` no se obtuvieron resultados para ninguno de los dos formatos (*list* y *kt*). Es relevante mencionar que la versión del k^2-tree que se utiliza en esta tesis de magíster es la que realiza una compactación en aquellas zonas que sólo contienen 0s (ceros), por lo tanto una situación en la que predomina la existencia de 1s (unos) es poco eficiente en compactación (y a su vez, en el uso de memoria), pero sí existe una versión del k^2-tree que compacta zonas que sólo contienen 0s y también aquellas zonas que sólo contienen 1s.

En el conjunto de datos `land-use` se muestran resultados para las operaciones de Diferencia, Intersección, Unión y Complemento.

4.3.1. Tamaño de la Entrada

Para este criterio se agrupan los tiempos obtenidos según el tamaño de entrada para las operaciones. El tamaño de la Entrada se considera como la suma de los tamaños de cada fichero que interviene en la operación. Para la línea de comparación *kt-list-kt* se utiliza el tamaño de los ficheros k^2-tree.

La Figura 4.6 muestra, para cada conjunto de datos, la gráfica que incluye los tiempos de las listas de adyacencia. En ambos casos se tiene que el rango de tamaño de los ficheros que almacenan las listas de adyacencia es superior al rango de los ficheros que contienen los k^2-tree en los conjunto de datos `uk-snaps` y `land-use`. Las demás gráficas que se presentan para el criterio de Tamaño de la Entrada no incluyen la listas de adyacencia con la finalidad de facilitar la lectura de los resultados en las figuras, por lo que sólo se consideran las líneas de comparación



(a) Diferencia

(b) Intersección

(c) Diferencia Simétrica

(d) Unión

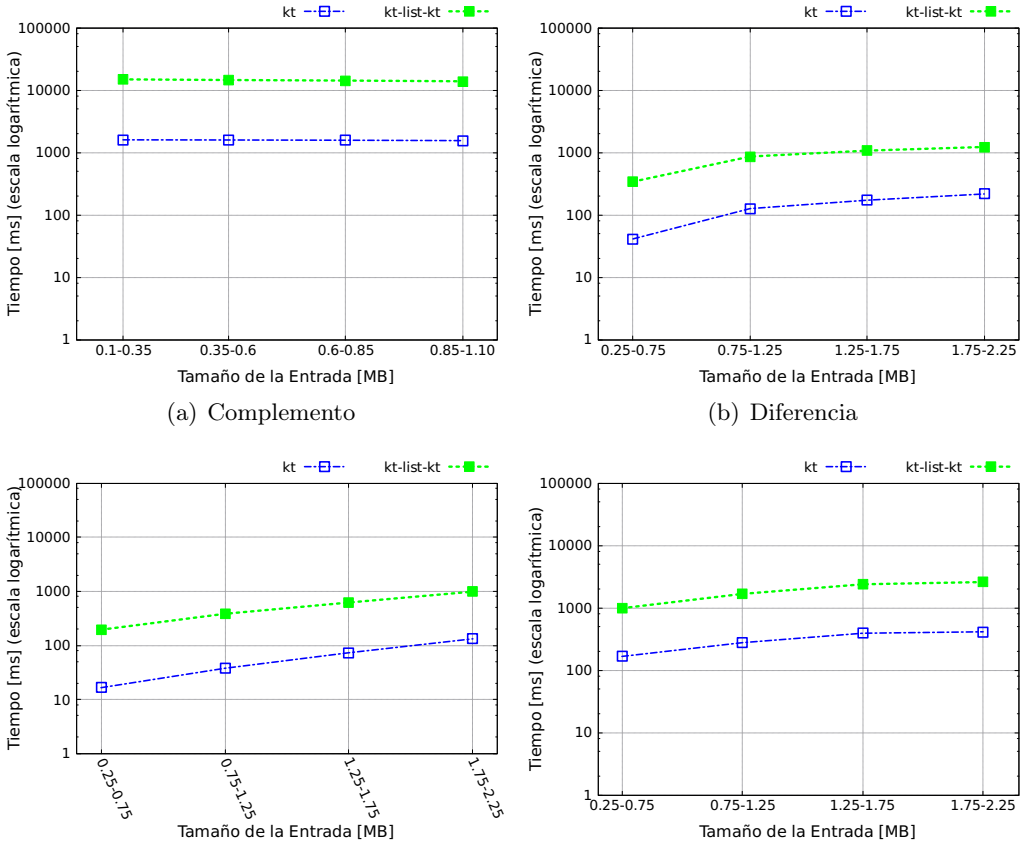
Figura 4.7: Tiempos obtenidos en diferentes operaciones sobre el conjunto de datos `uk-snaps`. Se utiliza el criterio de Tamaño de la Entrada en el eje X.

kt y *kt-list-kt*.

En la Figura 4.6 se ve que para los mismos datos, todos los casos incrementan sus tiempos en la medida que los ficheros son más grandes. Hay que tener en cuenta que el rango de tamaños en ficheros de la línea de comparación *list* considera el mismo conjunto de datos que el rango de tamaño para la línea de comparación *kt*, y es por la compresión de estos últimos que su rango es menor.

En la Figura 4.7 se presentan los tiempos de las operaciones de Diferencia, Intersección, Diferencia Simétrica y Unión de las líneas de comparación *kt* y *kt-list-kt* en el conjunto de datos `uk-snaps`. Se puede ver en todas las operaciones que los tiempos tienen casi un orden de magnitud de diferencia favorable a *kt*. Para ambos casos existe una tendencia a aumentar el tiempo promedio de las operaciones a medida que crece el tamaño de los ficheros de entrada. Así también se puede apreciar una similitud en el comportamiento de ambas líneas de comparación, y que el tiempo empleado por las operaciones está entre los 110 y los 1.200 ms.

La Figura 4.8 muestra los tiempos de las operaciones de Complemento, Diferencia, Intersección



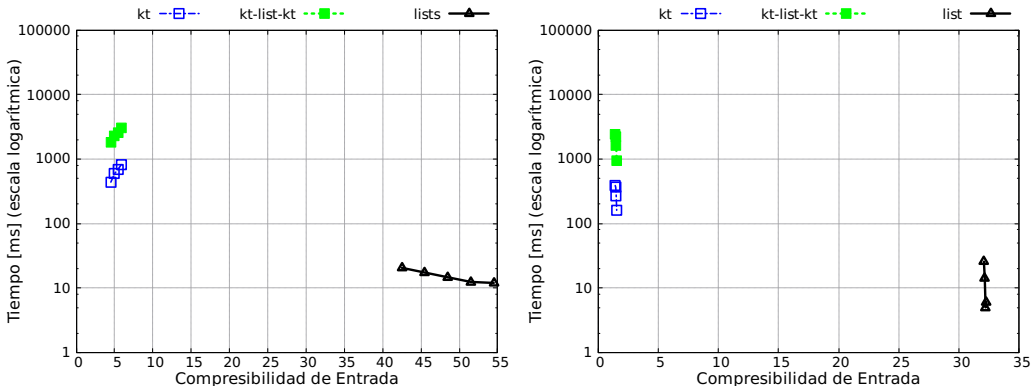
(a) Complemento

(b) Diferencia

(c) Intersección

(d) Unión

Figura 4.8: Tiempos obtenidos en diferentes operaciones sobre el conjunto de datos `land-use`. Se utiliza el criterio de Tamaño de la Entrada en el eje X.



(a) Unión en conjunto de datos `uk-snaps` (b) Unión en conjunto de datos `land-use`

Figura 4.9: Gráficas que incluyen el tiempo de las listas de adyacencia con el criterio de Compresibilidad de Entrada.

y Unión para el conjunto de datos de `land-use` en los criterios *kt* y *kt-list-kt*. Se puede apreciar que para la operación de Complemento existe una tendencia a disminuir el tiempo de ejecución en la medida que aumenta el tamaño de los ficheros de entrada. En las demás operaciones existe un incremento del tiempo a medida que crece el tamaño de los ficheros de entrada. La operación de Intersección es la que muestra una mayor pendiente en la gráfica, que se mantiene. La operación de Diferencia tiene una pendiente similar en el primer segmento de la gráfica, en los segmentos restantes la pendiente disminuye y se mantiene. Para las cuatro operaciones existe cerca de un orden de magnitud de diferencia en el tiempo empleado, donde la línea de comparación *kt* muestra mejor resultado que el *kt-list-kt*.

4.3.2. Compresibilidad de Entrada

En esta sección se utiliza como criterio de evaluación la Compresibilidad en los elementos de entrada de la operación, por medio del factor *Bits Por Relación* (BPR). Considerando $R_{espacio}$ como el tamaño del fichero que contiene la representación A, y $R_{relaciones}$ el número de relaciones existentes en la representación A, el factor BPR en una operación que tiene como entrada las representaciones A y B se calcula: $\frac{A_{espacio}+B_{espacio}}{A_{relaciones}+B_{relaciones}}$.

Nótese que el factor representa un mejor resultado mientras menor sea el valor (en cuanto a compresión respecta). A medida que se tiene un mayor factor de Compresibilidad de Entrada se incrementa el espacio requerido por cada vínculo en una representación.

La Figura 4.9 presenta gráficas que incluyen los tiempos de: *kt*, *kt-list-kt* y *list* donde se compara el tiempo empleado para ejecutar una operación contra la Compresibilidad de Entrada de la operación ejecutada. En estas gráficas se aprecia la diferencia que existe en los rangos de compresión de *list* comparadas con los rangos de compresión de *kt* y *kt-list-kt* para los conjunto de datos `land-use` y `uk-snaps`. Es por lo anterior, y tal como en la sección previa, que se han omitido en las demás gráficas del criterio Compresibilidad de Entrada los tiempos para las ejecuciones de *list*.

Recalcamos nuevamente que la comparación contra listas de adyacencia no es el foco de estas gráficas, sino que se incluye para tener en cuenta que el proceso de compresión y descompresión son significativamente importantes en los resultados que se aprecian para los tiempos de *kt-list-kt*.

La Figura 4.10 muestra los tiempos agrupados para el conjunto de datos `uk-snaps` en las operaciones de Diferencia, Intersección, Diferencia Simétrica y Unión. En esta figura se puede notar que la ejecución de las operaciones tiende a mantener la diferencia de tiempo entre ambos casos (*kt* y *kt-list-kt*), el cual se aproxima a un orden de magnitud favorable a *kt*. En general, para las cuatro operaciones el tiempo aumenta a medida que la Compresibilidad de Entrada incrementa.

En la Figura 4.11 se muestran los tiempos para las operaciones de Diferencia, Intersección, Unión y Complemento para el conjunto de datos `land-use`. Tal como el caso anterior la diferencia se mantiene aproximadamente en un orden de magnitud entre ambos casos, y su comportamiento es muy similar. La operación de Complemento muestra un tiempo de ejecución constante en ambos casos, y las demás operaciones muestran que a mayor Compresibilidad de Entrada, menor es el tiempo de la ejecución.

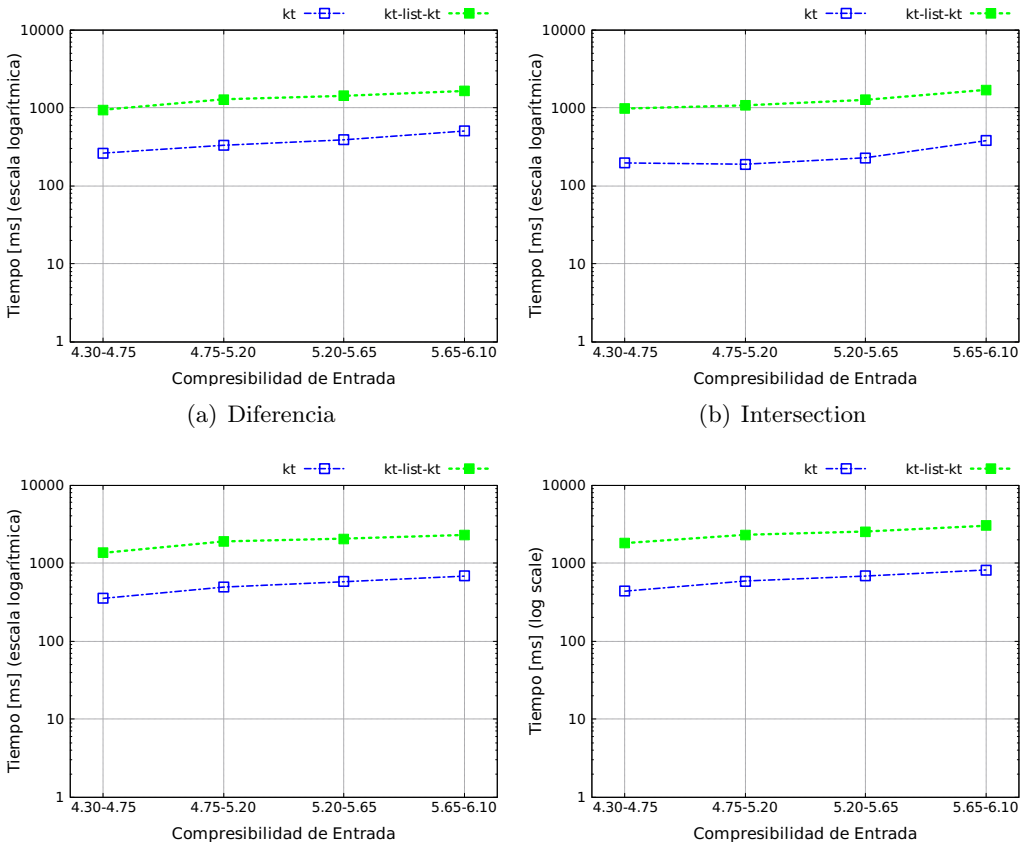
Adicionalmente, se incorpora la Figura 4.12 que muestra gráficamente el comportamiento que existe entre la cantidad de elementos relacionados y el tamaño del fichero para cada representación en los conjuntos de datos. Se puede apreciar que para el conjunto de datos `land-use` el factor de Compresibilidad en la Entrada no presenta una gran variación en la medida que se aumenta el número de elementos relacionados, en cambio el conjunto de datos `uk-snaps` muestra una gran variación en en su factor de Compresibilidad de Entrada, donde se puede ver una tendencia en la que el factor aumenta en la medida que la cantidad de elementos relacionados crece. Recordar que el factor de Compresibilidad de Entrada representa mejor compresión a menor valor.

4.3.3. Densidad de Entrada

En este criterio se considera una agrupación de los tiempos obtenidos en los experimentos según la Densidad de Entrada para las operaciones. Teniendo $densidad_R$, como el porcentaje de relaciones sobre el total de posibles relaciones (tercera columna de la Tabla 4.2), se define la Densidad de Entrada de una operación como la suma de las densidades de cada representación de entrada, es decir, $DensidadEntrada(Operación(A,B)) = densidad_A + densidad_B$.

La Figura 4.13 muestra los tiempo de las operaciones de Diferencia, Intersección, Diferencia Simétrica y Unión en el conjunto de datos `uk-snaps`. En estas operaciones se presenta una diferencia inferior a un orden de magnitud entre los tiempos de *kt* y *kt-list-kt*. El comportamiento para estos casos es muy similar: a medida que aumenta la Densidad de Entrada para las operaciones, aumenta el tiempo de ejecución. La operación sobre *list* resulta muy eficiente en tiempo tardando 1 a 2 órdenes de magnitud menos que el caso *kt-list-kt*.

Para el conjunto de datos `land-use`, que se muestra en la Figura 4.14, se muestran los resultados de las operaciones de Complemento, Diferencia, Intersección y Unión. En este caso la operación de Complemento se mantiene constante en tiempo para *list*, *kt* y *kt-list-kt*. En las demás operaciones los tiempos muestran un incremento a medida que se aumenta la Densidad de Entrada. Se puede ver que *kt-list-kt* siempre requiere más tiempo que *kt*. Cabe destacar que para la Intersección la implementación de las operaciones en *kt* presenta muy buenos resultados si se



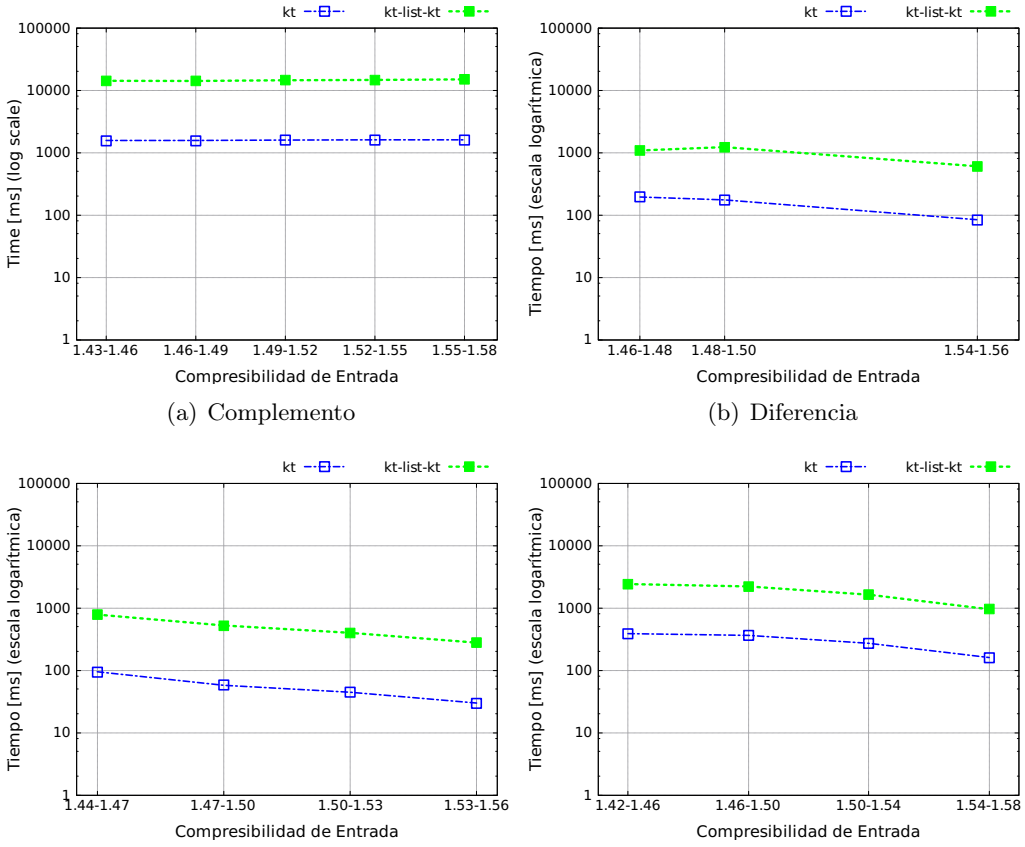
(a) Diferencia

(b) Intersection

(c) Symmetric Difference

(d) Union

Figura 4.10: Tiempos obtenidos en diferentes operaciones sobre el conjunto de datos `uk-snaps`. Se utiliza el criterio de Compresibilidad de Entrada en el eje X.



(a) Complemento (b) Diferencia

(c) Intersección (d) Unión

Figura 4.11: Tiempos obtenidos en diferentes operaciones sobre el conjunto de datos `land-use`. Se utiliza el criterio de Compresibilidad de Entrada en el eje X.

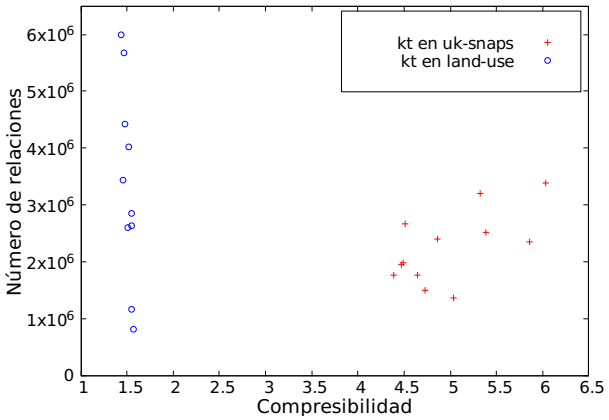


Figura 4.12: Número de elementos relacionados y compresibilidad de una representación binaria para los conjunto de datoss `uk-snaps` and `land-use`.

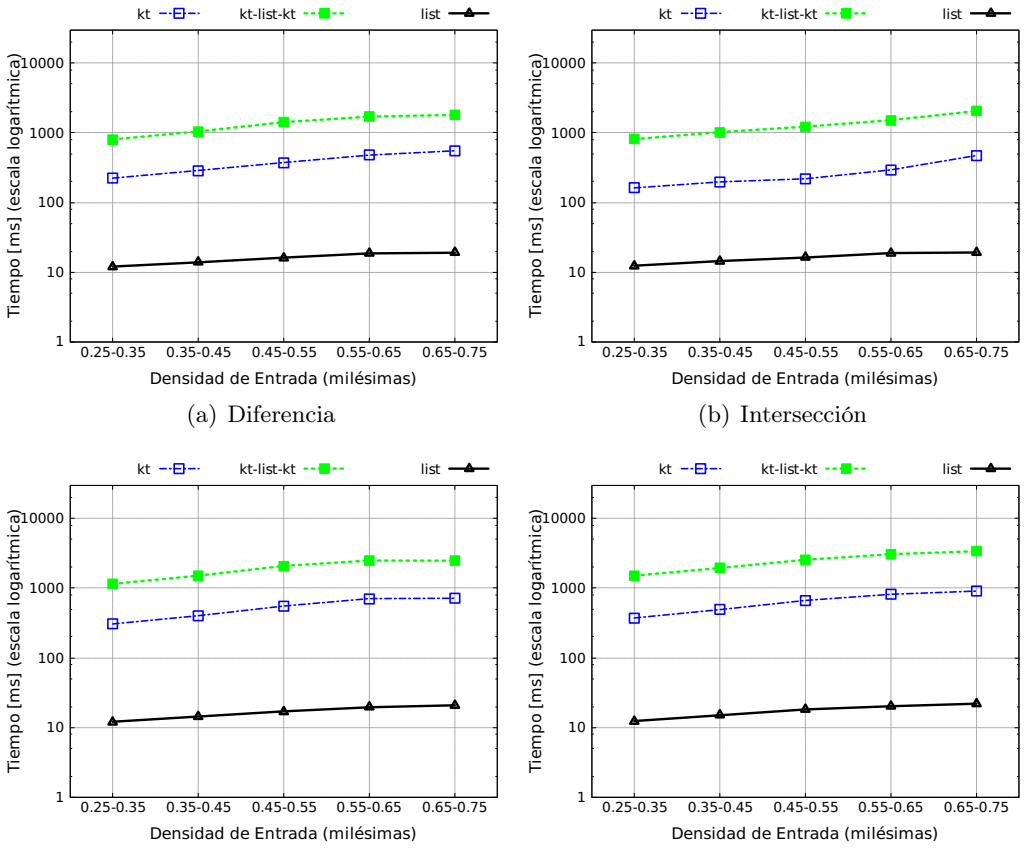
comparan contra las demás operaciones, teniendo un orden de magnitud menos que en la Unión.

4.3.4. Densidad de Salida

En este criterio se considera una agrupación de los tiempos obtenidos en las pruebas según la Densidad de Salida para las operaciones. En este caso se define la Densidad de Salida de una operación como el porcentaje de vínculos que posee su resultado sobre la cantidad de vínculos posibles en la relación.

La Figura 4.15 presenta los resultados para las operaciones de Diferencia, Intersección, Diferencia Simétrica y Unión en el conjunto de datos `uk-snaps`. En todos los casos el tiempo requerido por las operaciones aumenta mientras mayor sea la Densidad de Salida. Las operaciones sobre *kt-list-kt* toman un mayor tiempo del que se requiere en *kt*, pero en general la diferencia de tiempo se mantiene, mostrando un comportamiento similar en todas las operaciones.

En el conjunto de datos `land-use` los tiempos, en la Figura 4.16, muestran un comportamiento constante para la operación de Complemento en todos sus casos. Para las demás operaciones presentadas se tiene un incremento del tiempo a medida que aumenta la Densidad de Salida, pero la diferencia de tiempo entre *kt* y *kt-list-kt* se mantiene similar. La diferencia de tiempo entre *kt* y *kt-list-kt* se mantiene cercano a un orden de magnitud para todas las operaciones, donde el tiempo de *kt* siempre es inferior.



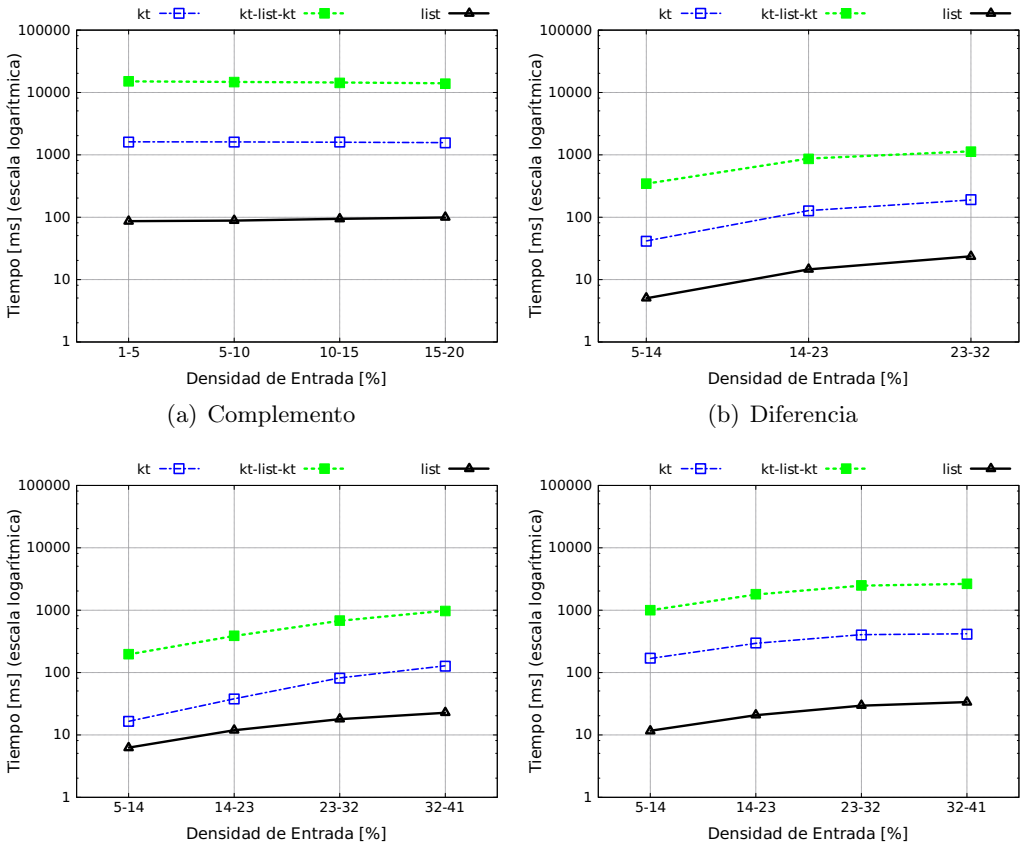
(a) Diferencia

(b) Intersección

(c) Diferencia Simétrica

(d) Unión

Figura 4.13: Tiempos obtenidos en diferentes operaciones sobre el conjunto de datos `uk-snaps`. Se utiliza el criterio de Densidad de Entrada en el eje X.



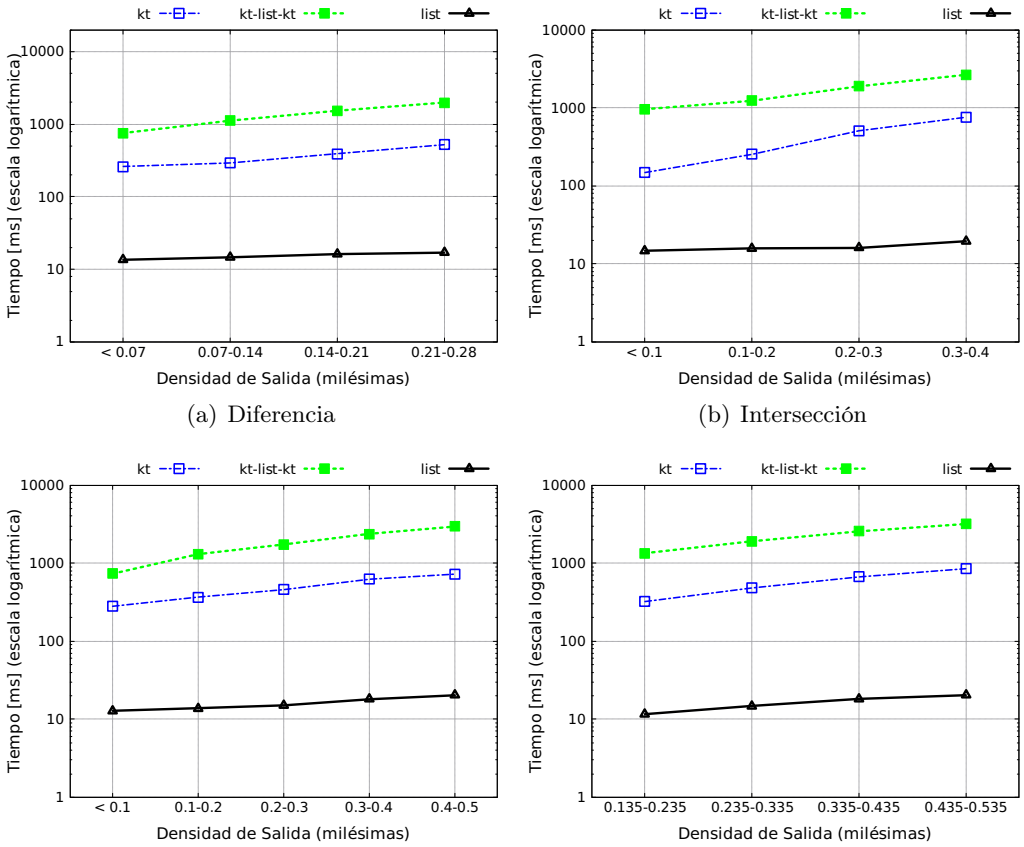
(a) Complemento

(b) Diferencia

(c) Intersección

(d) Unión

Figura 4.14: Tiempos obtenidos en diferentes operaciones sobre el conjunto de datos `land-use`. Se utiliza el criterio de Densidad de Entrada en el eje X.



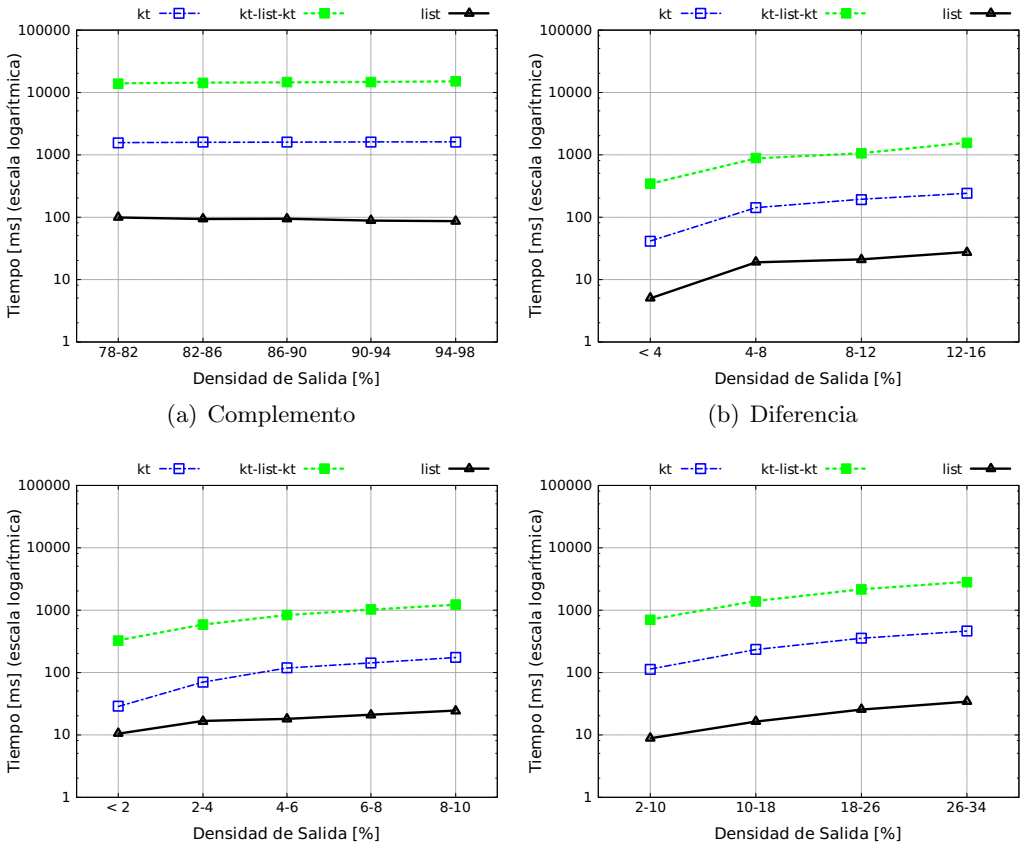
(a) Diferencia

(b) Intersección

(c) Diferencia Simétrica

(d) Unión

Figura 4.15: Tiempos obtenidos en diferentes operaciones sobre el conjunto de datos `uk-snaps`. Se utiliza el criterio de Densidad de Salida en el eje X.



(a) Complemento

(b) Diferencia

(c) Intersección

(d) Unión

Figura 4.16: Tiempos obtenidos en diferentes operaciones sobre el conjunto de datos `land-use`. Se utiliza el criterio de Densidad de Salida en el eje X.

Capítulo 5

Operaciones de Conjunto sobre el Wavelet Tree de Relaciones Binarias

En este capítulo se describen los algoritmos propuestos para la EDC wavelet tree de Relaciones Binarias (BRWT por sus siglas en inglés), descrito en la Sección 2.2.2, para resolver operaciones de conjunto sobre relaciones binarias representadas en esta EDC, de manera que la operación se calcule sobre las relaciones binarias sin la necesidad de descompactarlas en listas de adyacencia, calcular el resultado de la operación deseada y compactarlo a un formato de BRWT. También se muestran y discuten los resultados de la experimentación realizada para la evaluación del rendimiento de los algoritmos propuestos.

5.1. Los Algoritmos

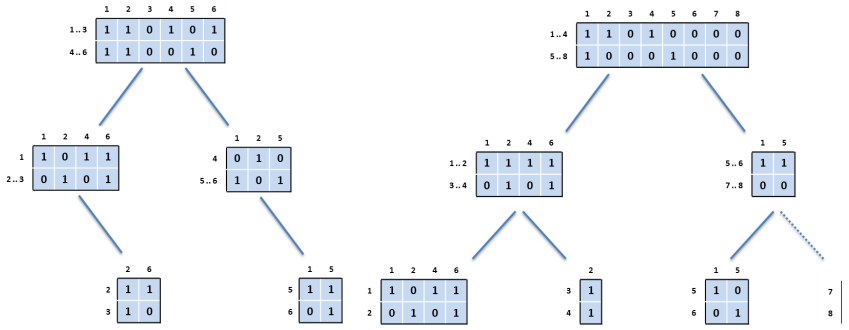
Los algoritmos propuestos para resolver operaciones de conjunto sobre relaciones binarias representadas en BRWT se basan en los propuestos para el k^2-tree por Brisaboa *et al.* [7], siendo modificados según fue necesario para el BRWT teniendo en cuenta las ventajas o desventajas que presenta las propiedades de esta EDC.

Lo anterior se fundamenta en el hecho que ambas estructuras son jerárquicas, permitiendo que los algoritmos para resolver las operaciones de conjunto aborden el problema de manera similar.

5.1.1. Aspectos Generales

A continuación se describen algunas consideraciones generales que se tienen para los algoritmos propuestos.

- Para las operaciones binarias (Unión, Intersección, Diferencia y Diferencia Simétrica) se opera sobre dos representaciones con la misma cantidad de objetos.
- Toda representación posee un número de objetos que es potencia de 2. Si una representación originalmente no lo es, se incrementa su número de objetos a la potencia de dos más próxima y se conserva en un parámetro el número de objetos del tamaño original en *originalObjects*.



(a) Sin extensión del número de objetos

(b) Con extensión del número de objetos

Figura 5.1: Ejemplos de BRWTs.

Si el árbol tuviera un número de objetos distinto de potencia de 2 algunas hojas estarían en el último nivel, y otras en el penúltimo. Este incremento del número de objetos hará que todos los nodos hoja del árbol estén al mismo nivel como lo muestra la Figura 5.1. Para aquellos nuevos objetos que se requiere añadir se definen valores 0 en el nodo raíz, por lo que no hay hijos asociados a ellos.

- Un nodo del BRWT se divide en dos partes con la misma cantidad de bits, por lo que se puede interpretar que un nodo posee dos sub-bitmaps para su representación. Por lo anterior, en una representación del BRWT se puede identificar cada nodo por sus dos sub-bitmaps como lo muestra la Figura 5.2. Y a cada sub-bitmap se le puede asignar un id que dependerá del número de objetos en el BRWT. Dado que para las operaciones binarias se tiene la misma cantidad de objetos, los ids de los sub-bitmaps definidos en ambas representaciones coincidirán siempre que estos ids se generen en base al número de bitmaps posibles, que a su vez se calcula en base al número de objetos de la representación. No sería así si el id fuera generado según el número de nodos reales de una representación, caso en el que algunos nodos podrían no estar presentes en alguna de las representaciones y por consiguiente, los ids de los sub-bitmaps no coincidirían, por ejemplo, si existiera un nivel más en la Figura 5.2 el siguiente sub-bitmap sería identificado con el número 14, independiente que los sub-bitmaps 12 y 13 no contengan elementos. Así mismo, las posiciones que se definen para cada sub-bitmap sería 43 para 12, 13 y 14.

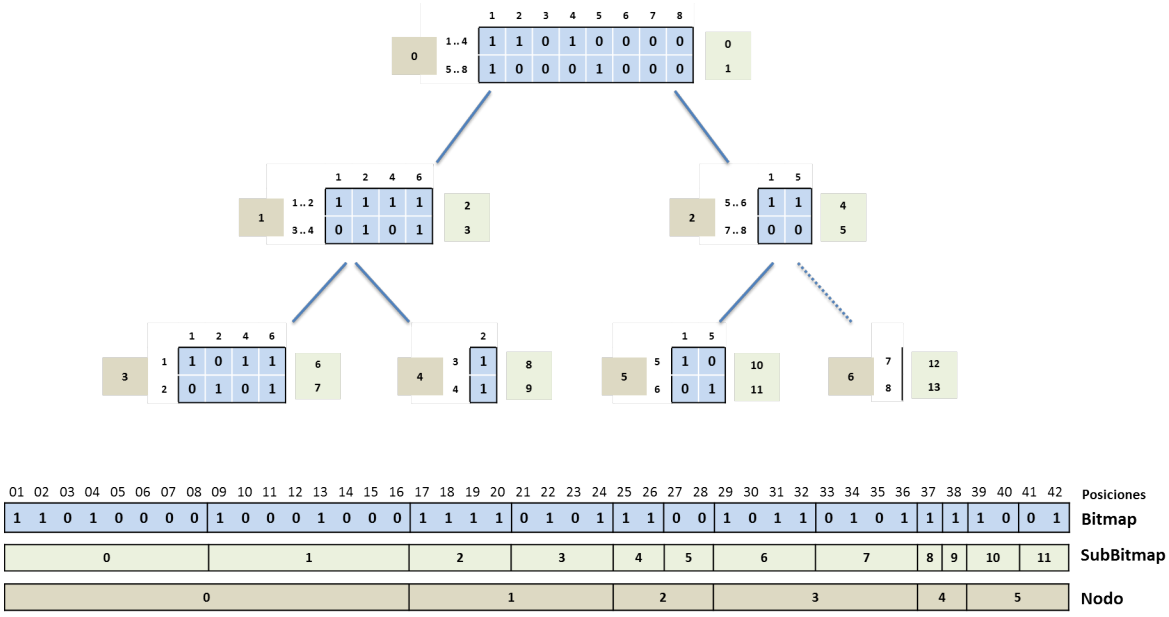


Figura 5.2: Ejemplo de BRWT, bitmap, sub-bitmaps y nodos.

Cabe destacar que el algoritmo de Unión, realiza un procesamiento en anchura de las representaciones para calcular el resultado. En cambio, los demás algoritmos realizan procesamientos en profundidad para determinar los resultados. Dado que esta exploración en profundidad se realiza por cada nodo, pero con el fin de determinar la presencia o ausencia de un elemento por nodo, no con el fin de procesar el nodo completo como sucede en el k^2-tree. Por los aspectos mencionados antes, para obtener la presencia o ausencia de una relación en un nodo, se deben conocer el resultado de cada sub-bitmap para todos los algoritmos que recorren en profundidad las representaciones.

Así también, los algoritmos que realizan recorridos en profundidad de la representación requieren de dos secciones:

- Un algoritmo que invoca la sección recursiva tantas veces como objetos posea la representación. Esto porque en el nodo raíz están definidos todos los elementos aún si sus valores son $\langle 0, 0 \rangle$ para los sub-bitmaps. Así también se aprovecha esta primera parte para obtener valores requeridos para el algoritmo recursivo que calcula el resto de los resultados.

- Un algoritmo que procesa los demás nodos, donde sólo se registran elementos que tienen al menos una relación en alguno de los sub-bitmaps. Como se dijo antes, este algoritmo sólo realiza el registro de un elemento cada vez; no procesa un nodo completo. Por lo mismo es que se requiere la primera parte no recursiva que invoca este algoritmo por cada elemento de la raíz.

En esta sección se hace uso de ejemplos que apoyan la descripción de los algoritmos de operaciones de conjunto sobre el BRWT para facilitar la comprensión. Estos ejemplos representan

las mismas relaciones binarias que fueron utilizadas como entrada para ejemplificar los algoritmos del k^2-tree, por lo que la Figura 5.3 (relación A en los ejemplos) es equivalente a la Figura 3.1, y la Figura 5.4 (relación B en los ejemplos) es la misma relación representada en la Figura 3.2. Así también se da que las figuras con los resultados de las operaciones de conjunto sobre el BRWT representan la misma relación binaria que los resultados en las figuras de ejemplo para el k^2-tree.

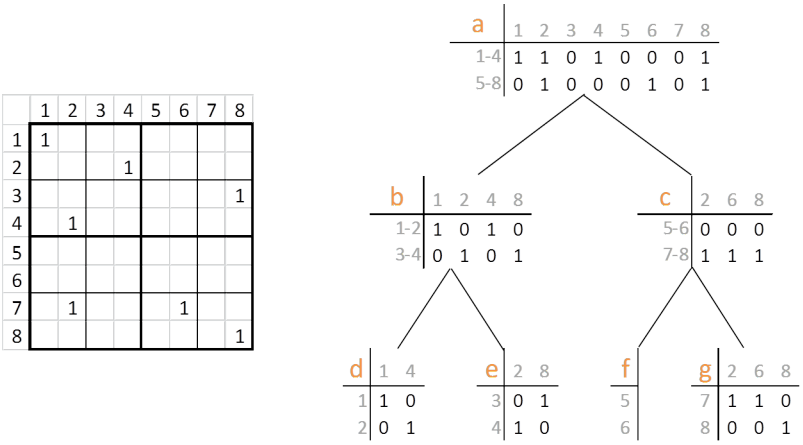


Figura 5.3: Ejemplo: BRWT A.

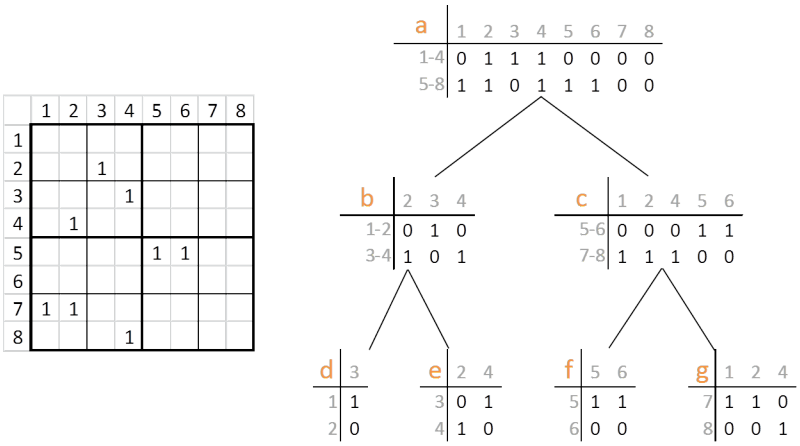


Figura 5.4: Ejemplo: BRWT B.

5.1.2. Unión

El Algoritmo 5.1.1 de Unión calcula el resultado realizando un recorrido del bitmap desde los bits que representan el nodo raíz, hasta los que representan los nodos hoja. Esto es posible ya que la Unión de dos conjuntos corresponde a todos los objetos que estén en alguna de las relaciones de origen/entrada. Dado esto, en los nodos superiores del BRWT se puede identificar qué elementos están presentes en el resultado sin la necesidad de explorar el árbol hasta sus nodos hoja.

Algorithm 5.1.1 UnionBRWT(A, B)

```
1: usingQA ← true
2: pA ← 0, pB ← 0
3: bA ← 0, bB ← 0
4: for i ← 0 ... numOfObjects do
5:     QA.Insert(pair⟨1, 1⟩)
6: end for
7: QA.Insert(pair⟨0, 0⟩)
8: while pA < |A| ∨ pB < |B| do
9:     if usingQA then
10:        pair ← QA.Remove()
11:    end if
12:    childrens ← 0
13:    while (usingQA ∧ (pair.A ∨ pair.B)) ∨ (!usingQA∧!QB.IsEmpty())) do
14:        if usingQA then
15:            QB.Insert(pair)
16:        else
17:            pair ← QB.Remove()
18:        end if
19:        bA ← pair.A ∧ A[pA]
20:        bB ← pair.B ∧ B[pB]
21:        if (bA ∨ bB) ∧ (pA < A.leafStarts ∨ pB < B.leafStarts) then
22:            QA.Insert(pair⟨bA, bB⟩)
23:            childrens ← childrens + 1
24:        end if
25:        R[posR] ← bA ∨ bB
26:        posR ← posR + 1
27:        if pair.A then
28:            pA ← pA + 1
29:        end if
30:        if pair.B then
31:            pB ← pB + 1
32:        end if
33:        if usingQA then
34:            pair ← QA.Remove()
35:        end if
36:    end while
37:    usingQA ← (∼ usingQA)
38:    if (pA < A.leafStarts ∨ pB < B.leafStarts) ∧ childrens > 0 then
39:        QA.Insert(pair⟨0, 0⟩)
40:    end if
41: end while
42: return R
```

En el Algoritmo 5.1.1 de Unión del BRWT considera los siguientes puntos para la sincronización del recorrido de los bitmaps que representan las relaciones de entrada:

1. La cantidad de elementos en los nodos del BRWT no es la misma en todos, como sucede con el k^2-tree (donde los nodos son de tamaño k^2).

2. Un nodo del BRWT se compone por dos bitmaps.

3. Los elementos presentes en cada nodo también están definidos en el nodo padre. Por lo tanto, si el nodo padre representa la existencia de un elemento en uno de sus bitmaps, el nodo hijo correspondiente define en cada uno de sus bitmaps algún valor para este elemento (0 ó 1).

4. Puede darse que un nodo no esté representado en alguna de las relaciones, pero si lo esté en la otra. Como consecuencia, el resultado si tendrá los elementos representados en la relación que lo posee.

Similar a como se hace en el Algoritmo 3.1.1 de Unión para el k^2-tree se utilizará una cola para identificar la presencia o ausencia de los elementos en cada relación. Cabe resaltar que la cola utilizada en el k^2-tree indica la presencia o ausencia de nodos en las relaciones, mientras que en el BRWT se utilizará la cola QA para determinar la presencia o ausencia de elementos en los nodos, lo que gráficamente equivale a una columna en los nodos del BRWT. Otra diferencia radica en que la cola utilizada en el BRWT (QA) no guarda el nivel, ya que este algoritmo utiliza una posición del bitmap en la que comienzan las hojas de cada relación para identificar el nivel máximo, esto gracias a que se creó el atributo $leafStarts$ para la implementación del BRWT. Es por lo anterior que QA almacena pares de bits, de la forma $\langle b_1, b_2\rangle$, que indican si el nodo está presente en la relación A por medio del bit b_1 (accedido como $pair.A$ en el Algoritmo 5.1.1), y también si el nodo está en la relación B por medio de b_2 (accedido como $pair.B$ en el Algoritmo 5.1.1). Lo anterior permite un recorrido sincronizado de los bitmaps de cada relación binaria independiente de los nodos presentes/ausentes en cada una, abordando de manera efectiva los puntos 3 y 4 antes mencionados.

La estrategia que se utiliza para reconocer el inicio y fin de cada nodo hace uso de la misma cola de pares QA gracias a que ésta sólo almacena los pares $\langle 1, 1\rangle$, $\langle 0, 1\rangle$ o $\langle 1, 0\rangle$, donde queda disponible el par $\langle 0, 0\rangle$ ya que, tal como en el Algoritmo 3.1.1, no es necesario insertar en la cola un elemento cuando ninguna de las relaciones A y B define un nodo. Por tanto, para resolver el punto 1, se utiliza el par $\langle 0, 0\rangle$ en el Algoritmo 5.1.1 para indicar el fin de un nodo (ó comienzo de uno).

Dado que un nodo posee dos bitmaps que se deben procesar, los pares contenidos en la cola deben ser utilizadas dos veces durante el recorrido secuencial de los bitmaps. Para no duplicar los pares en la cola QA se utiliza la cola QB que "recoge" los pares a medida que éstos son desencolados de QA hasta que finaliza un bitmap, e inmediatamente se procesan los pares de QB para el segundo bitmap del mismo nodo. Sin esta segunda cola, en la primera se deberían duplicar los pares para procesar ambos bitmaps del nodo que le correspondan. De esta manera se aborda el punto 2

Sólo se insertan elemento en la cola QA mientras los nodos procesados tengan hijos. Es decir que desde la raíz hasta el penúltimo nivel se insertan y procesan pares en/desde QA, pero en el último nivel sólo se procesan los pares que ya están almacenados en QA, sin realizar nuevas inserciones. Esto no aplica para QB, ya que "recoge" los elementos que son procesados por QA para utilizarlos en el segundo bitmap de cada nodo de los BRWT.

En la Figura 5.5 se ve la matriz de adyacencia y el BRWT que representa la relación del resultado al aplicar el algoritmo de Unión teniendo como entrada las relaciones binarias de las figuras 5.3 y 5.4, como A y B respectivamente. A continuación se describe parte del proceso que realiza el algoritmo, el que comienza insertando tantos pares $\langle 1, 1\rangle$ en QA como objetos participan en las relaciones binarias. Estos pares insertados serán los que permiten el procesamiento del primer nodo de ambas relaciones, recordando que el nodo raíz del BRWT define todos los objetos. Luego se inserta el par $\langle 0, 0\rangle$ en QA para indicar el fin del primer nodo. De manera general, por cada nodo del BRWT, se procesan todos los pares desde QA siendo insertados en QB a medida

que se extraen y siempre que el contenido del par sea distinto de $\langle 0, 0\rangle$. Una vez que se encuentra este par se realizan las mismas operaciones para determinar el resultado pero haciendo uso de los pares almacenados en la cola QB mientras ésta no sea vacía. El control de este intercambio en el uso de las colas se hace mediante la variable $usingQA$ (Iniciada en la Línea 1) que se utiliza para indicar si es QA o QB la cola desde la que se extraen los pares en las líneas 9 y 33.

El recorrido se realiza sobre los bitmaps A y B mediante el uso de los índices $posA$ y $posB$ respectivamente. En aquellos casos donde una relación no define una rama será el par que se extraiga de la cola que esté en uso la que indique si los bits en las posiciones $A[posA]$ y $B[posB]$ deben ser considerados en las instrucciones. Éstos índices son incrementados en cada iteración en la que son utilizados, es decir que cuando el par contiene 1 para A se incrementa $posA$ (Línea 28) y cuando el par contiene el valor 1 para B se incrementa $posB$ (Línea 31). Cabe mencionar que el resultado será almacenado en el bitmap R el que será recorrido con el índice $posR$ a medida que se genere el resultado.

El algoritmo, luego de las inserciones de los 8 pares $\langle 1, 1\rangle$ para el procesamiento del primer nodo de las relaciones (Ver figuras 5.3 y 5.4, las que poseen 8 columnas en su nodo a), inicia con el nodo a de ambas relaciones, extrayendo un par desde QA que contiene $\langle 1, 1\rangle$, donde el primer bit de A es 1 y el bit de B es 0, luego se inserta el mismo par en la cola QB. Dado que en este caso el bit de A es 1 y el de B es 0, se inserta en QA el par $\langle 1, 0\rangle$, posteriormente se inserta el resultado en R que corresponde a la operación lógica OR sobre los bits (Línea 25). El siguiente par será $\langle 1, 1\rangle$, que se inserta en QB, dado que ambas relaciones poseen 1 en el bit correspondiente se inserta $\langle 1, 1\rangle$ en QA, y finalmente se inserta el resultado en R, que para este caso es 1. Se procede de esta misma manera hasta que se finaliza el procesamiento del primer bitmap del nodo a en ambas relaciones, lo que se reconoce porque el noveno par que se extrae de QA corresponde al par $\langle 0, 0\rangle$ que se insertó luego de los ocho pares $\langle 1, 1\rangle$ insertados para procesar la raíz. Para el procesamiento del segundo bitmap del nodo a en ambas relaciones, se utilizan los pares almacenados en QB, siguiendo la misma estrategia que en la primera parte, con la salvedad que los pares extraídos desde QB simplemente son descartados una vez que se utilizan, y que el fin del segundo bitmap (y por tanto el fin del nodo) se reconoce cuando QB está vacía. Cuando se pasa al procesamiento del nodo b de ambas relaciones, primero se extrae un par de QA correspondiente al primero insertado en el desarrollo de este ejemplo, que contiene $\langle 1, 0\rangle$, por lo tanto se interpreta que en A el bit de la posición $posA$ debe ser considerado en el resultado, pero el bit de B en la posición $posB$ no. Como en los casos descritos antes, se inserta un par en QB con el mismo contenido del par en proceso, el que será reutilizado para las posiciones del nodo b de ambas relaciones cuando se procese el segundo bitmap. Luego se inserta en QA el par $\langle 1, 0\rangle$. Así se procesan cada uno de los nodos hasta finalizar los bitmaps de ambas relaciones.

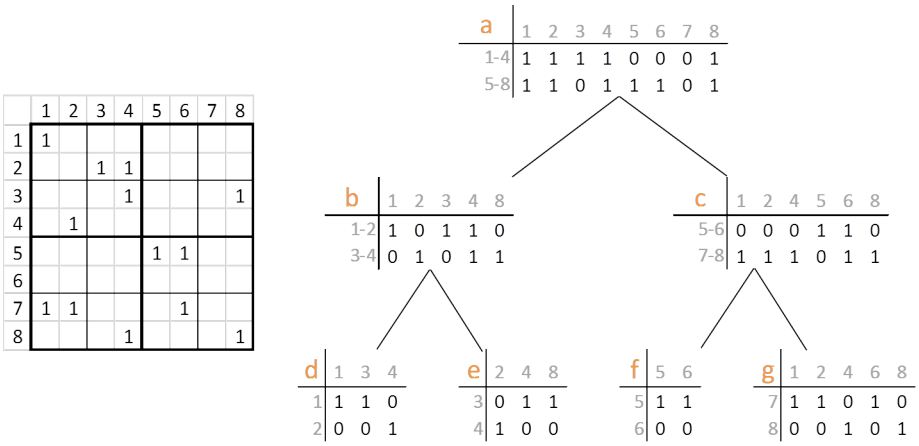


Figura 5.5: BRWT del resultado de la Operación de Unión sobre las relaciones A (Figura 5.3) y B (Figura 5.4).

La variable *childrens* del Algoritmo 5.1.1 se utiliza para evitar insertar un par $\langle 0, 0\rangle$ en QB en el caso que los nodos de ambas representaciones no posean hijos (no definan una rama). Para relaciones binarias que definen pocos elementos se podrían producir situaciones en las que QB posea mayormente pares $\langle 0, 0\rangle$, realizando iteraciones que no ejecutan instrucciones útiles para la operación.

5.1.3. Algoritmos con Recorrido en Profundidad

A continuación se describen algunos elementos que son comunes en todos los algoritmos que realizan un recorrido en profundidad de la representación del BRWT. Los algoritmos a considerar son Diferencia, Diferencia Simétrica, Intersección y Complemento, de las cuáles sólo el complemento corresponde a una operación unaria.

Los parámetros que se deben obtener con anterioridad a la ejecución de cada operación son:

- pA/pB corresponde a la lista de posiciones de los sub-bitmaps de cada nodo en el bitmap que representa el BRWT A/B obtenido luego de llamar a la función $positionsOfWT$. Estas posiciones cumplen que $pX[i] \leq pX[i+1]$, $0 \leq i < numSubBitmaps$. Se puede dar que dos posiciones contiguas sean iguales cuando un nodo no está definido en el BRWT, por lo que el sub-bitmap ocupa 0 bits del bitmap general. Esto se puede ver en los nodos 12 y 13 de la Figura 5.2, donde éstos nodos, al no contener elementos, tendrían la misma posición que el siguiente nodo, que corresponde a 43 según la misma figura.

- bA/bB son los bits que indican si el nodo padre define 0 ó 1 para el elemento analizado en los sub-bitmaps de la relación A/B al invocar el algoritmo recursivo.

- $idBitmap$ corresponde al identificador del bitmap que se procesa en el BRWT, como el ejemplo de la Figura 5.2. Este parámetro se utiliza para indicar la posición en el arreglo pA/pB que almacena la posición de cada sub-bitmap.

- $idFirstLeaf$ contiene el id del primer sub-bitmap del primer nodo hoja en el BRWT. El valor es dependiente del número de objetos que posea el BRWT y se obtiene con la invocación a la función $firstLeaf$. Dado que las representaciones operadas (en caso que sea una operación binaria) poseen igual número de objetos, el id de cada sub-bitmap es coincidente en ambas representaciones. Este parámetro ayuda a identificar cuándo se debe continuar la recursión para explorar hijos.

A continuación se describen las variables principales que utilizan los algoritmos con recorrido en profundidad para resolver las operaciones de conjunto.

- $idLeftChild$ y $idRightChild$ son los ids de los sub-bitmaps que representan el hijo izquierdo y derecho del nodo respectivamente. Por ejemplo, en la Figura 5.2 para el nodo 0 los valores de estas variables serían 2 para $idLeftChild$ y 4 para $idRightChild$.

- rA/rB indica si se debe considerar o no el bit apuntado por $pA/pB[idNodo]$. Podría ser que el padre posee 0 para el elemento actual o que el padre no existe en una relación, pero se está explorando dicha rama porque la otra relación si la define.

- bA_1/bA_2 y bB_1/bB_2 se utilizan para almacenar el valor del bit que se analiza en la operación en combinación con rA/rB. Por ejemplo, en la Figura 5.3, su nodo b define la columna 1, pero la Figura 5.4 no lo hace, por tanto bA_1 será 1 y bA_2 será 0, ya que el primer sub-bitmap de b en A define 1 y el segundo 0 para la columna 1, mientras que en la relación B las variables bB_1 y bB_2 serán 0 independiente del bit que se esté apuntando por el indice correspondiente, debido a que el nodo no define la columna 1 en b de B. Para el mismo nodo, sucede lo inverso con la columna 3, ya que la relación B si la define, mientras que A no lo hace.

- kl y kr son utilizados para almacenar los resultados parciales de cada sub-bitmap para el nodo en proceso. Éstos valores serán registrados en el resultado si al menos uno de ellos es 1, ó en el caso que se esté procesando la raíz del resultado.

- R corresponde al bitmap donde se almacena el resultado de la operación. Al igual que A y B, R se divide en $numNodos \times 2$ cantidad de sub-bitmaps, donde el $idNodo$ indica a qué sub-bitmap se concatena el resultado parcial de la operación por medio del operador $||$.

Así como en los algoritmos presentados para las operaciones de conjunto sobre k^2-tree en la Sección 3.1, los algoritmos para relaciones binarias representadas mediante BRWT requieren de algunas funciones adicionales, las que serán descritas a continuación. En general, estas funciones cumplen el mismo objetivo que las funciones *SkipNodes*, *FillIn* y *Copy* pero no lo hacen de la misma forma, debido a la diferencia que hay en la estructura.

- ***Copy***: Esta función se utiliza en las operaciones de Diferencia (Algoritmo 5.1.3) y de Diferencia Simétrica (Algoritmo 5.1.5), y tiene por objetivo realizar una copia de algunos elementos desde una de las relaciones de entrada hacia el resultado. A diferencia del Algoritmo 3.1.5 *Copy* que realiza la copia de una rama completa, esta función sólo copia la

columna deseada desde aquellos nodos que se encuentran en la rama hacia el resultado. Por ejemplo, si se utiliza esta función desde el nodo a de la Figura 5.3 para ejecutar una copia de la columna 1, la función recorrería el nodo b y copiaría la columna 1 en el resultado y luego exploraría el nodo d copiando también el resultado. El nodo e no sería visitado ya que desde el nodo b se conoce que e no posee elementos de la columna 1. Al momento de realizar la copia de los bits que corresponden, la función actualiza las posiciones de aquellos nodos que definen la columna que se copia hacia el resultado.

- ***Skip***: La función, utilizada por los Algoritmos de Diferencia 5.1.3 e Intersección 5.1.4, realiza una actualización del arreglo de posiciones de manera que la rama correspondiente se salte la columna indicada en aquellos nodos que la componen. Por ejemplo, utilizando la Figura 5.3, la función *Skip* en el nodo a para la columna 1 recorrería la rama formada por b, d y e en busca de aquellos que definen la misma columna. En este punto, hay que recordar que los arreglos de posiciones (en este caso pA) contendrán la posición de la columna igual o superior a 1 en cada nodo, así es que para el nodo b y d apuntarán a los bits de la columna 1, el nodo e a los bits de la columna 2. Es por lo anterior, que sólo se actualizarían los nodos b y d, donde los arreglos de posiciones pasarían a apuntar a las columnas 2 y 4 respectivamente, y el nodo e permanecería indicando la columna 2. De esta forma el algoritmo que haya invocado a *Skip* podrá continuar procesando los elementos que apuntan los arreglos de posiciones los que ya han saltado las posiciones en aquellos nodos que definen la columna no deseada.

- ***FillIn***: Genera valores 1 para una columna en todos los nodos que pertenecen a una rama, lo que sólo es utilizado por el Algoritmo 5.1.7 de la operación de Complemento. Por ejemplo, si se utiliza *FillIn* en el nodo a de la Figura 5.3 para el segundo sub-bitmap de la columna 1, se generaría en el resultado bits con 1 para los dos sub-bitmaps de todos los nodos de la rama correspondiente, que en este caso se compondría por c, f y g.

Para los algoritmos de las operaciones Diferencia, Intersección y Diferencia Simétrica se deben realizar repetidas llamadas recursivas para procesar todo el nodo raíz. Esto se debe a que los algoritmos recursivos en cada llamada obtienen el resultado de un elemento en el nodo, no procesan un nodo completo. La cantidad de llamadas corresponde al número de objetos de la relación, o lo que es igual, el número de elementos en la raíz. El Algoritmo 5.1.2 prepara las variables para la ejecución de los algoritmos recursivos y realiza la cantidad de llamadas necesarias para las operaciones. En este algoritmo se debe reemplazar la función *RecOperationBRWT* (Línea 7) por el nombre del algoritmo correspondiente según sea la operación que se debe realizar. En adelante, se hablará de los algoritmos de las operaciones como si fuese el mismo algoritmo el que procesa las relaciones de entrada.

5.1.4. Diferencia

El Algoritmo 5.1.3 de Diferencia para el BRWT considera los casos que a continuación se describen para obtener el resultado de la operación. *Caso 1* si se analiza un elemento del nodo hoja se realiza el cálculo del resultado de manera directa con los bits correspondientes, esto aplicando un *AND* lógico del bit de la primera relación con la negación del bit de la segunda

Algorithm 5.1.2 OperationBRWT(A, B)

```
1: pA ← positionsOfWT(A)
2: pB ← positionsOfWT(B)
3: bA ← 1, bB ← 1
4: idBitmap ← 0
5: idFirstLeaf ← firstLeaf(numOfObjects)
6: for i ← 0 ... numOfObjects do
7:     RecOperationBRWT(A, B, pA, pB, bA, bB, R, idBitmap, idFirstLeaf)
8: end for
9: return R
```

relación para ambos sub-bitmaps (Líneas 25 y 26). Si no se analiza un nodo hoja se tiene; *Caso 2*: si ambos bits son 1 se explora el hijo correspondiente con una llamada recursiva y se almacena el resultado que retorna el algoritmo (Líneas 10 y 18). *Caso 3*: cuando sólo A es 1 se debe realizar la copia de toda la rama hijo correspondiente con el uso de la función ***Copy*** (Líneas 12 y 20). *Caso 4*: si sólo el bit de B es 1 entonces se deben actualizar los valores del arreglo de posiciones pB descartando los elementos de la rama correspondiente (Líneas 15 y 22) haciendo uso de la función ***Skip***. Existe un caso que no se define directamente en el algoritmo; cuando ambos bits son 0, esto se debe a que los valores de kl y kr son inicializados en 0 (Línea 3), por lo que tal caso esta cubierto desde el principio.

La Figura 5.6 muestra la matriz de adyacencia y el BRWT resultado de la operación de Diferencia sobre las relaciones binarias de la Figuras 5.3, como la relación A y la Figura 5.4 como la relación B.

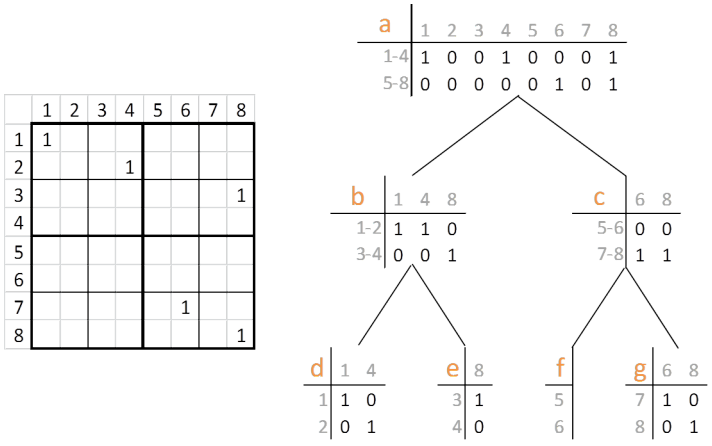


Figura 5.6: BRWT del resultado de la Operación de Diferencia sobre los BRWT A (Figura 5.3) y B (Figura 5.4).

Para obtener el resultado de esta figura el Algoritmo 5.1.3 comienza analizando la columna 1 de los nodos a en las relaciones A (Figura 5.3) y B (5.4), donde A tiene 1 y 0, y B tiene 0 y 1 (Recordar que para las operaciones recursivas se utiliza el Algoritmo 5.1.2, que en este caso realiza la llamada 8 veces al Algoritmo 5.1.3 para procesar las 8 columnas de los nodos raíz de ambas relaciones). Estos bits producen dos comparaciones, donde a cada uno se le aplican los casos antes descritos para esta operación. La primera comparación utiliza el bit del primer sub-bitmap de

Algorithm 5.1.3 RecDifferenceBRWT$(A, B, pA, pB, rA, rB, R, idBitmap, idFirstLeaf)$

```
1: idLeftChild ← (idBitmap + 1) * 2
2: idRightChild ← (idBitmap + 2) * 2
3: kl ← 0, kr ← 0
4: bA_1 ← A[pA[idBitmap]] ∧ rA
5: bA_2 ← A[pA[idBitmap + 1]] ∧ rA
6: bB_1 ← B[pB[idBitmap]] ∧ rB
7: bB_2 ← B[pB[idBitmap + 1]] ∧ rB
8: if idBitmap < idFirstLeaf then
9:     if bA_1 ∧ bB_1 then
10:        kl ← RecDifferenceBRWT(A, B, pA, pB, bA_1, bB_1, R, idLeftChild, idFirstLeaf)
11:    else if bA_1 then
12:        Copy(A, pA, R, idLeftChild, idFirstLeaf)
13:        kl ← 1
14:    else if bB_1 then
15:        Skip(B, pB, idLeftChild, idFirstLeaf)
16:    end if
17:    if bA_2 ∧ bB_2 then
18:        kr ← RecDifferenceBRWT(A, B, pA, pB, bA_2, bB_2, idRightChild, idFirstLeaf)
19:    else if bA_2 then
20:        Copy(A, pA, R, idRightChild, idFirstLeaf)
21:    else if bB_2 then
22:        Skip(B, pB, idRightChild, idFirstLeaf)
23:    end if
24: else
25:    kl ← bA_1 ∧ ∼ bB_1
26:    kr ← bA_2 ∧ ∼ bB_2
27: end if
28: if kl ∨ kr ∨ idBitmap == 0 then
29:    R[idBitmap] ← R[idBitmap]||kl
30:    R[idBitmap + 1] ← R[idBitmap + 1]||kr
31: else if rA ∨ idBitmap == 0 then
32:    pA[idBitmap] ← pA[idBitmap] + 1
33:    pA[idBitmap + 1] ← pA[idBitmap + 1] + 1
34: else if rB ∨ idBitmap == 0 then
35:    pB[idBitmap] ← pB[idBitmap] + 1
36:    pB[idBitmap + 1] ← pB[idBitmap + 1] + 1
37: end if
38: return kl ∨ kr
```

cada relación, donde A es 1 y B es 0, por lo tanto se aplica el *Caso 3* que utiliza la función *Copy*. En este caso, la función no realiza una copia de toda la rama hacia el resultado R, sino que sólo copia desde A aquellos elementos que coinciden con la columna 1 desde cada nodo de la rama, donde estos nodos serían b con 1 y 0 y el nodo d con los valores 1 y 0. El nodo e pertenece a la rama involucrada pero no define elementos para al columna 1. Para el segundo par de bits que se deben analizar, donde se tiene 0 para A y 1 para B, se descartan aquellos bits de la relación B que pertenecen a la columna 1 en la rama formada por los nodos c, f y g según indica el *Caso 4* (Donde sólo se actualizan los valores de las posiciones para los nodos c y g, ya que f no define la columna 1). Luego de lo anterior, se procede a procesar la columna 2 de las relaciones A y B. El bit del primer sub-bitmap es 1 en ambas relaciones, por lo que se debe explorar según indica el *Caso 2*. En este caso se explorará el hijo izquierdo de ambas relaciones correspondiente al nodo b, donde en ambas relaciones se tiene 0 para el primer sub-bitmap, por lo tanto el resultado parcial almacenado en la variable kl se mantiene en 0, pero el segundo sub-bitmap en ambos casos es 1, por lo que nuevamente se procede según el *Caso 2*, explorando el nodo e de ambas relaciones

que corresponde a un nodo hoja, por lo que se calcula directamente el resultado, según el *Caso 1*, esto resulta 0 para ambos sub-bitmaps. Por lo anterior, la operación no realiza registro de los resultados parciales, y retorna 0 como resultado. Esto hace que en la llamada que procesaba el nodo b también tenga ambos resultados parciales en 0, por lo que procede de la misma forma; no realiza registro y retorna 0 al resultado. Así es como para el nodo a, se tienen como resultado parcial para kl el valor 0, y sucederá de manera análoga para el resultado parcial kr, por lo que finalmente, el resultado para la columna 2 es 0 en ambos sub-bitmaps del nodo a.

5.1.5. Intersección

El Algoritmo 5.1.4 para la Intersección analiza el resultado de manera recursiva, recorriendo en profundidad las representaciones de entrada. A continuación se describen los casos considerados cuando no se está en un nodo hoja; *Caso 1*: Si ambos bits son 1 entonces debe realizar una llamada para que se exploren la ramas respectiva (Líneas 10 y 17), *Caso 2*: cuando uno de los bits es 1 se debe realizar una actualización de las posiciones pA/pB para descartar los bits que representan la rama (Líneas 12, 14, 19 y 21). El caso cuando ambos bits son 0 está cubierto desde el principio, cuando se asignan valores 0 a las variables kl y kr (Línea 3). Si se está en un nodo hoja, *Caso 4*, el resultado corresponde a una operación AND lógico de ambos bits (Líneas 24 y 25).

La Figura 5.7 muestra la matriz de adyacencia y el BRWT resultado de la operación de Intersección sobre las relaciones binarias de las figuras 5.3 y 5.4. El Algoritmo 5.1.4 comienza con la primera columna de los nodos a de las dos relaciones de entrada, donde para el primer sub-bitmap A es 1 y B es 0, y según el *Caso 2* se debe descartar la rama del hijo izquierdo en A con la función *Skip*, la cual descartaría la primera columna de los nodos b y d. Luego, con el bit del segundo sub-bitmap de las relaciones, se tiene 0 en A y 1 en B, por lo que ahora se descartan elementos del hijo derecho de B; la columna 1 en los nodos c y g. Continuando con la segunda columna de los nodos a, en el primer sub-bitmap los bits son 1 en ambas relaciones, por lo que según el *Caso 1* se exploran las ramas correspondientes, en este caso los hijos izquierdos. En el nodo b se tiene, para la columna 2, valores 0 para los primeros sub-bitmaps (por lo que el resultado parcial kl es 0) y 1 en el segundo sub-bitmap utilizando el *Caso 1* nuevamente, el que en esta situación explorará los hijos derechos del nodo b en ambas relaciones, el nodo e. En este nodo, que corresponde a una hoja, se aplica el *Caso 4* y, dado que los bits para la columna 2 coincide en A y B (el primer sub-bitmap es 0 y el segundo es 1 en las dos relaciones), el resultado corresponde a 0 y 1, lo que se registra en R y se retorna 1. Este retorno se toma en el proceso que tenía el nodo b y registra 1 para el resultado parcial kr, por tanto se añade 0 en R para el primer sub-bitmap y 1 para el segundo, luego se retorna 1, que es tomado por el resultado parcial kl del nodo a. Luego se analizan los bits para el segundo sub-bitmap en los nodos a de A y B, que nuevamente son 1 en ambos casos. Este procesamiento también resulta en una intersección de la columna 2 en los nodos c y g, por lo que kr también es 1 y el resultado se registra en R. Finalmente, luego de procesar los elementos restantes, se obtiene el resultado de la Figura 5.7.

Algorithm 5.1.4 RecIntersectionBRWT$(A, B, pA, pB, rA, rB, R, idBitmap, idFirstLeaf)$

```
1: idLeftChild ← (idBitmap + 1) * 2
2: idRightChild ← (idBitmap + 2) * 2
3: kl ← 0, kr ← 0
4: bA1 ← A[pA[idBitmap]] ∧ rA
5: bA2 ← A[pA[idBitmap + 1]] ∧ rA
6: bB1 ← B[pB[idBitmap]] ∧ rB
7: bB2 ← B[pB[idBitmap + 1]] ∧ rB
8: if idBitmap < idFirstLeaf then
9:     if bA1 ∧ bB1 then
10:        kl ← RecIntersectionBRWT(A, B, pA, pB, bA1, bB1, R, idLeftChild, idFirstLeaf)
11:    else if bA1 then
12:        Skip(A, pA, idLeftChild, idFirstLeaf)
13:    else if bB1 then
14:        Skip(B, pB, idLeftChild, idFirstLeaf)
15:    end if
16:    if bA2 ∧ bB2 then
17:        kr ← RecIntersectionBRWT(A, B, pA, pB, bA2, bB2, idRightChild, idFirstLeaf)
18:    else if bA2 then
19:        Skip(A, pA, idRightChild, idFirstLeaf)
20:    else if bB2 then
21:        Skip(B, pB, idRightChild, idFirstLeaf)
22:    end if
23: else
24:    kl ← bA1 ∧ bB1
25:    kr ← bA2 ∧ bB2
26: end if
27: if kl ∨ kr ∨ idBitmap == 0 then
28:    R[idBitmap] ← R[idBitmap]||kl
29:    R[idBitmap + 1] ← R[idBitmap + 1]||kr
30: end if
31: if rA ∨ idBitmap == 0 then
32:    pA[idBitmap] ← pA[idBitmap] + 1
33:    pA[idBitmap + 1] ← pA[idBitmap + 1] + 1
34: end if
35: if rB ∨ idBitmap == 0 then
36:    pB[idBitmap] ← pB[idBitmap] + 1
37:    pB[idBitmap + 1] ← pB[idBitmap + 1] + 1
38: end if
39: return kl ∨ kr
```

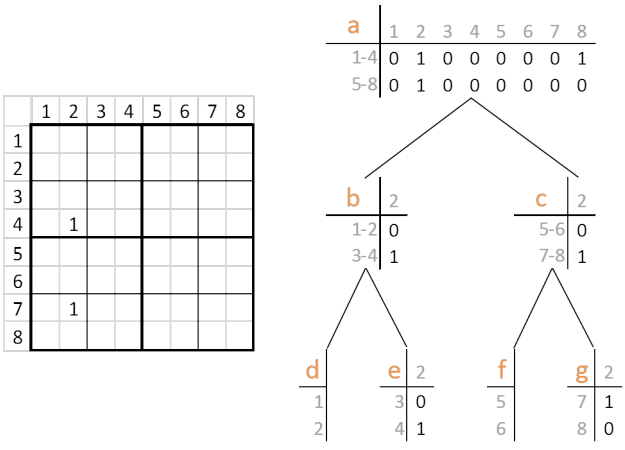


Figura 5.7: BRWT del resultado de la Operación de Intersección sobre los BRWT A (Figura 5.3) y B (Figura 5.4).

5.1.6. Diferencia Simétrica

El Algoritmo 5.1.5 de Diferencia Simétrica considera, para los nodos que no son hojas, los casos que se describen a continuación: *Caso 1*: Cuando ambos bits son 1 se exploran los nodos hijos con una llamada recursiva (Líneas 10 y 19), *Caso 2* si sólo uno de los bits es 1, entonces se realiza una llamada a la función ***Copy*** para hacer una copia de la columna que se procesa en la rama hijo desde A ó B hacia el resultado (Líneas 12, 14, 21 y 23), *Caso 3*: en otro caso el resultado parcial pasa a ser 0 (Líneas 16 y 25).

Si los bits analizados pertenecen a un nodo hoja, *Caso 4*, el resultado se calcula directamente con la operación *XOR* lógico sobre los bits que se analizan (Líneas 28 y 29).

Algorithm 5.1.5 RecSymmDiffBRWT$(A, B, pA, pB, rA, rB, R, idBitmap, idFirstLeaf)$

```
 1: idLeftChild ← (idBitmap + 1) * 2
 2: idRightChild ← (idBitmap + 2) * 2
 3: kl ← 1, kr ← 1
 4: bA1 ← A[pA[idBitmap]] ∧ rA
 5: bA2 ← A[pA[idBitmap + 1]] ∧ rA
 6: bB1 ← B[pB[idBitmap]] ∧ rB
 7: bB2 ← B[pB[idBitmap + 1]] ∧ rB
 8: if idBitmap < idFirstLeaf then
 9:     if bA1 ∧ bB1 then
10:         kl ← RecSymmDiffBRWT(A, B, pA, pB, bA1, bB1, R, idLeftChild, idFirstLeaf)
11:     else if bA1 then
12:         Copy(A, pA, R, idLeftChild, idFirstLeaf)
13:     else if bB1 then
14:         Copy(B, pB, R, idLeftChild, idFirstLeaf)
15:     else
16:         kl ← 0
17:     end if
18:     if bA2 ∧ bB2 then
19:         kr ← RecSymmDiffBRWT(A, B, pA, pB, bA2, bB2, idRightChild, idFirstLeaf)
20:     else if bA2 then
21:         Copy(A, pA, R, idRightChild, idFirstLeaf)
22:     else if bB2 then
23:         Copy(B, pB, R, idRightChild, idFirstLeaf)
24:     else
25:         kr ← 0
26:     end if
27: else
28:     kl ← (∼ bA1 ∧ bB1) ∨ (bA1 ∧ ∼ bB1)
29:     kr ← (∼ bA2 ∧ bB2) ∨ (bA2 ∧ ∼ bB2)
30: end if
31: if kl ∨ kr ∨ idBitmap == 0 then
32:     R[idBitmap] ← R[idBitmap]||kl
33:     R[idBitmap + 1] ← R[idBitmap + 1]||kr
34: else if rA ∨ idBitmap == 0 then
35:     pA[idBitmap] ← pA[idBitmap] + 1
36:     pA[idBitmap + 1] ← pA[idBitmap + 1] + 1
37: else if rB ∨ idBitmap == 0 then
38:     pB[idBitmap] ← pB[idBitmap] + 1
39:     pB[idBitmap + 1] ← pB[idBitmap + 1] + 1
40: end if
41: return kl ∨ kr
```

El resultado de la operación de Diferencia Simétrica sobre las relaciones de la Figura 5.3 como A y la Figura 5.4 como B se puede ver en La Figura 5.8, R según el Algoritmo 5.1.5. El algoritmo

comienza procesando la primera columna del nodo a en las relaciones A y B, donde para el primer sub-bitmap se da la combinación 1 (A) y 0 (B), y para el segundo sub-bitmap se da 0 (A) y 1 (B). En ambos casos se aplica el *Caso 2*; copiando los bits que corresponden a la columna 1 desde los nodos hijos de a. En el caso de la relación A se copian los elementos desde los nodos b y d, y desde la relación B se copian los bits de los nodos c y g. Para la segunda columna del nodo a, el bit del primer sub-bitmap en A es 1, y en B también es 1, y como indica el *Caso 1* se realiza la exploración en los nodos hijo (b) mediante una nueva llamada recursiva. En esta instancia, ambas relaciones poseen, en su columna 2, valores 0 en el primer sub-bitmap, y 1 en el segundo sub-bitmap, así es que se aplica nuevamente el *Caso 1* para los bits del segundo sub-bitmap, lo que hará que se analicen los nodos e en las relaciones A y B. Para este caso nuevamente coinciden los bits en ambos sub-bitmaps, pero en este caso el resultado se calcula según el *Caso 4* (e es un nodo hoja en las relaciones de entrada); se realiza la operación lógica *XOR* sobre los bits, obteniendo 0 para los dos sub-bitmaps. Por tanto, no se registran los resultados parciales en R y se retorna 0. En los nodos b, ambos resultados parciales corresponden a 0, por lo que también retorna 0, y en el nodo a se tiene que kl es 0, faltando procesar el segundo sub-bitmap de la columna 2. El resultado de la segunda columna en a es el mismo que ha sucedido con el segundo sub-bitmap, por tanto el resultado parcial kr también es 0, pero en este caso sí se registra el resultado en R porque se está procesando el nodo raíz de las relaciones. De esta forma, el algoritmo obtiene, luego de procesar las columnas pendientes de la raíz, el resultado de la operación de Diferencia Simétrica expuesto en la Figura 5.8.

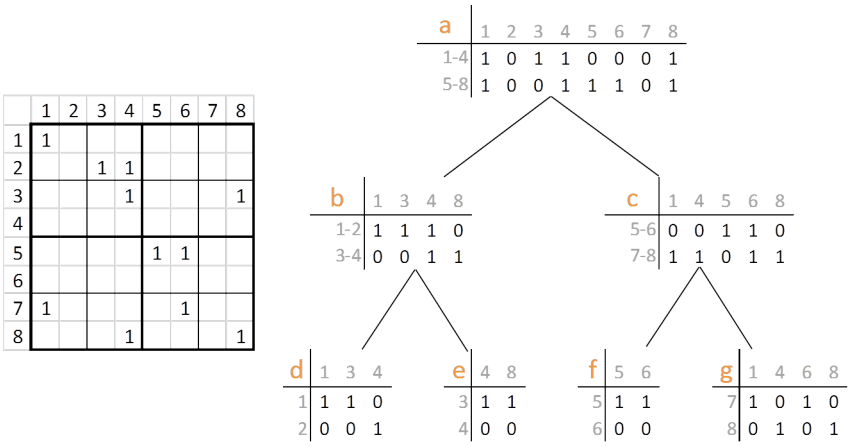


Figura 5.8: BRWT del resultado de la Operación de Diferencia Simétrica sobre los BRWT A (Figura 5.3) y B (Figura 5.4).

5.1.7. Complemento

A continuación el Algoritmo 5.1.6 realiza las llamadas al Algoritmo 5.1.7 para el cálculo del Complemento (Línea 6), siendo el segundo algoritmo el que realiza el proceso recursivo. En este caso el Algoritmo 5.1.6 es distinto del Algoritmo 5.1.2 en dos aspectos; *(1)* la entrada de datos sólo considera una relación binaria para esta operación, y *(2)* el Algoritmo 5.1.6 invoca la función recursiva tantas veces como objetos posee la representación (según el atributo *originalObject*),

lo cual no considera aquellos objetos generados al extender la relación a una potencia de 2, según la descripción de la Figura 5.1. Lo segundo evita que el proceso para calcular el complemento de una relación binaria genere relaciones entre aquellos elementos que originalmente no pertenecían a la relación binaria. Por lo tanto existe en el Algoritmo 5.1.6 un segundo ciclo que realiza la inserción de valores 0 que corresponden a esos objetos adicionales (sin relaciones) que fueron añadidos para que todas hojas de la estructura se ubiquen en el mismo nivel del BRWT.

Algorithm 5.1.6 ComplementBRWT(A)

```
1: pA ← positionsOfWT(A)
2: bA ← 1
3: idBitmap ← 0
4: idFirstLeaf ← firstLeaf(numOfObjects)
5: for i ← 0 ... originalObjects do
6:     RecComplementBRWT(A, pA, bA, R, idBitmap, idFirstLeaf)
7: end for
8: for i ← originalObjects ... numOfObjects do
9:     R[0] ← R[0]||0
10:    R[1] ← R[1]||0
11: end for
12: return R
```

Las variables que utiliza el Algoritmo 5.1.7 de Complemento son análogas a las utilizadas en los demás algoritmos que recorren el árbol en profundidad. Este algoritmo considera los siguientes casos generales para obtener el resultado de la operación de complemento sobre un relación binaria representada mediante BRWT; *Caso 1*: si el bit en proceso no está en un nodo hoja y su valor es 1 se realiza una llamada recursiva para procesar el nodo hijo (Líneas 8 y 13), *Caso 2*: si el bit es 0 y no es una hoja del BRWT, entonces se llama a la función ***FillIn*** la que inserta en la columna correspondiente bits con valor 1 en todos los nodos hijo del nodo actual (Líneas 10 y 15). *Case 3*: si el elemento está en un nodo hoja se aplica sobre el bit la operación lógica NOT (Líneas 19 y 22).

La Figura 5.9 muestra la matriz de adyacencia y el BRWT con el resultado de realizar la operación de Complemento sobre la relación binaria de la Figura 5.3 (A). Comenzando con el nodo a en el primer sub-bitmap la columna 1 posee valor 1 por lo que se realiza una llamada recursiva para que se explore la misma columna del nodo b, donde el bit del primer sub-bitmap también es 1, y como no es hoja se aplica nuevamente el *Caso 1*, donde esta nueva llamada procesará el nodo d de A. Dado que el nodo es una hoja se aplica el *Caso 3* para ambos bits de la columna 1, teniendo como resultados parciales 0 y 1, los que se registran en el nodo d del resultado, y se retorna 1, que se asigna a kl, y para el bit del segundo sub-bitmap en b se procede según el *Caso 2* llamando la función *FillIn* la que genera los bits de la columna 1 en el nodo e del resultado. Así es como la columna 1 en el nodo b registra 1 para ambos sub-bitmaps y retorna 1. Posteriormente, en el nodo a se procesa el bit del segundo sub-bitmap en la columna 1, y dado que su valor es 0, se procede con la llamada a *FillIn* (*Caso 2*) que generará los bits con valor 1 en las columnas 1 de los nodos c, f y g del resultado R. De esta manera, luego de procesar todos los bits de la relación A (Figura 5.3) se obtiene el resultado de la Figura 5.9.

Algorithm 5.1.7 RecComplementBRWT$(A, pA, bA, R, idBitmap, idFirstLeaf)$

```
1: idLeftChild ← (idBitmap + 1) * 2
2: idRightChild ← (idBitmap + 2) * 2
3: kl ← 0, kr ← 0
4: bA_1 ← A[pA[idBitmap]] ∧ bA
5: bA_2 ← A[pA[idBitmap + 1]] ∧ bAde
6: if idBitmap < idFirstLeaf then
7:     if bA_1 then
8:         kl ← RecComplementBRWT(A, pA, bA_1, R, idLeftChild, idFirstLeaf)
9:     else if idNode < originalObjects + idFirstLeaf then
10:        kl ← FillIn(A, pA, R, idLeftChild, idFirstLeaf)
11:    end if
12:    if bA_2 then
13:        kr ← RecComplementBRWT(A, pA, bA_2, idRightChild, idFirstLeaf)
14:    else if idNode + 1 < originalObjects + idFirstLeaf then
15:        kr ← FillIn(A, pA, R, idRightChild, idFirstLeaf)
16:    end if
17: else
18:    if idBitmap < originalObjects + idFirstLeaf then
19:        kl ←~ bA_1
20:    end if
21:    if idBitmap + 1 < originalObjects + idFirstLeaf then
22:        kr ←~ bA_2
23:    end if
24: end if
25: if kl ∨ kr ∨ idBitmap == 0 then
26:    R[idBitmap] ← R[idBitmap]||kl
27:    R[idBitmap + 1] ← R[idBitmap + 1]||kr
28: end if
29: if bA ∨ idBitmap == 0 then
30:    pA[idBitmap] ← pA[idBitmap] + 1
31:    pA[idBitmap + 1] ← pA[idBitmap + 1] + 1
32: end if
33: return kl ∨ kr
```

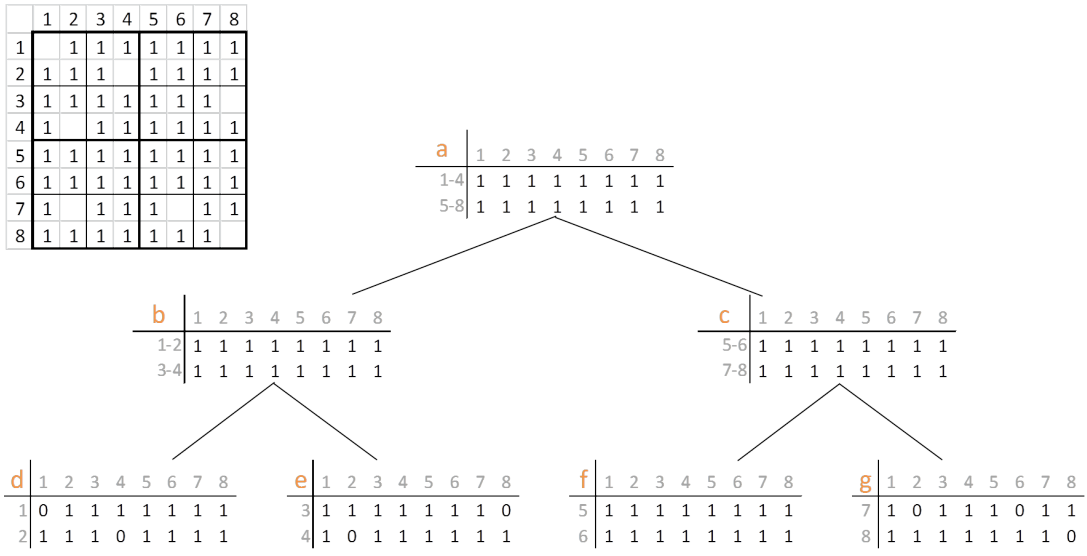


Figura 5.9: BRWT del resultado de la operación de Complemento sobre el BRWT de la Figura 5.3.

5.2. Experimentación

En esta sección se describe la evaluación experimental realizada sobre la implementación de los algoritmos antes descritos en el BRWT. Se describen en primer lugar, los conjuntos de datos utilizados para realizar los experimentos y, posteriormente, los diferentes procedimientos considerados para obtener los tiempos de ejecución de los algoritmos.

5.2.1. Datos Experimentales

Los datos utilizados para realizar las pruebas formales de los algoritmos se dividen en dos conjuntos reales, los cuales son `land-use` y `mini-snaps`, que se detallan a continuación.

land-use

Este conjunto de datos es el mismo que se describe en la sección 4.2.1 para el k^2-tree.

La Tabla 5.1 describe el tamaño de los ficheros para el conjunto de datos `land-use` que incluye la información de los ficheros BRWT.

	list	brwt	kt
`ag_1986`	17.725.072	1.326.236	818.544
`ag_2001`	13.781.972	1.013.588	626.596
`nf_1986`	23.973.972	1.718.128	1.073.284
`nf_2001`	16.137.572	1.241.816	762.280
`sc_1986`	10.593.772	846.848	510.160
`sc_2001`	11.475.072	916.768	553.516
`fp_1986`	10.441.172	806.460	491.984
`fp_2001`	22.762.672	1.678.944	1.044.488
`bg_1986`	4.706.372	387.600	226.680
`bg_2001`	3.283.072	273.188	159.712

Tabla 5.1: Tamaño de los ficheros para el conjunto de datos `land-use` en Bytes.

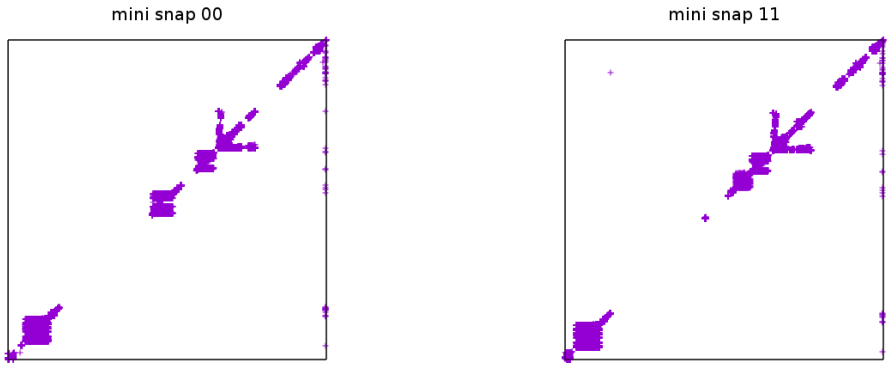
mini-snaps

Este conjunto de datos corresponde a una porción de cada elemento del conjunto de datos descrito en la sección 4.2.1, donde se consideraron sólo los primeros 5.000 objetos, por lo que el total de relaciones posibles para este conjunto de datos es de 25 millones (5 mil $\times$ 5 mil). A continuación, la Tabla 5.2 describe las propiedades de este conjunto de datos.

Así también se muestra una distribución de las relaciones existentes en el conjunto de datos `mini-snaps` en la Figura 5.10.

	cantidad m	densidad $m/n^2 \times 100$	list (bytes)	kt (bytes)	brwt (bytes)
`mini_snap00`	26.848	0,107 %	127.404	10.816	17.684
`mini_snap01`	14.601	0,058 %	78.416	6.584	11.116
`mini_snap02`	27.980	0,112 %	131.932	11.032	18.024
`mini_snap03`	21.407	0,086 %	105.640	9.116	15.552
`mini_snap04`	21.831	0,087 %	107.336	9.252	16.004
`mini_snap05`	21.924	0,088 %	107.708	9.604	16.364
`mini_snap06`	27.432	0,110 %	129.740	10.668	17.336
`mini_snap07`	28.505	0,114 %	134.032	11.168	18.408
`mini_snap08`	29.432	0,118 %	137.740	12.348	20.348
`mini_snap09`	17.554	0,070 %	90.228	6.648	10.912
`mini_snap10`	25.116	0,100 %	120.476	10.120	17.072
`mini_snap11`	25.443	0,102 %	121.784	10.624	17.296

Tabla 5.2: Propiedades del conjunto de datos `mini-snaps`.



(a) `mini_snap00` (b) `mini_snap11`

Figura 5.10: Ejemplo de dos distribuciones con el conjunto de datos `mini-snaps`.

5.2.2. Líneas de Comparación

Para los algoritmos implementados sobre el BRWT se utiliza como única línea de comparación el tiempo de ejecución de la misma operación sobre ficheros de tipo k^2-tree.

En este caso sólo se considera el tiempo de ejecución que le toma a la operación calcular el resultado una vez que se tienen en memoria principal las representaciones en el formato que corresponda (*kt* o *brwt*). No se considera el tiempo de carga hacia memoria principal, ni el tiempo que toma generar en memoria secundaria el fichero resultante.

5.2.3. Entorno de Pruebas

Las pruebas se realizaron sobre un equipo con procesador Intel® Core i7-4510U de 2.00GHz de frecuencia. Memoria RAM DDR3 con 8GB de capacidad y 1600MHz de frecuencia. El sistema operativo es Fedora 23 de 64 bits. Los algoritmos se implementan sobre el lenguaje de programación C y compilados sobre gcc versión 5.3.1.

Los valores presentados a continuación corresponden al promedio de 20 ejecuciones de la misma operación.

5.3. Resultados

A continuación se presentan los resultados de los experimentos realizados para los algoritmos implementados sobre el BRWT. En este caso se presentan las operaciones de Unión, Diferencia, Intersección, Diferencia Simétrica y Complemento para ambos conjuntos de datos.

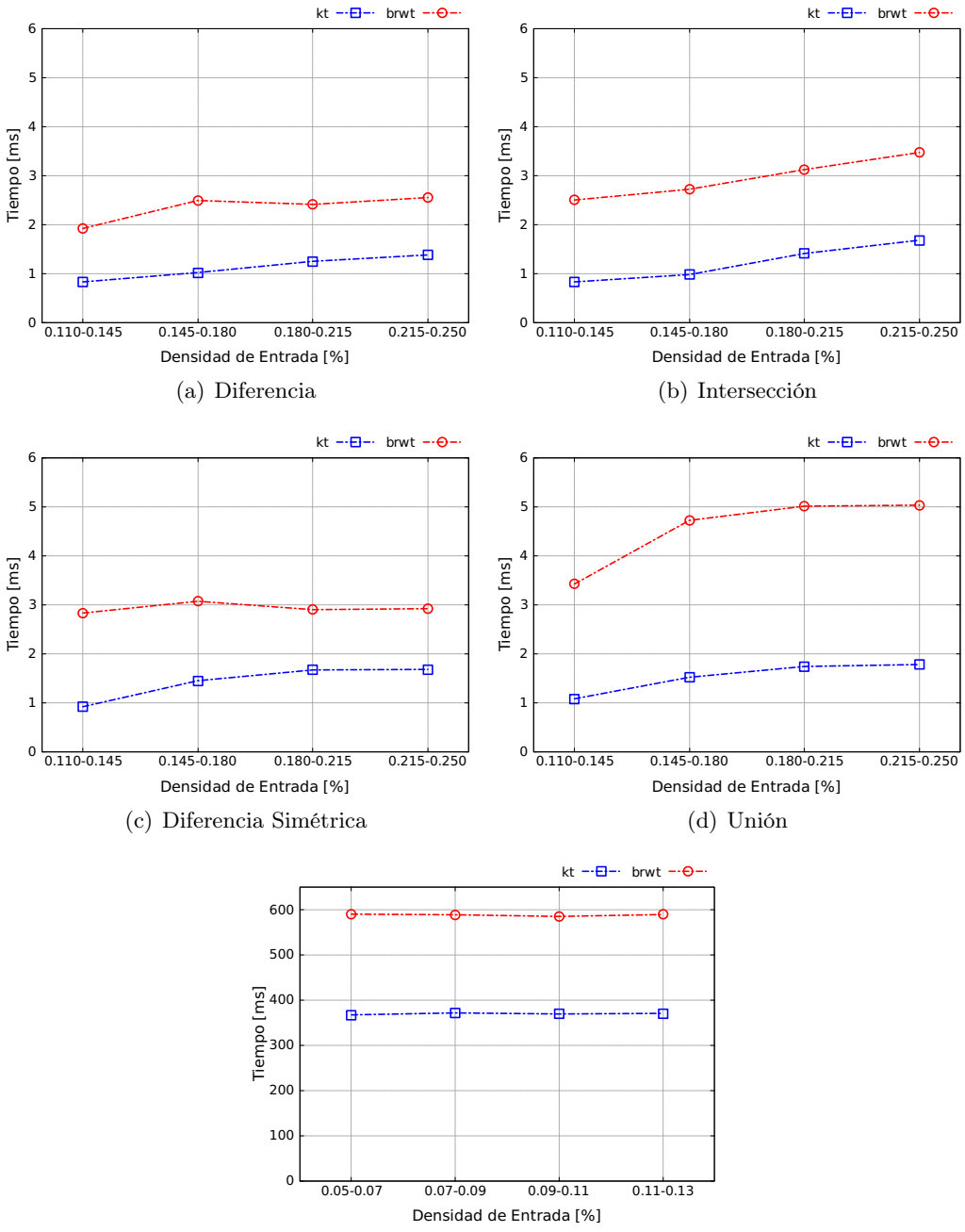
Los criterios a evaluar son la densidad de entrada y la densidad de salida explicadas en las secciones 4.3.3 y 4.3.4 respectivamente.

5.3.1. Densidad de Entrada

A continuación se presentan gráficas con los resultados de las operaciones según el criterio de Densidad de Entrada.

La Figura 5.11 muestra que el tiempo de ejecución de las operaciones implementadas sobre el BRWT se incrementa a medida que aumenta la densidad de entrada para todas las operaciones de Unión, Diferencia e Intersección. En el caso de la Diferencia Simétrica y del Complemento los tiempos para el conjunto de datos `mini-snaps` tienden a mantenerse con muy baja variación. Respecto de la proporción de tiempo en relación al k^2-tree se mantiene entre 2 a 3 veces el tiempo.

En la Figura 5.12 se pueden ver comportamientos similares respecto del tiempo en las operaciones para el conjunto de datos `land-use`. Para la operación de Complemento la diferencia entre los tiempos del BRWT y el k^2-tree se mantiene al aumentar la densidad de entrada. Sólo al principio se ve una leve disminución en la diferencia de tiempos de ejecución. Para el resto de las operaciones la diferencia de tiempo entre las estructuras va en aumento a medida que se incrementa la densidad de entrada. Los tiempos del BRWT son siempre superiores, siendo la mayor diferencia para la operación de Unión, donde hay poco más de 2 veces del tiempo que le toma al k^2-tree en una densidad de entrada del 32 a 41 %.



(a) Diferencia

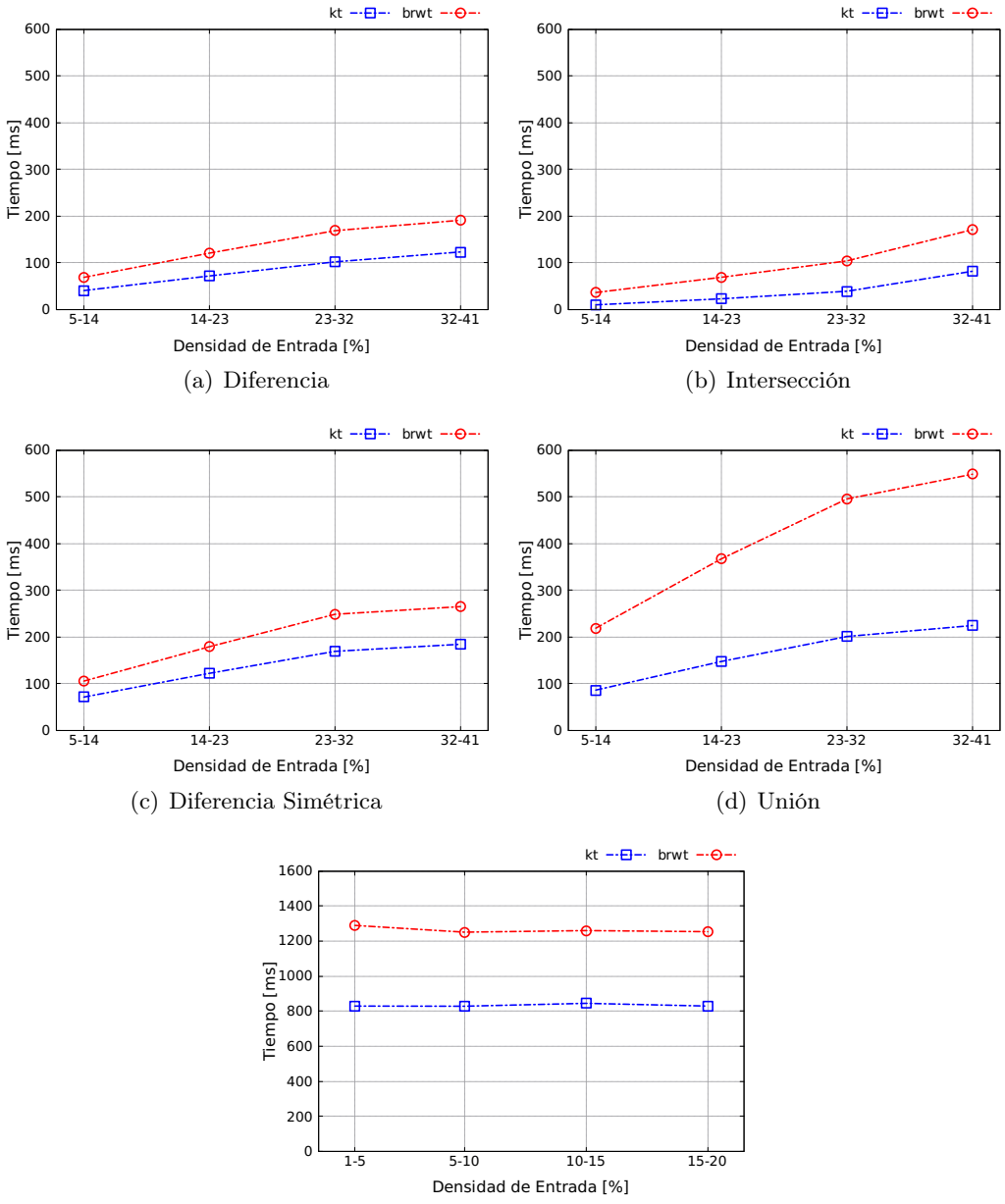
(b) Intersección

(c) Diferencia Simétrica

(d) Unión

(e) Complemento

Figura 5.11: Tiempos obtenidos sobre el conjunto de datos `mini-snaps`. Se utiliza el criterio de Densidad de Entrada en el eje X.



(a) Diferencia

(b) Intersección

(c) Diferencia Simétrica

(d) Unión

(e) Complemento

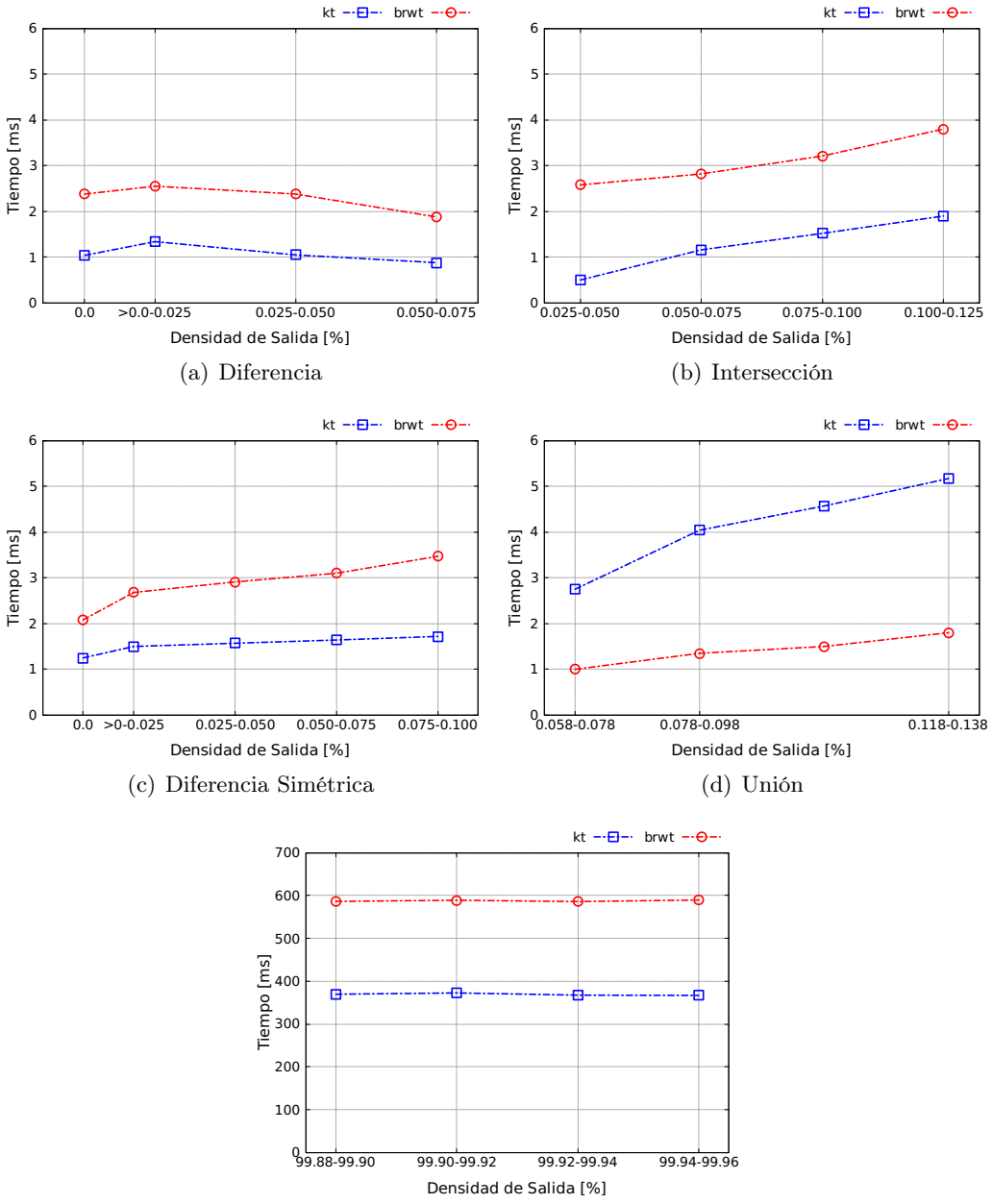
Figura 5.12: Tiempos obtenidos sobre el conjunto de datos `land-use`. Se utiliza el criterio de Densidad de Entrada en el eje X.

5.3.2. Densidad de Salida

En esta sección se presentan los resultados experimentales de los algoritmos considerando el criterio de Densidad de Salida detallado en 4.3.4.

Para el conjunto de datos `mini-snaps` (Figura 5.13) los tiempos del BRWT siempre son mayores que los tiempos del k^2-tree, siendo hasta en 3 veces mayor en la operación de Unión. Para la operación de Complemento los tiempos se mantienen casi constantes en ambos casos. Podemos ver en la Figura 5.13 que los algoritmos presentan un comportamiento similar en ambas EDC frente al criterio de Densidad de Salida. Podemos ver que las operaciones de Intersección, Diferencia Simétrica y de Unión aumentan los tiempos de operación a medida que se incrementa la densidad de Salida, y la Operación de Diferencia tiende a disminuir los tiempos a medida que incrementa la densidad de salida.

La Figura 5.14 muestra el comportamiento de los algoritmos para el BRWT y el k^2-tree. El comportamiento para el Complemento, en ambos casos tiende a mantenerse constante. La operación de Diferencia muestra un incremento en bajas densidades, pero posteriormente se mantiene casi constante para densidades de salida superiores a 1,5 %. En las demás operaciones (Intersección, Diferencia Simétrica y Unión) se ve un incremento constante de los tiempos a medida que se aumenta la Densidad de Entrada. Respecto de las diferencias de tiempo, el BRWT siempre presenta mayores tiempos que el k^2-tree. Para la Diferencia, Intersección y el Complemento la diferencia de tiempo tiende a mantenerse, o varía muy poco. Para la Diferencia Simétrica la diferencia entre los tiempos aumenta de manera paulatina, en cambio para la operación de Unión la diferencia entre los tiempos se incrementa de manera notoria según la gráfica.



(a) Diferencia

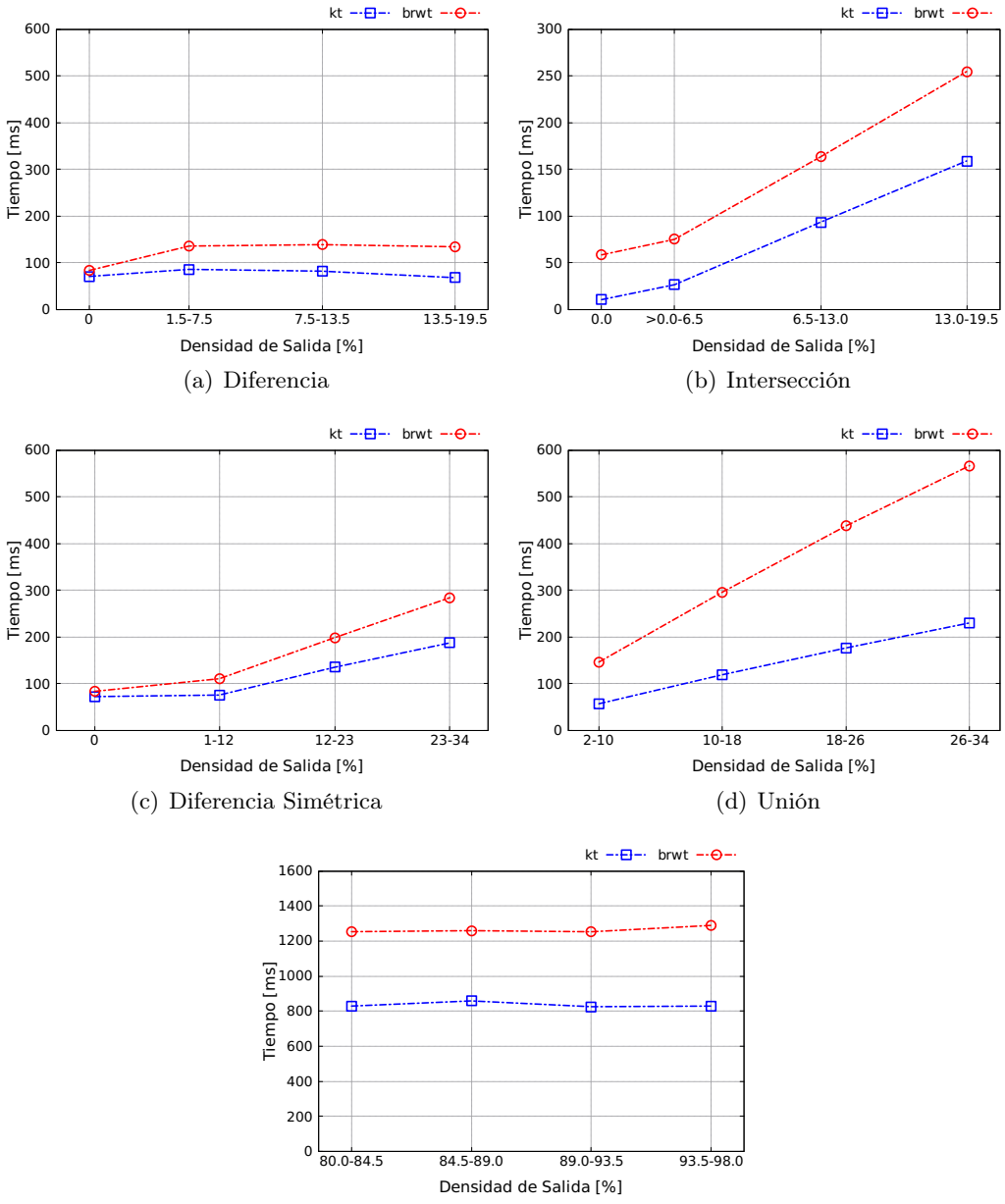
(b) Intersección

(c) Diferencia Simétrica

(d) Unión

(e) Complemento

Figura 5.13: Tiempos obtenidos sobre el conjunto de datos de datos `mini-snaps`. Se utiliza el criterio de Densidad de Salida en el eje X.



(a) Diferencia

(b) Intersección

(c) Diferencia Simétrica

(d) Unión

(e) Complemento

Figura 5.14: Tiempos obtenidos sobre el conjunto de datos `land-use`. Se utiliza el criterio de Densidad de Salida en el eje X.

5.4. Experimentación sobre el Espacio de Memoria Utilizado

Uno de los principales elementos a considerar cuando se habla de EDC es la cantidad de memoria que requiere una estructura para su funcionamiento y operación. Las implementaciones de operaciones de conjunto sobre EDC que se han desarrollado han mostrado un buen desempeño respecto del tiempo de ejecución frente a los escenarios en que han sido evaluados en este trabajo de tesis de magíster.

A continuación se evalúa el rendimiento respecto del espacio requerido en memoria principal para la ejecución de las implementaciones antes descritas. Dicha evaluación utiliza los mismos conjuntos de datos descritos en las Secciones 5.2 y 4.2, y se realiza sobre el BRWT (*brwt*) y el k^2-tree (*kt*). Se muestra en las gráficas la memoria requerida por las implementaciones para obtener el resultado utilizando como línea de comparación una implementación que calcula las operaciones sobre listas de adyacencia (*list*). Se hace uso de esta línea de comparación debido a que cualquier implementación de una EDC que no cuente con las operaciones de conjunto deberá descompactar las representaciones para operarlas sobre listas de adyacencia y luego compactar el resultado, por lo que *list* representa una cota inferior en cuanto al uso de memoria requerida para una EDC sin las operaciones de conjunto en su implementación.

La evaluación se realiza considerando las operaciones de Unión, Diferencia, Intersección y Diferencia Simétrica sobre los conjuntos de datos `mini-snaps` y `land-use` para el BRWT y el k^2-tree. Así también se incluyen los resultados de las mismas operaciones para el conjunto de datos `uk-snaps` sobre la implementación del k^2-tree.

Los valores utilizados para generar las gráficas de memoria fueron obtenidos al ejecutar las operaciones con el comando *time* de la versión 1.7 de **GNU**. Este comando entrega, además del tiempo de ejecución de un proceso, distintos detalles relativos al uso de memoria. En este caso se ha utilizado el valor máximo de la memoria utilizada por el proceso durante su ejecución.

Por cada operación en un conjunto de datos, se realizó la ejecución de cada combinación posible entre las representaciones que conforman un conjunto (Por ejemplo, para `land-use` existen 10 representaciones, por lo que se pueden generar 100 combinaciones posibles). De los valores obtenidos en todas esas combinaciones las gráficas presentan el promedio, valor máximo y valor mínimo.

En las gráficas no incluyen los resultados del Complemento ya que sus valores están muy por encima de las demás operaciones, pero en general los resultados son muy similares a las demás operaciones recursivas en los tres conjuntos de datos.

5.4.1. Resultados del Uso de Memoria

La Figura 5.15 muestra los resultados de la memoria utilizada por las implementaciones en el conjunto de datos `mini-snaps`. Para la operación de Unión ([U]) existe muy poca diferencia entre la memoria que se utiliza para las implementaciones *list* y *brwt*, en cambio *kt* requiere de menos memoria. Respecto de las operaciones recursivas (Diferencia, Intersección y Diferencia Simétrica) existe una desventaja evidente hacia el *brwt* que sobrepasa por mucho a las otras dos. En estas operaciones recursivas la implementación *kt* también utiliza menos memoria de lo que requiere *list*, siendo muy similares los resultados en los tres casos. Respecto de los mínimos y máximos

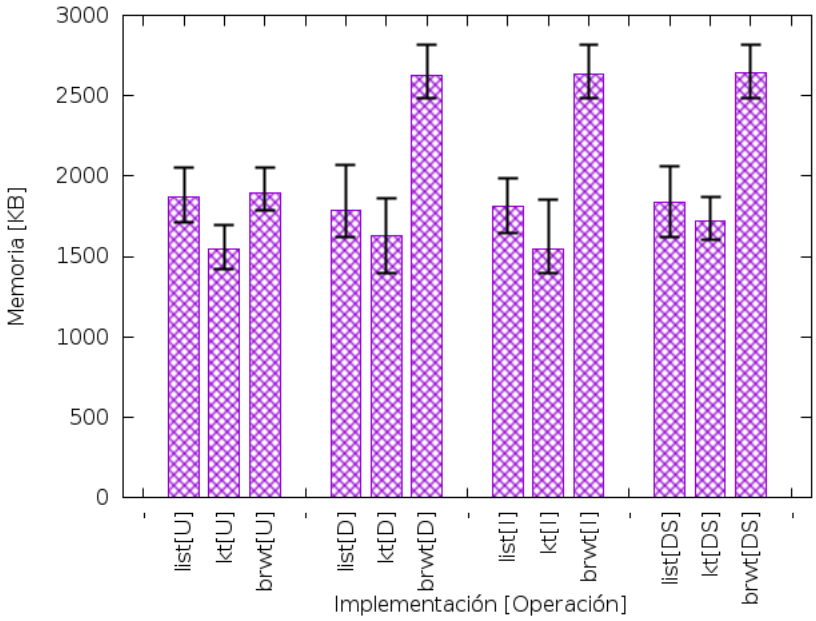


Figura 5.15: Memoria utilizada en las operaciones de Unión [U], Diferencia [D], Intersección [I] y Diferencia Simétrica [DS] para cada línea de comparación (*kt*, *list* y *brwt*) en el conjunto de datos `mini-snaps`.

de los resultados por cada operación, se ve una similitud en el rango que presentan todas las operaciones.

La Figura 5.16 presenta los resultados del uso de memoria en el conjunto de datos `land-use`, donde en general el promedio de espacio de *kt* y *brwt* es notoriamente inferior que el promedio de *list*. En este caso la operación de Unión presenta resultados muy favorables para *kt* y *brwt* frente al proceso *list*. En las operaciones de Diferencia, Intersección y Diferencia Simétrica la implementación *kt* es la que menos memoria requiere, le sigue de cerca *brwt* y posteriormente *list* presenta requerimientos varias veces más que las otras dos implementaciones. Otro aspecto que se puede resaltar es que los rangos entre el mínimo y máximo de memoria que se requiere para las operaciones de la implementación *list* es bastante amplio, a diferencia de los rangos para las EDC.

En general, se puede apreciar en ambas gráficas una ventaja para la implementación *kt*. La implementación *list* presenta un rendimiento cercano a *kt* en el set de datos `mini-snaps` para todas las operaciones (Figura 5.15), en cambio, para el set de datos `land-use` presenta una eficiencia muy inferior en cuanto al uso de memoria que las otras dos implementaciones. Para la implementación del *brwt* existe un buen rendimiento, muy cercano a *kt*, en el set `land-use`, en cambio en las operaciones recursivas de `mini-snaps` el *brwt* presenta un mayor uso de memoria que las otras dos implementaciones. En este mismo set de datos, *brwt* no muestra una gran diferencia en la operación de Unión, lo que se puede dar debido a que en las operaciones recursivas la implementación del *brwt* requiere de una estructura adicional muy grande para calcular las

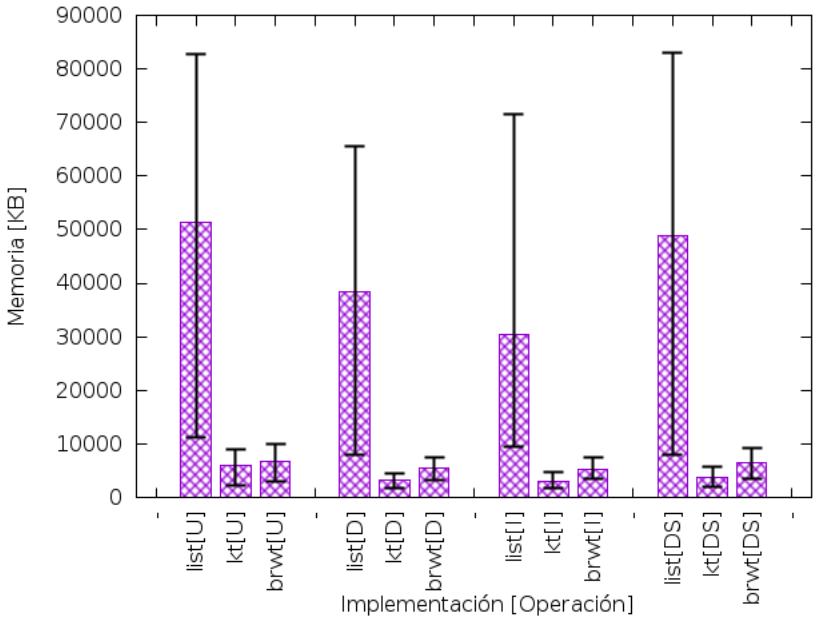


Figura 5.16: Memoria utilizada en las operaciones de Unión [U], Diferencia [D], Intersección [I] y Diferencia Simétrica [DS] para cada línea de comparación (*kt*, *list* y *brwt*) en el conjunto de datos `land-use`.

operaciones recursivas (el arreglo que almacena las posiciones de cada nodo del *BRWT*).

Adicionalmente la Figura 5.17 muestra el rendimiento en el uso de memoria para las implementaciones *list* y *kt* ejecutadas sobre el set de datos `uk-snaps` descrito en la Sección 4.2. De manera general se puede apreciar que para todas las operaciones la implementación *list* utiliza al rededor de cuatro veces más memoria que *kt* para resolver todas las operaciones. En particular, este conjunto de datos presenta una densidad siempre inferior al 0,0004 % (ver Tabla 4.2) y con 1 billón de posibles relaciones.

Con todo lo anterior se puede indicar que en un conjuntos de datos grande (`uk-snaps`), las operaciones implementadas sobre el k^2-tree presenta muy buenos resultados por sobre la implementación en listas de adyacencia. En un set de datos pequeño pero muy denso (`land-use`) el BRWT y el k^2-tree presentan muy buen desempeño en comparación a las listas de adyacencia. Pero para un set de datos pequeño y poco denso (`mini-snaps`) las implementaciones en EDC presentan poca diferencia contra una implementación en listas de adyacencia, incluso se da que BRWT presenta peores resultados.

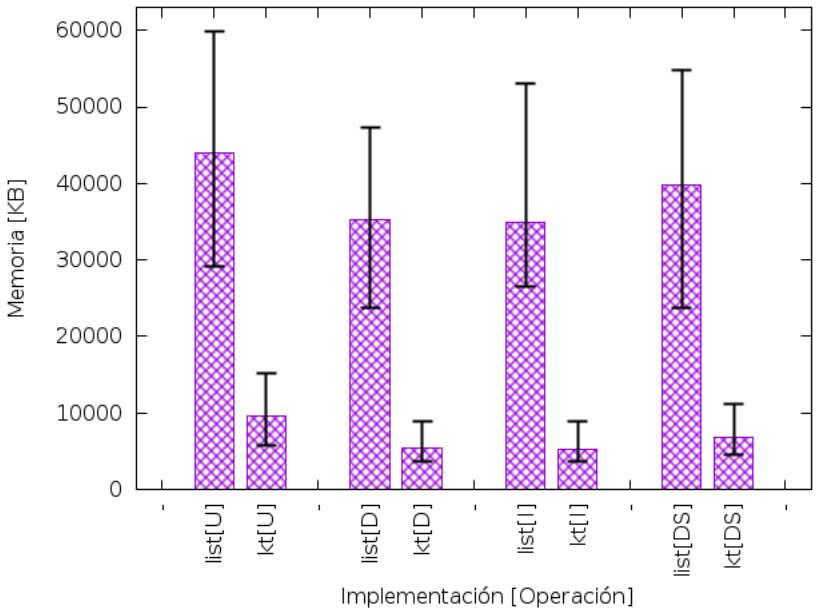


Figura 5.17: Memoria utilizada en las operaciones de Unión [U], Diferencia [D], Intersección [I] y Diferencia Simétrica [DS] para las líneas de comparación *kt* y *list* en el conjunto de datos `uk-snaps`.

Parte IV

Conclusiones y Trabajo Futuro

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

Esta tesis de magíster presenta, en primer lugar, la implementación de algoritmos para resolver operaciones de conjunto de relaciones binarias (Unión, Diferencia, Intersección, Diferencia Simétrica y Complemento) sobre el k^2-tree, los que fueron propuestos anteriormente por Brisaboa *et al.* [7]. Luego se han expuesto también los algoritmos para las mismas operaciones que se diseñaron e implementaron sobre el BRWT, estructura propuesta por Barbay, Claude y Navarro [1].

Todas los algoritmos se implementaron en el lenguaje C, lo que incluyó los algoritmos para operaciones de conjunto en el k^2-tree, una implementación básica del BRWT y la implementación de los algoritmos de operaciones de conjunto sobre ésta última EDC.

Así también se presentan en este trabajo los resultados experimentales de ambas implementaciones, realizadas sobre conjuntos de datos reales correspondientes a la representación de un grafo web y un ráster para representar el uso de suelos. En dicha experimentación se evaluó el tiempo de ejecución y el espacio de memoria necesarios para realizar operaciones de conjunto sobre relaciones binarias representadas en estas EDC.

En la evaluación de tiempo, los resultados muestran que el k^2-tree incluso llega a ser un orden de magnitud mejor que el caso base considerado; realizar la operación sobre una implementación de la EDC que no dispone de las operaciones de conjunto, por lo que debe descompactar los datos, obtener el resultado y compactar la solución a la EDC de origen.

Los resultados relativos al espacio de almacenamiento muestran que los algoritmos implementados requieren una menor cantidad de memoria para ejecutar las operaciones de conjunto de lo que se necesita al no disponer de las operaciones de conjunto en la implementación de la EDC.

Por lo anteriormente expuesto en esta tesis de magíster se puede concluir que es posible implementar algoritmos eficientes para realizar operaciones de conjunto sobre relaciones binarias representadas mediante EDC. Esta eficiencia se da tanto en términos del tiempo de ejecución, como del espacio de memoria necesario para su ejecución.

6.2. Trabajo Futuro

Considerando el trabajo realizado en esta tesis de magíster, se plantea como posible trabajo futuro:

- El diseño e implementación de operaciones de conjunto para otras variantes de las EDC utilizadas en esta tesis. Existen variantes de las EDC utilizadas en esta tesis de magíster que podrían extender sus capacidades con la implementación de operaciones de conjunto sobre ellas, como por ejemplo el k^2-tree que compacta zonas de 0s y 1s.
- Dado que se ha trabajado con relaciones binarias, se pueden desarrollar algoritmos que comprueben ciertas propiedades de las relaciones almacenadas en EDC, como por ejemplo, si son transitivas, simétricas, reflexivas, etc. Así también se podría calcular, por ejemplo, el cierre transitivo de una relación.
- También resultaría muy interesante el explorar la alternativa de paralelizar los algoritmos con el propósito de reducir los tiempos de ejecución.

Bibliografía

[1] Barbay, J., Claude, F., Navarro, G.: Compact binary relation representations with rich functionality. Information and Computation 232, 19–37 (2013)

[2] Blandford, D.K.: Compact Data Structures with Fast Queries. Ph.D. thesis, PhD thesis, School of Computer Science, Carnegie Mellon University, 2006. Also as Tech. Report CMU-CS-05-196 (2006)

[3] Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. Proceedings of the 20th international conference on World Wide Web pp. 587–596 (2010)

[4] Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: Srinivasan, S., Ramamritham, K., Kumar, A., Ravindra, M.P., Bertino, E., Kumar, R. (eds.) Proceedings of the 20th international conference on World Wide Web. pp. 587–596. ACM Press (2011)

[5] Boldi, P., Santini, M., Vigna, S.: A large time-aware graph. SIGIR Forum 42(2), 33–38 (2008)

[6] Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: Proc. of the Thirteenth International World Wide Web Conference (WWW 2004). pp. 595–601. ACM Press, Manhattan, USA (2004)

[7] Brisaboa, N.R., de Bernardo, G., Gutiérrez, G., Ladra, S., Penabad, M.R., Troncoso, B.A.: Efficient set operations over k2-trees. In: Data Compression Conference (DCC), 2015. pp. 373–382. IEEE (2015)

[8] Brisaboa, N.R., Cánovas, R., Claude, F., Martínez-Prieto, M.A., Navarro, G.: Compressed string dictionaries. In: Experimental Algorithms, pp. 136–147. Springer (2011)

[9] Brisaboa, N.R., Cerdeira-Pena, A., de Bernardo, G., Navarro, G.: Compressed representation of dynamic binary relations with applications. Information Systems 69, 106–123 (2017)

[10] Brisaboa, N.R., Ladra, S., Navarro, G.: k^2-trees for compact web graph representation. In: International Symposium on String Processing and Information Retrieval. pp. 18–30. Springer (2009)

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

[12] Casey, W., Shelmire, A.: Signature limits: an entire map of clone features and their discovery in nearly linear time. In: Malicious and Unwanted Software (MALWARE), 2016 11th International Conference on. pp. 1–10. IEEE (2016)

[13] Claude, F., Ladra, S.: Practical representations for web and social graphs. In: Proceedings of the 20th ACM international conference on Information and knowledge management. pp. 1185–1190. ACM (2011)

[14] Claude, F., Navarro, G.: Extended compact web graph representations. Algorithms and Applications 6060, 77–91 (2010)

[15] Costa, L.B., Al-Kiswany, S., Ripeanu, M.: Gpu support for batch oriented workloads. In: Performance Computing and Communications Conference (IPCCC), 2009 IEEE 28th International. pp. 231–238. IEEE (2009)

[16] De Bernardo, G., Álvarez-García, S., Brisaboa, N.R., Navarro, G., Pedreira, O.: Compact querieable representations of raster data pp. 96–108 (2013)

[17] Denzumi, S., Kawahara, J., Tsuda, K., Arimura, H., Minato, S.i., Sadakane, K.: DenseZDD: A Compact and Fast Index for Families of Sets, pp. 187–198. Springer International Publishing, Cham (2014)

[18] Faust, C.: Estructuras comprimidas para grafos de la Web. Ph.D. thesis (2008)

[19] Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes pp. 841–850 (2003)

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

[21] Larsson, N.J., Moffat, A.: Off-line dictionary-based compression. Proceedings of the IEEE 88(11), 1722–1732 (2000)

[22] Lin, T.W.: Set operations on constant bit-length linear quadtrees. Pattern Recognition 30(7), 1239–1249 (1997)

[23] Minato, S.i., Arimura, H.: Efficient method of combinatorial item set analysis based on zero-suppressed bdds pp. 4–11 (2005)

[24] Navarro, G.: Compact Data Structures. Cambridge University Press (2016)

[25] Piskac, R., de Moura, L., Bjørner, N.: Deciding effectively propositional logic using DPLL and substitution sets. Journal of Automated Reasoning 44(4), 401–424 (2010)

[26] Rossignac, J.R., Requicha, A.A.: Solid modeling. Tech. rep., Georgia Institute of Technology (1999)

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

[28] Samet, H.: An overview of quadtrees, octrees, and related hierarchical data structures. NATO ASI Series 40, 51–68 (1988)

[29] Shannon, C.E.: A mathematical theory of communication. ACM SIGMOBILE Mobile Computing and Communications Review 5(1), 3–55 (2001)