Universidad del Bío-Bío
Facultad de Ciencias Empresariales
Escuela de Ingeniería Civil en Informática

“Implementación de operaciones binarias y de consulta sobre un k^2-tree”

Proyecto de título para optar al Título de Ingeniero Civil en Informática

Tamara Alejandra Vivanco Neira - Gonzalo Antonio Candia Yáñez
Profesora Guía: Brunny Angélica Troncoso Pantoja
Febrero, 2015
Concepción, Chile

Agradecimientos

En primer lugar agradecemos el tiempo, esfuerzo y dedicación empleado por nuestra profesora guía, Brunny Troncoso Pantoja, así como también su sinceridad al comunicarnos nuestros aciertos y errores, y por depositar su confianza en nosotros hasta el último día. Sin su ayuda este proyecto de título no hubiera sido posible.

Además, agradecemos la ayuda indirecta de parte de la doctora Susana Ladra y el doctor Gilberto Gutiérrez, así como también a los compañeros y amigos que respondieron nuestras dudas y consultas: sus aportes e ideas facilitaron el desarrollo de este proyecto de título.

No es posible dejar de agradecer el apoyo constante brindado por nuestras familias en el camino que hemos recorrido desde hace 5 años y que ahora finaliza.

Y finalmente, a nuestros amigos por darnos apoyo durante estos meses, y subirnos el ánimo cada vez que fue necesario. Los estimamos a todos.

¡Muchísimas Gracias!

Tamara Vivanco y Gonzalo Candia.

Resumen

En la actualidad la web se expande prácticamente de forma descontrolada, por lo que su tamaño y alcance es cada vez más difícil de cuantificar. Algunos análisis del comportamiento de la web necesitan almacenar información sobre la existencia de un vínculo entre una página y otra. De forma natural, surge la idea de representar dichos vínculos a través de líneas o aristas; es por esto que se utilizan grafos. Es difícil dimensionar el tamaño de un grafo web que represente todas conexiones de todas las páginas existentes, más aún imaginar esa información almacenada en algún lugar. Lo lógico sería buscar la forma de reducir el tamaño de los archivos donde se almacenen, como por ejemplo comprimiendo esta información, pero conservando la funcionalidad de la estructura de datos original.

El k^2-tree representa de forma compacta la matriz de adyacencia de un grafo web dejando fuera las grandes áreas vacías (donde no existe ninguna conexión). Al dejar de lado estas áreas no se pierde la capacidad de navegación a través de los nodos. Sobre las matrices de adyacencia es posible realizar operaciones binarias como lo son el complemento, la diferencia, la unión y la intersección.

La finalidad de este proyecto en particular es implementar estas mismas operaciones sobre k^2-tree, para luego confrontar la implementación original y la comprimida en términos de desempeño, tiempo de respuesta, entre otros.

Tabla de contenido

Índice de figuras

Índice de tablas

1. Introducción

El propósito de este documento es presentar al lector el proceso para la implementación de operaciones binarias y de consulta sobre un k^2-tree. A continuación se detallan las secciones en las que se ha dividido el documento.

En el capítulo dos, se presentan los antecedentes generales para introducir a la base teórica del k^2-tree, sobre la cual se sustenta el desarrollo posterior de este documento, así como destacar las áreas en las que se aporte conocimiento.

Posteriormente, en el capítulo tres se definen los objetivos del proyecto, tanto los generales como los específicos. Además, se presentan las definiciones, siglas y acrónimos que se utilizan durante todo el documento.

El capítulo cuatro explica aspectos relativos a la construcción, tanto del k^2-tree como de los algoritmos. Se detallan el procedimiento, las formas de entrada y salida, y las estructuras de datos utilizadas.

El capítulo cinco explica el proyecto propiamente tal, se detalla el funcionamiento de cada algoritmo implementado, se presentan ejemplos gráficos con sus respectivas trazas de ejecución para facilitar la comprensión por parte del lector. De la misma forma, se documentan los datos utilizados para los casos de prueba.

En el capítulo sexto se expone la estructura de datos original, es decir, la matriz de adyacencia sin compresión y se detalla el funcionamiento de las operaciones binarias sobre estas, para la futura comparación.

En el capítulo siete se presentan los casos de pruebas planteados para realizar la evaluación de los algoritmos implementados. Las pruebas se realizan sobre la estructura de datos original y la estructura comprimida para el posterior análisis.

Posteriormente, en el capítulo ocho, se realiza un análisis de los resultados obtenidos en el capítulo anterior. Este análisis consiste en una evaluación crítica y una explicación de los resultados obtenidos por la implementación del k^2-tree y se comparan estos con los datos obtenidos por la implementación original.

Finalmente, en el capítulo nueve se plantean ideas de trabajos futuros relacionados con el k^2-tree y se presentan las conclusiones obtenidas y se sintetiza la totalidad de los planteamientos realizados en la extensión del proyecto.

2. Antecedentes Generales

En los últimos años, la Web ha cobrado mayor importancia, convirtiéndose de esta manera en un foco de intensa actividad de investigación, realizada tanto por académicos, como centros de investigación industrial. Esta actividad se enfoca principalmente en el desarrollo de técnicas eficientes para recuperar información a través de Internet, utilizando algún tipo de exploración o búsqueda que está especialmente adaptado a la estructura específica de la Web. Estas técnicas poseen muchas aplicaciones potenciales y reales, por ejemplo, en motores de búsqueda, en el diseño de los rastreadores eficaces, en la determinación de cibercomunidades, entre otras.

Un grafo web es un grafo dirigido cuyos vértices corresponden a las páginas de la World Wide Web, y donde una arista dirigida conecta una página X a una página Y si es que existe un hipervínculo en la página X que posea referencia a la página Y. Entre las utilidades de este tipo de grafos se encuentran las tareas de análisis de comportamiento de la web, como por ejemplo el cálculo de *PageRank*, algoritmo utilizado por Google Search para clasificar los sitios web en los resultados de las búsquedas. Según Claude [3], para una representación plana de la Web pública y estática se requerirían más de 600 GB, lo que justifica la utilización de estructuras compactas para la representación y manipulación de su información en memoria principal.

En la Figura 1 se observa un ejemplo de grafo web de 50 nodos y deja en evidencia el número de conexiones que existen en esa pequeña cantidad de nodos. El crecimiento de la WWW es exponencial, por lo tanto la cantidad de nodos del grafo web que la representa también lo es. Almacenar este volumen de información sigue siendo un problema a pesar de que cada vez existan dispositivos con mayor capacidad de almacenamiento.

Figura 1: Grafo web de 50 nodos.

En total, según estimaciones de 2014, el número total de páginas web es de más de 45.000 millones [4].

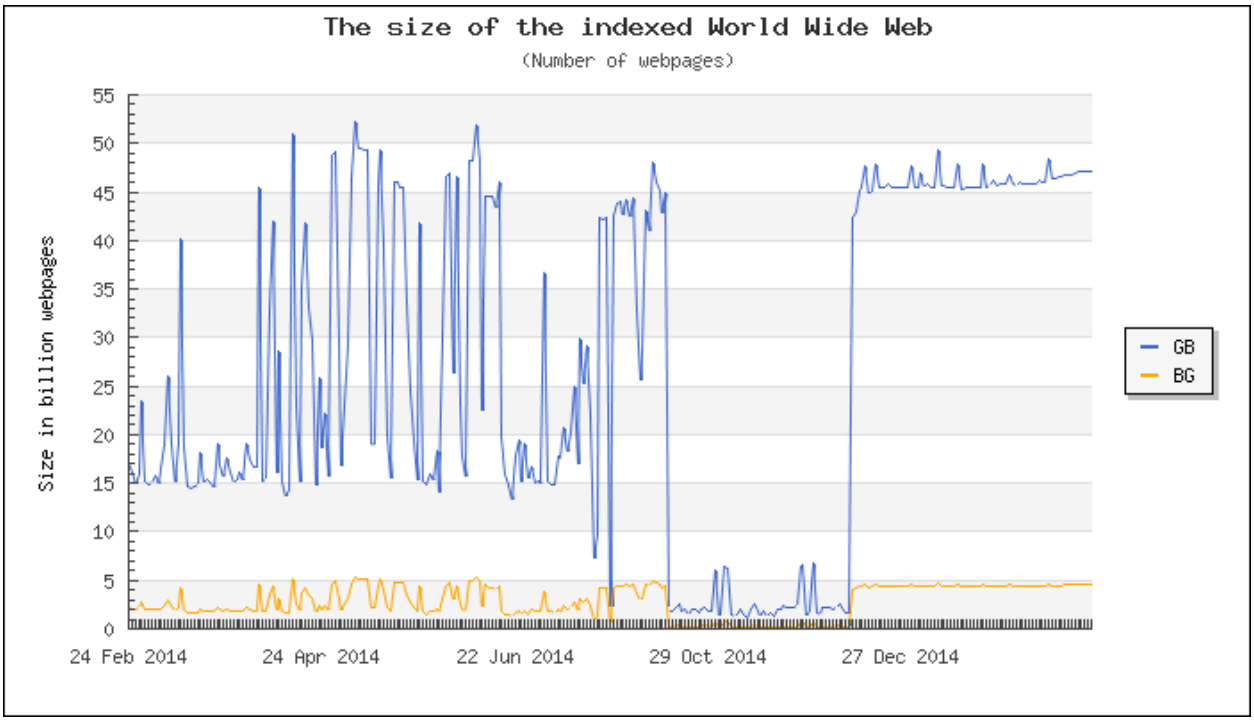

Figura 2: Gráfico del tamaño indexado de World Wide Web. [4]

GB: Ordenado en Google y Bing

BG: Ordenado en Bing y Google

Al momento de escoger una representación u otra, el contar con representaciones adecuadas que permitan interactuar con el resultado es un factor determinante, ya que esto tiene una relación directa en la comprensión y el entendimiento de la información resultante.

Por otro lado, Cristian Bustos [2] y otros señalan que “La brecha entre los tiempos de CPU y los de I/O se ha mantenido creciente durante las últimas décadas. Asimismo han aparecido nuevos niveles en la jerarquía de memoria (cachés de tamaño cada vez más considerable). Por ello, se ha hecho cada vez más atractivo el uso de estructuras de datos que ocupen poco espacio, incluso a veces comprimiendo la información sobre la que actúan. Si bien trabajar sobre esta información compacta es más laborioso, el hecho de poder mantenerla en una memoria órdenes de magnitud más rápida la convierte en una alternativa muy conveniente a las implementaciones clásicas.”

Una posible solución a los problemas de espacio de almacenamiento es la utilización de estructuras de datos comprimidas. Gonzalo Navarro [5] señala, “Las estructuras de datos compactas, son variantes de las estructuras clásicas solo que funcionan en espacio reducido y se pueden dividir en dos grupos: sucintas y comprimidas. Una estructura de datos se dice comprimida cuando toma espacio proporcional al de la secuencia comprimida (existe cierta

libertad para elegir un método razonable de compresión). En cambio, una estructura de datos es sucinta si necesita espacio asintóticamente despreciable sobre los datos en bruto.” [5].

Las estructuras comprimidas son estructuras modificadas que buscan almacenar la mayor cantidad de información en el menor espacio posible. Estas estructuras retienen la funcionalidad y el acceso directo, dando soporte a la consulta de información sin necesidad de descomprimirla, obteniendo de esta forma respuestas más rápidas. La estructura de datos k^2-tree fue diseñada para representar de manera compacta y eficiente relaciones binarias poco densas.

El k^2-tree es una representación compacta de una matriz de adyacencia de un grafo web, el cual soporta en su estructura la navegación hacia adelante y atrás entre los nodos, así como también permite visualizar los diferentes enlaces existentes entre un rango de nodos. Utiliza para esto la agrupación que surge de la matriz de adyacencia del grafo web, dejando fuera las grandes áreas vacías (donde no existe ninguna conexión) [1].

A continuación se explicará de forma gráfica el concepto de k^2-tree a través de un ejemplo sencillo.

La Figura 2 representa un grafo web pequeño compuesto de 4 nodos (n) o páginas web: 0, 1, 2 y 3. Esto implica una matriz de adyacencia cuadrada de tamaño n x n, donde n = 4, como se observa en la Figura 4.
El contenido de cada celda a_{ij} es 1 si existe conexión entre una página web i y una j, y un 0 en caso contrario.

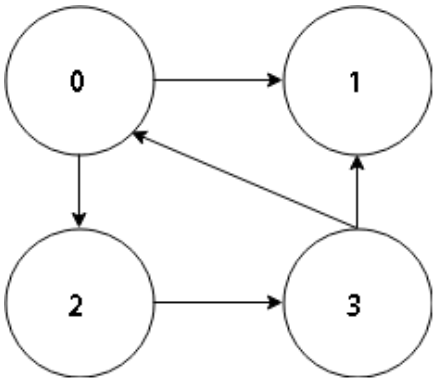

Figura 3: Ejemplo de grafo simple.

A	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	0	0	1
3	1	1	0	0

Figura 4: Matriz de adyacencia de grafo de 4 nodos.

La matriz de adyacencia se representa a través de un árbol k^2-ario, por esto es necesario asignar un valor para k.

Dado un k = 2, se procede con la subdivisión de la matriz de adyacencia en k^2 submatrices, esto es k filas y k columnas de submatrices de tamaño n^2/k^2.

En este caso particular se obtienen cuatro submatrices, donde cada una es hija del nodo raíz A y su valor es 1, si y solo si existe al menos un 1 en las celdas de la submatriz. Un 0 significa que toda la submatriz contiene 0's y por lo tanto, la descomposición del árbol termina allí, entonces los 0's se transforman en hojas del árbol.

A	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	0	0	1
3	1	1	0	0

Figura 5: Matriz dividida con k = 2.

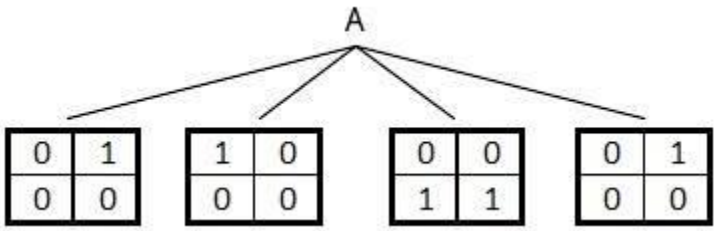

Figura 6: Representación de la división en submatrices.

Los hijos de un nodo se ordenan en el árbol comenzando con las submatrices de la primera fila, de izquierda a derecha, a continuación las submatrices de la segunda columna de izquierda a derecha, y así sucesivamente.

El tamaño n debe ser una potencia de k para que sea posible la subdivisión. La altura del árbol que se construirá será h = [$\log_k$ n]. En este caso particular h es igual a 2 ($\log_2$ 4).

Finalmente la matriz es representada por un árbol k^2-ario (ver Figura 7), el cual es llamado k^2-tree. En síntesis, la representación fue diseñada para la compresión de matrices con una gran cantidad de 0's en muy pocos bits.

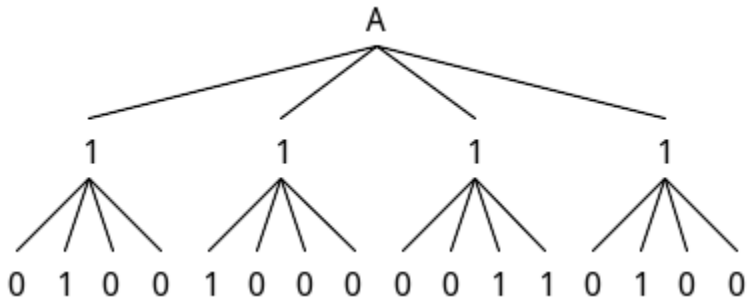

Figura 7: Árbol k2-ario.

La figura 12 generaliza el proceso de subdivisión de una matriz de adyacencia en k^2 submatrices indicando su orden.

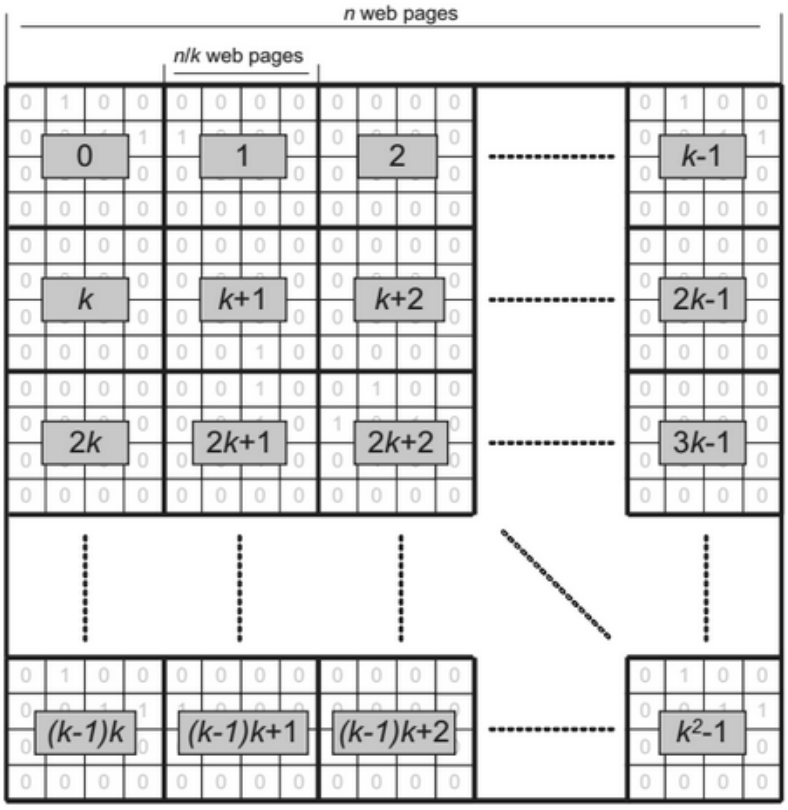

Figura 8: Representación de subdivisión de matriz de adyacencia.

Utilizando dos vectores de bits, se representa la matriz de adyacencia por completo a través del k^2-tree de una forma muy compacta. Dichos vectores se definen a continuación.

T (árbol): almacena todos los bits del k^2-tree exceptuando aquellos que se encuentran en el último nivel. Los bits son posicionados siguiendo un recorrido nivel por nivel; primero, los k^2 valores binarios de los hijos del nodo raíz y luego, los valores del segundo nivel, y así sucesivamente.

L (hojas del último nivel): almacena el último nivel del árbol. En consecuencia, esto representa los valores de las celdas originales de la matriz de adyacencia.

En el caso del ejemplo presentado, el contenido de los vectores T y L es el siguiente (ver Figura 7):

T = 1111

L = 0100 1000 0011 0100

A continuación se presenta un nuevo ejemplo de construcción de un k2-tree con n = 8 y sus respectivas matrices de adyacencia y árbol resultante.

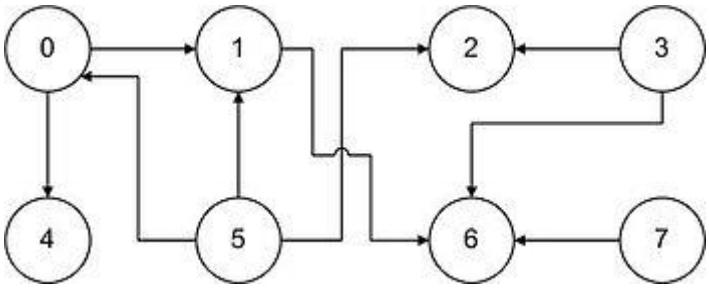

Figura 9: Grafo de 8 nodos.

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

Figura 11: Matriz de adyacencia 8x8.

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

Figura 10: Subdivisión de matriz de adyacencia con k = 2.

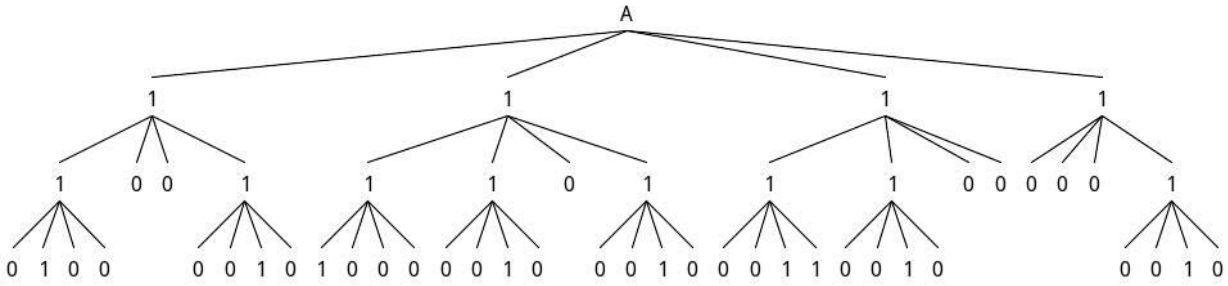

Figura 12: k^2-tree resultante.

Al concebir una estructura de datos nueva se hace necesario comprobar su efectividad, por este motivo se debe comprobar si es capaz de retener la funcionalidad, acceso directo y a la vez ser capaz de dar soporte a la consulta de información y ejecución de operaciones propias de una matriz de adyacencia sin comprimir.

3. Definición del Proyecto

3.1. Objetivos del Proyecto

3.1.1. Objetivos Generales

Estudiar la estructura de datos k^2-tree y sus implementaciones disponibles, para desarrollar mecanismos de consultas de datos y las operaciones binarias de unión, intersección, diferencia y complemento, directamente sobre el índice comprimido, sin acudir a la representación original.

3.1.2. Objetivos Específicos

- Estudiar los fundamentos teóricos de la estructura k^2-tree y de aspectos de su implementación y mecanismos de navegación.
- Instalar y ejecutar el sistema de Construcción de k^2-tree, y generar casos de prueba para la implementación de las operaciones binarias propuestas a continuación.
- Desarrollar las operaciones binarias: unión, intersección, diferencia y complemento directamente sobre la representación k^2-tree.
- Evaluar el rendimiento de las operaciones implementadas en términos del tamaño original de la matriz de entrada en relación al tamaño del k^2-tree (porcentaje de compresión de datos) y al tiempo de respuesta obtenido para estas operaciones en ambos contextos.
- Desarrollar operaciones de consulta de datos directamente sobre la representación k^2-tree, tanto por filas, columnas o respecto de una celda específica.
- Evaluar el rendimiento de las consultas implementadas en términos del tiempo de respuesta obtenido para diferentes volúmenes de datos.
- Analizar y comparar resultados de la experimentación.

3.2. Definiciones, Siglas y Abreviaciones

3.2.1. Definiciones

- **Matriz de Adyacencia:** es una matriz cuadrada que se utiliza como una forma de representar relaciones binarias.
- **Bit vector:** es un vector que almacena bits de manera compacta. En este caso particular, un *bit vector* es un tipo de dato en donde el valor de cada una de sus posiciones se encuentra almacenado un bit.

3.2.2. Acrónimos

- **WWW**: World Wide Web, sistema de distribución de documentos de hipertexto o hipermedios interconectados y accesibles vía internet.
- **I/O**: Input/Output, es la comunicación entre sistemas de procesamiento de información.

4. Aspectos de Construcción

A continuación se presentan aspectos relativos a la construcción del k^2-tree y los algoritmos propuestos. Para la construcción del k^2-tree se utiliza el software creado por Campos y Ulloa [6] con modificaciones en la entrada y la salida, y otras optimizaciones relativas al código.

4.1. Construcción de k^2-tree

4.1.1. Procedimiento y Estructuras de Datos utilizadas

1. Se lee la matriz desde el archivo binario, y se almacena en una lista de caracteres ligada. (bm).
2. En la función `Bitmap()` bm se copia en un vector auxiliar, y luego, este se traspasa a las estructuras de tipo ListAdy T y L. (ver Figura 13). Se agregan usando respectivamente las funciones `AddT()` y `AddL()`.
    a. La estructura creada para almacenar T es una lista enlazada que contiene la dirección de memoria de la lista que almacena la representación de los bits de la matriz de adyacencia (como caracteres). Además, contiene un entero que indica el nivel al que pertenecen los bits, y está enlazada al siguiente nivel.

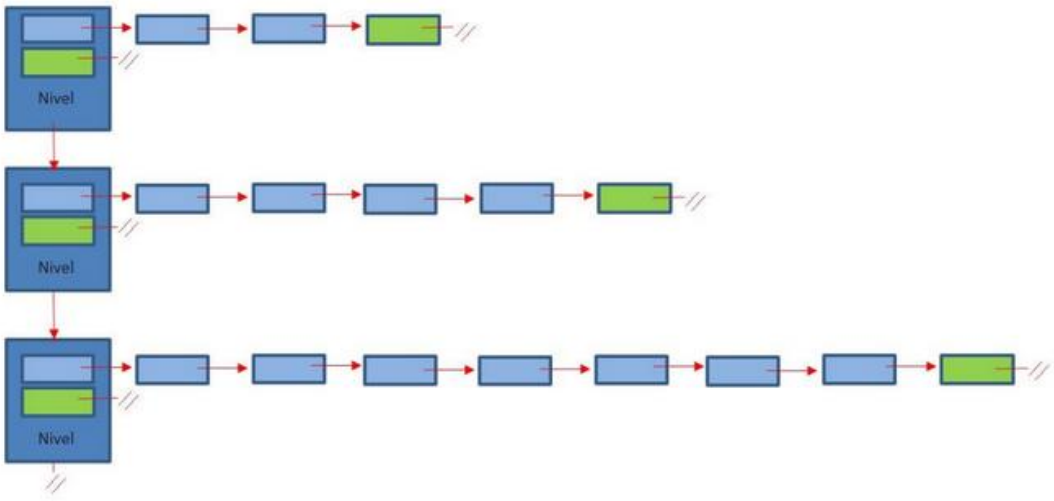

Figura 13: Representación gráfica de estructura creada para almacenar T (árbol).

b. La estructura creada para almacenar L es una lista ligada simple que almacena los nodos del último nivel del k^2-tree, es decir, los nodos hojas.

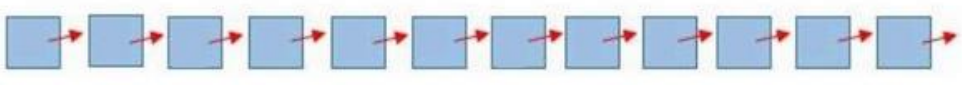

Figura 14: Representación gráfica de la estructura creada para almacenar L (nodos hojas).

3. Una vez creadas las listas por nivel (ListAdy) T y L, se crea el mapa de bits.
   a. Se almacena un número entero, donde cada uno de sus 32 bits (0's y 1's) corresponde a un bit extraído desde la lista de caracteres del punto 2.Una vez creado este entero, se añade a una lista con la estructura MyBM (ver Figura 10) que contiene un entero y el puntero al siguiente.
   b. De esta forma, queda como resultado una lista con enteros que representan cada uno de los bits.

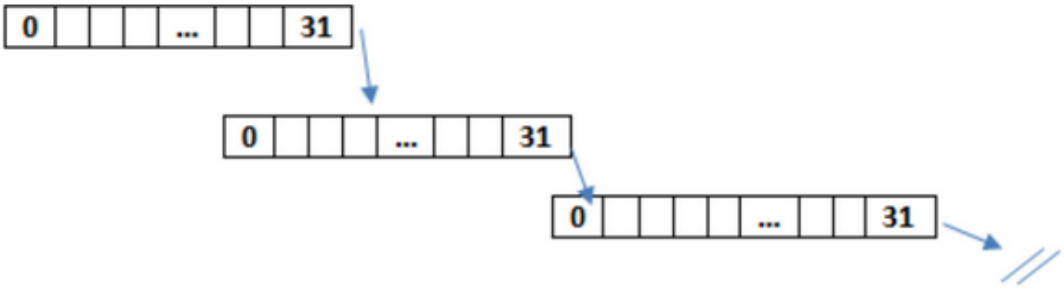

Figura 15: Lista ligada de enteros.

4. Completada la creación de las dos listas de tipo MyBM se asocia el Rank, que facilita la navegación a través de T y L. El Rank cuenta el número de veces que el bit b aparece hasta cierta posición, en este caso, hasta que aparece el bit 1. Particularmente, se asocia un Rank cada 20 nodos.

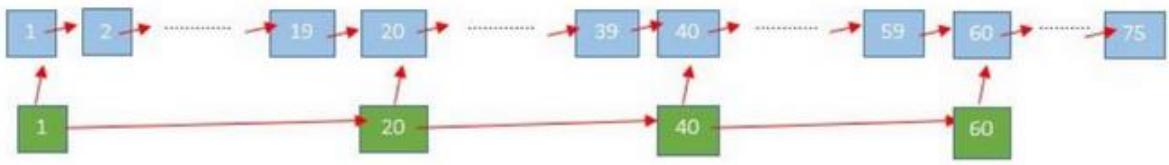

Figura 16: Representación de Rank aplicado a Bitmap.

5. Para extraer los datos de la estructura, se utilizan funciones propias de Rank, como RankSetT (para buscar en T) y RankSetL (para buscar en L).

4.1.2. Descripción de las formas de Entrada/Salida

- Entrada: Originalmente los datos de entrada eran recibidos desde un archivo de texto .txt, pero con el propósito de estandarizar los procesos se implementó una nueva forma de entrada, donde los datos son recibidos desde un archivo binario. El archivo contiene en la primera línea el orden de la matriz de adyacencia y en las líneas posteriores, los valores de las relaciones binarias que esta representa. Para almacenar la información se usan cadenas de caracteres, utilizando cada uno de los 8 bits de este tipo de dato. Los datos se extraen del archivo para posteriormente ser procesados. El proceso de extracción consiste en la lectura del orden de la matriz, la reserva de memoria para la representación del bitmap, lectura y almacenamiento de la matriz de adyacencia.
- Creación archivos de entrada: Los archivos de entrada son generados directamente en binario a través de un software que distribuye cantidades de 1's de forma uniforme (a través de todas las posiciones).
- Salida: Los resultados de cada proceso son tres archivos binarios, los cuales almacenan los vectores T y L y la información del árbol.

4.2. Construcción de algoritmos

4.2.1. Manipulación de estructuras

Para conocer el valor del bit en una determinada posición de un árbol, se utiliza una función llamada `GlobalSearch(pos, tamt)`. Dicha función recibe una posición del árbol de la cual se desea conocer el valor del bit contenido (0 o 1), se analiza si la posición corresponde a un nodo interno o una hoja, dependiendo del caso se usa la función new`Rank(A, i)` tanto para nodos internos u hojas. `NewRank` se mueve a través de los vectores de bit T y L utilizando el Rank. Se busca en T si el tamaño es menor al tamaño de T, y busca en L cuando el tamaño de T es superado. Una vez localizada la posición buscada, se comprueba el valor del bit en dicha posición.

Algoritmo 1: GlobalSearch

```
1: GlobalSearch(pos, pA)
2: if ( pos < tamt ) then
3:     return newRank(T, pos) {returns the value in the position}
4: else return newRank(L, pos)
5: end if
```

Para concatenar los diferentes niveles del k^2-tree se emplea la función `catLevel(bm, level, bit)`. Esta función recibe la estructura que almacena el árbol, el nivel en que debe realizarse la concatenación y los bits a concatenar, recorre el árbol hasta encontrar el nivel en donde se debe concatenar y añade el vector al final de la estructura a través de la función `AddNodo`. El tercer parámetro de la función `catLevel(comp, level, bit)` corresponde a cada uno de los valores del vector t, utilizado en el cálculo del complemento, la diferencia y la intersección, los cuales fueron almacenados en cada una de las llamadas a la función recursiva.

Algoritmo 2: catLevel

```
1: catLevel(bm, level, bit)
2: aux ← bm
3: while aux ≠ NULL do
4:     if ( aux → level = level ) then
5:           AddNodo(aux → LA, aux → last, bit)
6:     end if
7: end while
```

Dada la posición de un nodo padre, el algoritmo `Child(pA)` retorna un puntero con la primera posición en donde se encuentran almacenados sus hijos. Se calcula la posición del nodo hijo multiplicando la cantidad de 1s, obtenidos a través de la función `ApplyRank(A, pA)` por k^2.

Algoritmo 3: Child

```
1: Child(pA)
2:     ones ← ApplyRank(A, pA) {returns the number of 1's until the pA position}
3:     pA ← ones * k²
4: return pA
```

En la creación del k^2-tree surge la necesidad de utilizar una estructura de la cual se conozca el nivel en que se debe concatenar algún elemento. Esta característica es ofrecida por la estructura utilizada para el almacenamiento del bitmap T. Por lo tanto, se implementa una función que utiliza el mismo procedimiento que crea a T, pero añadiendo un nivel extra para incluir los nodos del bitmap L, último nivel del complemento. La función que crea la estructura donde se almacenará el complemento es `CreateColumnOfC(pot, k)`.

Para la utilización de ejemplos personalizados, es decir, teniendo el control sobre las posiciones de los 1s, se utiliza matrices en archivos de texto .txt.

4.3. Resumen de estructuras

Definición	Contenido	Descripción
MyList	char var nodo *next	Lista ligada simple con var de tipo char. Se usa para almacenar los valores leídos desde la matriz.
ListAdy	MyList LA MyList ultimo int nivel supernodo *next	Lista ligada que une los niveles. Cada nivel tiene a su vez una lista de tipo MyList.
MyBM	int var nodobm *next	Lista ligada simple con var de tipo int. Se usa para almacenar los valores enteros de la compresión de la estructura ListAdy. (Estructura comprimida).
Rank	MyBM nodo int cantidad nodoR *next	Lista que se utiliza para recorrer la representación comprimida.

Tabla 1: Resumen de estructuras.

5. Algoritmos Binarios

5.1. Complemento

5.1.1. Descripción de la Operación

La negación lógica o complemento es una función unaria que invierte el valor lógico de su argumento.

Sea A una matriz booleana de n x n elementos. Se define ~A por:

$$\sim A[i,j] = \begin{cases} 1, si\ A[i,j] = 0 \\ 0, si\ A[i,j] = 1 \end{cases}$$

Por ejemplo, si se tiene A como matriz binaria como muestra la figura 17.

A	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	0	0	1
3	1	1	0	0

Figura 17: Matriz binaria A complemento.

El complemento de dicha matriz, es decir ~A, se observa en la figura 18.

~A	0	1	2	3
0	1	0	0	1
1	1	1	1	1
2	1	1	1	0
3	0	0	1	1

Figura 18: Matriz binaria ~A complemento.

5.1.2. Ejemplo en k^2-tree

Dada la siguiente matriz de adyacencia A, con tamaño n = 8.

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

Figura 19: Matriz de adyacencia original complemento.

Con k = 2 se obtiene el siguiente k^2-tree de altura H = 3.

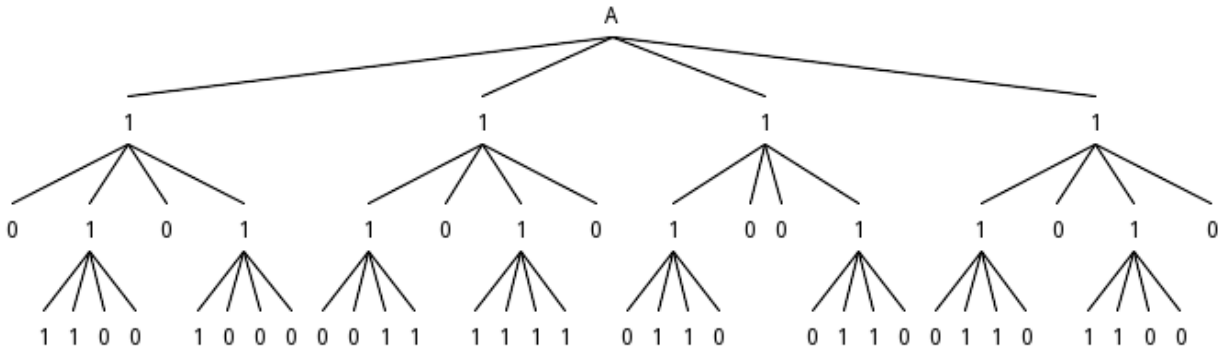

Figura 20: Árbol original complemento

La representación de la matriz de adyacencia para el complemento (A^c) se presenta a continuación.

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

Figura 21: Matriz de adyacencia resultante complemento.

Se observa que la matriz generada es efectivamente el complemento de la matriz presentada en el principio, por lo tanto el algoritmo queda comprobado.

El árbol complemento es el siguiente:

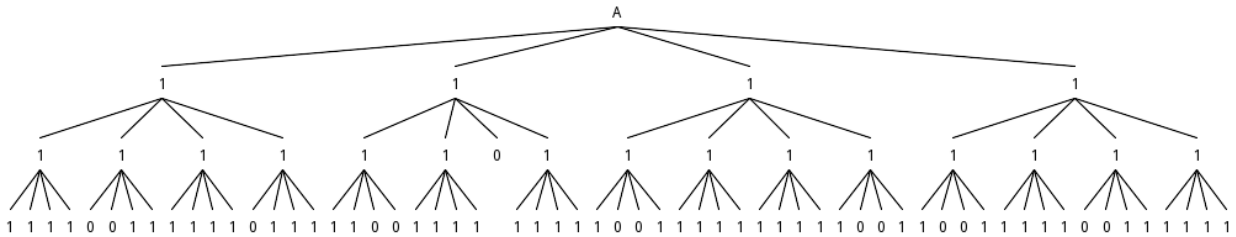

Figura 22: Árbol resultante complemento.

5.1.3. Descripción del Algoritmo

Para obtener el complemento de un k^2-tree A frecuentemente se requiere aplicar Primero en Profundidad Transversal desde el nodo actual de entrada a los nodos en el último nivel; debido a esto, se utiliza la estrategia de descomposición por niveles. El proceso finaliza cuando todas las ramas del k^2-tree A han sido visitadas. El algoritmo de complemento procesa cada nodo del k^2-tree A como sigue:

Caso 1: Si el valor de entrada es 0, entonces se usa una función llamada `FillIn()` para rellenar completamente con 1's el sub-árbol, desde el nodo actual hasta el último nivel.

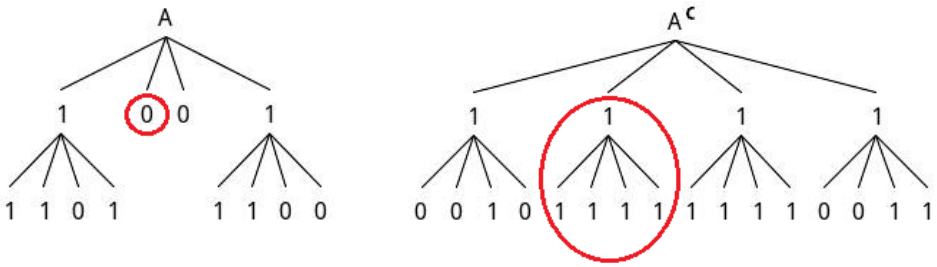

Figura 23: Ejemplo de Caso 1 Complemento.

Caso 2: Si el valor de entrada es 1, entonces se tiene un "candidato" para tomar el valor de 0, y es necesario comprobar esto por medio de la Profundidad Transversal. Existe la posibilidad de que los valores cambien de 0 a 1 en el último nivel, y en ese caso los niveles superiores no cambian. Si esto sucede se aplica una llamada recursiva para computar el complemento de un sub-árbol. Para el último nivel, se asigna directamente el valor negado (~) de la entrada.

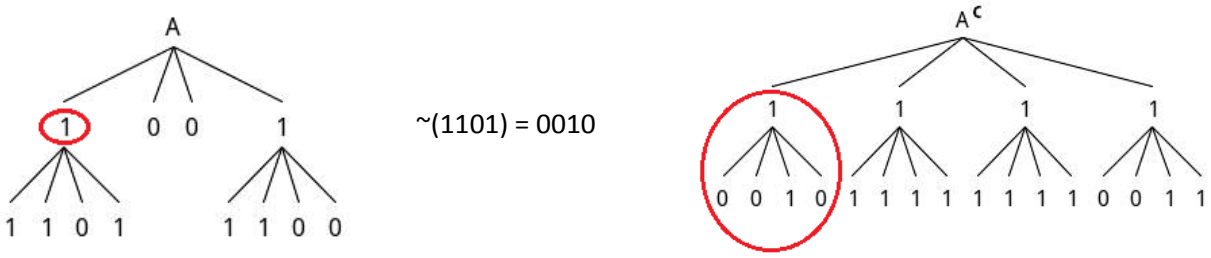

Figura 24: Ejemplo de Caso 2 Complemento.

5.1.4. Explicación y Traza Algoritmo Complemento

5.1.4.1. Descripción de las formas de Entrada/Salida

- Entrada: En la implementación del algoritmo Complemento se recibe como dato de entrada un bitmap T y L, estructuras de datos que son extraídas de un archivo binario.
- Salida: Al finalizar el cálculo de la existencia del Complemento se genera un bitmap que almacena el k^2-tree complemento, el que es almacenado en un archivo binario.

5.1.4.2. Aspectos de construcción del algoritmo Complemento

Algoritmo 4: FillIn

```
1: FillIn(l)
2: if ( l = H -1 ) then
3: t2 {t2 is a bitmap with k² size full of 1's}
4:     C[l] ← C[l] || t2
5: else
6:     for i ← 0 ... k² -1 do
7:           C[l] ← C[l] || 1
8:                 FillIn( l + 1 )
9:     end for
10: end if
```

El algoritmo `Filln(level, tree)` recibe como parámetro el nivel en el cual se debe comenzar a rellenar con 1s de forma recursiva y la estructura de la que se obtiene información del árbol, como la altura y el valor de k. El algoritmo concatena 1s con `catLevel` en el nivel que corresponda al momento en que se realizó la llamada.

5.1.4.3 Algoritmo Complemento

Algoritmo 5: Complemento

```
1: boolean Complement(l, pA)
2: writesomething ← 0, t {t is a bitmap with k² size}
3: for i ← 0 ... k² -1 do
4: t[i] ← 1
5:     if ( A[pA] = 1 ) then
6:           if l < H - 1 then
7:                 t[i] ← Complement(l + 1, Child(pA)) {Internal nodes}
8:           else
9:                 t[i] ← ~ A[pA]
10:          end if
11:    else if ( l < H - 1 ) then
12:          FillIn( l + 1 ) {Fill in the subtree with 1 values}
13:    end if
14:    writesomething ← writesomething ∨ t[i]
15:    pA ← pA + 1
16: end for
17: if ( writesomething = 1 ) then
18:    C[l] ← C[l] || t
19: end if
20: return writesomething
```

5.1.4.4. Traza de Ejecución Algoritmo Complemento

A continuación, en la figura 25, se observa el seguimiento al aplicar el algoritmo Complemento sobre el k^2-tree presentado. Las primeras cinco columnas son para el seguimiento de las variables utilizadas en el algoritmo, mientras que la última columna es utilizada para especificar el momento en que se realizan llamadas a funciones recursivas.

l	pA	ws	t[i]	i	Obs.
0	0	0			
0	0	1	t[0] = 1	0	Comp(1, 4), retorna ws = 1
	1	1	t[1] = 1	1	Comp(1, 8), retorna ws = 1
	2	1	t[2] = 1	2	Comp(1, 12), retorna ws = 1
	3	1	t[3] = 1	3	Comp(1, 16), retorna ws = 1
1	4	1	t[0] = 1	0	FillIn(2)
	5	1	t[1] = 1	1	Comp(2, 20), retorna ws = 1
	6	1	t[2] = 1	2	FillIn(2)
	7	1	t[3] = 1	3	Comp(2, 24), retorna ws = 1
2	20	0	t[0] = 0	0	
	21	0	t[1] = 0	1	
	22	1	t[2] = 1	2	
	23	1	t[3] = 1	3	
2	24	0	t[0] = 0	0	
	25	0	t[1] = 1	1	
	26	1	t[2] = 1	2	
	27	1	t[3] = 1	3	
1	8	1	t[0] = 1	0	Comp(2, 28), retorna ws = 1
	9	1	t[1] = 1	1	FillIn(2)
	10	1	t[2] = 0	2	Comp(2, 32), retorna ws = 0
	11	1	t[3] = 1	3	FillIn(2)
2	28	0	t[0] = 1	0	
	29	1	t[1] = 1	1	
	30	1	t[2] = 0	2	
	31	1	t[3] = 0	3	
2	32	0	t[0] = 0	0	
	33	0	t[1] = 0	1	
	34	0	t[2] = 0	2	
	35	0	t[3] = 0	3	
1	12	1	t[0] = 1	0	Comp(2, 36), retorna ws = 1
	13	1	t[1] = 1	1	FillIn(2)
	14	1	t[2] = 1	2	FillIn(2)
	15	1	t[3] = 1	3	Comp(2, 40), retorna ws = 1
2	36	1	t[0] = 1	0	
	37	1	t[1] = 0	1	
	38	1	t[2] = 0	2	
	39	1	t[3] = 1	3	
2	40	1	t[0] = 1	0	
	41	1	t[1] = 0	1	
	42	1	t[2] = 0	2	
	43	1	t[3] = 1	3	
1	16	0	t[0] = 1	0	Comp(2, 44), retorna ws = 1
	17	1	t[1] = 1	1	FillIn(2)
	18	1	t[2] = 1	2	Comp(2, 48), retorna ws = 1
	19	1	t[3] = 1	3	FillIn(2)
2	44	1	t[0] = 1	0	
	45	1	t[1] = 0	1	
	46	1	t[2] = 0	2	
	47	1	t[3] = 1	3	
2	48	0	t[0] = 0	0	
	49	0	t[1] = 0	1	
	50	1	t[2] = 1	2	
	51	1	t[3] = 1	3	

Figura 25: Traza de ejecución de complemento.

5.2. Diferencia

5.2.1. Descripción de la Operación

Sea A, B y C matrices booleanas de n x n elementos. Se define A ∧ ~B = C (A – B = C) la diferencia entre A y B, por:

$$C[i,j] = \begin{cases} 1, si\ A[i,j] = 1\ y\ B[i,j] = 0 \\ 0, si\ A[i,j] = B[i,j] = 1, o\ A[i,j] = 0 \end{cases}$$

Por ejemplo, si se tiene A y B como matrices binarias de entrada, como muestra la figura 26

A	0	1	2	3
0	1	0	1	0
1	0	1	0	1
2	1	1	1	1
3	0	0	0	0

B	0	1	2	3
0	1	0	0	1
1	0	1	0	1
2	0	0	1	0
3	0	1	1	1

Figura 26: Matrices binarias A y B Diferencia

Para mayor comprensión, en la figura 27 se muestra la matriz negada de B, ~B.

~B	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	1	0	0	0

Figura 27: Matriz binaria ~B Diferencia

Finalmente, completado el cálculo de la diferencia se obtiene la matriz C.

C	0	1	2	3
0	0	0	1	0
1	0	0	0	0
2	1	1	0	1
3	0	0	0	0

Figura 28: Matriz binaria resultante C Diferencia

5.2.2. Ejemplo en k^2-tree

Dada la siguiente matriz de adyacencia A, con tamaño n = 8.

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

Figura 29: Matriz de adyacencia A diferencia.

Con k = 2 se obtiene el siguiente k^2-tree A de altura H = 3.

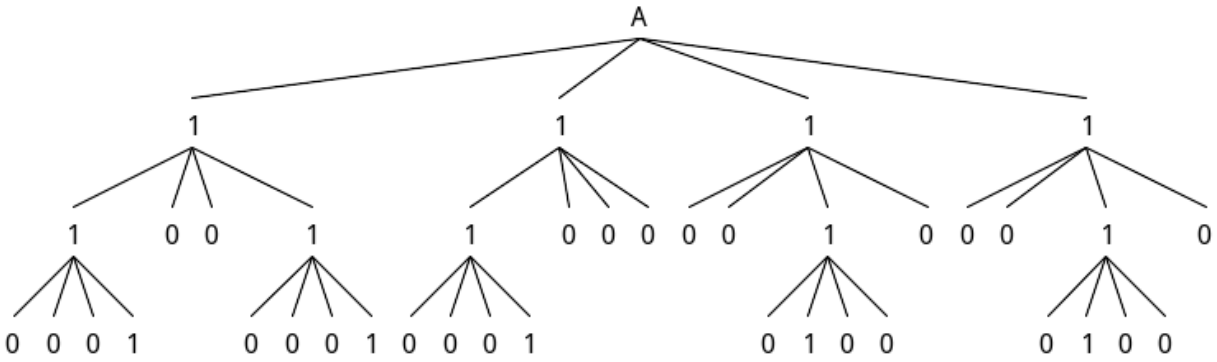

Figura 30: Árbol original A diferencia.

Dada la siguiente matriz de adyacencia B, con tamaño n = 8.

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

Figura 31: Matriz de adyacencia B diferencia.

Con k = 2 se obtiene el siguiente k^2-tree B de altura H = 3.

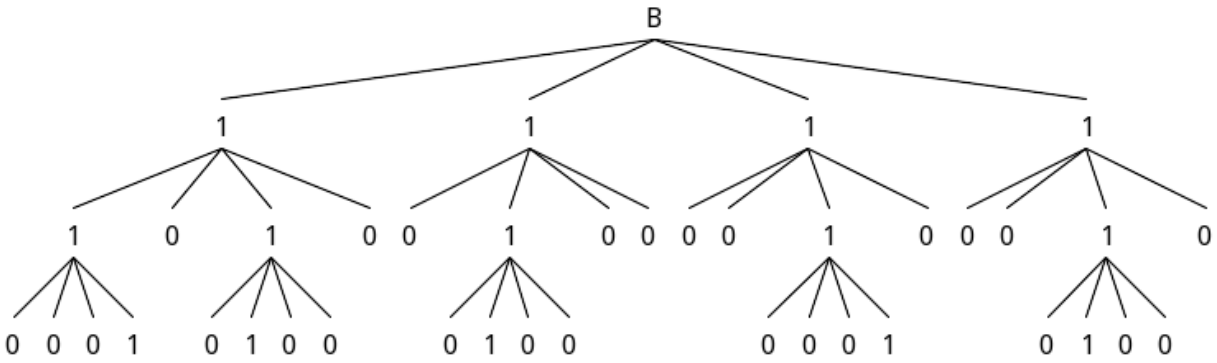

Figura 32: Árbol original B diferencia.

La representación de la matriz de adyacencia C (A - B) de la diferencia se presenta a continuación.

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

Figura 33: Matriz de adyacencia resultante diferencia.

El árbol para la matriz resultado C de la diferencia es el siguiente:

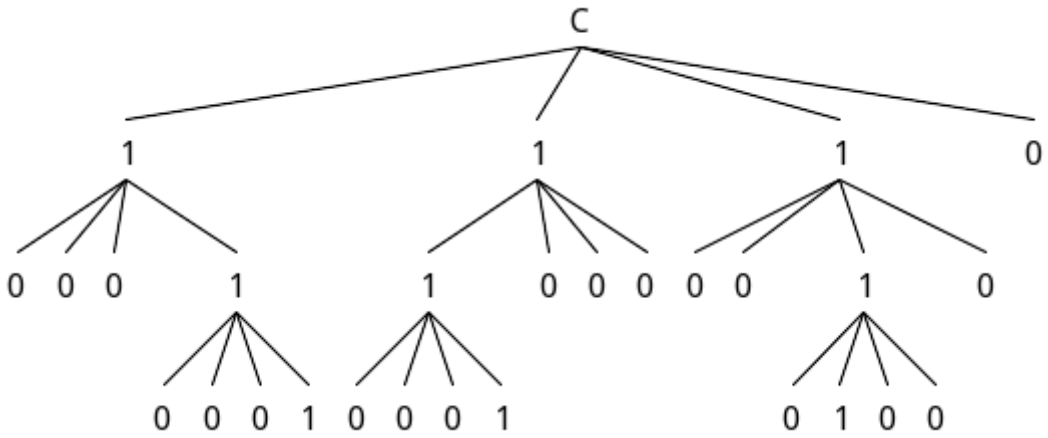

Figura 34: Árbol resultante diferencia.

5.2.3. Descripción del Algoritmo

Para determinar la diferencia entre un k^2-tree A y B, A - B, existen diferentes situaciones debido a que se requiere efectuar un recorrido en profundidad sobre las entradas. A causa de lo anterior, se aplica una estrategia de descomposición por niveles. Como los k^2-tree A y B tienen una altura H, se utilizan H vectores de bits diferentes para concatenar los resultados de las comparaciones por nivel.

El algoritmo comienza comparando los k^2 nodos de las raíces de cada k^2-tree analizando los siguientes 3 casos:

Caso 1: Si el valor de entrada en el nodo k^2-tree A es 0, entonces el resultado es 0 y en el nivel correspondiente de C se concatena este valor, no existiendo un análisis más profundo.

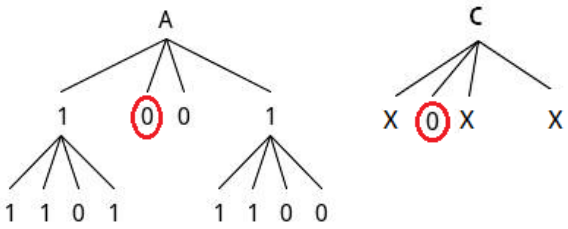

Figura 35: Ejemplo de Caso 1 Diferencia

Caso 2: Si el valor de entrada en la raíz del k^2-tree A es 1, y el valor de entrada en la raíz del k^2-tree B es 0, se procede a copiar el sub-árbol del k^2-tree A completo desde el nodo actual hasta llegar a los nodos asociados al último nivel, concatenando los resultados a cada nivel correspondiente.

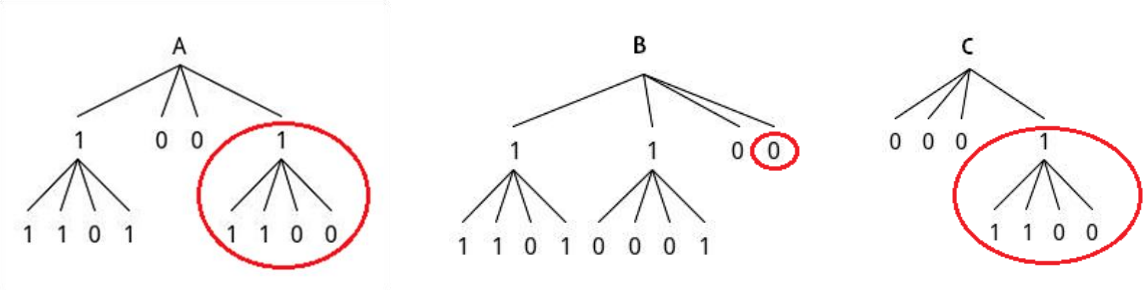

Figura 36: Ejemplo de Caso 2 Diferencia

Caso 3: Si el valor de entrada en la raíz del k^2-tree A es 1, y el valor de entrada en la raíz del k^2-tree B es 1 también, entonces se encuentra un “candidato” a tomar el valor 1 y debe evaluarse

en profundidad. Aquí se genera una llamada recursiva al algoritmo, que vuelve a aplicar alguno de los casos anteriores hasta llegar al último nivel (H - 1), en el cual se puede comparar directamente los valores con el criterio habitual de diferencia: 1 - 1 = 0, 1 - 0 = 1 expresada como A[pA] ∧ ~B[pB]. Si al menos uno de los valores examinados es 1, entonces retorna un 1 como resultado a los niveles superiores, sino se retorna un 0; esta propagación continúa hasta llegar al nivel superior y examinar cada una de las k^2 ramas existentes.

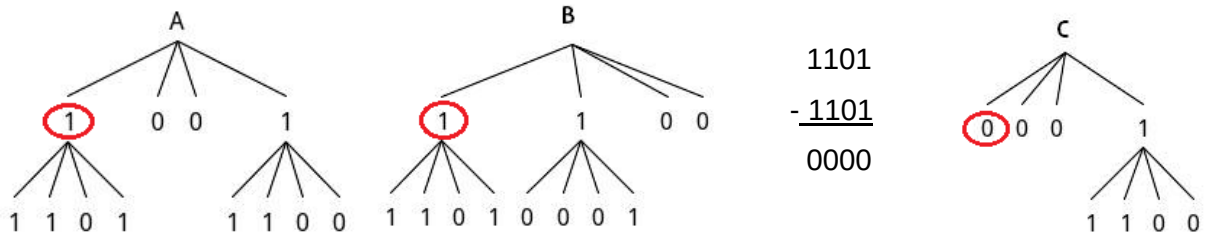

Figura 37: Ejemplo de Caso 3 Diferencia

5.2.4. Explicación y Traza Algoritmo Diferencia

5.2.4.1. Descripción de las formas de Entrada/Salida

- Entrada: En la implementación del algoritmo Diferencia se recibe como datos de entrada dos bitmaps (T y L) correspondientes a los dos árboles sobre los cuales se efectuará la operación. Estas estructuras de datos son extraídas de un archivo binario.
- Salida: Al finalizar el cálculo de la existencia de diferencia entre los árboles utilizados, se genera un bitmap que almacena el k^2-tree diferencia, que es almacenado en un archivo binario.

5.2.4.2. Aspectos de Construcción de Algoritmo Diferencia

Algoritmo 6: Copy

```
1: boolean Copy(l, pA)
2: if (l ≤ H - 1) then
3:     for i ← 0 ... k² - 1 do
4:           C[l] ← C[l] || A [pA + i]
5:           if ( l < H ) ∧ (A [pA + i] = 1) then
6:                 return Copy(l + 1, Child(A, pA + i)
7:           end if
8:     end for
```

```
9: else
10:    return 1
11: end if
```

Debido a las características del algoritmo diferencia se debe utilizar la función `Copy(l,pA)` que se encarga de copiar de forma recursiva los valores almacenados en los nodos a partir de un nivel l y una posición pA dadas. Se copian uno a uno los valores de los nodos del árbol A en el nivel correspondiente, y de cada uno de los hijos, si éstos existiesen y no fuesen ya nodos hojas.

En la línea 5 del Algoritmo Diferencia, se utiliza la función `GlobalSearch(x)` para determinar el valor del bit en la posición, tanto en el árbol A (pA) como en el árbol B (pB), y comprobar de esta forma las sentencias que se encuentran dentro de la condición if. La función se llama recursivamente con el uso de `Child()`, que como fue mencionado, determina la posición del primer hijo de un determinado nodo.

El algoritmo utiliza las mismas funciones que el complemento pero individualizadas para un árbol A y un árbol B.

5.2.4.3. Algoritmo Diferencia

Algoritmo 7: Diferencia

```
1: boolean Difference(l, pA, pB)
2: writesomething ← 0, t {t is a bitmap with k² size}
3: for i ← 0 ... k² -1 do
4:     t[i] ← 0
5:     if ( A[pA] ∧ B[pB] ) then
6:           if l < H - 1 then
7:                 t[i] ← Difference(l + 1, Child(pA), Child(pB)) {Internal nodes}
8:           else
9:                 t[i] ← A[pA] ∧ ~ B[pB]{Last level}
10:          end if
11:    else if ( A[pA] = 1 ) ∧ ( B[pB] = 0 )then
12:          if ( l < H - 1 )then
13:                t[i] = Copy( l + 1, Child(pA)) {Copy Subtree}
14:          else
15:                t[i] ← 1
```

```
16:          end if
17:    end if
18:    writesomething ← writesomething ∨ t[i]
19:    pA ← pA + 1, pB ← pB + 1
20: end for
21: if ( writesomething = 1 ) then
22:    C[l] ← C[l] || t
23: end if
24: return writesomething
```

5.2.4.4. Traza de Ejecución Algoritmo Diferencia

En la figura 38 se observa el seguimiento al aplicar el algoritmo Diferencia sobre los k^2-tree presentados. Las primeras seis columnas son para el seguimiento de las variables utilizadas en el algoritmo, mientras que la última columna es utilizada para especificar el momento en que se realizan llamadas a funciones recursivas.

l	pA	pB	ws	t[i]	i	Obs.
0	0	0	0			
0	0	0	1	t[0] = 1	0	Llama a Diff(1,4,4)
	1	1	1	t[1] = 1	1	Llama a Diff(1,8,8)
	2	2	1	t[2] = 1	2	Llama a Diff(1,12,12)
	3	3	1	t[3] = 0	3	Llama a Diff(1,16,16)
1	4	4	0	t[0] = 0	0	Llama a Diff(2,20,20)
	5	5	0	t[1] = 0	1	
	6	6	0	t[2] = 0	2	
	7	7	1	t[3] = 1	3	Copy(2, 24) retorna 1, retorna 1 a llamada Diff(1,4,4)
2	20	20	0	t[0] = 0	0	
	21	21	0	t[1] = 0	1	
	22	22	0	t[2] = 0	2	
	23	23	0	t[3] = 0	3	Retorna 0 a llamada Diff(2,20,20)
1	8	8	1	t[0] = 1	0	Copy(2, 28), retorna 1
	9	9	1	t[1] = 0	1	
	10	10	1	t[2] = 0	2	
	11	11	1	t[3] = 0	3	Retorna 1 a llamada Diff(1,8,8)
1	12	12	0	t[0] = 0	0	
	13	13	0	t[1] = 0	1	
	14	14	1	t[2] = 1	2	Llama a Diff(2,32,32)
	15	15	1	t[3] = 0	3	Retorna 1 a llamada Diff(1,12,12)
2	32	32	0	t[0] = 0	0	
	33	33	1	t[1] = 1	1	
	34	34	1	t[2] = 0	2	
	35	35	1	t[3] = 0	3	Retorna 1 a llamada Diff(2,32,32)
1	16	16	0	t[0] = 0	0	
	17	17	0	t[1] = 0	1	Llama a Diff(2,36,36)
	18	18	0	t[2] = 0	2	
	19	19	0	t[3] = 0	3	Retorna 0 a llamada Diff(1,16,16)
1	36	36	0	t[0] = 0	0	
	37	37	0	t[1] = 0	1	
	38	38	0	t[2] = 0	2	
	39	39	0	t[3] = 0	3	Retorna 0 a llamada Diff(2,36,36)

Figura 38: Traza de ejecución diferencia.

5.3. Unión

5.3.1. Descripción de la Operación

Sea A, B y C matrices booleanas de n x n elementos. Se define A ∨ B = C la unión de A y B, por:

$$C[i,j] = \begin{cases} 1, si\ A[i,j] = 1\ o\ B[i,j] = 1 \\ 0, si\ A[i,j] = B[i,j] = 0 \end{cases}$$

Por ejemplo, si se tiene A y B como matrices binarias de entrada, como muestra la figura 39

A	0	1	2	3
0	1	0	1	0
1	0	1	0	1
2	1	1	1	1
3	0	0	0	0

B	0	1	2	3
0	1	0	0	1
1	0	1	0	1
2	0	0	1	0
3	0	1	1	1

Figura 39: Matrices binarias A y B Unión

Finalmente, la matriz C representa el resultado de la unión de la matriz A y la matriz B.

C	0	1	2	3
0	1	0	1	1
1	0	1	0	1
2	1	1	1	1
3	0	1	1	1

Figura 40: Matriz binaria resultante C Unión

5.3.2. Ejemplo en k^2-tree

Dada la siguiente matriz de adyacencia A, con tamaño n = 8.

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

Figura 41: Matriz de adyacencia A unión.

Con k = 2 se obtiene el siguiente k^2-tree A de altura H = 3.

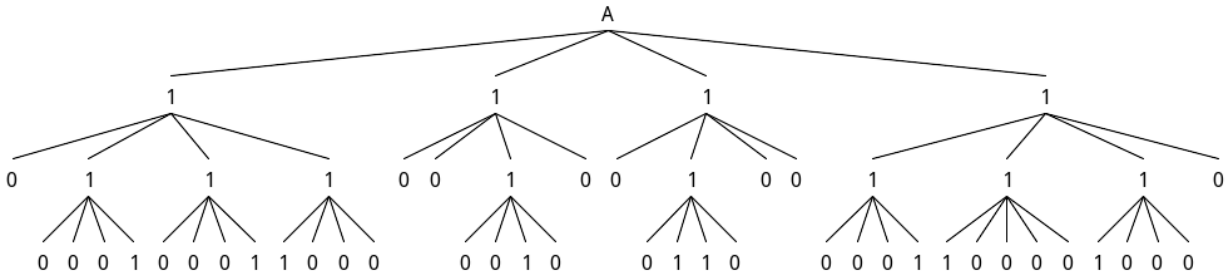

Figura 42: Árbol original A unión.

Dada la siguiente matriz de adyacencia B, con tamaño n = 8.

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

Figura 43: Matriz de adyacencia B unión.

Con k = 2 se obtiene el siguiente k^2-tree B de altura H = 3.

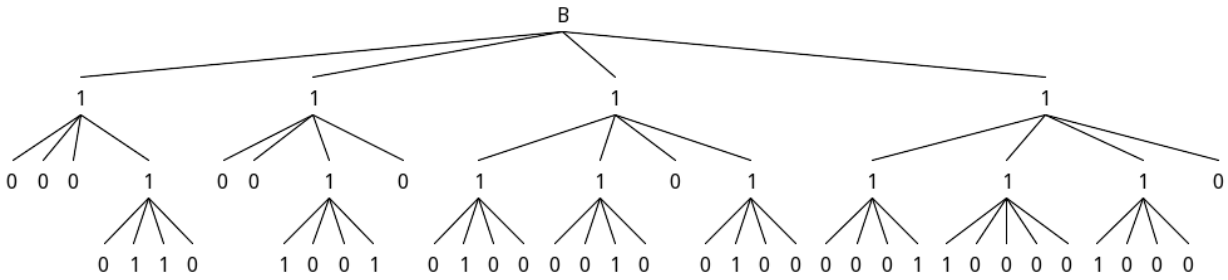

Figura 44: Árbol original B unión.

La representación de la matriz de adyacencia C (A ∪ B) de la unión se presenta a continuación.

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

Figura 45: Matriz de adyacencia resultante unión.

El árbol producto de la unión es el siguiente.

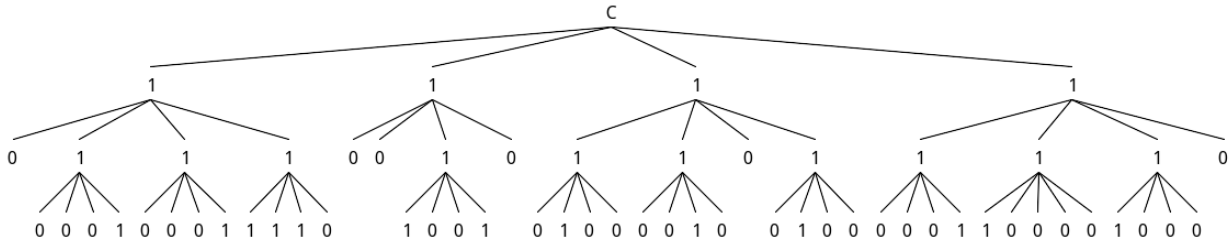

Figura 46: Árbol resultante unión.

5.3.3. Descripción del Algoritmo

El algoritmo de unión recorre en amplitud los bitmaps A y B de forma sincronizada. Para el recorrido de los bitmaps se mantiene una cola, denominada Q, donde se almacenan las tuplas 〈l, rA, rB〉. Los valores rA y rB indicaran si los nodos internos procesados de A y B, respectivamente, tienen hijos o no. El algoritmo comienza insertando la tupla 〈0, 1, 1〉 en Q. Lo cual significa que los nodos raíz de los árboles A y B, que se encuentran en el nivel 0, tienen hijos. Entonces, hasta que no existen tuplas para procesar, el algoritmo procesa la cola como sigue. Cada tupla corresponde a solo uno de los cuatro casos, dependiendo de los valores de rA y rB:

Caso 1: rA = 1 y rB = 1. Si ambos nodos internos tienen hijos, el algoritmo ejecuta la operación OR entre cada uno de los bits de los hijos apuntados por pA y pB respectivamente, y el resultado es añadido al bitmap C.

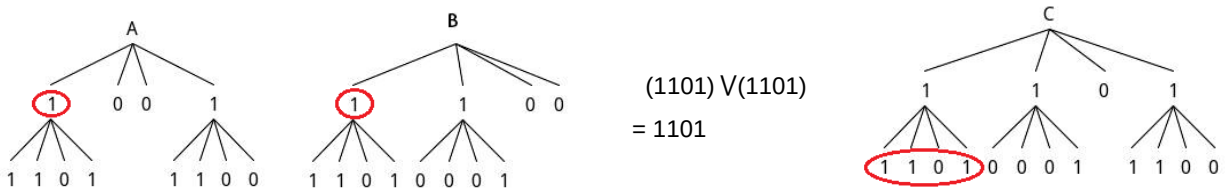

Figura 47: Ejemplo de Caso 1 Unión

Caso 2: rA = 0 y rB = 0. Si algunos de los nodos internos tiene hijos, el algoritmo añade un 0 al final del bitmap C. Si esto ocurre, los índices pA y pB no son incrementados.

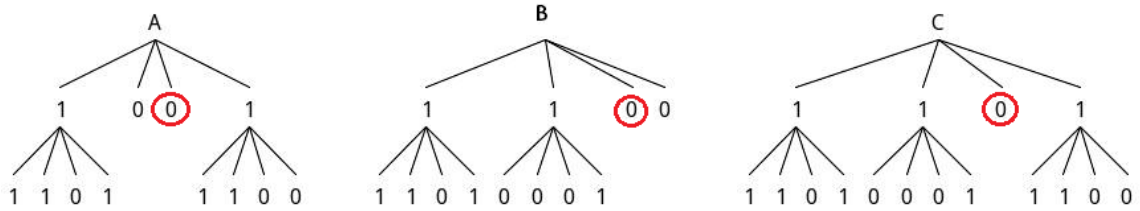

Figura 48: Ejemplo de Caso 2 Unión

Caso 3 y Caso 4: Solo un k^2-tree tiene hijos. Bajo el supuesto de que el nodo de A no tiene hijos (rA = 0, rB = 1), el algoritmo copia los k^2 bits de B apuntados por pB, añadiendo el resultado al bitmap C. Se debe notar que en este caso, solo el índice pB es incrementado. Cuando rA =1, rB = 0 se procede de forma análoga con los roles intercambiados entre los nodos de A y B. En estos casos, si la operación OR para un par de hijos resulta 1 y si este no es el último de los k^2-trees correspondientes, una nueva tupla es insertada en la cola.

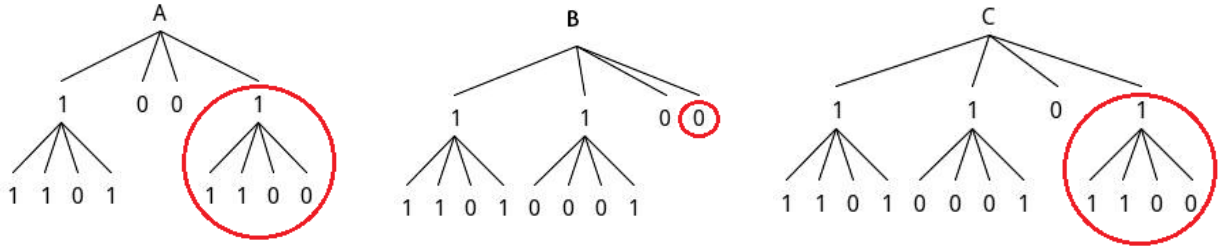

Figura 49: Ejemplo de caso 3 y 4 Unión

5.3.4. Explicación y Traza Algoritmo Unión

5.3.4.1. Descripción de las formas de Entrada/Salida

- Entrada: En la implementación del algoritmo Unión se reciben como datos de entrada dos bitmaps (T y L) correspondientes a los dos árboles sobre los cuales se efectuará la operación. Estas estructuras de datos son extraídas desde un archivo binario.

- Salida: Al finalizar el cálculo de unión entre los árboles utilizados, se genera un bitmap que almacena el k^2-tree correspondiente a la unión, el que es almacenado en un archivo binario.

5.3.4.2. Aspectos de Construcción de Algoritmo Unión

El algoritmo de Unión se diferencia de los algoritmos anteriores porque para realizar el procedimiento utiliza colas. Es por esto que se debió crear la estructura de una cola, con sus funciones correspondientes `Insert(l,bA,bB)` y `Delete()`, para poder operar con ella dentro del algoritmo. Las operaciones sobre colas restantes no fueron implementadas ya que no serían de utilidad.

`Insert(l,bA,bB)`, función que permite añadir un nodo al final de la cola. Los parámetros corresponden a la información que se almacena por cada nodo; nivel, el estado del bit en A y el estado del bit en B.

`Delete()`, función que permite eliminar y retornar los valores presentes en el nodo frontal de la cola, es decir el primer elemento que entró.

El algoritmo de unión se ejecuta mientras exista alguna tupla en la pila (no vacía), recorriendo así, todo el árbol, y comparando todas las posiciones. Para facilitar el manejo de las tuplas se utiliza un vector de tipo entero con tres posiciones.

Para las operaciones básicas de búsqueda se utiliza las mismas funciones que en los algoritmos anteriores.

Un aspecto que se debe destacar es que el bitmap obtenido es almacenado en una lista ligada secuencialmente, es decir, se almacena un nivel de T tras otro y a continuación L. Esto se traduce en que no es posible acceder a un nivel específico del árbol obtenido.

5.3.4.3. Algoritmo Unión

Algoritmo 8: Unión

```
1: boolean Union(pA, pB)
2: Q.Insert( ⟨0, 1, 1⟩ )
3: pA ← 0, pB ← 0 {pA is an index to traverse A and pB to B}
4: while Q ≠ NULL do
5:     ⟨l, rA, rB⟩ ← Q.Delete()
```

```
6:     for i ← 0 ... k² -1 do
7:           bA ← 0, bB ← 0 {bitmaps bA and bB, respectively}
8:           if rA = 1 then
9:                 bA ← A[pA], pA ← pA + 1
10:          end if
11:          if rB = 1 then
12:                bB ← B[pB], pB ← pB + 1
13:          end if
14:          C ← C || (bA ∨ bB)
15:          if (l < H) ∧ (bA ∨ bB = 1) then
16:                Q.Insert( ⟨l + 1, bA, bB⟩ )
17:          end if
18:    end for
19: end while
20: return C
```

5.3.4.4. Traza de Ejecución Algoritmo Unión

En la figura 50 se observa el seguimiento al aplicar el algoritmo Unión sobre los k^2-tree presentados. Las primeras ocho columnas son para el seguimiento de las variables utilizadas en el algoritmo, mientras que la última columna es utilizada para especificar las inserciones y eliminaciones realizadas sobre la cola Q.

l	pA	pB	rA	rB	bA	bB	i	Q
0	0	0						Se elimina tupla <0,1,1>
0	0	0	1	1	1	1	0	Se agrega tupla <1,1,1>
	1	1			1	1	1	Se agrega tupla <1,1,1>
	2	2			1	1	2	Se agrega tupla <1,1,1>
	3	3			1	1	3	Se agrega tupla <1,1,1>
1	4	4	1	1	0	0	0	Se elimina tupla <1,1,1>
	5	5			1	0	1	Se agrega tupla <2,1,0>
	6	6			1	0	2	Se agrega tupla <2,1,0>
	7	7			1	1	3	Se agrega tupla <2,1,1>
1	8	8	1	1	0	0	0	Se elimina tupla <1,1,1>
	9	9			0	0	1	
	10	10			1	1	2	Se agrega tupla <2,1,1>
	11	11			0	0	3	
1	12	12	1	1	0	1	0	Se elimina <1,1,1>. Se agrega tupla <2,0,1>
	13	13			1	1	1	Se agrega tupla <2,1,1>
	14	14			0	0	2	
	15	15			0	1	3	Se agrega tupla <2,0,1>
1	16	16	1	1	1	1	0	Se elimina <1,1,1>. Se agrega tupla <2,1,1>
	17	17			1	1	1	Se agrega tupla <2,1,1>
	18	18			1	1	2	Se agrega tupla <2,1,1>
	19	19			0	0	3	
2	20	20	1	0	0	0	0	Se elimina tupla <2,1,0>
	21				0	0	1	
	22				0	0	2	
	23				1	0	3	
2	24	20	1	0	0	0	0	Se elimina tupla <2,1,0>
	25				0	0	1	
	26				0	0	2	
	27				1	0	3	
2	28	20	1	1	0	1	0	Se elimina tupla <2,1,1>
	29	21			0	0	1	
	30	22			1	0	2	
	31	23			0	1	3	

Figura 50: Traza de ejecución unión.

l	pA	pB	rA	rB	bA	bB	i	Q
2	32	24	1	1	0	1	0	Se elimina tupla <2,1,1>
	33	25			0	0	1	
	34	26			1	0	2	
	35	27			0	1	3	
2	36	28	0	1	0	0	0	Se elimina tupla <2,0,1>
		29			0	1	1	
		30			0	0	2	
		31			0	0	3	
2	36	32	1	1	0	0	0	Se elimina tupla <2,1,1>
	37	33			1	0	1	
	38	34			1	1	2	
	39	35			0	0	3	
2	40	36	0	1	0	0	0	Se elimina tupla <2,0,1>
		37			0	1	1	
		38			0	0	2	
		39			0	0	3	
2	40	40	1	1	0	0	0	Se elimina tupla <2,1,1>
	41	41			0	0	1	
	42	42			0	0	2	
	43	43			1	1	3	
2	44	44	1	1	1	1	0	Se elimina tupla <2,1,1>
	45	45			0	0	1	
	46	46			0	0	2	
	47	47			0	0	3	
2	48	48	1	1	1	1	0	Se elimina tupla <2,1,1>
	49	49			0	0	1	
	50	50			0	0	2	
	51	51			0	0	3	

Figura 51: Traza de ejecución unión (cont.).

5.4. Intersección

5.4.1. Descripción de la Operación

Sea A, B y C matrices booleanas de n x n elementos. Se define A ∧ B = C la intersección de A y B, por:

$$C[i,j] = \begin{cases} 1, si\ A[i,j] = B[i,j] = 1 \\ 0, si\ A[i,j] = 0\ o\ B[i,j] = 0 \end{cases}$$

Por ejemplo, si se tiene A y B como matrices binarias de entrada, como muestra la figura 52.

A	0	1	2	3
0	1	0	1	0
1	0	1	0	1
2	1	1	1	1
3	0	0	0	0

B	0	1	2	3
0	1	0	0	1
1	0	1	0	1
2	0	0	1	0
3	0	1	1	1

Figura 52: Matrices binarias A y B Intersección.

Finalmente, la matriz C representa el resultado de la intersección de las matrices A y B.

C	0	1	2	3
0	1	0	0	0
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0

Figura 53: Matriz binaria resultante C Intersección.

5.4.2. Ejemplo en k^2-tree

Dada la siguiente matriz de adyacencia A, con tamaño n = 8.

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

Figura 54: Matriz de adyacencia A intersección.

Con k = 2 se obtiene el siguiente k^2-tree A de altura H = 3.

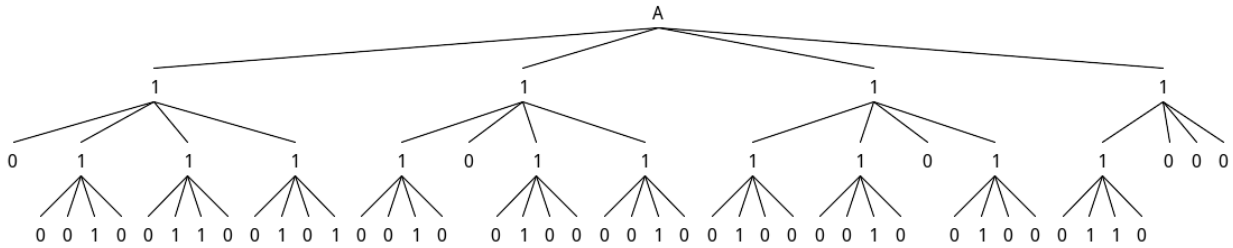

Figura 55: Árbol original A intersección.

Dada la siguiente matriz de adyacencia B, con tamaño n = 8.

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

Figura 56: Matriz de adyacencia B intersección.

Con k = 2 se obtiene el siguiente k^2-tree B de altura H = 3.

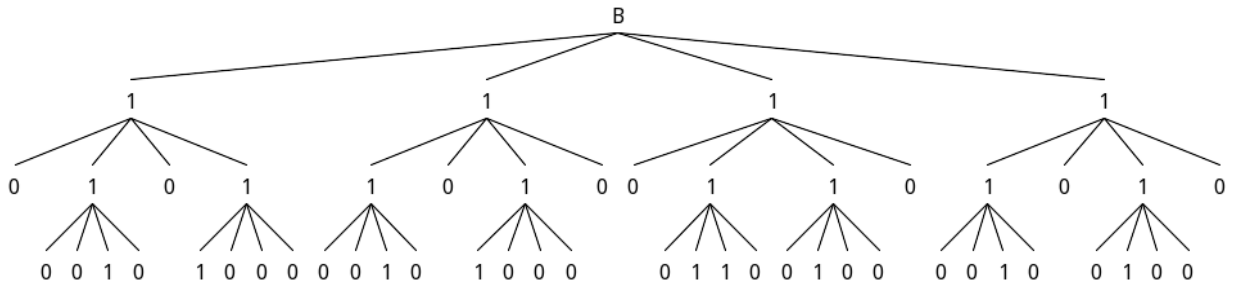

Figura 57: Árbol original B intersección.

La representación de la matriz de adyacencia C (A ∩ B) de la intersección se presenta a continuación.

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

Figura 58: Matriz de adyacencia resultante intersección.

El árbol obtenido al aplicar la intersección se representa como sigue.

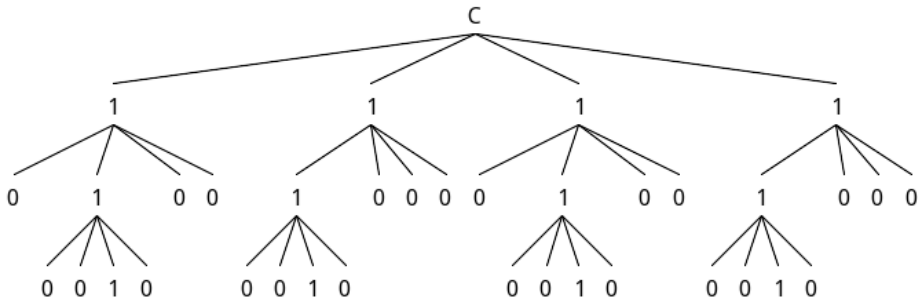

Figura 59: Árbol resultante intersección.

5.4.3. Descripción del Algoritmo

El algoritmo de intersección realiza un recorrido en profundidad los árboles A y B sincronizadamente, intersectando cada par de nodos A y B. Debido a las características del recorrido, los sub-árboles son procesados desde izquierda a derecha por cada nivel. La primera llamada al algoritmo es `Intersection(1, pA, pB)`, en donde todos los valores de pA y pB son establecidos como nodo inicial para cada nivel.

El algoritmo intersección compara los k^2 nodos en cada paso y considera uno de los siguientes casos.

Caso 1: Si los nodos de entrada pertenecen al último nivel, entonces se puede computar el valor de la intersección directamente usando el operador AND.

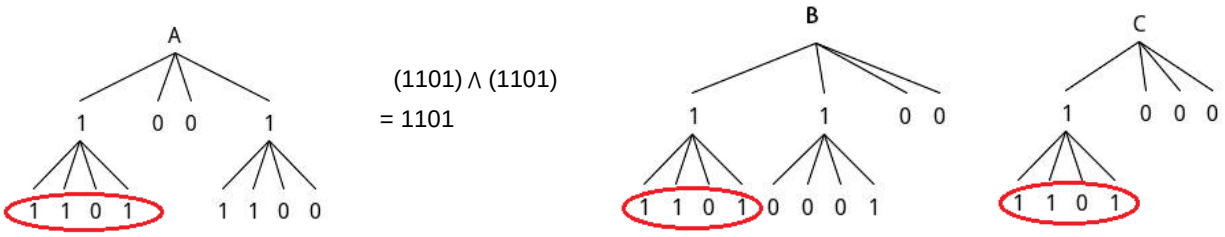

Figura 60: Ejemplo de Caso 1 Intersección.

Caso 2: Si uno de los valores de entrada del k^2-tree A y B en el nivel i es 0, entonces el resultado es 0 y se concatena el valor 0 en C[i]. En este caso, se necesita actualizar los punteros pA[i + 1], … , pA[H] o pB[i + 1], … , pB[H], para omitir por completo el sub árbol no-vacío.

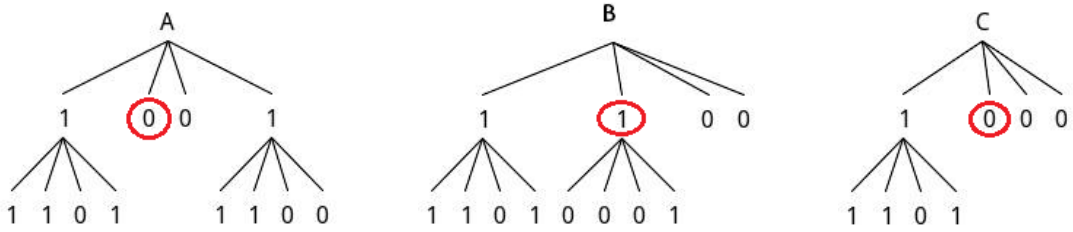

Figura 61: Ejemplo de Caso 2 Intersección.

Caso 3: Si ambos valores son 1, entonces se tiene un "candidato" para tomar el valor de 1, pero se necesita verificarlo a través de un profundo recorrido. Aquí se ejecuta una llamada recursiva con el siguiente nivel como parámetro de entrada. Así el algoritmo retorna si es añadido algún bit al bitmap C[j] para un cierto nivel j, esta llamada recursiva determinará si la intersección de los sub árboles de ambos nodos es vacía (la llamada retorna 0) o no (retorna 1), determinando el valor de la intersección para los valores de entrada originales.

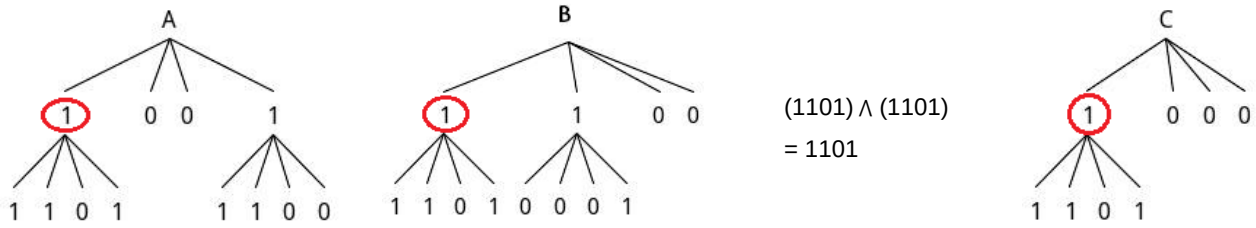

Figura 62: Ejemplo de Caso 3 Intersección.

5.4.4. Explicación y Traza Algoritmo Intersección

5.4.4.1. Descripción de las formas de Entrada/Salida

- Entrada: En la implementación del algoritmo Intersección se recibe como datos de entrada dos bitmaps (T y L) correspondientes a los dos árboles sobre los cuales se efectuará la operación. Estas estructuras de datos son extraídas de un archivo binario.
- Salida: Al finalizar el cálculo de la intersección entre los árboles utilizados, se genera un bitmap que almacena el k^2-tree resultante, el que es desplegado por pantalla además de ser almacenado en un archivo binario.

5.4.4.2. Aspectos de Construcción del Algoritmo Intersección

La construcción del algoritmo Intersección es casi idéntica a la del algoritmo Diferencia, debido a que no posee la condición donde se utiliza la función `Copy(x)`. Al implementar el algoritmo diferencia prácticamente se implementó el de intersección.

5.4.4.3. Algoritmo Intersección

Algoritmo 9: Intersección

```
1: boolean Intersection(l, pA, pB)
2: writesomething ← 0, t {t is a bitmap with k² size}
3: for i ← 0 ... k² -1 do
4:     t[i] ← 0
5:     if ( A[pA] ∧ B[pB] ) then
6:           if l < H - 1 then
7:                 t[i] ← Intersection(l + 1, Child(pA), Child(pB)) {Internal nodes}
8:           else
9:                 t[i] ← A[pA] ∧ B[pB]{Last level}
10:          end if
11:    end if
12:    writesomething ← writesomething ∨ t[i]
13:    pA ← pA + 1, pB ← pB + 1
14: end for
15: if ( writesomething = 1 ) then
16:    C[l] ← C[l] || t
17: end if
18: return writesomething
```

5.4.4.4. Traza de Ejecución Algoritmo Intersección

En la figura 63 se observa el seguimiento al aplicar el algoritmo Intersección sobre los k^2-tree presentados. Las primeras seis columnas son para el seguimiento de las variables utilizadas en el algoritmo, mientras que la última columna es utilizada para especificar el momento en que se realizan las llamadas recursivas.

l	pA	pB	ws	t[i]	i	Obs.
0	0	0	0			
0	0	0	1	t[0] = 1	0	Llama a Inter(1,4,4)
	1	1	1	t[1] = 1	1	Llama a Inter(1,8,8)
	2	2	1	t[2] = 1	2	Llama a Inter(1,12,12)
	3	3	1	t[3] = 1	3	Llama a Inter(1,16,16)
1	4	4	0	t[0] = 0	0	
	5	5	1	t[1] = 1	1	Llama a Inter(2,20,20)
	6	6	1	t[2] = 0	2	
	7	7	1	t[3] = 0	3	Llama a Inter(2,28,24). Retorna 1 a llamada Inter(1,4,4)
2	20	20	0	t[0] = 0	0	
	21	21	0	t[1] = 0	1	
	22	22	1	t[2] = 1	2	
	23	23	1	t[3] = 0	3	Retorna 1 a llamada Inter(2,20,20)
2	28	24	0	t[0] = 0	0	
	29	25	0	t[1] = 0	1	
	30	26	0	t[2] = 0	2	
	31	27	0	t[3] = 0	3	Retorna 0 a llamada Inter(2,28,24)
1	8	8	1	t[0] = 1	0	Llama a Inter(2,32,28)
	9	9	1	t[1] = 0	1	
	10	10	1	t[2] = 0	2	Llama a Inter(2,36,32)
	11	11	1	t[3] = 0	3	Retorna 1 a llamada Inter(1,8,8)
2	32	28	0	t[0] = 0	0	
	33	29	0	t[1] = 0	1	
	34	30	1	t[2] = 1	2	
	35	31	1	t[3] = 0	3	Retorna 1 a llamada Inter(2,32,28)
2	36	32	0	t[0] = 0	0	
	37	33	0	t[1] = 0	1	
	38	34	0	t[2] = 0	2	
	39	35	0	t[3] = 0	3	Retorna 0 a llamada Inter(2,36,32)
1	12	12	0	t[0] = 0	0	
	13	13	1	t[1] = 1	1	Llama a Inter(2,48,36)
	14	14	1	t[2] = 0	2	
	15	15	1	t[3] = 0	3	Retorna 1 a llamada Inter(1,12,12)
2	48	36	0	t[0] = 0	0	
	49	37	0	t[1] = 0	1	
	50	38	1	t[2] = 1	2	
	51	39	1	t[3] = 0	3	Retorna 1 a llamada Inter(2,48,36)
1	16	16	0	t[0] = 1	0	Llama a Inter(2,56,44)
	17	17	1	t[1] = 0	1	
	18	18	1	t[2] = 0	2	
	19	19	1	t[3] = 0	3	Retorna 1 a llamada Inter(1,16,16)
2	56	44	0	t[0] = 0	0	
	57	45	0	t[1] = 0	1	
	58	46	1	t[2] = 1	2	
	59	47	1	t[3] = 0	3	Retorna 1 a llamada Inter(2,56,44)

Figura 63: Traza de ejecución intersección.

6. Construcción de Baselines para Comparación

El objetivo de la creación y utilización de una línea de base es proporcionar una base de información o punto de partida contra el cual observar, evaluar y comparar el comportamiento de la implementación de las operaciones binarias sobre el k^2-tree. Por lo tanto, el baseline describe la implementación de operaciones binarias que trabajan sobre la matriz binaria de forma directa, sin efectuar compresión alguna sobre el archivo a analizar.

6.1. Algoritmo de creación de matrices binarias

El programa utilizado para la creación de las matrices contra las que se realizan las comparaciones recibe tres parámetros: el tamaño de la matriz, la cantidad de 1s y el nombre del archivo de salida. Una restricción es que el tamaño de la matriz debe ser múltiplo de 8 para poder almacenarla en un char (8 bits) y utilizarlo de forma completa.

El programa reserva memoria para el bitmap y establece el valor de cada bit en 0, luego utiliza el algortimo `setBits(bm, n, ones)` para añadir la cantidad de 1s necesaria.

Algoritmo 10: Creación Baseline

```
1: setBits(bm, n, ones)
2: r, p
3: while i < ones do
4:     r ← rand() % n² {r is a random value inside the bounds of the matrix}
5:     p ← r / W {p is the random position where 1s are located}
6:     bit ← r - p * W + 1 {W is the number of bits}
7:     msk ← ( 0x01 ) << ( W - 1 ) {msk}
8:     if ( bm[p] = 0) then
9:           bm[p] ← bm[p] | msk
10:          i++
11: end if
12: end while
```

Una vez finalizado el posicionamiento de 1s, el programa procede a generar un archivo binario con la matriz en su interior en el formato descrito anteriormente.

6.2. Algoritmos de operaciones binarias

El programa utilizado en el cálculo del complemento sobre una matriz binaria recibe tres parámetros: el nombre del archivo de entrada, el nombre del archivo de salida y el tamaño del buffer de lectura.

Para el cálculo del complemento se obtiene el valor negado de cada bit de cada char del vector.

Algoritmo 11: Baseline Complemento

```
1: unOperator(input, output, buffersize, n)
2: buf1[buffersize], bl1 {buf1 is used to access a bit, bl1 is the amount of data
read}
3: do{
4:     bl1 ← readFile(input)
5:     for i ← 0 ... bl1 do
6:           buf1[i] ← ~buf1[i]
7:           writeFile(output)
8:     end for
9: while( bl1 ≤ buffersize )
```

El programa utilizado en el cálculo de la unión, intersección y diferencia es uno solo, debido a que reciben los mismos parámetros. Para ejecutar cada uno se debe ingresar el número asignado a cada operación. Se reciben cinco parámetros: el nombre del archivo de entrada de la matriz A, el nombre del archivo de entrada de la matriz B, el nombre del archivo de salida, el tamaño del buffer de lectura y la opción que identifica a la operación.

1. Para el cálculo de la unión se recorre cada bit de cada char del vector tanto de la matriz A como la matriz B, para cada par de bits se utiliza el operador OR.
2. Para el cálculo de la intersección se recorre cada bit de cada char del vector tanto de la matriz A como la matriz B, para cada par de bits se utiliza el operador AND.
3. Para el cálculo de la diferencia se recorre cada bit de cada char del vector tanto de la matriz A como la matriz B, para cada par de bits se utiliza el operador AND pero se utiliza el valor negado del bit en B.

Algoritmo 12: Baseline Unión, Intersección, Diferencia

```
1: BinOperator(input1, input2, output, buffersize, n, op)
2: buf1[buffersize], bl1 {buf1 is used to access a bit, bl1 is the amount of data
read}
3: buf2[buffersize] {buf2 is used to access a bit}
4: do{
5:     bl1 ← readFile(input1)
6:     bl2 ← readFile(input2)
7:     switch op
8:           case 1 {union}
9:                 for i ← 0 ... bl1 do
10:                      buf1[i] ←  buf1[i] | buf2[i]
11:                end for
12:          case 2 {intersection}
13:                for i ← 0 ... bl1 do
14:                      buf1[i] ←  buf1[i] & buf2[i]
15:                end for
16:          case 3 {difference}
17:                for i ← 0 ... bl1 do
18:                      buf1[i] ←  buf1[i] & ~buf2[i]
19:                end for
20:          writeFile(output)
21: while( bl1 ≤ buffersize )
```

7. Pruebas

7.1. Elementos de prueba

Las pruebas realizadas son de evaluación experimental. El enfoque escogido para las pruebas es de caja negra, ya que se conoce la función específica para la que fue realizado cada módulo. Se proporcionan las entradas y se estudian las salidas para ver si concuerda con lo esperado. Para la realización de las pruebas solo se considera el desempeño de los algoritmos implementados, dejando de lado el proceso de creación del k^2-tree a partir de una matriz de adyacencia. Todos los algoritmos utilizan un k^2-tree almacenado como bitmaps T y L que se encuentran almacenados en un archivo. Cada uno de los algoritmos lee uno o dos archivos según corresponda.

El objetivo es obtener y comparar el tiempo y memoria utilizada por los distintos algoritmos al momento de generar el k^2-tree resultante. Las pruebas consideran matrices de adyacencia de diferentes tamaños y diferentes densidades de matriz utilizando distribución uniforme. Se entiende como densidad de matriz a la cantidad de enlaces que están presentes en esta, esto es, a matrices de adyacencia con mayor densidad, existe una mayor cantidad de conexiones entre sus nodos.

Ante la necesidad de utilizar diferentes densidades de matriz sobre las matrices en las pruebas, se utilizan programas que generan matrices con estas características.

7.2. Especificación de pruebas

Se procederá a realizar pruebas de desempeño y funcionalidad de los diversos algoritmos, utilizando para esto, distintos casos de prueba, de distinto tamaño y densidad de matriz, distribuidos de forma uniforme.

Por cada módulo, algoritmo, se medirá lo siguiente:

- Tiempo empleado para el cálculo y almacenamiento del k^2-tree resultante, medido en microsegundos (µs). Una vez son leídos los archivos de entrada se comienza con la medición, esta incluye el recorrido sobre la estructura, la realización de las comparaciones y asignaciones correspondientes.
- Tamaño original de la matriz de entrada versus el tamaño del k^2-tree, porcentaje de compresión de datos, en relación al tiempo de respuesta obtenido por cada uno.

- Densidad dentro de la matriz de adyacencia. El tipo de distribución utilizada es Uniforme. En la distribución Uniforme, la posición de cada 1 en la matriz es igualmente probable para cada intervalo de igual longitud en la distribución.

En la siguiente tabla se especifican los casos de prueba realizados.

Orden de la matriz	Densidad de matriz	Distribución	Valores de K
104, 520, 1.000, 3.000, 5.000	5%, 25%, 50%, 75%, 95%	Uniforme	2

Tabla 2: Especificación de casos de prueba.

En la primera columna se encuentra el orden de la matriz se refiere al valor de n; si el orden es de 104, la matriz posee nxn elementos, es decir, 10.816 casillas. En la segunda columna, la densidad de la matriz es definida de forma porcentual, además, para cada prueba se incluye la equivalencia numérica del porcentaje. En la tercera columna, se especifica la distribución y finalmente, en la cuarta columna, el valor de k utilizado.

Características del equipo donde se realizaron las pruebas:

- Arquitectura: 64 bits
- Sistema Operativo: Linux Ubuntu 14.04
- Memoria RAM: 3814MiB
- Procesador: Intel® Core™i3-3110M CPU @ 2.40GHz

El formato para los gráficos de tiempo versus densidad de matriz para cada orden de la matriz es el siguiente:

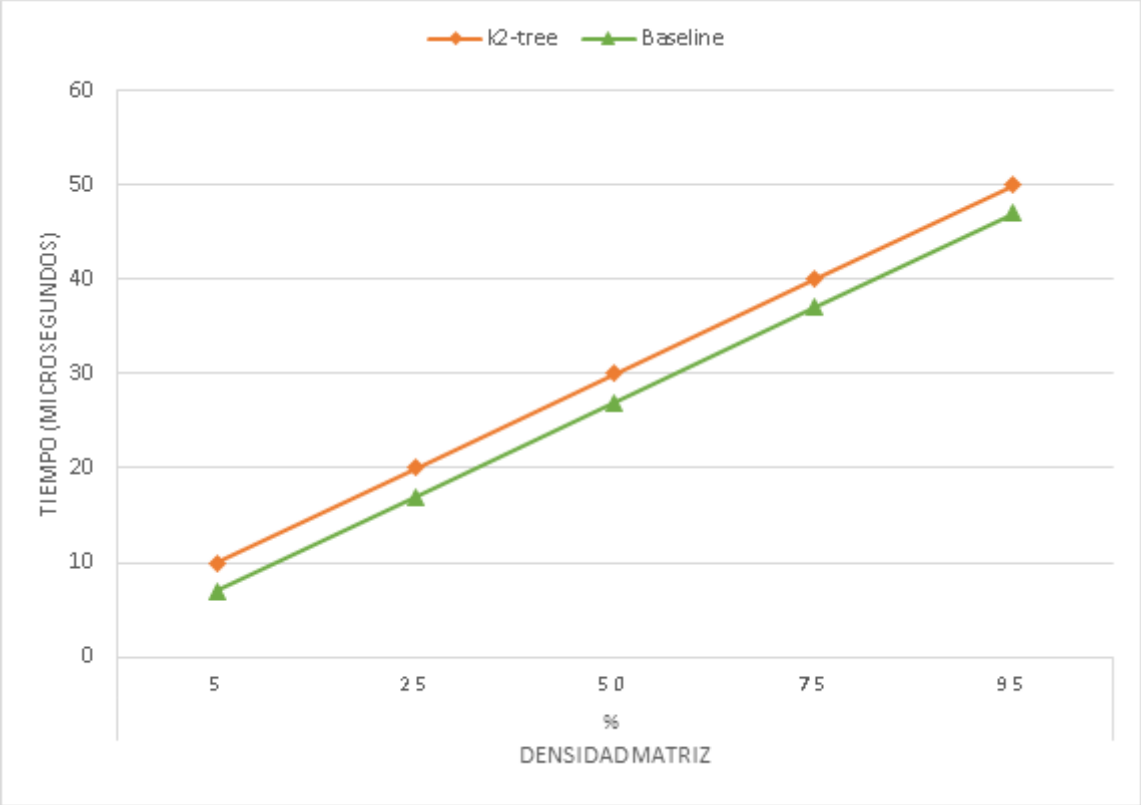

Figura 64: Formato de gráficos.

El eje y es utilizado para indicar el tiempo en microsegundos y el eje x contiene las diferentes densidades expresadas en porcentaje. En el caso de los gráficos de tamaño versus implementación, el eje y es utilizado para los tamaños y el eje x para el orden de la matriz. La elección de utilizar gráficos de líneas fue tomada debido a las características que esta ofrece, como mostrar de forma sencilla la evolución y el comportamiento de los datos.

A continuación se presenta el detalle de las pruebas con sus respectivos gráficos para cada uno de los algoritmos analizados, para posteriormente, realizar un análisis de los resultados en el siguiente capítulo.

7.3. Detalle de las Pruebas

7.3.1. Pruebas algoritmo Complemento

7.3.1.1. Pruebas Densidad v/s Tiempo algoritmo Complemento

Complemento	n = 104	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	541	4.613,876	14
25%	2.704	6.391,048	20,980
50%	5.408	6.160,974	21,219
75%	8.112	6.040,096	27,894
95%	10.276	5.960,941	22,172

Tabla 3: Mediciones de tiempo complemento n 104.

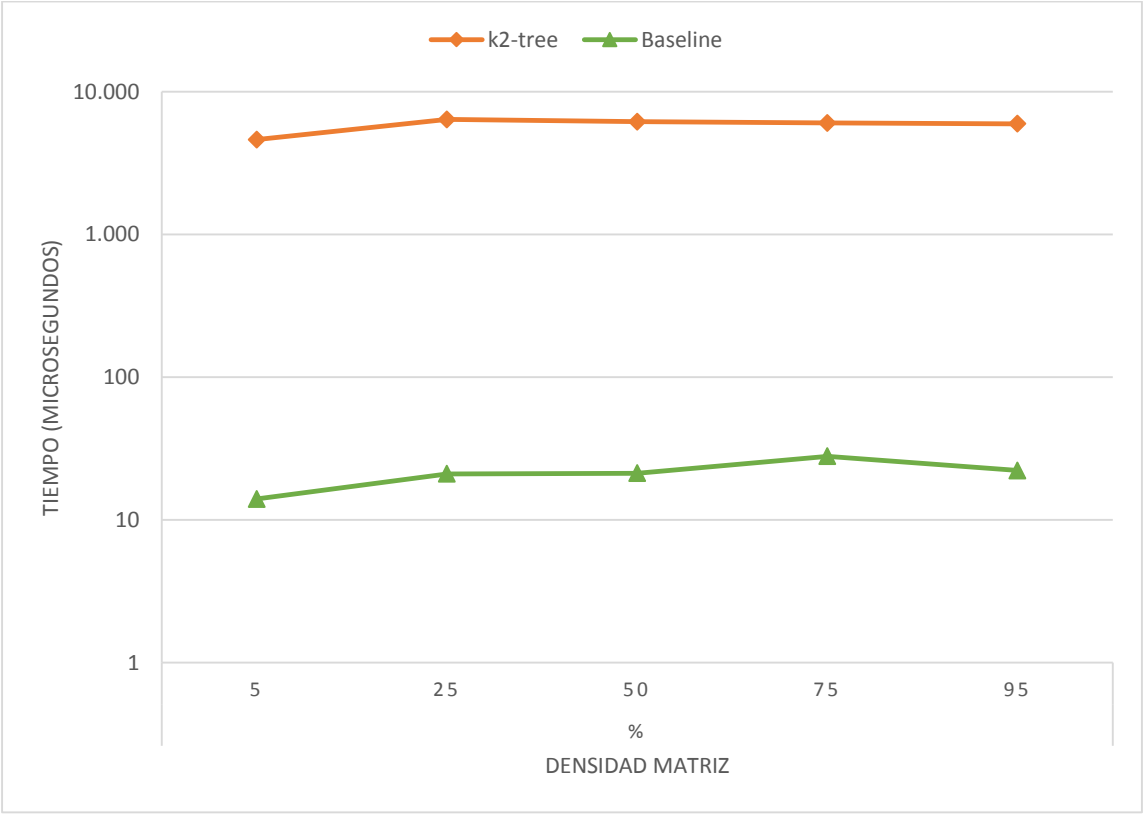

Figura 65: Gráfico mediciones de tiempo complemento n 104.

Complemento	n = 520	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	13.520	83.407,164	478,983
25%	67.600	123.557,090	365,019
50%	135.200	131.442,070	378,131
75%	202.800	117.808,103	365,018
95%	256.880	125.780,821	184,059

Tabla 4: Mediciones de tiempo complemento n 520.

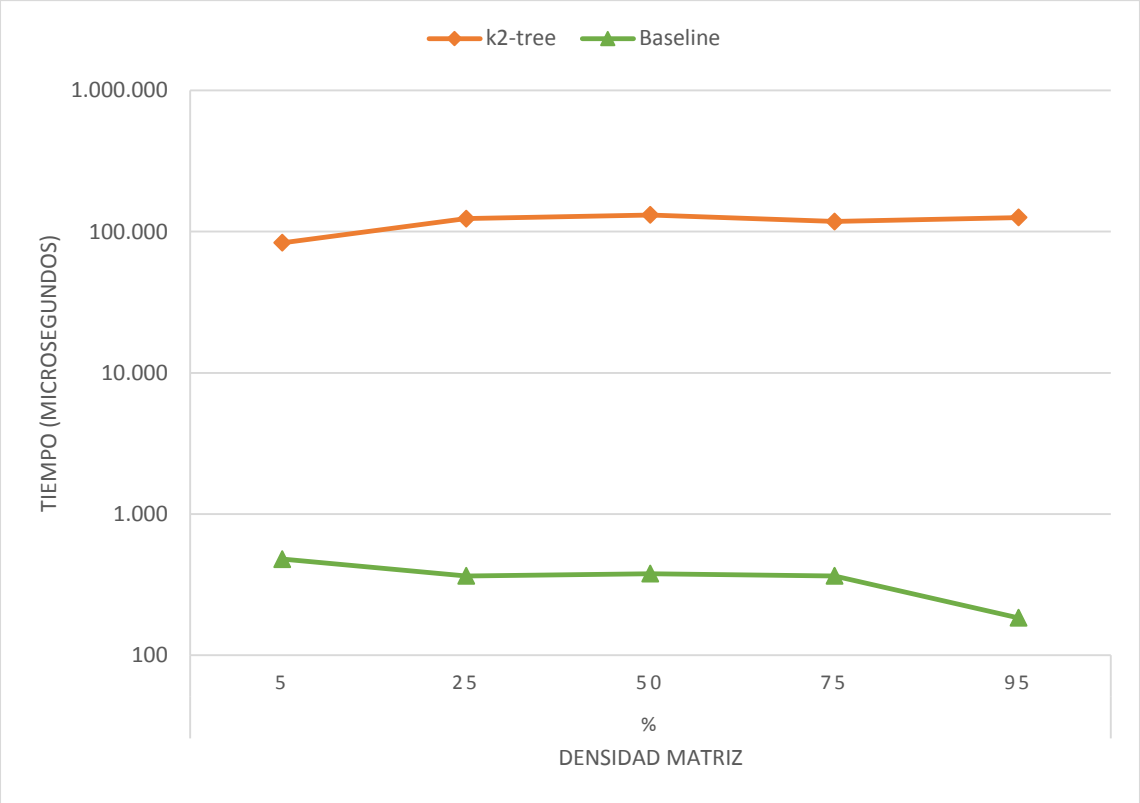

Figura 66: Gráfico mediciones de tiempo complemento n 520.

Complemento	n = 1.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	50.000	411.911,011	629,186
25%	250.000	721.745,014	704,050
50%	500.000	727.962,017	1.250,982
75%	750.000	703.343,868	1.204,013
95%	950.000	710.436,105	1.205,921

Tabla 5: Mediciones de tiempo complemento n 1000.

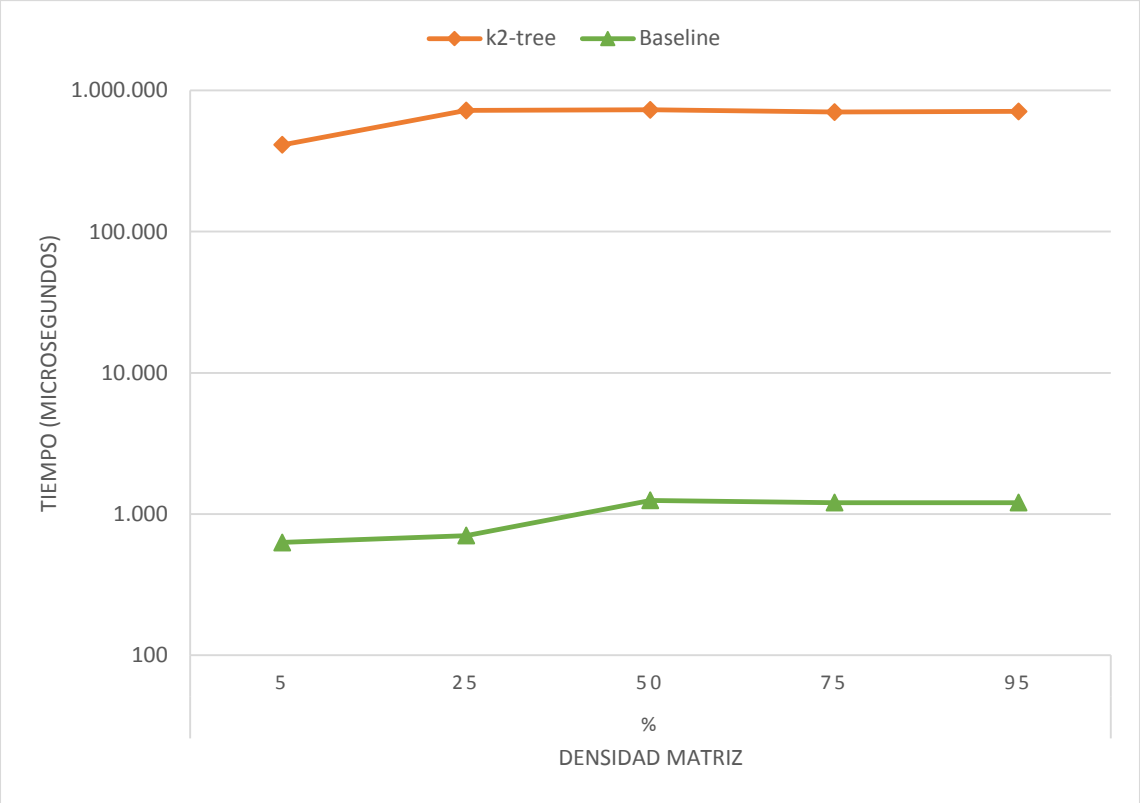

Figura 67: Gráfico mediciones de tiempo complemento n 1000.

Complemento	n = 3.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	450.000	17.677.752,017	8.422,136
25%	2.250.000	37.148.880,004	10.849,952
50%	4.500.000	37.666.620,016	5.980,968
75%	6.750.000	37.412.599,086	8.882,999
95%	8.550.000	37.425.884,008	7.128,000

Tabla 6: Mediciones de tiempo complemento n 3000.

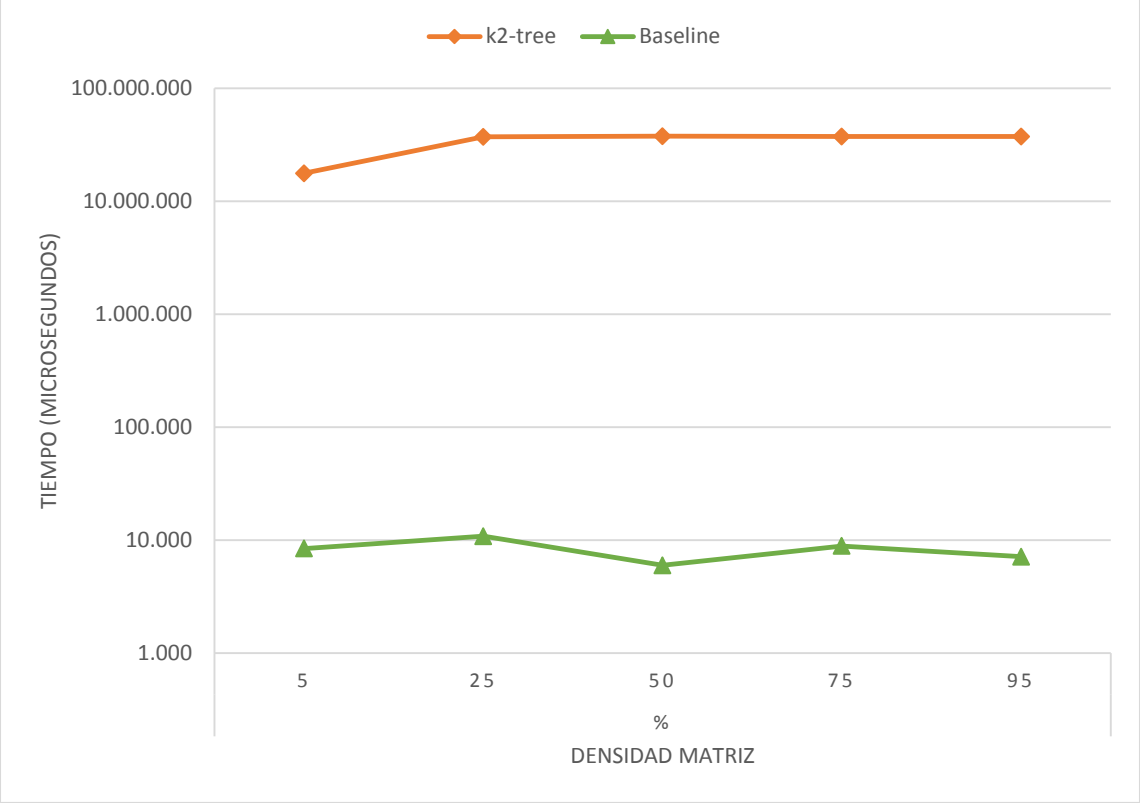

Figura 68: Gráfico mediciones de tiempo complemento n 3000.

Complemento	n = 5.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	1.250.000	127.341.792,106	22.526,025
25%	6.250.000	275.974.436,044	23.120,164
50%	12.500.000	281.072.520,017	20.684,003
75%	18.750.000	278.445.549,011	19.556,999
95%	23.750.000	279.176.872,968	17.398,834

Tabla 7: Mediciones de tiempo complemento n 5000.

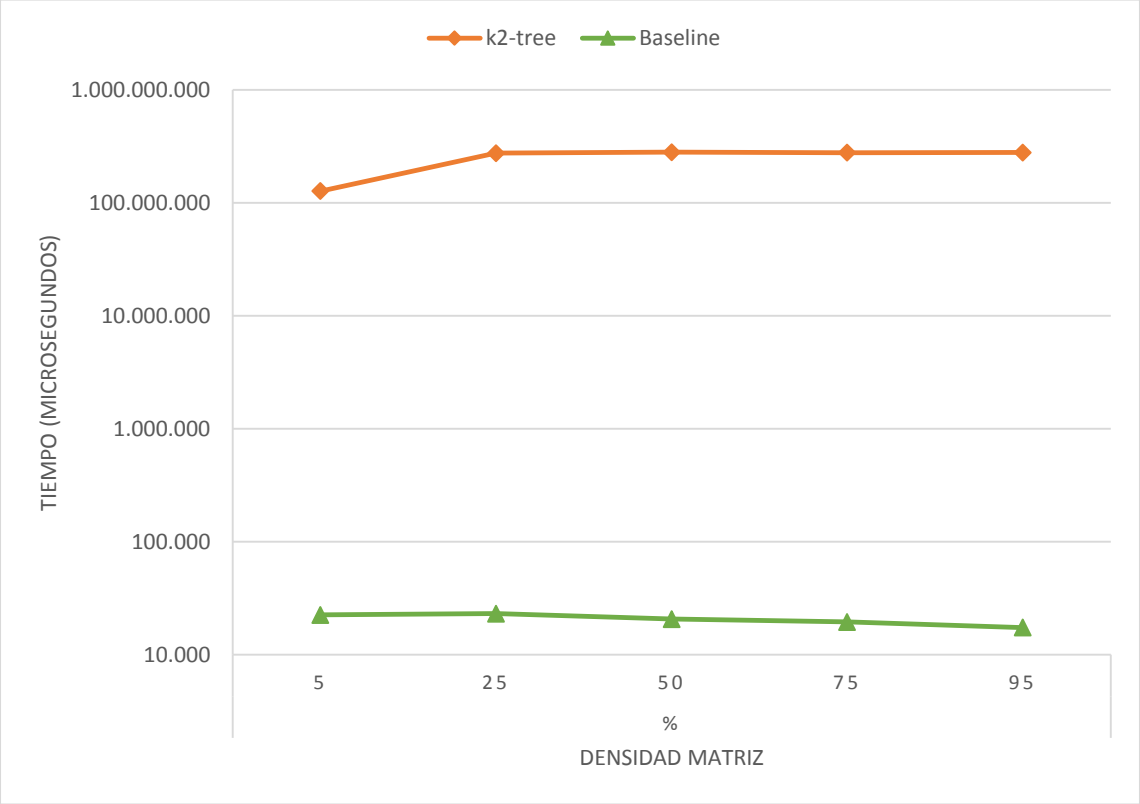

Figura 69: Gráfico mediciones de tiempo complemento n 5000.

7.3.1.2. Análisis pruebas Densidad v/s Tiempo Complemento

Para las pruebas que confrontan el tiempo que se emplea al calcular el complemento con la densidad de la matriz de adyacencia:

- Es posible observar que en la representación con k^2-tree, para densidades bajas contenidas en la matriz de adyacencia (5%) se obtiene un mejor tiempo de ejecución que a mayor densidad, para el mismo orden de la matriz.
- La representación original, baseline, se obtiene un tiempo de ejecución prácticamente constante, para distintos órdenes de matriz.
- El tiempo de ejecución aumenta exponencialmente a medida que aumenta el orden de la matriz de adyacencia.
- Dentro de un mismo orden de matriz, se observa que a menor porcentaje de densidad de matriz, menor es el tiempo que ocupa tanto en el baseline como en el k^2-tree.

7.3.1.3. Pruebas Orden de Matriz v/s Tamaño algoritmo Complemento

Orden de la matriz	Tamaño archivo Entrada Baseline (kB)	Tamaño de archivo Salida Baseline (kB)	Tamaño archivo Entrada k^2-tree (kB)*	Tamaño de archivo Salida k^2-tree (kB)*	Porcentaje compresión archivos de Salida (%)
104	1,4	1,4	0,568	0,596	57,43
520	33,8	33,8	13,8	18,9	44,08
1.000	125,0	125,0	50,8	42,6	65,92
3.000	1.100	1.100	457,7	409,8	62,74
5.000	3.100	3.100	1.270,6	1.197,1	61,38

Tabla 8: Mediciones tamaño complemento.

* El tamaño de archivo de entrada y el de salida k^2-tree, son la suma del tamaño del archivo que almacena T y L.

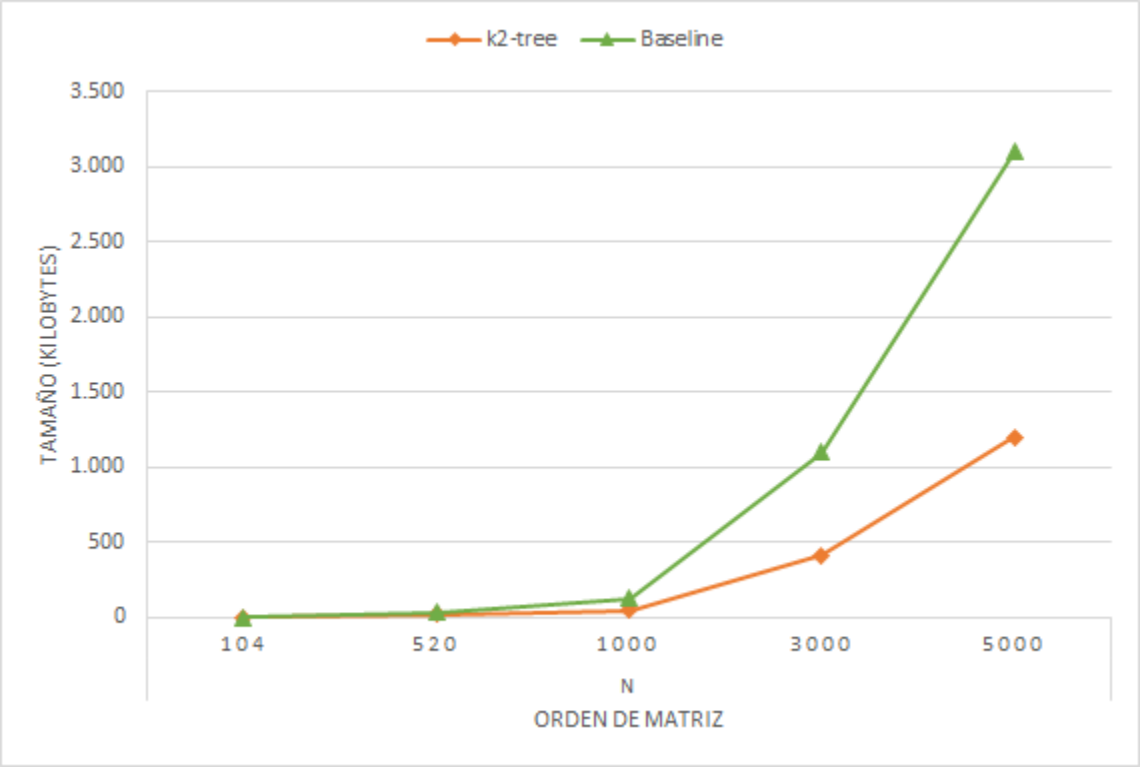

Tabla 9: Gráfico mediciones de tamaño complemento.

7.3.1.4. Análisis pruebas Orden de Matriz v/s Tamaño algoritmo Complemento

Para las pruebas que confrontan el orden de la matriz con el tamaño de los archivos de salida luego del cálculo del complemento:

- Comparando los resultados luego del procesamiento del algoritmo con los datos de entrada, se puede verificar que a menor densidad, el tamaño de los archivos resultantes es mayor. Esto se explica, debido a que el mismo procesamiento del algoritmo devuelve un k^2-tree con mayor densidad (elemento negado del proceso original), ocupando por tanto más espacio. Lo mismo sucede viceversa, es decir que a mayores densidades de matriz, el tamaño del archivo resultante es menor.

7.3.2. Pruebas algoritmo Diferencia

7.3.2.1. Pruebas Densidad v/s Tiempo algoritmo Diferencia

Diferencia	n = 104	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	541	7.848,024	256,061
25%	2.704	9.815,931	264,883
50%	5.408	12.327,909	242,949
75%	8.112	11.688,947	300,884
95%	10.276	10.169,029	242,949

Tabla 10: Mediciones de tiempo diferencia n 104.

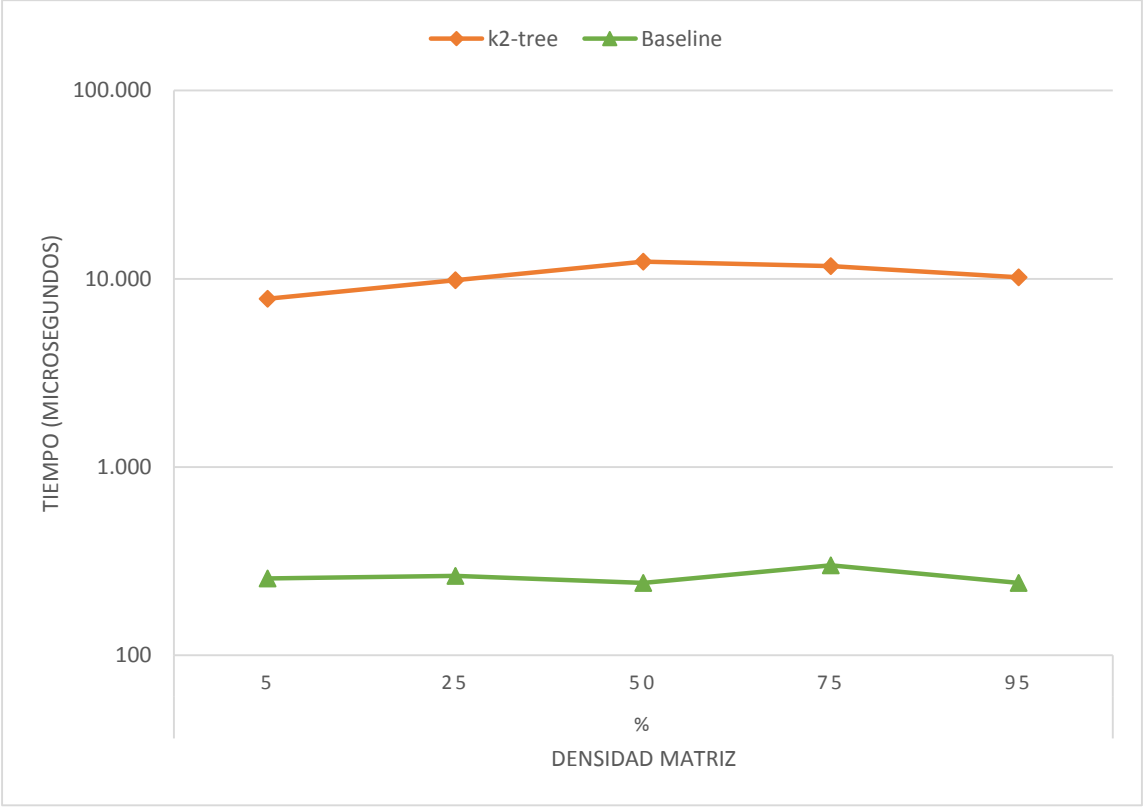

Figura 70: Gráfico mediciones de tiempo complemento n 104.

Diferencia	n = 520	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	13.520	165.019,035	309,944
25%	67.600	251.583,099	594,854
50%	135.200	229.223,966	627,994
75%	202.800	215.896,844	597,954
95%	256.880	211.066,007	586,986

Tabla 11: Mediciones de tiempo diferencia n 520.

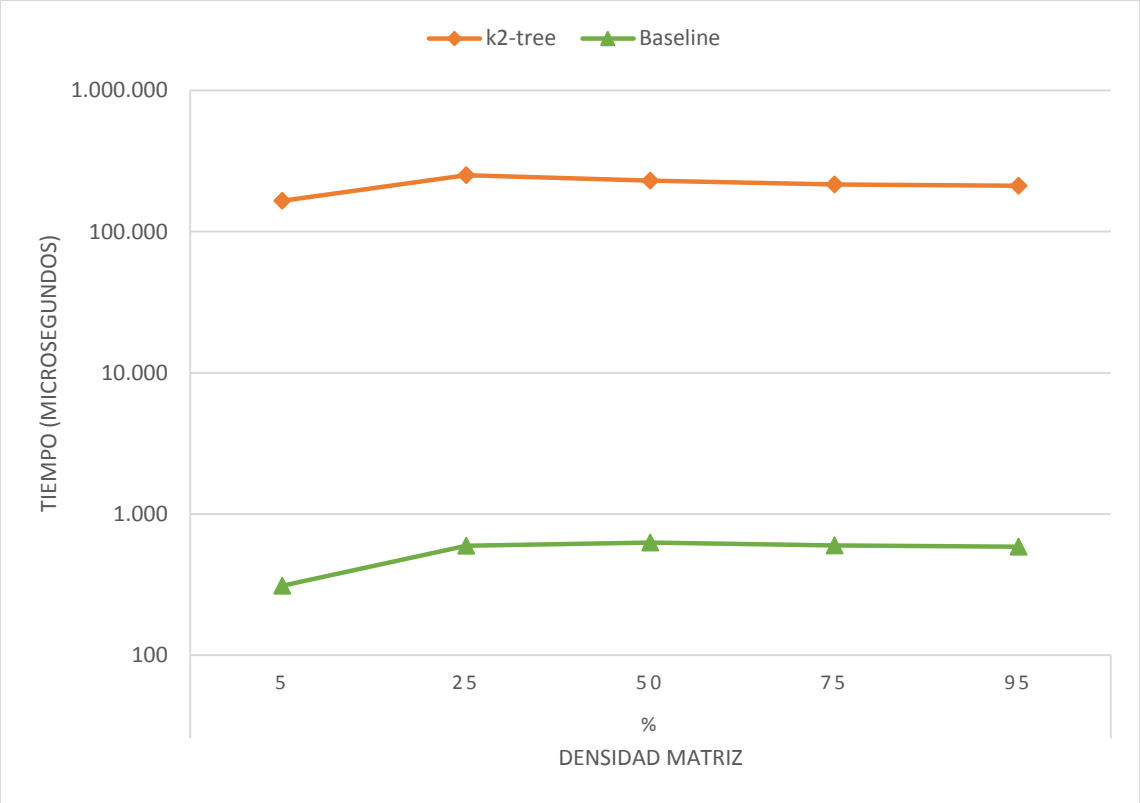

Figura 71: Gráfico mediciones de tiempo diferencia n 520.

Diferencia	n = 1.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	50.000	876.065,969	1.515,150
25%	250.0000	1.607.909,917	1.502,990
50%	500.0000	1.465.328,931	1.543,998
75%	750.000	1.397.323,131	1.205,205
95%	950.000	1.392.953,157	1.512,050

Tabla 12: Mediciones de tiempo diferencia n 1000.

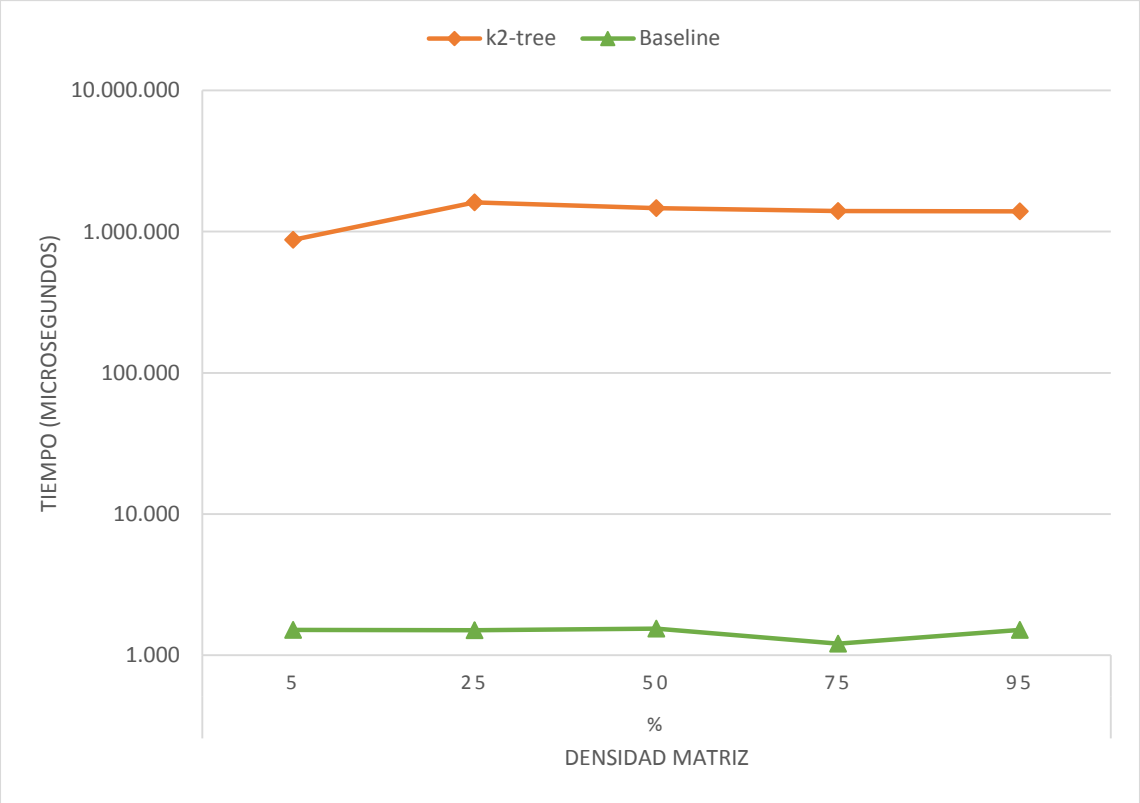

Figura 72: Gráfico mediciones de tiempo diferencia n 1000.

Diferencia	n = 3.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	450.000	35.723.181,962	8.008,003
25%	2.250.000	86.246.426,820	11.045,932
50%	4.500.000	77.398.851,156	8.932,8289
75%	6.750.000	74.904.621,839	9.495,0199
95%	8.550.000	73.058.510,780	11.314,868

Tabla 13: Mediciones de tiempo diferencia n 3000.

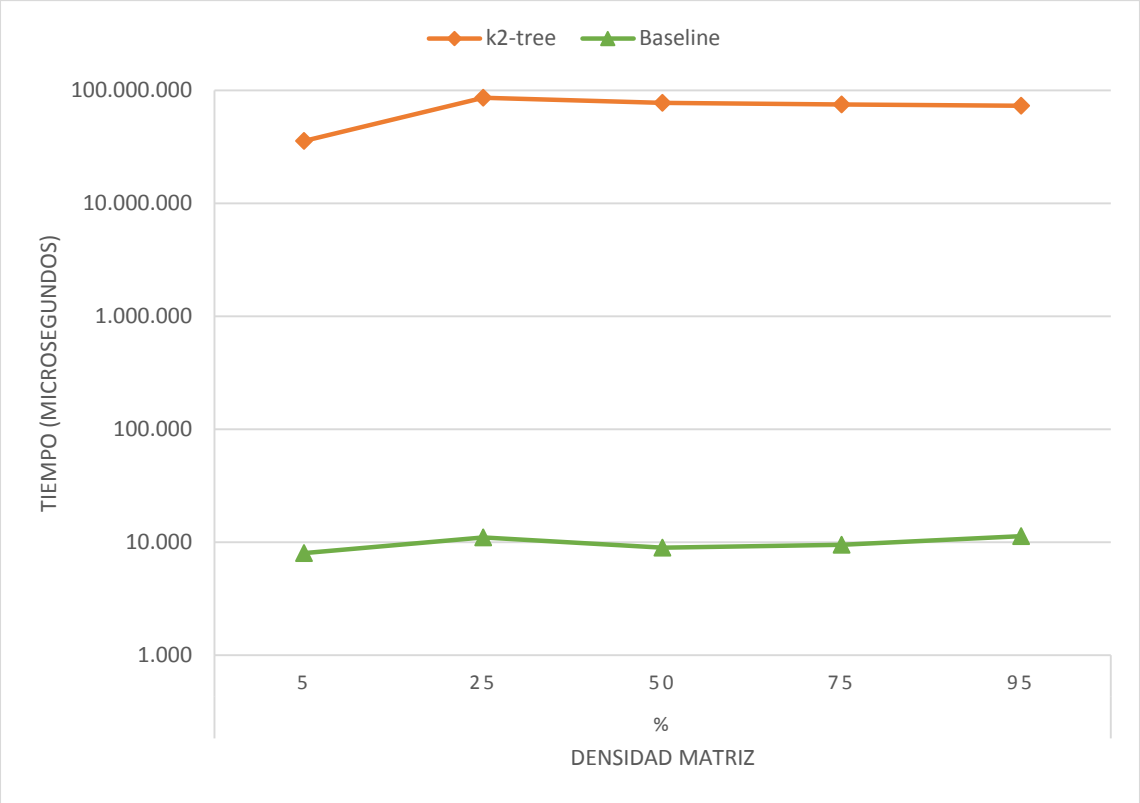

Figura 73: Gráfico mediciones de tiempo diferencia n 3000.

Diferencia	n = 5.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	1.250.000	252.308.656,930	19.227,981
25%	6.250.000	628.544.302,940	24.168,968
50%	12.500.000	574.013.015,985	24.724,960
75%	18.750.000	542.084.116,935	20.795,822
95%	23.750.000	537.595.818,996	23.901,939

Tabla 14: Mediciones de tiempo diferencia n 5000.

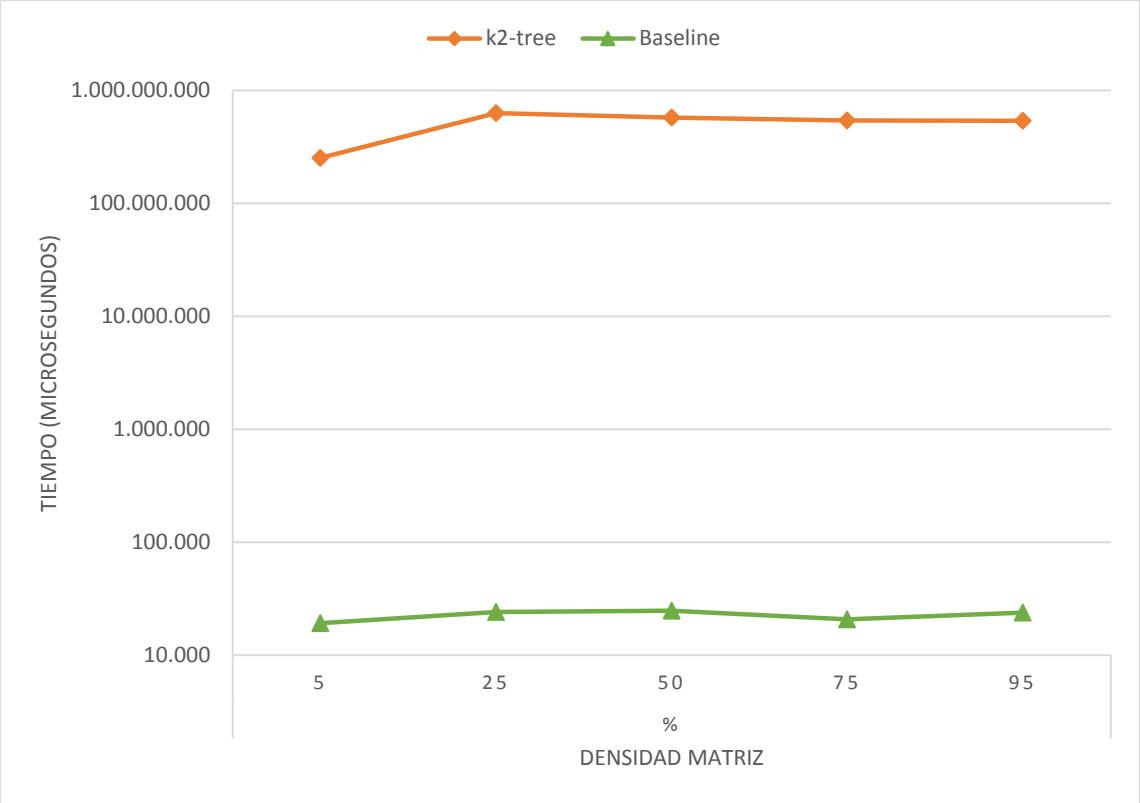

Figura 74: Gráfico mediciones de tiempo n 5000.

7.3.2.2. Análisis pruebas Densidad v/s Tiempo algoritmo Diferencia

Para las pruebas que confrontan el tiempo que se emplea al calcular la diferencia con la densidad de la matriz de adyacencia:

- Para bajas densidades de matriz se obtiene mejor tiempo de ejecución para el k^2-tree, al aumentar la densidad de 5% a 25%, el tiempo se incrementa exponencialmente. Luego, de 50% al 95% de densidad se obtiene una leve disminución del tiempo.
- El tiempo usado por el baseline es, como en el complemento, es similar al gráfico de una función uniforme.

7.3.2.3. Pruebas Orden de Matriz v/s Tamaño algoritmo Diferencia

Orden de la matriz	Tamaño archivo Entrada Baseline (kB)	Tamaño de archivo Salida Baseline (kB)	Tamaño archivo Entrada k^2-tree (kB)*	Tamaño de archivo Salida k^2-tree (kB)*	Porcentaje compresión archivos de Salida (%)
104	1,4	1,4	0,568	0,116	91,71
520	33,8	33,8	13,8	6,9	79,59
1.000	125,0	125,0	50,8	26,1	79,12
3.000	1.100	1.100	457,7	409,8	62,75
5.000	3.100	3.100	1.270,6	648,7	79,07

Tabla 15: Mediciones tamaño diferencia.

* El tamaño de archivo de entrada y el de salida k^2-tree, son la suma del tamaño del archivo que almacena T y L.

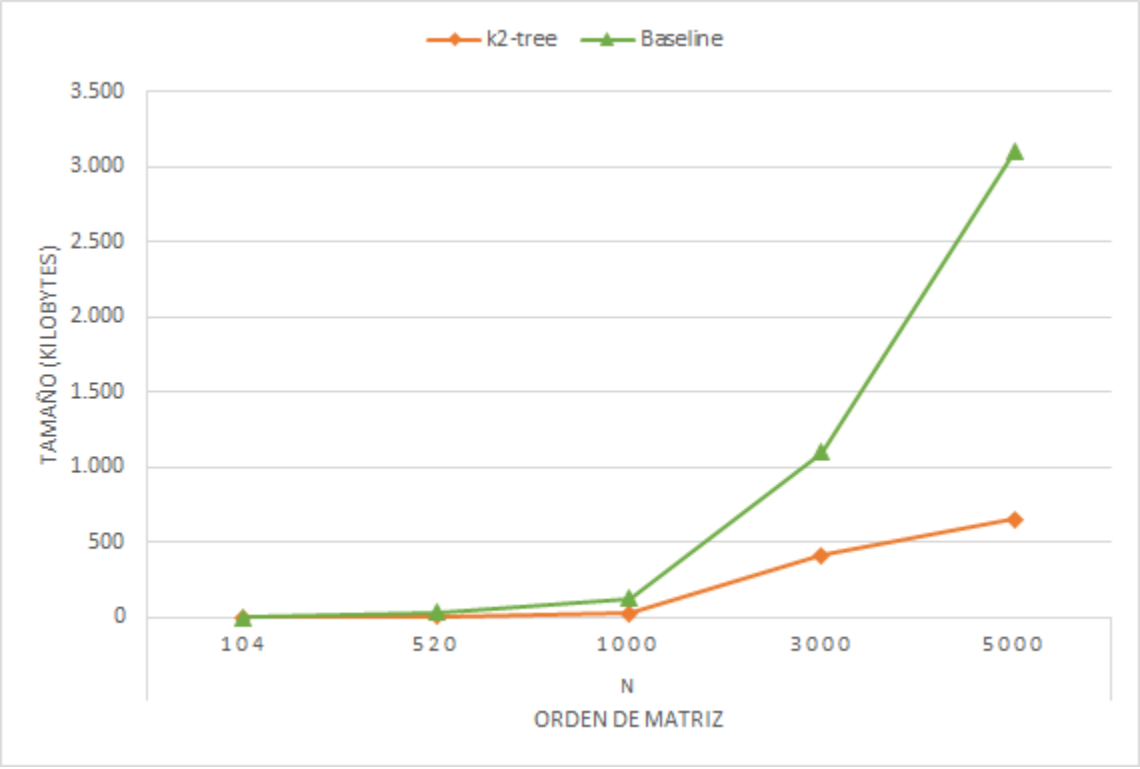

Figura 75: Gráfico mediciones tamaño diferencia.

7.3.2.4. Análisis pruebas Orden de Matriz v/s Tamaño algoritmo Diferencia

Para las pruebas que confrontan el orden de la matriz con el tamaño de los archivos de salida luego del cálculo de la diferencia:

- En relación a los tamaños de los resultados de representación original, el comportamiento es similar al obtenido en el complemento.
- Los porcentajes de compresión de los datos son similares para las diferentes densidades de matriz (76,5% en promedio).

7.3.3. Pruebas algoritmo Unión

7.3.3.1. Pruebas Densidad v/s Tiempo algoritmo Unión

Unión	n = 104	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	541	6.193,876	269,175
25%	2.704	9.138,107	243,902
50%	5.408	9.184,122	228,881
75%	8.112	7.922,887	265,814
95%	10.276	6.034,851	267,029

Tabla 16: Mediciones de tiempo unión n 104.

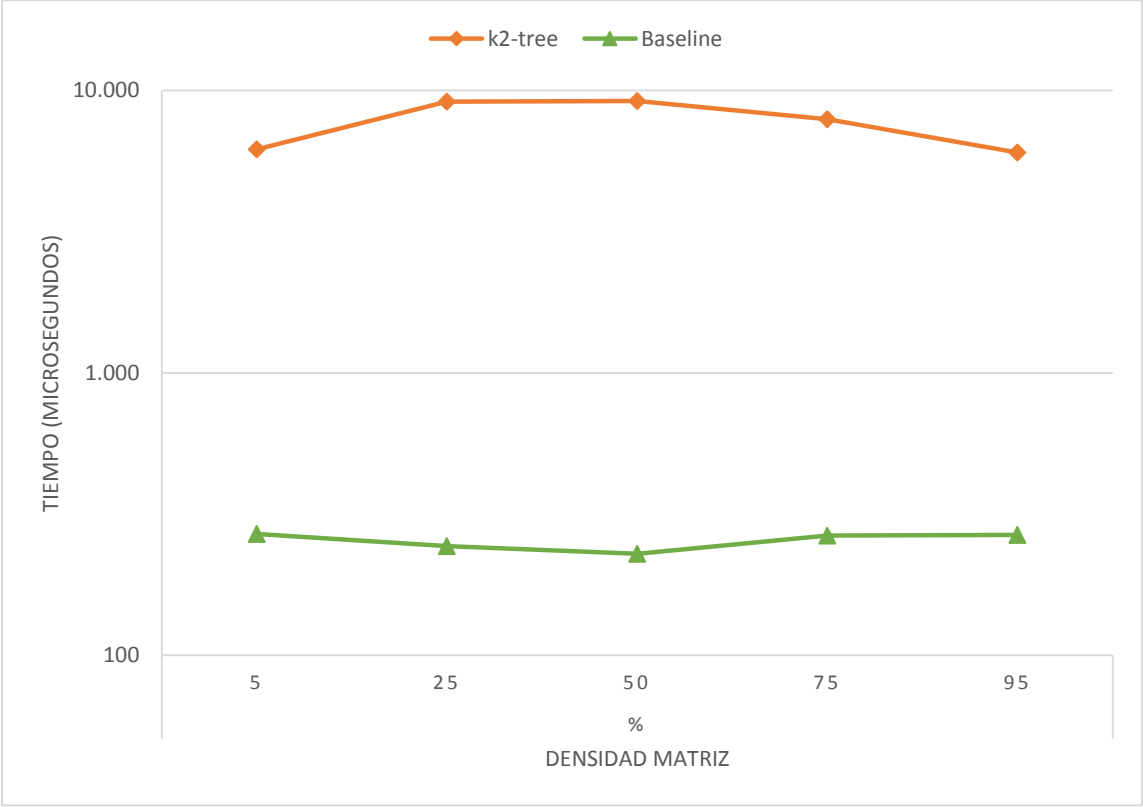

Figura 76: Gráfico mediciones de tiempo unión n 104.

Unión	n = 520	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	13.520	108.582,019	637,054
25%	67.600	180.691,004	585,079
50%	135.200	179.953,098	588,894
75%	202.800	181.446,075	619,888
95%	256.880	177.721,023	604,152

Tabla 17: Mediciones de tiempo unión n 520.

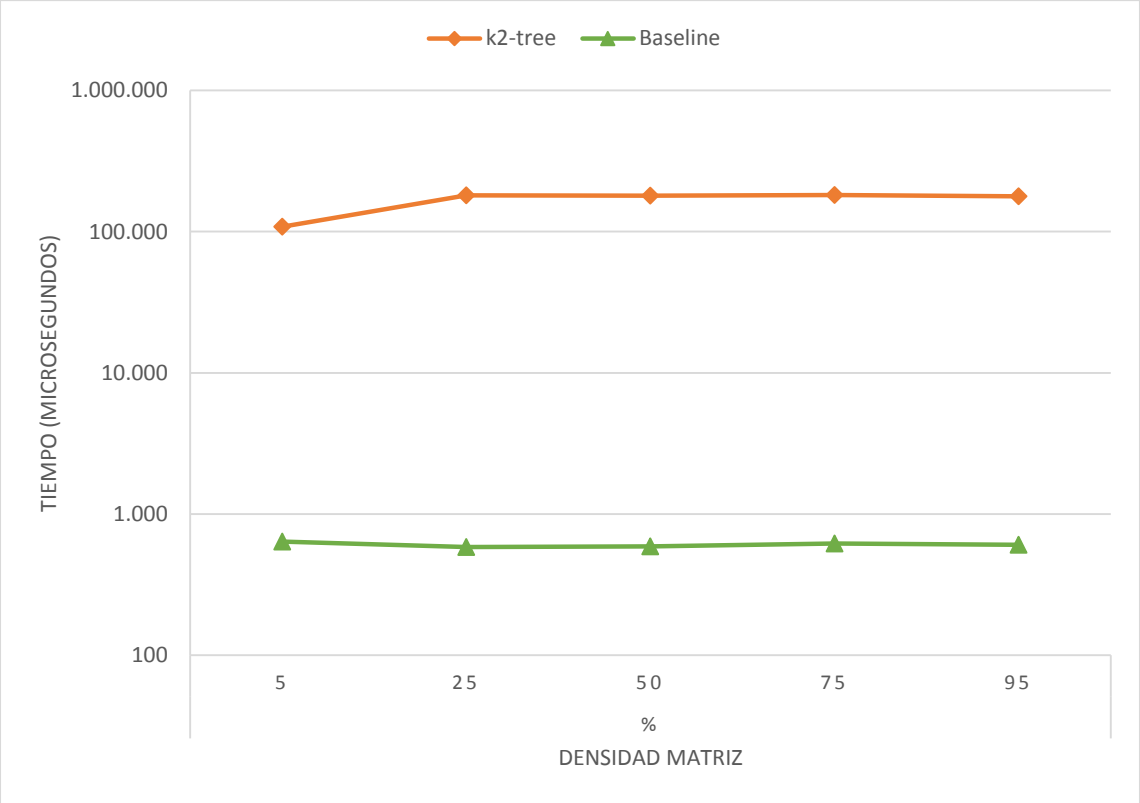

Figura 77: Gráfico mediciones de tiempo unión n 520.

Unión	n = 1.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	50.000	642.899,036	1.523,972
25%	250.000	1.229.735,851	1.446,962
50%	500.000	1.234.740,972	1.513,004
75%	750.000	1.235.476,017	1.475,811
95%	950.000	1.233.588,934	1.457,929

Tabla 18: Mediciones de tiempo unión n 1000.

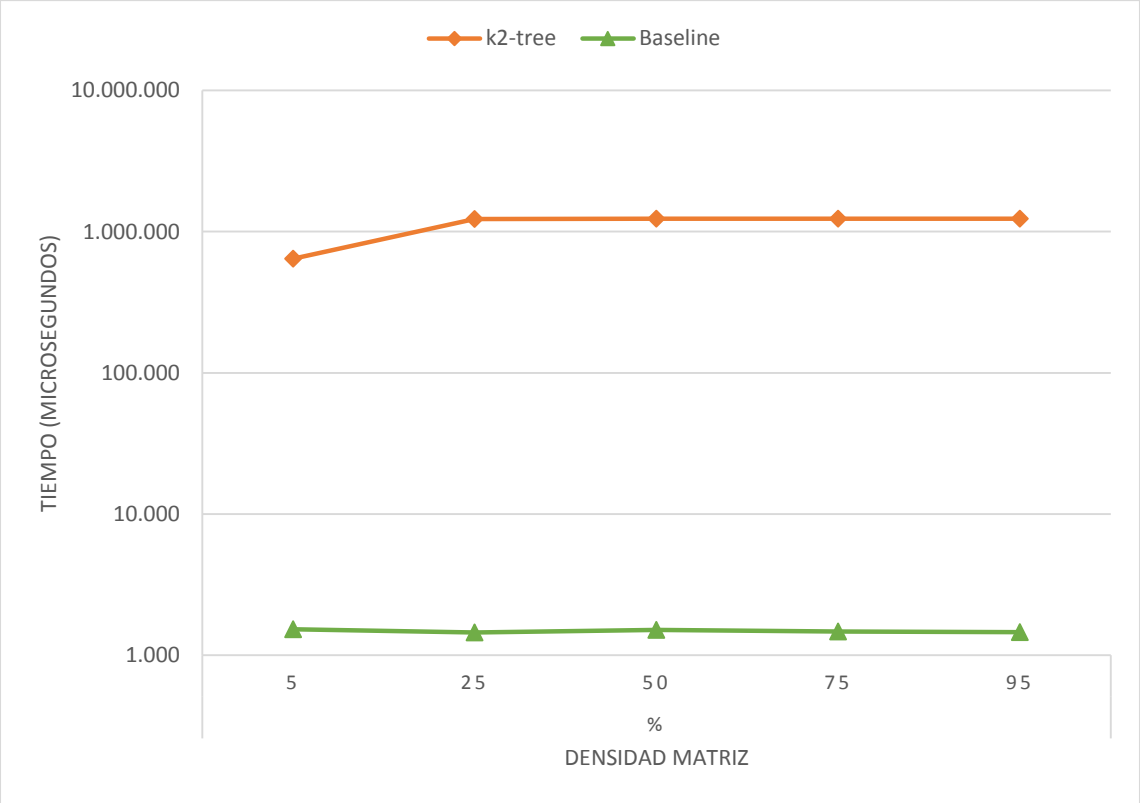

Figura 78: Gráfico mediciones de tiempo unión n 1000.

Unión	n = 3.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	450.000	31.558.917,045	10.610,818
25%	2.250.000	68.878.183,126	11.342,048
50%	4.500.000	70.189.818,143	11.605,024
75%	6.750.000	70.375.468,015	10.591,983
95%	8.550.000	70.021.248,817	9.415,149

Tabla 19: Mediciones de tiempo unión n 3000.

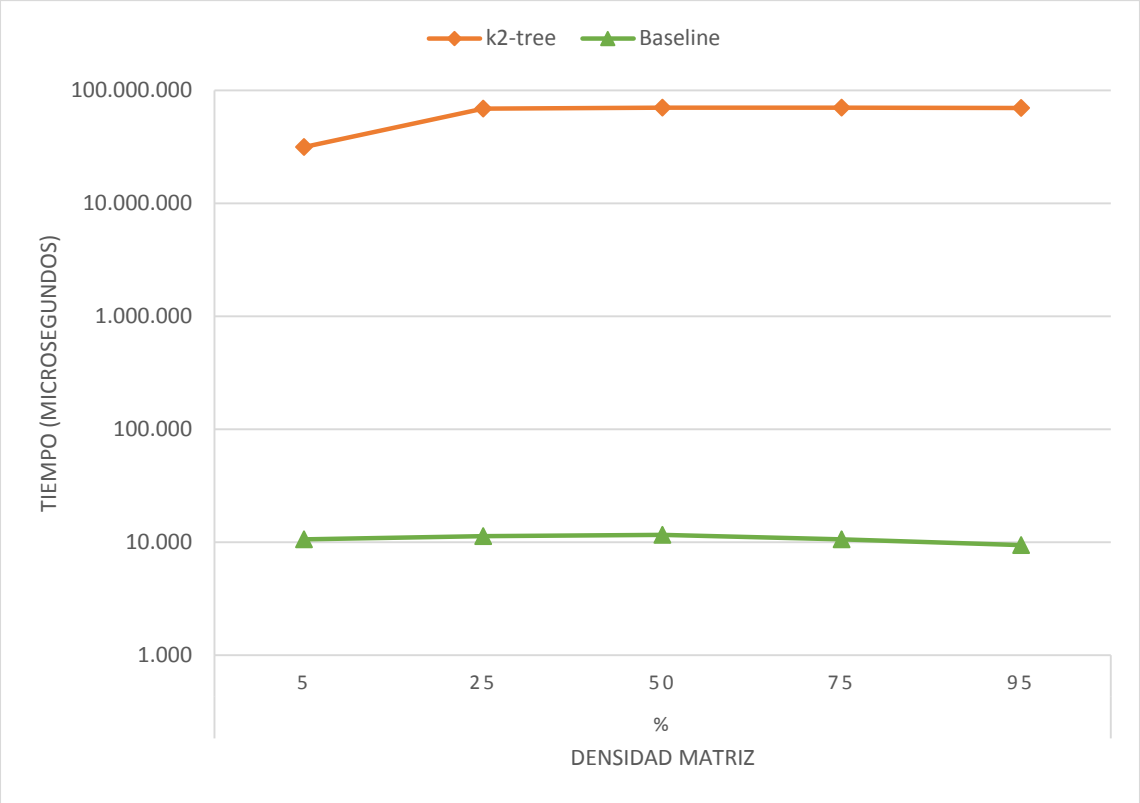

Figura 79: Gráfico mediciones de tiempo unión n 3000.

Unión	n = 5.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	1.250.000	233.956.813,812	21.180,152
25%	6.250.000	519.101.983,070	27.582,884
50%	12.500.000	520.009.629,011	22.250,891
75%	18.750.000	520.041.448,116	20.936,966
95%	23.750.000	524.712.630,033	20.599,127

Tabla 20: Mediciones de tiempo unión n 5000.

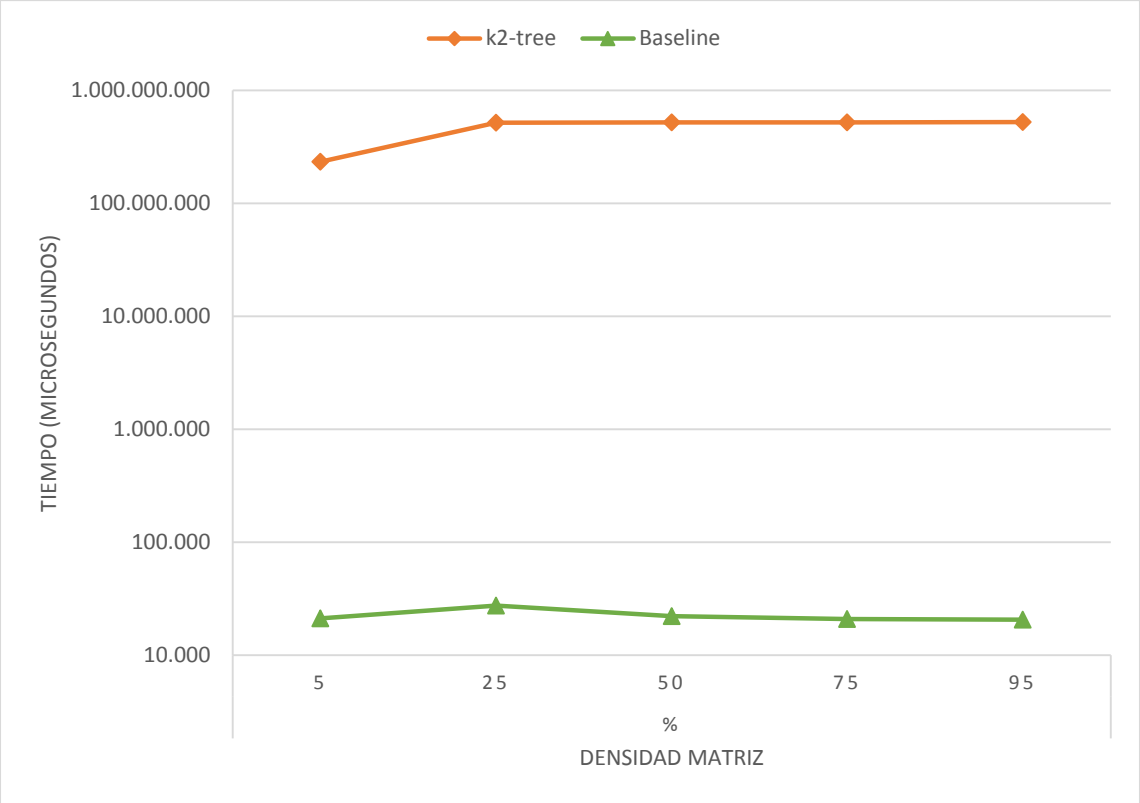

Figura 80: Gráfico mediciones de tiempo unión n 5000.

7.3.3.2. Análisis pruebas Densidad v/s Tiempo algoritmo Unión

Para las pruebas que confrontan el tiempo que se emplea al calcular la unión con la densidad de la matriz de adyacencia:

- Los tiempos aumentan aproximadamente un 50%, desde una densidad de 5% a 25%, y luego se mantienen con una tendencia constante. Esto es válido para todos los n, con excepción de 140, donde se ve un comportamiento más irregular.

7.3.3.3. Pruebas Orden de Matriz v/s Tamaño algoritmo Unión

Orden de la matriz	Tamaño archivo Entrada Baseline (kB)	Tamaño de archivo Salida Baseline (kB)	Tamaño archivo Entrada k^2-tree (kB)*	Tamaño de archivo Salida k^2-tree (kB)*	Porcentaje compresión archivos de Salida
104	1,4	1,4	0,568	0,313	77,64
520	33,8	33,8	13,8	7,489	77,843
1.000	125,0	125,0	50,8	35,138	71,889
3.000	1.100	1.100	457,7	250	77,273
5.000	3.100	3.100	1.270,6	687,5	77,822

Tabla 21: Mediciones tamaño unión.

* El tamaño de archivo de entrada y el de salida k^2-tree, son la suma del tamaño del archivo que almacena T y L.

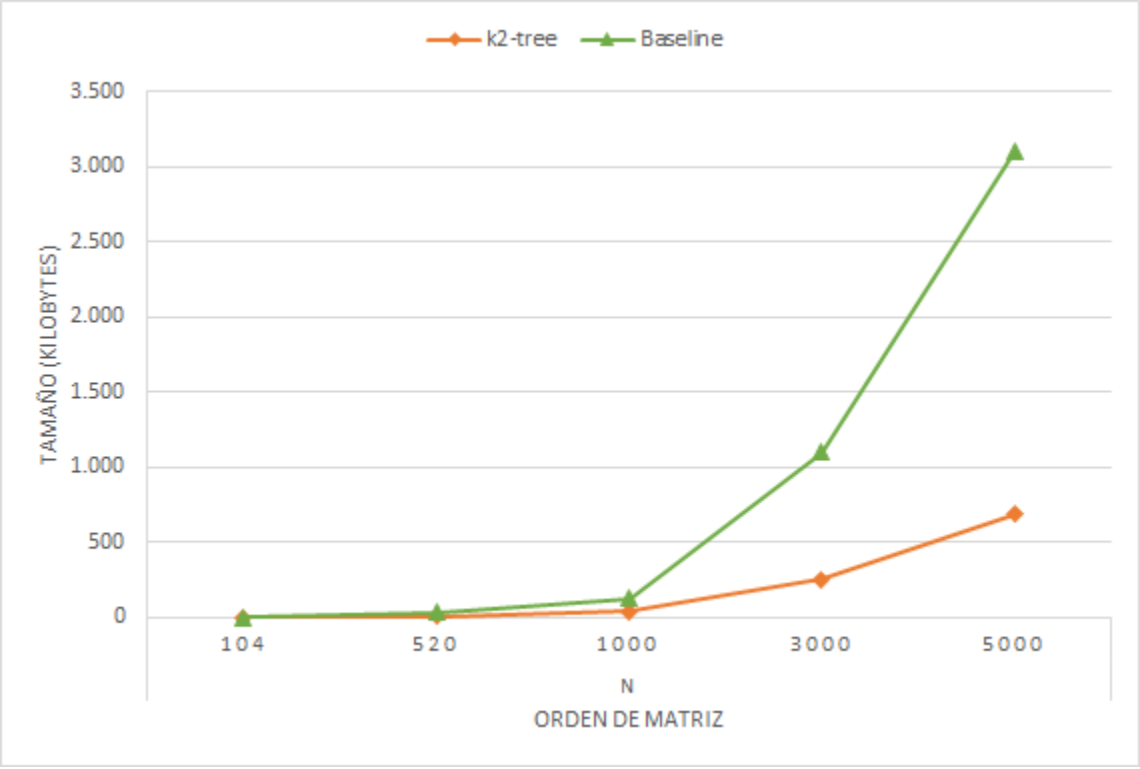

Figura 81: Gráfico mediciones tamaño unión.

7.3.3.4. Análisis pruebas Orden de Matriz v/s Tamaño algoritmo Unión

Para las pruebas que confrontan el orden de la matriz con el tamaño de los archivos de salida luego del cálculo de la unión:

- El porcentaje de compresión de los archivos se mantiene constante en aproximadamente un 77%.

7.3.4. Pruebas algoritmo Intersección

7.3.4.1. Pruebas Densidad v/s Tiempo algoritmo Intersección

Intersección	n = 104	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	541	4.832,983	240,087
25%	2.704	4.914,045	229,835
50%	5.408	4.902,840	231,981
75%	8.112	4.914,045	241,041
95%	10.276	4.961,967	241,995

Tabla 22: Mediciones de tiempo intersección n 104.

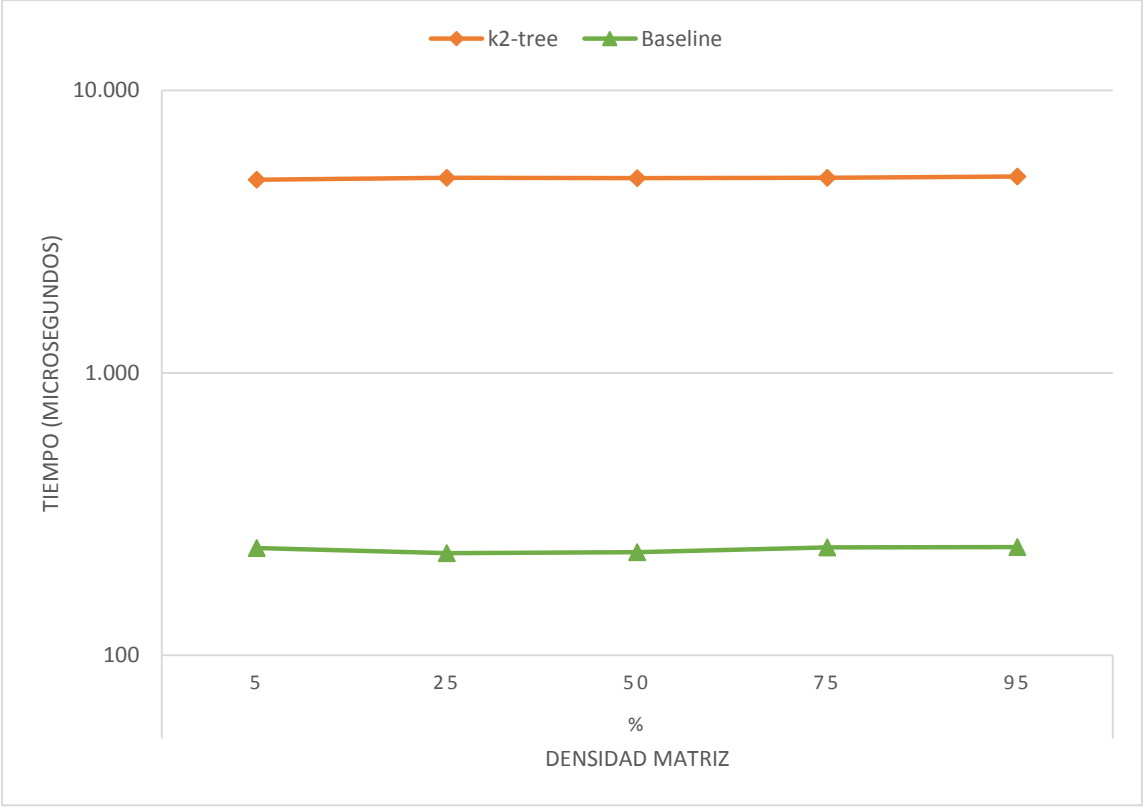

Figura 82: Gráfico mediciones de tiempo intersección n 104.

Intersección	n = 520	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	13.520	76.122,046	326,156
25%	67.600	75.402,021	895,023
50%	135.200	74.456,930	607,967
75%	202.800	82.046,031	586,987
95%	256.880	78.516,006	586,033

Tabla 23: Mediciones de tiempo intersección n 520.

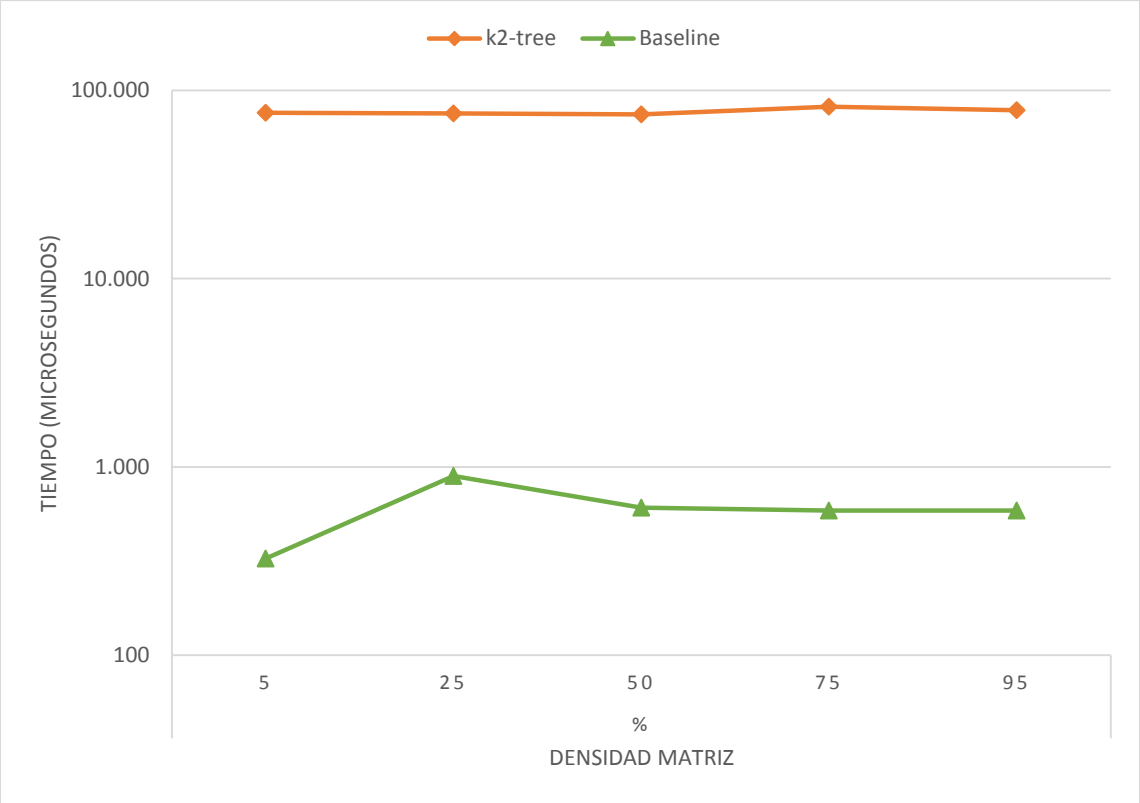

Figura 83: Gráfico mediciones de tiempo intersección n 520.

Intersección	n = 1.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	50.000	610.965,013	1.473,904
25%	250.0000	602.473,020	1.476,049
50%	500.0000	603.663,921	1.492,977
75%	750.000	617.671,012	1.482,964
95%	950.000	605.350,971	1.581,907

Tabla 24: Mediciones de tiempo intersección n 1000.

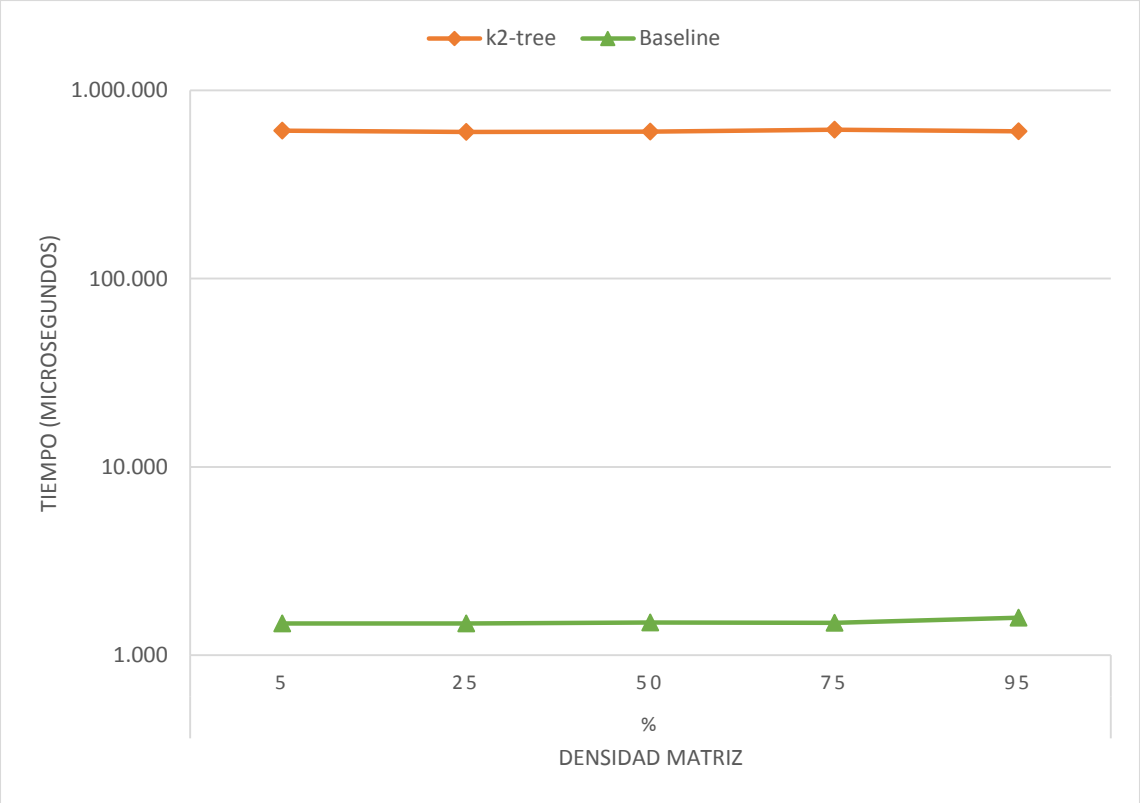

Figura 84: Gráfico mediciones de tiempo intersección n 1000.

Intersección	n = 3.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	450.000	3.762.698,173	10.702,133
25%	2.250.000	3.719.992,876	9.149,074
50%	4.500.000	3.738.810,062	10.936,975
75%	6.750.000	3.693.124,055	11.321,068
95%	8.550.000	3.713.045,120	11.488,914

Tabla 25: Mediciones de tiempo intersección n 3000.

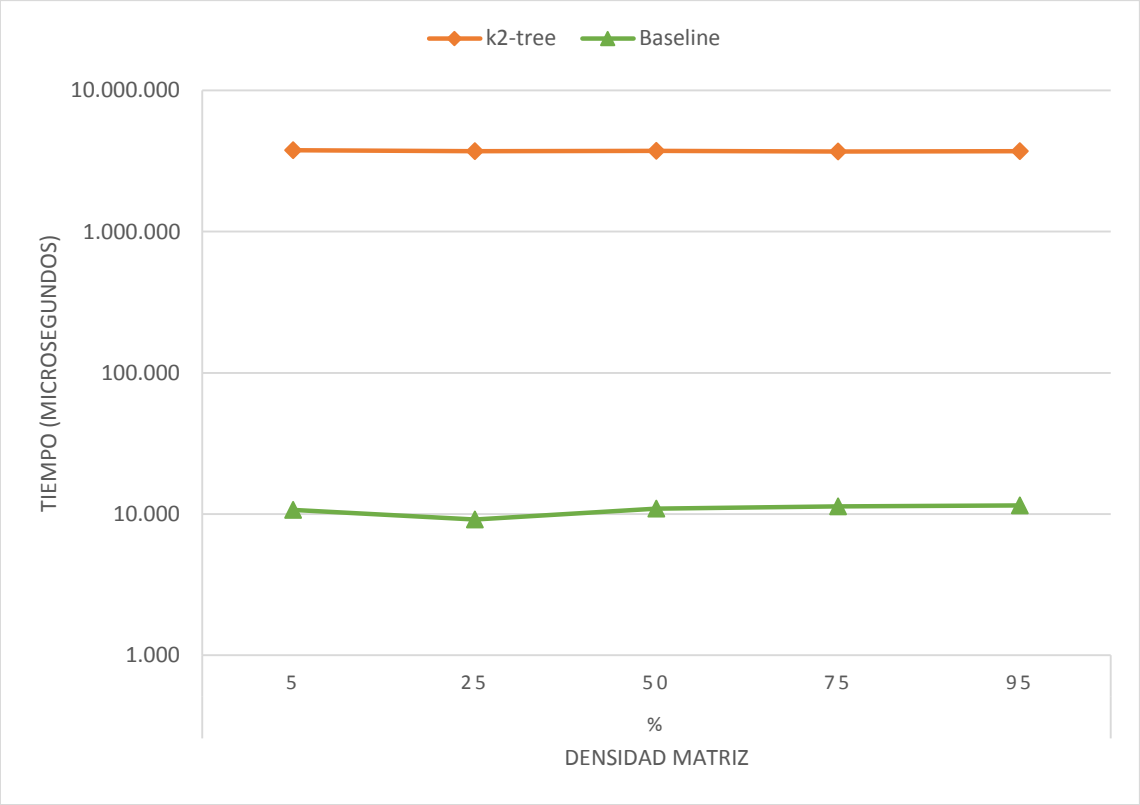

Figura 85: Gráfico mediciones de tiempo intersección n 3000.

Intersección	n = 5.000	k^2-tree	Baseline
Densidad matriz	Cantidad de 1s	Tiempo D. Uniforme (µs)	Tiempo D. Uniforme (µs)
5%	1.250.000	7.978.863,000	23.180,008
25%	6.250.000	8.073.734,998	18.998,146
50%	12.500.000	8.013.105,154	20.844,936
75%	18.750.000	8.016.180,992	21.025,181
95%	23.750.000	7.977.558,135	19.582,987

Tabla 26: Mediciones de tiempo intersección n 5000.

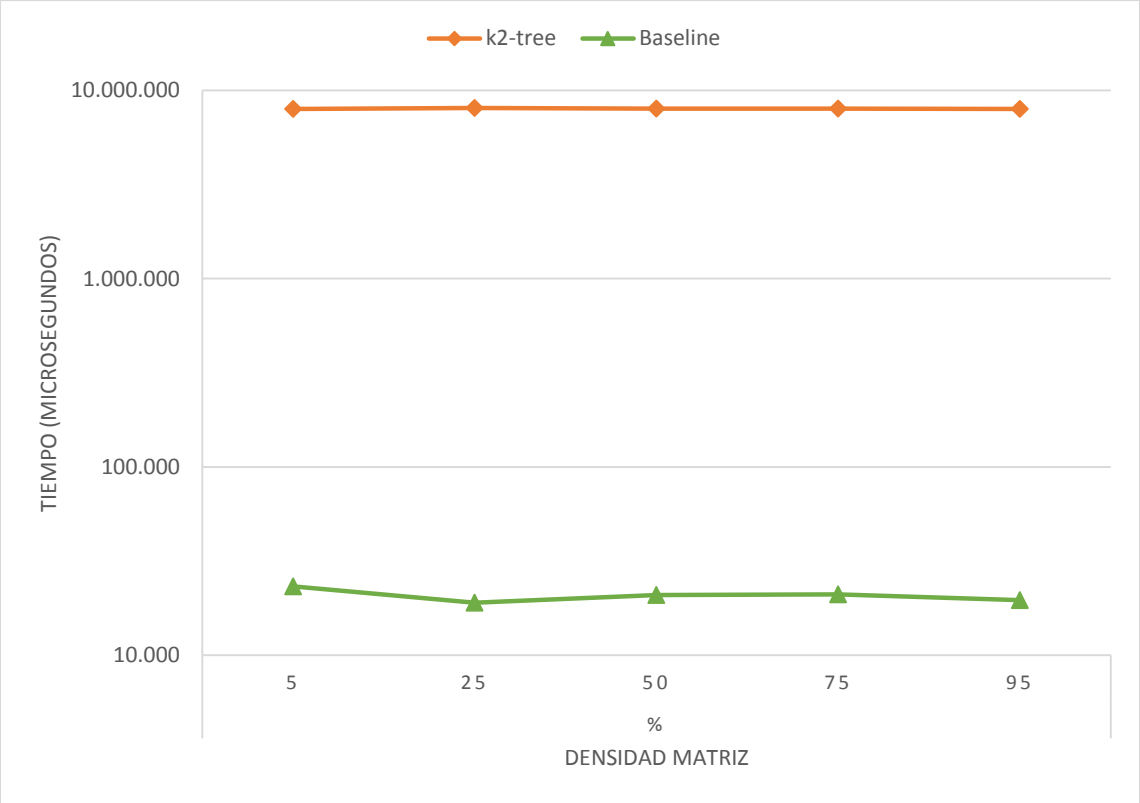

Figura 86: Gráfico mediciones de tiempo intersección n 5000.

7.3.4.2. Análisis pruebas Densidad v/s Tiempo algoritmo Intersección

Para las pruebas que confrontan el tiempo que se emplea al calcular la unión con la densidad de la matriz de adyacencia:

- Los tiempos aumentan aproximadamente un 50%, desde una densidad de 5% a 25%, y luego se mantienen con una tendencia constante. Esto es válido para todos los n, con excepción de 140, donde se ve un comportamiento más irregular.

7.3.4.3. Pruebas Orden de Matriz v/s Tamaño algoritmo Intersección

Orden de la matriz	Tamaño archivo Entrada Baseline (kB)	Tamaño de archivo Salida Baseline (kB)	Tamaño archivo Entrada k^2-tree (kB)*	Tamaño de archivo Salida k^2-tree (kB)*	Porcentaje compresión archivos de Salida
104	1,4	1,4	0,568	0,508	63,714
520	33,8	33,8	13,8	6,9	79,585
1.000	125,0	125,0	50,8	43,7	65,04
3.000	1.100	1.100	457,7	265,3	75,882
5.000	3.100	3.100	1.270,6	576,2	81,413

Tabla 27: Mediciones tamaño intersección.

* El tamaño de archivo de entrada y el de salida k^2-tree, son la suma del tamaño del archivo que almacena T y L.

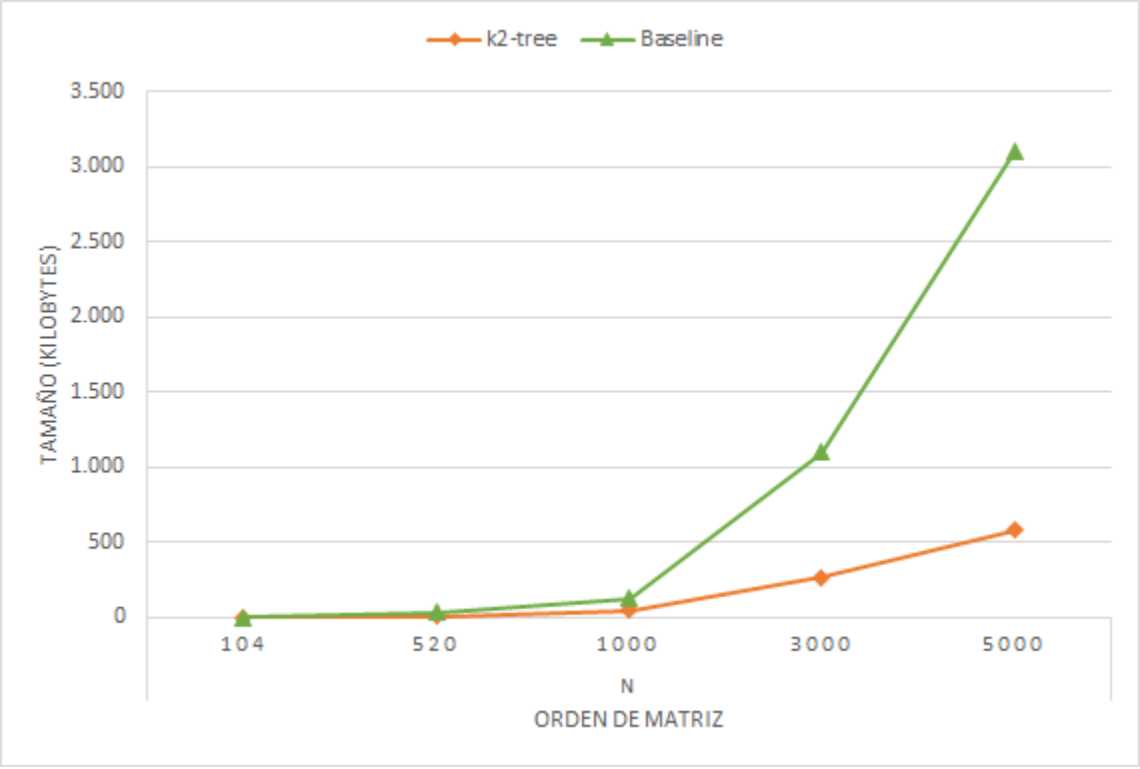

Figura 87: Gráfico mediciones tamaño intersección.

7.3.4.4. Análisis pruebas Orden de Matriz v/s Tamaño algoritmo Intersección

Para las pruebas que confrontan el orden de la matriz con el tamaño de los archivos de salida luego del cálculo de la intersección:

- Nuevamente, el comportamiento fue similar al de los algoritmos anteriores, de esta manera no se obtiene nuevas conclusiones.

8. Análisis General de Resultados

- El k utilizado es siempre 2 para así poder enfocar el análisis en las variables que se debían medir.
- Los tiempos obtenidos por el k^2-tree se alejan bastante de los obtenidos por la representación original, baseline, esto se debe al uso de estructuras de construcción y búsqueda poco eficientes. Con n = 104 y una densidad de 5% se tiene que en promedio las operaciones sobre el k^2-tree tardan 5872,18 microsegundos mientras que el baseline demora 194,83 microsegundos. Siendo hasta 30 veces más lenta el cálculo sobre el k^2-tree.
- En el k^2-tree, el algoritmo de diferencia y de unión son los que obtienen los peores tiempos de ejecución. Con un n de 5.000 y un 95% de densidad se emplean casi 9 minutos en realizar estas operaciones, mientras que para el complemento y la intersección se emplean 5 y menos de 1 minuto respectivamente. La gran cantidad de tiempo empleado por el algoritmo de unión se debe a que se realiza un recorrido en amplitud de los k^2-tree, verificando el valor de todos los nodos presentes en el árbol. Por otro lado, el tiempo empleado por el algoritmo de diferencia se debe a la cantidad de llamadas a la función `Copy` que se encarga de copiar de forma recursiva los valores almacenados en los nodos a partir de uno específico hasta llegar a sus nodos hojas.
- El algoritmo de intersección sobre k^2-tree obtiene tiempos de ejecución casi constantes. Esto es debido a que en este algoritmo solo es recursiva la llamada a sí mismo y no ocupa funciones recursivas adicionales cuando realiza el cálculo. Por lo tanto, al utilizar estructuras de datos ineficientes y ejecutar funciones recursivas sobre estas, aumenta el tiempo de cálculo de forma alarmante.
- Para las pruebas que confrontan el orden de la matriz con el tamaño de los archivos de salida del k^2-tree y la representación original, se obtienen gráficas similares entre las diferentes operaciones efectuadas, observándose un comportamiento exponencial.
- Al utilizar la representación original se obtienen tiempos de ejecución similares para todos los algoritmos. La causa es que en este algoritmo se revisa cada posición de la matriz independiente de la operación o el contenido de las casillas.
- Se aprecia una clara disminución del tamaño de la representación original comparado con los T y L generados por el k^2-tree. Esto se explica, debido al mismo funcionamiento del algoritmo, que elimina sectores de 0s, ahorrando así espacio en disco.
- El porcentaje de compresión de los archivos de salida del k^2-tree y los de la representación original, para todos los algoritmos es de un promedio aproximado de 70%.

9. Conclusiones y Trabajos Futuros

Durante el desarrollo del proyecto se profundizó en los aspectos teóricos así como en la utilidad práctica de los conceptos relacionados con el k^2-tree. Se comprobó el ahorro de utilización de memoria en disco al almacenar las estructuras, demostrando su utilidad para manipular grandes volúmenes de información con menores tamaños de almacenamiento.

El proyecto alcanzó su objetivo general, al implementar las operaciones binarias sobre la estructura comprimida, sin necesidad de acudir a la representación original. A pesar de esto, al finalizar las pruebas quedó en evidencia la ineficiencia en cuanto a tiempo de ejecución al utilizar el modelo de construcción de k^2-tree con listas enlazadas sobre la cual se basa este proyecto. Aunque el Rank permite ahorrar tiempo de búsqueda, el hecho de que existan muchos nodos retrasa demasiado el proceso. Por lo tanto, se descarta el uso de estas estructuras creadas para almacenar el k^2-tree.

De forma paralela, al optimizar el código de creación, se reforzaron buenas prácticas de programación, las cuales son fundamentales y permiten mejorar los tiempos de ejecución, y por consiguiente, mejorar los resultados obtenidos.

En otras implementaciones se ha demostrado que el k^2-tree cumple con el objetivo de representar una matriz de adyacencia de forma compacta y soporta en su estructura la navegación hacia adelante y atrás entre los nodos, así como también permite visualizar los diferentes enlaces existentes entre un rango de nodos. Aunque a la fecha de hoy no se ha demostrado que la realización de operaciones binarias sobre esta sea igual de eficiente al momento de ahorrar tiempo de procesamiento. La teoría indica que con una implementación adecuada, como por ejemplo utilizando otro tipo de estructura para almacenar el árbol sería factible. Por lo tanto, se propone como trabajo futuro:

- Optimizar la creación de k^2-tree, utilizada en este proyecto para mejorar los tiempos de búsqueda y respuesta, evitando recorrer listas ligadas de gran envergadura.
- Desarrollar las operaciones binarias directamente sobre otras implementaciones existentes.
- Creación de nuevas implementaciones de k^2-tree en el lenguaje de programación C o en algún otro lenguaje que ofrezca características adecuadas para el manejo de bits.

- Estudio de viabilidad de la utilización del k^2-tree en otros campos de investigación, ya que está desarrollado para permitir el acceso sobre la estructura comprimida.
- Implementación dinámica del k^2-tree que facilite la adición y eliminación de nodos, así como otras operaciones que se pueden aplicar sobre matrices de adyacencia.

10. Referencias

[1] Brisaboa N. R., Ladra S. y Navarro G., “K2-trees for compact web graph representation,” en String Processing and Information Retrieval, 2009, pp. 18–30.

[2] Bustos C., Esquivel S., Ludueña V., Reyes N., Roggero P., Chávez E. y Navarro G. “Análisis e Indexación de de Datos no Convencionales”<http://sedici.unlp.edu.ar/bitstream/handle/10915/19758/Documento_completo.pdf?sequence=1>

[3] Claude F. (2008), “Estructuras comprimidas para grafos de la web.” Tesis de Magíster, U. de Chile. <http://www.tesis.uchile.cl/bitstream/handle/2250/111736/claude_ff.pdf?sequence=1>

[4] De Kunder, M. (2015), World Wide Web Size, Daily Estimated Size of the World Wide Web. <http://www.worldwidewebsize.com/>

[5] Navarro, G. (2007), “Tutorial de Estructuras Compactas de Datos” <http://www.dcc.uchile.cl/~gnavarro/compactas.html>

[6] Ulloa J., Campos L. (2014). “Creación y manipulación de k^2-tree para la compresión de representaciones de datos espaciales”.

11. Bibliografía

- Biblioteca Universidad del Bío-Bío (marzo de 2012). Guía para la redacción en el estilo APA, 6º Edición. <http://werken.ubiobio.cl/html/docs/Apa_Edicion6.pdf>
- Carme Alvarez Faura, Josep Díaz Cort, María Serna Números, ISSN 0212-3096, Nº. 43-44, 2000 (Ejemplar dedicado a: Las matemáticas del siglo XX: una mirada en 101 artículos), págs. 477-480
- Wikipedia, the free encyclopedia. Webgraph. (marzo de 2014) <http://en.wikipedia.org/wiki/Webgraph>
- Yoichiro Hasebe (2013). RSyntaxTree. <http://yohasebe.com/rsyntaxtree/>

11. Anexos

11.1. Planificación Inicial del Proyecto

Para calendarizar las actividades que serían llevadas a cabo inicialmente se utilizó una Carta Gantt, en ella se detallan cada una de las actividades y el tiempo en que ejecutarán.

Id	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
1	Proyecto de Título	86 días	lun 01-09-14	lun 29-12-14		
2	Estudio	13 días	lun 01-09-14	mié 17-09-14		
3	Estudio del arte	10 días	lun 01-09-14	vie 12-09-14		
4	Definición de límites	3 días	lun 15-09-14	mié 17-09-14		
5	Análisis	25 días	lun 22-09-14	vie 24-10-14		
6	Análisis de implementaciones	10 días	lun 22-09-14	vie 03-10-14		
7	Diseño de experimentos	3 días	lun 06-10-14	mié 08-10-14		
8	Resultados de experimentos	7 días	jue 09-10-14	vie 17-10-14		
9	Evaluación de experimentos	4 días	lun 20-10-14	jue 23-10-14		
10	Elección de implementación	1 día	vie 24-10-14	vie 24-10-14		
11	Desarrollo	25 días	lun 27-10-14	vie 28-11-14		
12	Desarrollo de operaciones binarias	4 días	lun 27-10-14	jue 30-10-14		
13	Evaluación de rendimiento	5 días	lun 03-11-14	vie 07-11-14		
14	Desarrollo de Consulta de Datos	10 días	lun 10-11-14	vie 21-11-14		
15	Evaluacion de rendimiento	5 días	lun 24-11-14	vie 28-11-14		
16	Resultados	20 días	lun 01-12-14	lun 29-12-14		
17	Análisis general de resultados	5 días	lun 01-12-14	vie 05-12-14		
18	Conclusiones	5 días	lun 08-12-14	vie 12-12-14		
19	Borrador Informe Final	6 días	vie 19-12-14	vie 26-12-14		
20	Entrega informe final	0 días	lun 29-12-14	lun 29-12-14		

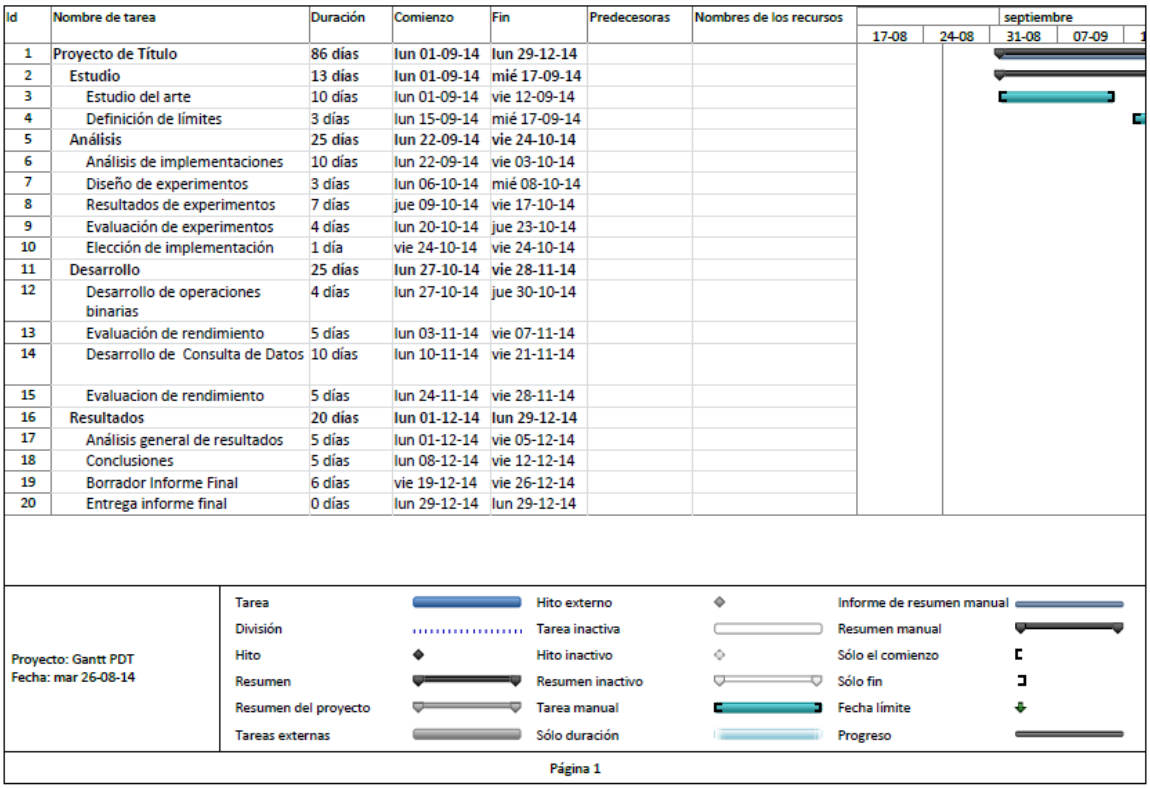

Figura 88: Planificación inicial A.

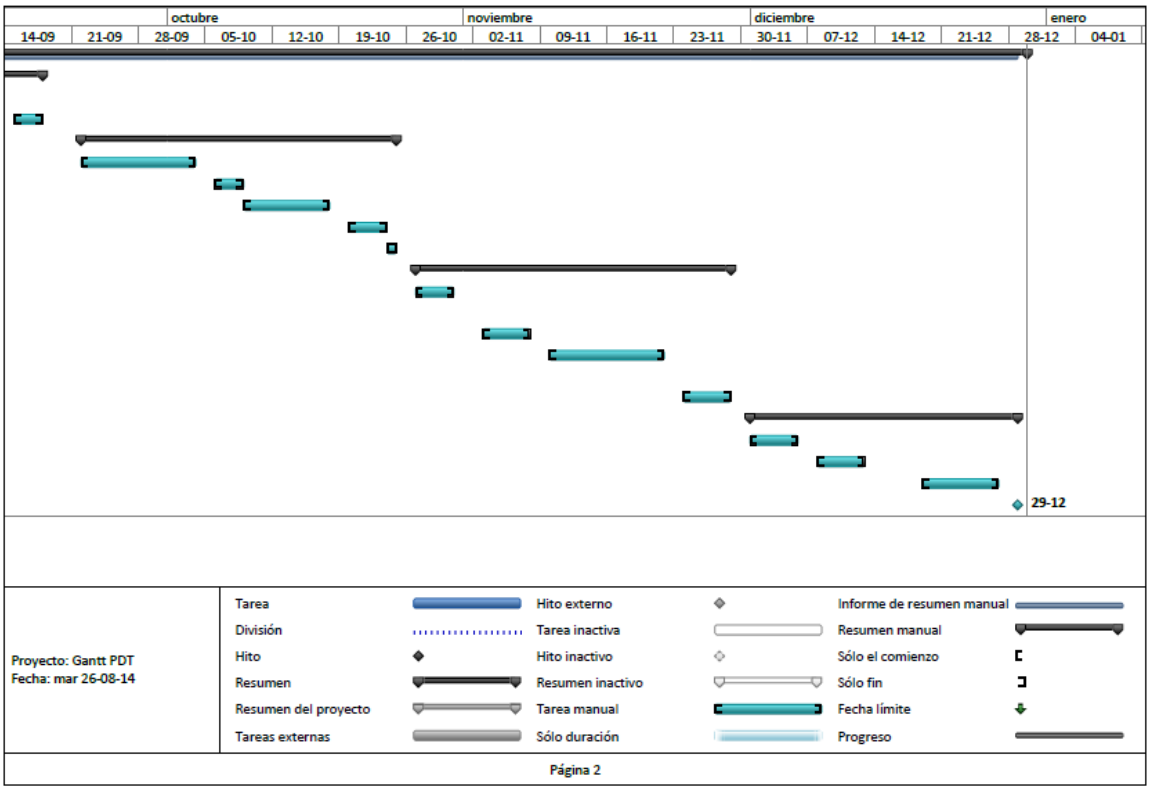

Figura 89: Planificación inicial B.

11.2. Manejo de Bits

En el trabajo "Creación y manipulación de k^2-tree para la compresión de representaciones de datos espaciales" se encuentran implementadas algunas funciones que fueron concebidas con el objetivo de facilitar el manejo de los bits.

Las funciones mencionadas utilizan operaciones bit a bit o *bitwise* para operar sobre números binarios a nivel de sus bits individuales.

- **`IS_SET(entero, índice):`** Función que permite determinar si un bit en particular está encendido (1) o apagado (0). Se comprueba el estado del bit en el *bit vector* entero en la posición dada utilizando el operador bit a bit AND (entero & índice).

- **`SET_BIT(entero, índice del bit):`** Función que permite encender un bit individual, cambiar su estado de apagado a encendido. Enciende el bit en el *bit vector* entero utilizando el operador bit a bit OR (entero |= índice) y el operador de asignación.

- **`REMOVE_BIT(entero, índice del bit):`** Función permite apagar un bit individual. Apaga el bit en el *bit vector* entero utilizando el operador bit a bit AND, el operador de asignación y el operador bit a bit NOT (entero &= ~índice).

- **`TOGGLE_BIT(entero, índice del bit):`** Función que permite conmutar el estado de un bit. Si el bit en el *bit vector* entero se encuentra encendido, lo apaga y viceversa, utilizando el operador de asignación y el operador bit a bit XOR (entero ^= índice).

- **`ViewBinario(entero):`** Función que permite ver el estado de cada uno de los 32 bit de un entero. Recibe un valor entero y recorre cada uno de sus 32 bits desplazándose de derecha a izquierda utilizando el operador bit a bit SHIFT LEFT. Mientras se realiza el recorrido va comprobando bit a bit si se encuentran encendidos o apagados e imprime su valor.

- **`Reset(entero):`** Función que permite apagar todos los bit de un entero. Recibe un valor entero y recorre cada uno de sus 32 bits desplazándose de derecha a izquierda utilizando el operador bit a bit SHIFT LEFT. Mientras se realiza el recorrido va apagando cada uno de los bits.