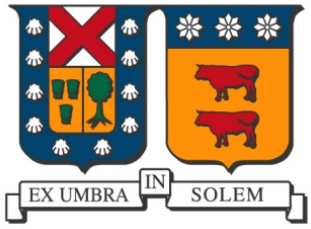
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



“ALGEBRA DE MATRICES UTILIZANDO ESTRUCTURAS COMPACTAS TIPO k^2- TREES”

DIEGO BELTRÁN MADRID

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Diego Gastón Arroyuelo Billiardi
Profesor Correferente: José Luis Martí Lara

Diciembre - 2024

DEDICATORIA

Dedico este trabajo a mi familia, porque sin ellos, no sería quien soy hoy en día.

AGRADECIMIENTOS

Quiero agradecer a mi profesor guía porque, a pesar de haberse retirado de la universidad, siguió ayudandome para concretar este trabajo, además de tenerme paciencia en las reuniones.

También, quiero agradecer a mis amigos y compañeros, que me dieron su apoyo constante en todas las etapas de este trabajo.

Por último, quiero agradecer a toda mi familia, que me ha ayudado mucho durante este tiempo en que he trabajado en este proyecto.

RESUMEN

Resumen— El álgebra de matrices dispersas es un problema muy común en varias áreas de la matemática como lo puede ser la computación científica y la inteligencia artificial. Este trabajo propone utilizar la estructura de datos k^2-tree, el cual representa el árbol en arreglos de bits, como nueva forma de representar matrices dispersas y en conjunto de nuevas operaciones, se propone como primer acercamiento, sumar dos matrices utilizando este nuevo enfoque, dando tiempos de ejecución razonables y dando una conclusión favorable para matrices dispersas.

Palabras Clave— Álgebra de matrices, matrices dispersas, k^2-tree, estructuras de datos.

ABSTRACT

Abstract— Sparse matrix algebra is a very common problem in several areas of mathematics such as scientific computing and artificial intelligence. This work proposes to use the data structure k^2-tree, which represents the tree in arrays of bits, as a new way of representing sparse matrices and in a set of new operations, it is proposed as a first approach, to add two matrices using this new approach, giving reasonable execution times and giving a favorable conclusion for sparse matrices.

Keywords— Matrix algebra, sparse matrices, k^2-tree, data structures.

GLOSARIO

CSR: ***Compressed Sparse Row*** - Método de compresión de matrices por fila
CSC: ***Compressed Sparse Column*** - CSR pero en vez de por fila es por columna
COO: ***Coordinated Format*** - Comprime ambos a la vez
CPU: ***Central Processing Unit*** - Procesador de un computador
GPU: ***Graphical Processing Unit*** - Procesador que se dedica exclusivamente a procesar video y todo lo que tenga que ver con gráfica.
BSR: ***Block Compressed Sparse Row***
METIS: Herramienta que sirve para particionar matrices dispersas.
CSB: ***Compressed Sparse Block***
SpMV: ***Sparse Matrix-Vector Multiplication***
sparse: Una matriz *sparse* es una matriz dispersa, es decir, una matriz donde la mayor parte de sus elementos es 0

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

En este trabajo, se aborda el problema del álgebra de matrices dispersa, el cual corresponde a una de las ramas de la matemática computacional que se ocupa del análisis, diseño y optimización de operaciones algebraicas sobre matrices que contienen predominantemente valores vacíos. Estas matrices se presentan mediante estructuras especializadas, como listas enlazadas, árboles o formatos compactos como CSR. Estas estructuras permiten reducir significativamente el uso de la memoria y acelerar ciertas operaciones, pero también imponen restricciones en el diseño de algoritmos debido a la necesidad de acceder y manipular los datos de forma no convencional, por ende, el principal problema radica en minimizar los costes de acceso y operación sobre estas estructuras.

Este problema impacta principalmente a las siguientes áreas de estudio:

- Modelado científico y simulaciones numéricas: En donde las relaciones entre varias variables de los modelos están localizadas en subregiones del problema modelado.
- Optimización y análisis de datos: En donde las matrices dispersas permiten modelar relaciones esporádicas entre características.
- Procesamiento de grafos: En donde, las matrices dispersas representan relaciones entre nodos, por lo que el enfoque disperso optimiza tanto el almacenamiento como los cálculos
- Computación en alto rendimiento: En donde se busca maximizar el rendimiento utilizando hardware moderno como GPUs y arquitecturas multicore.

Para resolver este problema, se propone utilizar la estructura de datos k^2-tree, propuesta por [Brisaboa *et al.*, 2013], el cual propone una representación de matrices binarias en forma de un árbol específico, donde los elementos del árbol se pueden representar con arreglos de bits. Sin embargo, para dar soporte a este tipo de matrices, se le ha decidido añadir un arreglo de valores numéricos donde se puedan almacenar todos los valores no nulos de la matriz. Para probar que esta estructura puede usarse, se ha diseñado el algoritmo de la suma, donde al tomar dos matrices con estructura k^2-tree, resuelva de manera óptima la suma de esas dos matrices.

Para ello, se ha utilizado un entorno de programación en *Linux* usando el lenguaje *C++* en conjunto con el compilador de g++, dando resultados prometedores en cuanto a tiempos de ejecución, pudiendo corroborar lo óptimo que es la solución propuesta. Por último, se implementó una pequeña herramienta que sirve para sumar cualquier matriz dispersa, con el fin de poder facilitar un poco, el tiempo que se requiere sumar matrices con referente a la suma original.

CAPÍTULO 1
DEFINICIÓN DEL PROBLEMA

En este capítulo, se definirá el problema del álgebra de matrices sobre matrices dispersas, y sus aplicaciones correspondientes como, por ejemplo, todas las aplicaciones que tiene en la computación científica y el machine learning. Esta área de estudio tiene muchos problemas, como los costos computacionales, el almacenamiento en la memoria y uso de la CPU, son principales focos de preocupación debido a la estructuración y manejo de las matrices en un contexto de optimización de la memoria. Para poder solucionar estos problemas, se han diseñado técnicas de almacenamiento y algoritmos especializados que ayudan a resolver este problema, sin embargo, no han sido suficientes con respecto a costos computacionales. Durante este capítulo, se conocerá en detalle el problema del álgebra de matrices dispersas, el proceso que existe actualmente para operar con estas matrices, los problemas que actualmente tiene este proceso, y las soluciones existentes al proceso que tiene actualmente esta área de estudio.

1.1. Contexto

El álgebra de matrices es una rama esencial en el área de las matemáticas en donde se estudian propiedades y operaciones de matrices. Una matriz es un arreglo rectangular de numeros organizados en filas y columnas, en la que es posible aplicar en múltiples ramas de la sociedad, como lo es la economía y la ciencia de datos, o incluso usarlo para buscar datos.

Por otro lado, el álgebra de matrices con respecto a las matrices dispersas se centra en la realización de operaciones algebraicas —como la suma, la multiplicación y la transposición— sobre matrices que contienen un gran número de elementos nulos. Este problema es relevante en campos como la computación científica, el procesamiento de datos y la optimización, donde el almacenamiento y la manipulación de grandes cantidades de datos escasos en memoria se vuelven críticos.

Las matrices dispersas presentan múltiples desafíos específicos, ya que los métodos convencionales de álgebra matricial pueden ser ineficientes tanto en términos de tiempo como de espacio. Las operaciones en matrices densas suelen asumir que la mayoría de los elementos están poblados, lo cual no es el caso en matrices dispersas. Por ello, es necesario emplear estructuras de datos especializadas que sólo almacenen los elementos no nulos y algoritmos optimizados que puedan acceder y operar eficientemente sobre estos elementos.

Las principales dificultades incluyen el diseño de representaciones eficientes para almacenar matrices dispersas, la implementación de algoritmos de acceso que reduzcan el tiempo de cómputo al evitar operaciones innecesarias en elementos nulos, y el desarrollo de métodos que permitan realizar operaciones algebraicas, manteniendo el formato disperso. Así, el ál-

gebra de matrices dispersas busca equilibrar la eficiencia computacional con la reducción del uso de memoria, para hacer posible el manejo de matrices de gran escala en aplicaciones donde la escasez de datos es inherente.

1.2. Situación Actual

Actualmente, el álgebra de matrices dispersas juega un rol crucial en áreas como la inteligencia artificial, el análisis de redes sociales, el modelado de sistemas físicos y la simulación en computación científica. La creciente disponibilidad de datos y el uso extendido de modelos basados en grafos y redes neuronales han impulsado la necesidad de manipular matrices extremadamente grandes y, a menudo, dispersas, donde sólo una fracción pequeña de los elementos es significativa. Este tipo de matrices aparece frecuentemente en aplicaciones donde los datos son inherentemente escasos, como en la representación de relaciones en redes o la factorización de grandes sistemas de ecuaciones en simulaciones científicas.

A pesar de los avances en hardware y procesamiento, trabajar con matrices dispersas sigue siendo un desafío computacional. Para abordar estos problemas, se han desarrollado representaciones especializadas de almacenamiento, como el formato CSR y CSC, que permiten almacenar únicamente los elementos no nulos y sus ubicaciones. Estas estructuras mejoran el acceso y reducen los requerimientos de memoria, aunque la eficiencia varía según la operación y la densidad de la matriz.

En cuanto a las operaciones algorítmicas, se han diseñado métodos avanzados de multiplicación y factorización de matrices dispersas, pero siguen siendo operaciones costosas, especialmente cuando las dimensiones son muy grandes. Además, la investigación se enfoca en optimizar la ejecución en arquitecturas paralelas y distribuidas, como las GPUs y sistemas de cómputo en la nube, que pueden manejar grandes cantidades de datos en paralelo. A pesar de estos avances, el álgebra de matrices dispersas aún enfrenta retos, como la dificultad de balancear la carga en sistemas paralelos debido a la naturaleza irregular de los datos y la complejidad adicional en algoritmos que deben minimizar los accesos a los elementos nulos.

Los desarrollos en álgebra de matrices dispersas han permitido progresos significativos en eficiencia, pero la solución de problemas a gran escala en la práctica sigue requiriendo innovaciones continuas en algoritmos y optimización de hardware, particularmente con la expansión de áreas como el aprendizaje automático, donde las operaciones de matrices dispersas son fundamentales.

1.3. Procesos

1.3.1. Proceso del modelamiento de los datos

En cuanto al proceso de modelado de datos y sus operaciones para el álgebra de matrices dispersas, en la actualidad, implica una serie de pasos orientados a representar, de manera eficiente los datos escasos en una matriz, maximizando la eficiencia en términos de almacenamiento y procesamiento. En este contexto, el objetivo es optimizar el acceso a los elementos no nulos y minimizar el uso de memoria. Para ello, el proceso de uso es el siguiente:

- **Análisis del problema y caracterización de los datos**: El primer paso es analizar el problema y caracterizar los datos para entender el nivel de dispersión y la distribución de los elementos no nulos en la matriz. Este análisis permite seleccionar el modelo de almacenamiento más adecuado. Por ejemplo, si los elementos no nulos están distribuidos en bandas o bloques, un formato que aproveche esta estructura puede ofrecer mejores rendimientos en tiempo y espacio.
- **Selección del Formato de Almacenamiento**: Existen múltiples formatos de almacenamiento especializados en matrices dispersas, y la elección depende del tipo de operación algebraica que se llevará a cabo y de la densidad de los elementos no nulos [Dongarra *et al.*, 2000]. Entre los formatos más comunes se encuentran:
  - **CSR**: Almacena las filas de la matriz como listas compactas, eficientes para operaciones fila a fila, como la multiplicación matriz-vector.
  - **CSC**; Similar a CSR, pero organiza la matriz por columnas, útil en operaciones columna a columna.
  - **Diagonal y Bloques**: Útil para matrices dispersas que presentan estructura en bloques o bandas diagonales, minimizando accesos innecesarios.
  - **COO**; Representa cada elemento no nulo con sus coordenadas, lo cual facilita la construcción inicial de la matriz dispersa y permite manipulaciones dinámicas de los datos.
- **Estructuración y Preprocesamiento de los Datos**: Para construir una matriz dispersa de manera eficiente, es fundamental preprocesar los datos en el formato elegido, indexando cada elemento no nulo según sus coordenadas y generando los vectores de posición y valores correspondientes. Este preprocesamiento implica ordenar los datos, eliminar duplicados o redundancias y, en algunos casos, realizar compresión para reducir el tamaño total en memoria.
- **Optimización del Acceso a Datos y Cache**: El acceso a datos en matrices dispersas se optimiza estructurando el modelo para maximizar la eficiencia de la memoria caché, especialmente en sistemas de hardware paralelo como las GPUs. Esto se logra mediante una ordenación cuidadosa de los elementos y la ubicación contigua de las

posiciones y valores no nulos, minimizando el número de accesos y garantizando que las operaciones de álgebra se ejecuten con rapidez.

- **Diseño de Algoritmos y Operaciones Algebraicas**: El siguiente paso consiste en diseñar o adaptar algoritmos que aprovechen el modelo de datos seleccionado. Para la multiplicación de matrices dispersas, por ejemplo, se pueden utilizar algoritmos que sólo acceden a las posiciones no nulas, eliminando operaciones innecesarias sobre elementos cero. Estos algoritmos deben ser lo suficientemente flexibles para permitir operaciones algebraicas de manera eficiente sin sobrecargar la memoria.
- **Validación y Optimización Iterativa**: Finalmente, se realiza una validación del modelo implementado mediante pruebas de eficiencia en términos de memoria y velocidad de cómputo. A partir de los resultados obtenidos, se pueden hacer optimizaciones adicionales, como ajustar el modelo de almacenamiento o afinar los algoritmos de acceso, para mejorar aún más la eficiencia.

En la actualidad, el proceso recién descrito, puede graficarse de la siguiente manera:

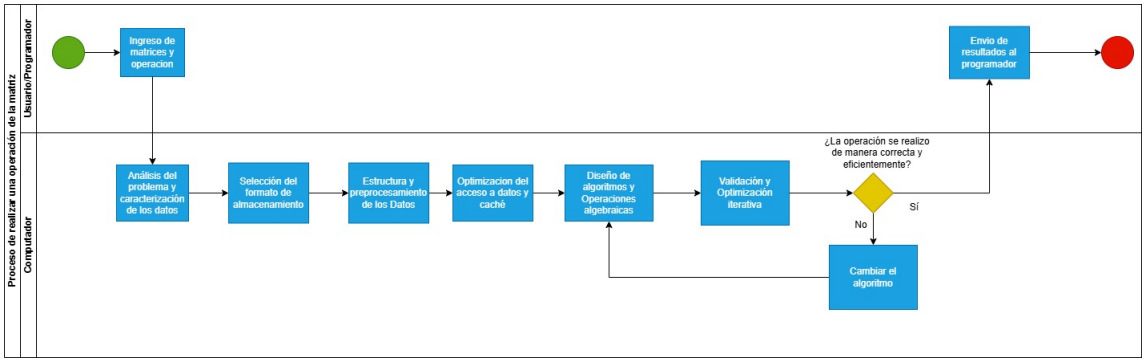


Figura 1: Flujo del proceso de realizar una operacion matricial dentro de un computador
Fuente: Elaboración propia

Cabe destacar que estos problemas reflejan la complejidad inherente al modelado y manejo de matrices dispersas en el álgebra lineal. Cada uno de estos desafíos requiere técnicas especializadas y algoritmos adaptados, y a pesar de los avances en investigación y hardware, el álgebra de matrices dispersas continúa siendo un área de mejora y desarrollo activo en el campo de la computación científica.

1.4. Problemas detectados

[Dongarra, 2020] indica que el modelado y la manipulación de matrices dispersas presentan una serie de problemas técnicos y de optimización que complican tanto el desarrollo de aplicaciones como la eficiencia en su ejecución. Estos problemas pueden agruparse en áreas

clave: la representación de datos, el acceso eficiente a elementos no nulos, el manejo de hardware, y la paralelización y escalabilidad en sistemas de gran escala.

- **Representación y Almacenamiento de Datos**: Un desafío fundamental en el álgebra de matrices dispersas es seleccionar una representación de datos que minimice el uso de memoria sin sacrificar la velocidad de acceso. Las matrices dispersas contienen principalmente elementos nulos, y las estructuras de datos tradicionales no son eficientes, ya que almacenan grandes cantidades de ceros. Elegir el formato de almacenamiento adecuado (CSR, CSC, COO, entre otros) es complicado porque cada formato ofrece ventajas y desventajas específicas para ciertos tipos de operaciones, y un formato ideal para una operación puede ser ineficiente para otra. Además, cuando las matrices presentan estructuras complejas o patrones dispersos irregulares, se hace difícil optimizar el almacenamiento y el acceso sin comprometer la flexibilidad.

- **Acceso Ineficiente a Elementos No Nulos**: El acceso eficiente a los elementos no nulos es esencial en las operaciones algebraicas con matrices dispersas. En muchos casos, los algoritmos tienen que "saltar" entre elementos no nulos de la matriz, lo que implica múltiples accesos indirectos a memoria y un bajo aprovechamiento de la memoria caché. Este acceso irregular ralentiza el procesamiento, especialmente en operaciones como la multiplicación de matrices, donde cada acceso debe realizarse de forma selectiva. Además, este problema se agrava en matrices dispersas de gran tamaño, donde la latencia de memoria puede volverse un cuello de botella.

- **Complejidad en la Optimización para Hardware Moderno**: Los avances en hardware, como las GPUs y los sistemas de cómputo paralelizados, pueden mejorar la eficiencia en las operaciones de matrices dispersas, pero aprovechar estas arquitecturas no es sencillo. Las GPUs, por ejemplo, funcionan mejor con datos dispuestos de manera contigua y predecible, lo cual no es común en las matrices dispersas debido a la distribución irregular de los elementos. Este desajuste entre el diseño de hardware y la estructura de las matrices dispersas introduce complejidad en la programación y exige un ajuste fino en los algoritmos para minimizar la transferencia de datos entre memoria principal y caché o memoria de la GPU.

- **Paralelización y Balanceo de Carga en Sistemas Distribuidos**: En sistemas distribuidos y de alto rendimiento, paralelizar las operaciones en matrices dispersas es otro desafío. Debido a la distribución desigual de los elementos no nulos, lograr un balance de carga adecuado entre los nodos o procesadores es difícil. En muchos casos, algunos nodos pueden quedarse inactivos esperando a que otros completen sus tareas, lo que reduce la eficiencia general del sistema. La naturaleza irregular de las matrices dispersas también dificulta dividir los datos de manera uniforme y hacer que cada procesador o nodo trabaje al máximo de su capacidad.

- **Complejidad en el Desarrollo de Algoritmos Eficientes**: Los algoritmos para operaciones básicas (como la suma, transposición, o multiplicación de matrices dispersas) requieren diseño especializado. La mayoría de los algoritmos tradicionales de álgebra

lineal no pueden aplicarse directamente a matrices dispersas sin incurrir en una gran cantidad de operaciones redundantes en elementos nulos. Esto exige desarrollar algoritmos adaptados para matrices dispersas, lo cual complica el desarrollo y, en muchos casos, requiere conocimientos avanzados tanto en matemáticas como en diseño de algoritmos y arquitectura de sistemas.

- **Mantenimiento de la integridad**: Cuando las matrices dispersas alcanzan un tamaño extremadamente grande (por ejemplo, en aplicaciones de inteligencia artificial o simulaciones físicas), mantener la escalabilidad del sistema se convierte en un problema serio. A medida que aumenta el tamaño de la matriz, también lo hace la necesidad de memoria y el tiempo de procesamiento, y los enfoques tradicionales de almacenamiento y computación pueden volverse insuficientes. Las técnicas de compresión y los modelos de almacenamiento especializado pueden aliviar parte del problema, pero aún existen límites de hardware que hacen que el manejo de grandes escalas siga siendo un reto.
- **Dificultad en la Validación y Optimización Iterativa**: Optimizar y validar modelos de matrices dispersas es una tarea iterativa y lenta. La eficiencia de un modelo depende de múltiples factores (estructura de la matriz, tipo de operación, arquitectura de hardware), y pequeñas modificaciones pueden causar grandes variaciones en el rendimiento. Probar exhaustivamente las combinaciones de estructuras de datos, algoritmos y configuraciones de hardware es un proceso costoso y consume recursos, lo que dificulta alcanzar una solución óptima para todas las aplicaciones.

Todos estos problemas actuales reflejan la complejidad inherente al modelado y manejo de matrices dispersas en el álgebra lineal. Cada uno de estos desafíos requiere técnicas especializadas y algoritmos adaptados, y a pesar de los avances en investigación y hardware, el álgebra de matrices dispersas continúa siendo un área de mejora y desarrollo activo en el campo de la computación científica.

1.5. Soluciones actuales

Las soluciones actuales a los problemas detectados en el álgebra de matrices dispersas se han centrado en el desarrollo de formatos de almacenamiento eficientes, la optimización del acceso a memoria, y el aprovechamiento de arquitecturas modernas para paralelizar las operaciones. Estas soluciones han permitido avances significativos, aunque cada una tiene limitaciones específicas según el tipo de aplicación y el entorno de hardware.

- **Representación y Almacenamiento Eficiente de Datos**: Para abordar el problema de almacenamiento en matrices dispersas, se han desarrollado varios formatos especializados. Entre los más utilizados están el CSR y el CSC, que sólo almacenan los valores no nulos y sus ubicaciones, reduciendo considerablemente el uso de memoria. También

está el COO, que almacena coordenadas individuales, es preferido cuando se necesita modificar la estructura de la matriz dinámicamente. Estos permiten una mayor eficiencia de almacenamiento, aunque cada uno es más adecuado para ciertos tipos de operaciones.

En aplicaciones específicas, como las que requieren acceso por bloques o patrones específicos (por ejemplo, matrices de banda o diagonales), se han implementado formatos especializados como el Diagonal o BSR. Estos formatos el acceso para matrices con patrones particulares, permitiendo que los algoritmos realicen operaciones más rápidamente al aprovechar la estructura de los datos.

- **Optimización del Acceso a Memoria y Uso de Caché**: El acceso eficiente a los elementos no nulos sigue siendo un desafío, especialmente en arquitecturas que dependen de la memoria caché. Para solucionar esto, se han implementado técnicas de reordenación y prefetching de datos, que mejoran la localidad de referencia y reducen los tiempos de espera en el acceso a memoria. Por ejemplo, algunos ingresan los elementos no nulos de tal forma que se minimizan los accesos aleatorios, lo cual es especialmente útil en sistemas de alto rendimiento donde la latencia de memoria puede limitar el procesamiento.

- **Optimización para Hardware**: Las GPUs y otras arquitecturas de cómputo paralelo representan una oportunidad importante para mejorar la eficiencia de las operaciones con matrices dispersas, aunque requieren estrategias especializadas para manejar el acceso irregular a memoria. Se han desarrollado bibliotecas como cuSPARSE para NVIDIA GPUs, que incluye optimizaciones específicas para el acceso disperso en paralelo, y SuiteSparse, que se adapta a diferentes arquitecturas para maximizar la eficiencia. Estas bibliotecas implementan algoritmos e que optimizan el uso de memoria y el procesamiento de elementos no nulos en bloques, permitiendo que las GPUs procesen grandes matrices dispersas de manera más eficiente.

- **Balanceo de Carga en Sistemas Distribuidos**: El problema de balanceo de carga en sistemas paralelos se ha abordado con algoritmos de particionado y redistribución de trabajo que dividen la matriz en submatrices de tamaño equilibrado según la densidad de los elementos no nulos. Herramientas como METIS y ParMETIS se utilizan para particionar matrices dispersas de forma que el trabajo se distribuya equitativamente entre los nodos de un clúster o una red de computadoras. Esto reduce el tiempo de inactividad y mejora la eficiencia general del sistema al asegurar que cada procesador trabaje con una carga equivalente.

- **Algoritmos Eficientes para Operaciones Básicas**: Se han adaptado y optimizado los algoritmos para operaciones comunes como la multiplicación de matrices dispersas, la factorización y la resolución de sistemas lineales dispersos. Métodos como el SpMV, propuesto por [Buluç y Gilbert, 2018], han sido optimizados mediante técnicas que eliminan operaciones redundantes sobre elementos nulos, y que también adaptan los cálculos para arquitecturas de paralelización masiva. Estas mejoras han resultado en

algoritmos más rápidos que puedan resolver problemas de gran escala en menor tiempo, aunque siguen dependiendo del patrón de dispersión y la densidad de los datos.

- **Soluciones para la Escalabilidad en Gran Escala**: Para resolver los problemas de escalabilidad en aplicaciones que utilizan matrices dispersas de gran tamaño, se han propuesto técnicas de compresión que reducen aún más la necesidad de memoria, y que permiten que los datos se procesen en paralelo. Algoritmos como el CSB agrupan bloques de elementos no nulos, facilitando el manejo de matrices grandes en sistemas de hardware limitados. Además, plataformas de cómputo en la nube permiten distribuir el alma y procesamiento de matrices dispersas en varios nodos, maximizando la capacidad de manejo de datos.

Todas estas soluciones a este proceso contemplan nuevas técnicas de optimización para estructuras que ya se conocen como la CSR o la CSC, sin embargo, no se han propuesto nuevas estructuras que permitirían aún más diseñar y optimizar las operaciones algebraicas básicas sobre estas matrices, lo cual, es fin de este trabajo.

1.6. Objetivos

Se presentan los siguientes objetivos para el desarrollo del proyecto:

1.6.1. Objetivo General

- Implementar un algebra de matrices poco densas sobre

  0representaciones comprimidas de las mismas

1.6.2. Objetivos Específicos

- Implementar las operaciones de matrices utilizando estructuras compactas tipo k^2-trees
- Mostrar empíricamente mediante experimentos las ventajas y desventajas al utilizar una estructura del enfoque implementado
- Implementar una herramienta con la que un usuario pueda interactuar y realizar álgebra de matrices.

CAPÍTULO 2
MARCO CONCEPTUAL

En este capítulo, se discutirá lo básico correspondiente al álgebra de matrices y la estructura de datos llamada k^2-*tree*, introducida por [Brisaboa *et al.*, 2013] y permite la compresión de matrices utilizando una estructura de tipo árbol el cual se almacena eficientemente en la memoria.

2.1. Breve introducción al algebra de matrices

Una matriz A es un arreglo numérico rectangular de m filas y n columnas, denotado como $A \in \mathbb{R}^{m \times n}$, en el cual, para poder acceder a un elemento dentro de esta matriz, se debe especificar la fila y la columna indicada. Por ejemplo considere la siguiente matriz $A \in \mathbb{R}^{2 \times 3}$ definida como:

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 1 & 8 \end{pmatrix}$$

Para acceder a un elemento de esa matriz se debe de especificar cual fila y columna uno desea buscar, en este caso, si se selecciona la fila 2 con la columna 3 el valor que devuelve la matriz seria 8.

De esta definición se pueden desprender los siguientes casos:

- Escalar: Un escalar correspondería a una matriz de 1 fila y 1 columna. Usualmente, son valores numéricos individuales en los que son extraídos del dominio de los números reales [Aggarwal, 2020] y tienen muchas aplicaciones en varios campos de la ciencia. Por ejemplo, la edad de un grupo de personas puede representarse como escalares.
- Vectores: Un vector corresponde a una matriz de n filas y 1 columna, en el caso de vectores columnas, o uno de 1 fila y n columnas. Usualmente, a los vectores se le conocen como un arreglo numérico de n datos en las que, a cada dato se le puede referir como entradas, componentes, o dimensiones del vector [Aggarwal, 2020]. A los vectores también se les puede llamar coordenadas de n dimensiones desde un punto de vista geométrico.

Dentro del álgebra de matrices podemos encontrar las siguientes operaciones:

- Suma y Resta: Dada dos matrices del mismo orden $A, B \in \mathbb{R}^{m \times n}$, la suma (o resta) de estas matrices corresponde a realizar la suma (o resta) de componente por componente, como se indica a continuación:

$$A \pm B = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} \pm \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{pmatrix} = \begin{pmatrix} a_{11} \pm b_{11} & \dots & a_{1n} \pm b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} \pm b_{m1} & \dots & a_{mn} \pm b_{mn} \end{pmatrix}$$

- Transpuesta: Dada una matriz de orden $A \in \mathbb{R}^{m \times n}$, transponer una matriz implica crear otra de orden $A^T \in \mathcal{M}^{n \times m}$ intercambiando filas y columnas, como se indica a continuación:

$$A^T = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}^T = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{pmatrix}$$

- Producto: Dentro de esta operacion se distinguen tres posibles casos:

  - Matriz-Escalar: La multiplicación de una matriz $A \in \mathbb{R}^{m \times n}$ con un escalar $k \in \mathbb{R}$ viene dado por:

$$kA = k \cdot \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} = \begin{pmatrix} k \cdot a_{11} & k \cdot a_{12} & \dots & k \cdot a_{1n} \\ k \cdot a_{21} & k \cdot a_{22} & \dots & k \cdot a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ k \cdot a_{m1} & k \cdot a_{m2} & \dots & k \cdot a_{mn} \end{pmatrix}$$

  - Matriz-vector: La multiplicación de una matriz $A \in \mathbb{R}^{m \times n}$ y un vector $\vec{x} \in \mathbb{R}^n$ resulta en un vector de dimensión $\vec{y} \in \mathbb{R}^m$ de la siguiente manera:

$$A \cdot \vec{x} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} a_{11} \cdot x_1 + \ldots + a_{1n} \cdot x_n \\ a_{21} \cdot x_1 + \ldots + a_{2n} \cdot x_n \\ \vdots \\ a_{m1} \cdot x_1 + \ldots + a_{mn} \cdot x_n \end{pmatrix}$$

    En donde muestra que cada componente se puede calcular de la siguiente manera:

$$y_i = \sum_{j=1}^{n} a_{ij} \cdot x_j, \forall i \in [1, m]$$

  - Matriz-Matriz: La multiplicación de una matriz $A \in \mathbb{R}^{m \times n}$ con otra matriz $B \in \mathbb{R}^{n \times p}$ correspondería hacer p productos matriz-vector y resulta en una matriz $C \in \mathbb{R}^{m \times p}$, el cual se puede desarrollar de la siguiente manera:

$$A \cdot B = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \dots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{np} \end{pmatrix} = \begin{pmatrix} c_{11} & \dots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{m1} & \dots & c_{mp} \end{pmatrix}$$

Donde cada componente c_{ij} de esa matriz se puede calcular de la siguiente manera:

$$c_{ij} = \sum_{j=1}^{n} a_{ij} \cdot b_{ji}$$

Existen diversos tipos de matrices utilizadas en matemáticas y otras disciplinas. Cada tipo de matriz tiene características y propiedades únicas que las distinguen entre sí. Una de ellas, corresponde a las matrices cuadradas, las que corresponden a matrices con igual número de filas y columnas. A continuación, se presentarán algunos tipos matrices cuadradas:

- Matriz “cero”: Es una matriz donde todos los elementos de esta tienen un valor 0.

$$A = \begin{pmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}$$

- Matriz “uno”: Es una matriz donde todos los elementos de esta tienen un valor 1.

$$A = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{pmatrix}$$

- Matriz diagonal: Es una matriz donde solo existen elementos en la diagonal de la matriz. Por ejemplo:

$$A = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{pmatrix}$$

  Donde $a_{ii} \neq 0, \forall i \in [1, n]$

- Matriz triangular superior: Es una matriz donde todos los elementos de la diagonal y por encima de esta son distintos a cero.

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{pmatrix}$$

- Matriz triangular inferior: Es una matriz donde todos los elementos de la diagonal y por debajo de esta son distintos a cero.

$$A = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

- Matriz simétrica: Una matriz es considerada simétrica cuando la matriz transpuesta es igual a la matriz original, es decir:

$$A^T = A$$

- Matriz antisimétrica: Una matriz es considerada antisimétrica cuando la matriz transpuesta es igual a matriz inversa aditiva de la original, es decir:

$$A^T = -A$$

- Matriz identidad: La matriz identidad corresponde a una matriz diagonal donde cada $a_{ii} = 1, \forall i \in [1, n]$. Usualmente se denota por I_n donde n corresponde al número de filas y columnas que este posee.

$$I_n = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

Teniendo en cuenta estas definiciones, es posible definir las siguientes operaciones.

- Inversa: Una matriz $A \in \mathbb{R}^{n \times n}$ tiene una inversa cuando el producto de la inversa con la original da como resultado a la matriz identidad:

$$AA^{-1} = A^{-1}A = I_n$$

  Teniendo en cuenta esta definición se puede definir la siguiente matriz:

  - Matriz ortogonal: También conocida como matriz unitaria, es una matriz donde la transpuesta de la matriz es igual a la inversa.

$$A^T = A^{-1}$$

- Determinante: La determinante de una matriz corresponde a una función que a cada matriz $A \in \mathbb{R}^{n \times n}$ se le asocia un número real denotado como $det(A)$ o $|A|$. Este numero se puede calcular de manera recursiva tomando en cuenta que:

  - $n = 1 \Rightarrow det(A = \big(a\big)) = a$
  - $n = 2 \Rightarrow det\left(A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\right) = ad - bc$
  - $n > 2 \Rightarrow det(A) = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} det(A_{[i][j]})$

  Donde esta función cumple con las siguientes propiedades:

  - $det(A^{-1}) = \dfrac{1}{det(A)}$
  - $det(A \cdot B) = det(A) \cdot det(B), B \in \mathbb{R}^{n \times n}$
  - $det(\alpha \cdot A) = \alpha^n \cdot det(A), \alpha \in \mathbb{R}$
  - $det(A^T) = det(A)$

- Traza: La traza de la matriz corresponde a una función que calcula la suma de todos los elementos de la diagonal, denotado por $tr(A)$, se puede expresar de la siguiente forma:

$$tr(A) = \sum_{i=1}^{n} a_{ii}$$

  Donde esta función cumple con las siguientes propiedades:

  - $tr(A + B) = tr(A) + tr(B)$
  - $tr(\alpha \cdot A) = \alpha \cdot tr(A)$
  - $tr(A^T) = tr(A)$

2.2. Algoritmos de multiplicación de matrices

Un primer acercamiento a la multiplicación de matrices correspondería a el producto matriz-matriz discutido anteriormente en este capítulo, el problema es que, para matrices grandes, independientemente si son dispersas o no, el costo computacional se eleva demasiado acorde al número de elementos que se tiene, es por esto que, para reducir ese coste, se han propuesto muchos algoritmos durante las últimas décadas que optimizan esta operación, haciendo posible reducir su costo operacional de manera asintótica. Los algoritmos que se revisarán son uno basado en el enfoque de dividir y conquistar, el de Strassen, y uno específico para matrices dispersas, propuesto por [Schoor, 1982].

2.2.1. Dividir y conquistar

El algoritmo de dividir y conquistar consiste en realizar una división por bloques a cada matriz involucrada en la operacion como tal. Dada dos matrices A y B, cada matriz se dividirá en 4 submatrices de tamaño $N/2 \times N/2$ cada una quedando como:

$$A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Donde cada C_{ij} corresponde a una submatriz el cual se define como:

- $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$
- $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$
- $C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$
- $C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$

Al aplicar este algoritmo para multiplicar las matrices, se obtiene que el tiempo de ejecución no supera el método convencional de la multiplicación, correspondiente a $O(n^3)$, para ello, la siguiente sección presentará uno mejorado a partir de este.

2.2.2. Strassen

Al igual que el algoritmo *Z-order*, el algoritmo de Strassen realiza la misma subdivisión mediante dividir y conquistar en las matrices involucradas en la operación. Dada dos matrices A y B, cada matriz se dividirá en 4 submatrices de tamaño $N/2 \times N/2$ cada una quedando como:

$$A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Sin embargo, Strassen define las siguientes 7 submatrices.

- $P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
- $P_2 = (A_{11} + A_{22})B_{11}$
- $P_3 = A_{11}(B_{12} - B_{22})$
- $P_4 = A_{22}(B_{21} - B_{11})$
- $P_5 = (A_{11} + A_{22})B_{22}$
- $P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
- $P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

En los cuales, cada C_{ij} los calcula de la siguiente manera:

- $C_{11} = P_1 + P_4 - P_5 + P_7$
- $C_{12} = P_3 + P_5$
- $C_{21} = P_2 + P_4$
- $C_{22} = P_1 - P_2 + P_3 + P_6$

La gracia de este algoritmo en comparación con *Z-order* es que se reduce el número de operaciones para encontrar el producto deseado en una operación, haciendo que el tiempo de ejecución disminuya de manera considerable, teniendo este tiempo de ejecución en el orden de $O(N^{\log_2 7})$

2.2.3. *Fast algorithm for sparse matrix multiplication*

En 1982, [Schoor, 1982] presento un algoritmo que permitía multiplicar dos matrices *sparse* en un tiempo promedio mejor que el convencional. En este caso, Amir Schoor ocupo una lista ortogonal para representar las matrices *sparse*.

El algoritmo asume que, al multiplicar ambas matrices, todos los elementos no nulos de una matriz se multiplican con todos los elementos no nulos de la i-ésima fila de la segunda matriz, y solo por esa fila. El algoritmo itera sobre todas las filas de la primera matriz, haciendo que,

por cada fila, el algoritmo busca el elemento no cero correspondiente a la misma columna donde se encuentre almacenada el elemento de esa fila. Luego de realizado esta búsqueda, el algoritmo almacena eficientemente el resultado de la multiplicación sumada mediante una variable auxiliar para su almacenamiento.

Gracias a esta observación, el tiempo promedio de la operación llega a ser más eficiente de lo que el algoritmo convencional implica, llegando a un orden de $O(D_1 D_2 MNK)$ cuando la convencional es de $O((D_1 + D_2)MNK)$, asumiendo que $A = M \times N$ y que $B = N \times K$, además de que D_1 y D_2 es son las densidades de las matrices A y B respectivamente.

2.3. Árboles k^2-arios

A continuación, se definirá una estructura de datos presentado por [Brisaboa *et al.*, 2013], en el que permite realizar representaciones de las matrices utilizando un árbol binario que es capaz de comprimirlas.

2.3.1. Definición conceptual

Un árbol k^2-ario es una estructura de dato de tipo árbol en el que es posible representar matrices de $\mathbb{R}^{n \times n}$ utilizando un árbol de altura $h = \lceil \log_k n \rceil$. En cada nodo de este árbol se le asigna un valor 1 si existen datos en las hojas y 0 cuando no existen datos se comportan como hojas, a excepción del último nivel, donde cada hoja marcará un 1 si existe un dato en esa celda de la matriz, o un 0 en caso contrario.

Para armar el árbol, se utiliza una estrategia de subdivisión de matrices que sigue la misma estrategia que *MX Quadtree* [Samet, 2006], el cual subdivide la matriz en k^2 partes iguales, tratando de buscar en cada submatriz si existen datos dentro o no, en donde, por cada submatriz a la que se encontraron datos, se vuelve a realizar la división de manera recursiva, hasta que ya no se pueda seguir subdividiendo la matriz, es decir, cuando la dimensión de cada submatriz es 1.

Por ejemplo, supongamos que se tiene la siguiente matriz $A \in \mathbb{R}^{4 \times 4}$.

$$A = \begin{pmatrix} 1 & 3 & 2 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 2 & 1 \end{pmatrix}$$

Tomando $k = 2$, se realiza la siguiente subdivisión:

$$A = \left(\begin{array}{cc|cc} 1 & 3 & 2 & 2 \\ 0 & 1 & 1 & 1 \\ \hline 0 & 0 & 1 & 3 \\ 0 & 0 & 2 & 1 \end{array}\right)$$

Quedando con las siguientes 4 submatrices:

$$A_1 = \begin{pmatrix} 1 & 3 \\ 0 & 1 \end{pmatrix},\ A_2 = \begin{pmatrix} 2 & 2 \\ 1 & 1 \end{pmatrix},\ A_3 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix},\ A_4 = \begin{pmatrix} 1 & 3 \\ 2 & 1 \end{pmatrix}$$

En donde solo las matrices A_1, A_2 y A_4 presentan datos no nulos, mientras que la matriz A_3 es una matriz sin datos dentro, por lo que ya no se sigue dividiendo.

Realizando recursivamente la división por cada submatriz con valores no nulos tenemos que:

$$A_1 = \left(\begin{array}{c|c} 1 & 3 \\ \hline 0 & 1 \end{array}\right),\ A_2 = \left(\begin{array}{c|c} 2 & 2 \\ \hline 1 & 1 \end{array}\right),\ A_4 = \left(\begin{array}{c|c} 1 & 3 \\ \hline 2 & 1 \end{array}\right)$$

Quedando las siguientes submatrices:

$$A_{11} = (1),\ A_{12} = (3),\ A_{13} = (0),\ A_{14} = (1)$$

$$A_{21} = (2),\ A_{22} = (2),\ A_{23} = (1),\ A_{24} = (1)$$

$$A_{31} = (1),\ A_{32} = (3),\ A_{33} = (2),\ A_{34} = (1)$$

Donde se obtiene un total de 12 escalares que corresponderían a las hojas del árbol, dado que las submatrices no se pueden seguir subdividiendo. A continuación, el árbol resultante del ejemplo se puede ver en la siguiente figura:

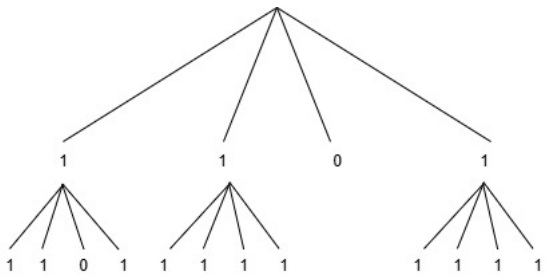


Figura 2: Representación del árbol k^2-*tree* del ejemplo
Fuente: Elaboración propia

2.3.2. Representación

[Brisaboa *et al.*, 2013] plantea que el árbol completo puede representarse usando 2 arreglos de bits, de la siguiente manera.

- T (*Tree*): Almacena todos los bits del árbol k^2-ario a excepción de los encontrados en la profundidad h, es decir, las hojas. La forma de almacenarlos es ir recorriendo los bits por niveles del árbol, es decir, se almacenan todos los bits que están en el nivel 1, luego los del nivel 2, y así hasta llegar al nivel $h-1$. Tomando de ejemplo el árbol anterior, la representación en bits sería: $T = \{1, 1, 0, 1\}$
- L (*Leaves*): Arreglo en el que entrega el último nivel del árbol, representado con un arreglo de bits, este indica en que posición de la matriz hay una conexión entre la fila y la columna de la matriz. En el caso de la matriz anterior quedaría como: $L = \{1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$

Esta representación permite una fácil navegación dentro del árbol, puesto que, dado una posición i en el árbol T, tal que $T[i] = 1$, entonces el hijo de ese nodo corresponde a $rank(T, i) \cdot k^2 + j$, donde $rank(T, i)$ corresponde al número de 1s que están en el árbol $T[0, i]$ y j corresponde al j-ésimo hijo a buscar. Cabe mencionar que, para encontrar al j-ésimo hijo de manera eficiente, la búsqueda debe soportar la operación de $rank(T, i)$, el cual se puede ejecutar en tiempo constante utilizando subespacios vectoriales sobre la secuencia de bits.

2.3.3. Operaciones

Para poder manejar el árbol k^2-ario de manera eficiente, [Brisaboa *et al.*, 2013] propuso unas primeras operaciones básicas sobre este, las cuales se mencionarán brevemente a continuación:

- ***CheckLink***: Comprueba si hay alguna conexión entre el nodo p y el q del árbol T, utilizando propiedades como la recursión y la función $rank(T, i)$.
- **Sucesores**: Retorna los hijos del nodo p del árbol T, utilizando la recursión como estructura algorítmica principal, y la función $rank(T, i)$.
- **Predecesores**: Retorna los padres del nodo q del árbol T, y, al igual que las operaciones anteriores mencionadas, también utiliza la recursión como estructura algorítmica principal, y la función $rank(T, i)$.
- **Rango**: Permite encontrar tanto predecesores como sucesores dentro de un rango de nodos dentro del árbol. Se comporta igual que las operaciones anteriores solo que en este caso no se visitan todos los nodos.

- ***LinkInRange***: Esta es una generalización de la última operación donde se comprueba si existe alguna conexión entre un nodo p, o un rango de nodos $[p_1, p_2]$, con otro rango de nodos $[q_1, q_2]$

Sin embargo, en el año 2015, [Brisaboa *et al.*, 2015] publica más operaciones que el árbol k^2-ario puede soportar. Varias de estas operaciones involucran a las relaciones que existen entre bits dentro del árbol, y además, operaciones involucradas al conjunto completo del árbol cuando el árbol solo tiene nodos con valor 1. En este caso, definieron los *bitmaps* como la representación de las relaciones de entrada del árbol k^2-ario solo considerando que las hojas tienen solo entradas binarias 0 o 1. Las operaciones definidas en [Brisaboa *et al.*, 2015] se describen brevemente a continuación:

- **Unión**: El algoritmo une dos *bitmaps* utilizando las relaciones binarias en un entorno parecido al BFS. En donde se van almacenando tuplas dentro de una cola los *bitmaps* más importantes mientras se procesa el algoritmo.
- **Intersección**: A diferencia de la unión, la intersección recorre el *bitmap* de una forma parecida a DFS, donde se procesan los subárboles dentro del algoritmo.
- **Diferencia**: En esta operación, busca la diferencia de dos relaciones binarias utilizando una estructura algorítmica similar a la intersección.
- **Complemento**: Al igual que la intersección y la diferencia, esta operación realiza un DFS sobre el árbol intercambiando valores de los nodos dependiendo del caso.
- **Copia**: Permite copiar un subárbol desde la posición actual hasta el último nivel del árbol, esto es útil para la operación de diferencia, ya que cubre uno de los casos de la funcionalidad del algoritmo
- ***SkipNodes***: Es una función propuesta para realizar la operación de intersección cuando ocurre un caso en particular dentro del análisis. El algoritmo permite ignorar el nodo y pasar al siguiente nodo.

Todas estas operaciones se implementan sobre un árbol T y tienen una versión donde el árbol solo tiene valores 1 en sus nodos.

CAPÍTULO 3
PROPUESTA DE SOLUCIÓN

Para mejorar estas operaciones matriciales dispersas, el enfoque propuesto es comprimir la matriz utilizando una estructura de datos basados en árboles k^2-trees, donde su representación permite una compresión óptima en la memoria debido a como estaba pensado. Según [Brisaboa *et al.*, 2013], esta estructura se enfoca en representar compactamente grafos que representan relaciones entre páginas web. Para que este enfoque pueda ser útil en operaciones de matrices como la suma, se propone realizar ciertos cambios en cómo se estructura este árbol, por ejemplo, un arreglo ordenado de cierta forma que pueda representar los valores no nulos de la matriz dispersa. Por otro lado, para demostrar que esta solución puede ser válida para realizar operaciones sobre esta estructura, se realizaron cambios en la representación y en la entrada de los datos. Además, se propone un algoritmo que resuelve la suma de matrices.

3.1. Estructura

Para poder realizar correctamente las operaciones básicas del álgebra de matrices utilizando la estructura de k^2-trees, definiremos nuevas operaciones sobre la estructura y además de formalizar operaciones ya existentes para la correcta funcionalidad de nuestra solución. Por último, se analizará que ocurre cuando la matriz no tiene el tamaño mínimo para estructurarlo como árbol k^2.

3.1.1. Curva Z-order

La curva Z-order es un método de asignación y ordenamiento de datos multidimensionales a solo una dimensión mientras se preservan la localidad de los puntos de los datos. El valor z de un punto en muchas dimensiones se calcula simplemente intercalando las representaciones binarias de sus valores de coordenadas. A continuación, se presenta una descripción gráfica de cómo funciona esta curva.

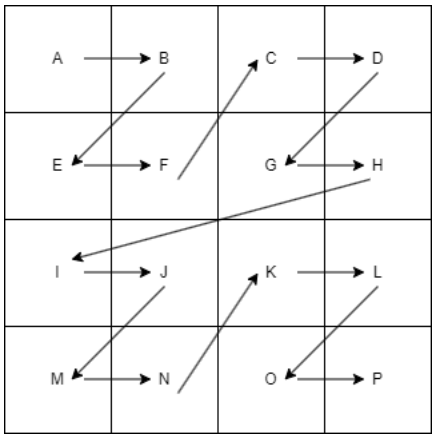


Figura 3: Recorrido de una matriz utilizando la secuencia *Z-order*
Fuente: Elaboración propia

Como se puede apreciar, la curva va ordenando en forma de Zig-Zag, coincidiendo con las posiciones de las hojas cuando se construye un k^2-tree.

3.1.2. Representación

Para que una matriz no binaria sea representada correctamente dentro de un árbol k^2-ario, se definirá un nuevo arreglo de valores $V \in \mathbb{R}^n$, donde solo se almacenan todos los elementos distintos a cero que contenga la matriz, y a su vez, todos estos valores deben de estar ordenados con la curva Z-order definida en la sección anterior. Por ejemplo, utilizando la matriz de ejemplo del capítulo anterior:

$$A = \begin{pmatrix} 1 & 3 & 2 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 2 & 1 \end{pmatrix}$$

Su nuevo arreglo V seria:

$$V = \{1, 3, 1, 2, 2, 1, 1, 1, 3, 2, 1\}$$

Debido a esta representación, podemos almacenar los valores de la matriz de manera eficiente, lo que reduce significativamente el uso de memoria. Al omitir los ceros, mantenemos solo las operaciones relevantes, enfocándonos en los valores que realmente pueden cambiar. Sin embargo, esta nueva representación plantea un desafío adicional: ¿cómo podemos relacionar el arreglo de bits L con el nuevo arreglo de valores V?

Para entender esta relación, se analizará el quinto elemento del arreglo L, donde se sabe que $L[5] = 1$. Esto indica que hay un valor distinto de cero en esa posición. Al calcular $rank(L, 5) = 4$ obtenemos que el cuarto elemento en V es el valor buscado.

Para verificar este resultado, si seguimos la curva de orden-Z en la matriz, observamos que el cuarto elemento no nulo se encuentra en el segundo cuadrante de la matriz. En términos del árbol k^2-tree este valor correspondería al segundo hijo del nodo raíz y sería la primera hoja de dicho hijo, ocupando la quinta posición en el arreglo de hojas.

Formalizando, sea $w \in \mathbb{R}$ un valor de la matriz donde $w \in V$. Para encontrar la posición i donde se relaciona el arreglo de bits L con el arreglo de valores V, se calcula $i = rank(L, j)$ donde j es la posición deseada en el arreglo de bits. Así, se cumple que $V[i] = w$.

3.1.3. Entrada de los datos

Para optimizar la creación de los árboles k^2-arios, podemos trabajar las matrices solo almacenando todos los valores $w \neq 0$. Dado un vector $\vec{v} = (p, q, w)$, se define el arreglo de datos Q como:

$$Q = \{(p_i, q_i, w_i), \forall i \in [1, n^2], n > 1\}$$

Donde sabemos que se cumple que $A[p_i, q_i] = w_i, \forall w_i \neq 0, i \in [1, n^2]$, p corresponde a la fila de la matriz A y q su columna. De esta forma, se asume que los valores nulos si son considerados, pero no de manera explícita.

3.1.4. Hijo de un nodo de un árbol k^2-ario

[Brisaboa *et al.*, 2013] plantea una acercamiento inicial a como obtener el hijo de un nodo de un arbol T. Sin embargo, los autores no consideraban que, al tratarse de arreglos finitos de bits, estos pueden representar otro valor dependiendo de donde se esté ubicado en este arreglo, por lo que, definiremos la función $child(T, x, i)$ como:

$$child(T, x, i) = \begin{cases} rank(T, x) \cdot k^2 + i & si \quad x \in [0, size(T)[\\ x & si \quad x \text{ es hoja} \\ -1 & eotc \end{cases}$$

De esta forma, nos aseguramos de eliminar todas las ambigüedades que se podrían obtener al momento de recorrer un árbol cuando deseemos realizar cualquier operacion, haciendo que retorne siempre un hijo valido para no salir de los límites del árbol.

3.1.5. Matrices fuera del rango esperado

Para matrices cuya dimensión sea distinta a $2^k, \forall k > 1$, la construcción de un árbol k^2-tree es posible siempre y cuando sea posible ampliar la matriz con la cantidad necesaria de filas y columnas para alcanzar el tamaño deseado dentro de esta matriz. Por ejemplo, sea:

$$A = \begin{pmatrix} 1 & 3 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

La construcción del árbol correspondería a agregar una nueva fila y una nueva columna de tal forma que su tamaño este en el rango esperado quedando como:

$$A_{ampliada} = \begin{pmatrix} 1 & 3 & 2 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

De esta manera, es posible representar cualquier matriz a un árbol k^2.

Por otra parte, el añadir filas y columnas con elementos nulos a una matriz no altera la suma total de los elementos de la matriz original. Esto es porque, al agregar filas y columnas formadas exclusivamente por ceros (elementos nulos), no se está añadiendo ningún valor a la suma de los elementos ya existentes en la matriz. Los ceros no contribuyen a la suma total, por lo que el resultado de sumar todos los elementos de la matriz ampliada sigue siendo el mismo que el de la matriz original. En términos matemáticos, la suma de la matriz original es invariable ante la inclusión de filas o columnas que no añaden valores distintos de cero.

A modo de ejemplo, consideremos las matrices A y B definidas como:

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 4 & 2 & 6 \\ 7 & 3 & 9 \end{pmatrix}, B = \begin{pmatrix} 1 & 8 & 2 \\ 3 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

La suma de ambas viene dada por:

$$A + B = \begin{pmatrix} 1 & 2 & 1 \\ 4 & 2 & 6 \\ 7 & 3 & 9 \end{pmatrix} + \begin{pmatrix} 1 & 8 & 2 \\ 3 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 10 & 3 \\ 7 & 7 & 10 \\ 10 & 5 & 10 \end{pmatrix}$$

Donde es posible ver que existen 9 elementos que son distintos a cero. Ahora, consideremos las matrices A' y B' como ampliaciones de las matrices A y B respectivamente, donde:

$$A' = \begin{pmatrix} 1 & 2 & 1 & 0 \\ 4 & 2 & 6 & 0 \\ 7 & 3 & 9 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, B' = \begin{pmatrix} 1 & 8 & 2 & 0 \\ 3 & 5 & 4 & 0 \\ 3 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

La suma de ambas matrices está dada por:

$$A' + B' = \begin{pmatrix} 1 & 2 & 1 & 0 \\ 4 & 2 & 6 & 0 \\ 7 & 3 & 9 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 8 & 2 & 0 \\ 3 & 5 & 4 & 0 \\ 3 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 10 & 3 & 0 \\ 7 & 7 & 10 & 0 \\ 10 & 5 & 10 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

En este caso, además de mantener el mismo número de elementos no nulos que en la operacion original, los elementos resultantes a la suma también son exactamente los mismos que en la matriz original, manteniendo tanto valor, como posición en fila y en columna.

3.2. Suma

3.2.1. Procedimiento

El enfoque propuesto para sumar dos matrices usando la estructura k^2-tree es un enfoque basado en las submatrices, las cuales se obtienen cuando se construye un k^2-tree sobre esa matriz, utilizando $k = 2$. Sean A y B matrices en el espacio $\mathcal{M}(\mathbb{R})_{n\times n}$, definimos:

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}, B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

Donde A_i y $B_i, \forall i \in \{1, 2, 3, 4\}$ son matrices de tamaño $n/2 \times n/2$ cada una. En el contexto de árboles k^2-tree, cada submatriz propuesta corresponde a un hijo de un árbol. Sea K una matriz en $\mathcal{M}(\mathbb{R})_{n\times n}$ con K_i submatrices $\forall i \in \{1, 2, 3, 4\}$:

- K_1 corresponde al primer hijo de la matriz-árbol K
- K_2 corresponde al segundo hijo de la matriz-árbol K
- K_3 corresponde al tercer hijo de la matriz-árbol K

- K_4 corresponde al cuarto hijo de la matriz-árbol K

Por ende, para poder sumar dos matrices, definimos lo siguiente:

$$C = A + B = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} + \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$$

Por lo que se pude asumir que:

- $C_1 = A_1 + B_1$
- $C_2 = A_2 + B_2$
- $C_3 = A_3 + B_3$
- $C_4 = A_4 + B_4$

Donde A_i, B_i, C_i son submatrices o hijos de los árboles A, B y C correspondientes. Para construir un algoritmo apropiado que represente correctamente las ecuaciones descritas, consideraremos un enfoque recursivo para la suma, por lo que definiremos la función $suma(T_1, T_2)$ como:

Algorithm 1 Suma de k^2-trees

procedure $suma(Ret, T_1, T_2, L_1, L_2, V_1, V_2, x_1, x_2, i, j)$
  if x_1 es raiz & x_2 es raiz **then**
    $i \leftarrow leftshift(i, 1)$
    $j \leftarrow leftshift(j, 1)$
    $suma(Ret, T_1, T_2, L_1, L_2, V_1, V_2, child(T_1, x_1, 1), child(T_2, x_2, 1), i|0, j|0)$
    $suma(Ret, T_1, T_2, L_1, L_2, V_1, V_2, child(T_1, x_1, 2), child(T_2, x_2, 2), i|0, j|1)$
    $suma(Ret, T_1, T_2, L_1, L_2, V_1, V_2, child(T_1, x_1, 3), child(T_2, x_2, 3), i|1, j|0)$
    $suma(Ret, T_1, T_2, L_1, L_2, V_1, V_2, child(T_1, x_1, 4), child(T_2, x_2, 4), i|1, j|1)$
  else if x_1 es hoja & x_2 es hoja **then**
    $BothLeaves(Ret, T_1, T_2, L_1, L_2, V_1, V_2, x_1, x_2, i, j)$
    return
  else if Solo x_1 es hoja **then**
    $OneLeaf(Ret, T_1, L_1, V_1, x_1, i, j)$
    return
  else if Solo x_2 es hoja **then**
    $OneLeaf(Ret, T_2, L_2, V_2, x_2, i, j)$
    return
  else
    $InNodes(Ret, T_1, T_2, L_1, L_2, V_1, V_2, x_1, x_2, i, j)$
  end if
end procedure

Figura 4: Algoritmo que suma 2 matrices usando la estructura k^2-tree
Fuente: Elaboración propia

Donde T_k, L_k, V_k y x_k son elementos del árbol k^2-tree representando a la matriz k, mientras que los parámetros i y j servirán para ir generando la posición de la suma durante el recorrido de ambos árboles. Las funciones $BothLeaves$, $OneLeaf$, e $InNodes$ son funciones que ayudan el manejo de la función principal a su mejor entendimiento. Estos algoritmos se pueden definir como:

Algorithm 2 $BothLeaves$

procedure Funcion $BothLeaves(Ret, T_1, T_2, L_1, L_2, V_1, V_2, x_1, x_2, i, j)$
$h_1 \leftarrow x_1 - size(T_1)$ ▷ Hoja actual arbol 1
$h_2 \leftarrow x_2 - size(T_2)$ ▷ Hoja actual arbol 2
if $L_1[h_1] == 1$ & $L_2[h_2] == 1$ **then**
$v_1 \leftarrow rank(L_1, h_1)$
$v_2 \leftarrow rank(L_2, h_2)$
$s = V_1[v_1] + V_2[v_2]$
$t \leftarrow (i, j, s)$
Guardar t en Ret
else if Solo $L_1[h_1] == 1$ **then**
$v_1 \leftarrow rank(L_1, h_1)$
$s = V_1[v_1]$ ▷ Implica que $V_2[v_2] = 0$
$t \leftarrow (i, j, s)$
Guardar t en Ret
else if Solo $L_2[h_2] == 1$ **then**
$v_2 \leftarrow rank(L_2, h_2)$
$s = V_2[v_2]$ ▷ Implica que $V_1[v_1] = 0$
$t \leftarrow (i, j, s)$
Guardar t en Ret
end if
return
end procedure

Figura 5: Función que gestiona lo que ocurre en las hojas durante la suma
Fuente: Elaboración propia

Algorithm 3 $OneLeaf$

procedure $OneLeaf(Ret, T, L, V, x, i, j)$
  $h \leftarrow x - size(T)$
  if $L[h] == 1$ **then**
    $v \leftarrow rank(L, h)$
    $s = V[v]$
    $t \leftarrow (i, j, s)$
    Guardar t en Ret
    return
  end if
  return
end procedure

Figura 6: Función que gestiona lo que ocurre en solo 1 hoja durante la suma
Fuente: Elaboración propia

Algorithm 4 $InNodes$

procedure $InNodes(Ret, T_1, T_2, L_1, L_2, V_1, V_2, x_1, x_2, i, j)$
  $i = leftshift(i, 1)$
  $j = leftshift(j, 1)$
  if No hay hijos disponibles **then**
    return
  end if
  if Existe un valor valido de $child(T_1, x_1, 1)$ o $child(T_2, x_2, 1)$ **then**
    $suma(Ret, T_1, T_2, L_1, L_2, V_1, V_2, child(T_1, x_1, 1), child(T_2, x_2, 1), i|0, j|0)$
  end if
  if Existe un valor valido de $child(T_1, x_1, 2)$ o $child(T_2, x_2, 2)$ **then**
    $suma(Ret, T_1, T_2, L_1, L_2, V_1, V_2, child(T_1, x_1, 2), child(T_2, x_2, 2), i|0, j|1)$
  end if
  if Existe un valor valido de $child(T_1, x_1, 3)$ o $child(T_2, x_2, 3)$ **then**
    $suma(Ret, T_1, T_2, L_1, L_2, V_1, V_2, child(T_1, x_1, 3), child(T_2, x_2, 3), i|1, j|0)$
  end if
  if Existe un valor valido de $child(T_1, x_1, 4)$ o $child(T_2, x_2, 4)$ **then**
    $suma(Ret, T_1, T_2, L_1, L_2, V_1, V_2, child(T_1, x_1, 4), child(T_2, x_2, 4), i|1, j|1)$
  end if
end procedure

Figura 7: Función que gestiona lo que ocurre en los nodos que no son hojas
Fuente: Elaboración propia

3.2.2. Análisis del tiempo de ejecución

Para poder determinar la complejidad temporal de la solución propuesta, se debe de tomar en cuenta que esta solución está basada en la estructura k^2-tree y su recorrido, por ende, cuando se hacen 4 llamadas a la recursión usando la función $InNodes$, el proceso se detiene en los siguientes casos:

- Cuando ambos nodos son hojas.
- Cuando x_1 o x_2 no pueda determinar elementos no nulos en sus hijos.

Por ende, el tiempo de ejecución de la suma está relacionado con:

- **Altura del árbol**: En este caso, como se mencionó anteriormente, la profundidad del árbol es $\log_k N$
- **Número de nodos visitados**: Para esta función, al llegar al nivel d, hay hasta k^{2d} nodos. En caso de que el árbol sea completo, esto implica que el número total de nodos no nulos del árbol es N

Por lo tanto, en el peor caso, cada nodo puede generar hasta 4 llamadas recursivas, lo que implica que el tiempo de ejecución asintótico es de $O(N)$, y a su vez, el tiempo de ejecución asintótico para las operaciones internas, como lo es el acceso a hijos, cálculo de índices, etc., es de tiempo constante, es decir, $O(1)$.

Comparando esta solución con la convencional, teniendo en cuenta que su tiempo de ejecución es $O(n \times m)$ independiente de los casos, la solución propuesta solo va recorriendo los nodos relevantes, lo que reduce N significativamente al nivel de la compresión debido a la estructura que utiliza, haciendo que en el mejor caso, cuando se tratan de matrices dispersas, este algoritmo puede acercarse a $O(\log_k(n \times m))$. Si se trabajan con matrices cuadradas, es decir, cuando $m = n$, este tiempo de ejecución se reduce a $O(\log_k n)$.

CAPÍTULO 4
VALIDACIÓN DE LA SOLUCIÓN

Para probar que la solución propuesta es factible, el proyecto se concretó en una serie de especificaciones de tanto compilador, como librerías utilizadas, sistema operativo, y lenguaje de programación utilizado, luego, se presentarán los resultados tanto comparativos como por tamaño de la matriz. Por último, se presentará un programa básico que pueda sumar matrices utilizando el algoritmo implementado.

4.1. Entorno de pruebas

Para el desarrollo de este proyecto, se utilizó el lenguaje C++, un lenguaje compilado que permite manejar objetos como clases similares a Java o Python, y que, a su vez, permite realizar múltiples operaciones matemáticas en distintas estructuras de datos.

Para este lenguaje, se utilizó la librería llamada *sdsl* [Simongog, 2015], en donde se encuentran implementados múltiples de las estructuras necesarias, tanto como los *bit vectors* como la estructura principal de este proyecto.

Gracias a esta librería, se debe de ejecutar en un entorno Unix para su correcto funcionamiento, en este caso se utilizó una *Virtual Machine* que simula el sistema operativo *Ubuntu Linux 22.04*, el cual permite que el programa no acceda a secciones no correspondidas de la memoria. Debido a esto, se utilizó el compilador incorporado dentro de este sistema operativo conocido como g++ el cual permite tener varias opciones de compilación.

4.2. Resultados comparativos

Para poder comparar la operacion propuesta con la convencional, se han generado 100 matrices de tamaño $2{,}000 \times 2{,}000$, donde el $0,01$ % de los elementos son elementos distintos a cero, luego, se seleccionaron aleatoriamente 2 matrices por operacion entre los cuales suman un total de 50 operaciones en total. El siguiente gráfico muestra como fue el desempeño de ambas operaciones.

Tabla 1: Primeras 25 sumas comparativas
Fuente: Elaboración propia.

Matriz 1	Matriz 2	Tiempo de ejecución suma k^2-tree	Tiempo de ejecución suma convencional
n75	n06	$0{,}001891$	$0{,}008247$
n56	n37	$0{,}001997$	$0{,}007009$
n21	n96	$0{,}001838$	$0{,}006950$
n43	n48	$0{,}001938$	$0{,}007013$
n82	n25	$0{,}001798$	$0{,}006863$
n87	n94	$0{,}002087$	$0{,}007806$
n61	n49	$0{,}001990$	$0{,}007113$
n52	n38	$0{,}002043$	$0{,}007468$
n91	n36	$0{,}002036$	$0{,}007680$
n32	n98	$0{,}001899$	$0{,}008164$
n23	n35	$0{,}001760$	$0{,}008234$
n46	n88	$0{,}001993$	$0{,}007545$
n92	n20	$0{,}001974$	$0{,}007449$
n97	n28	$0{,}002013$	$0{,}007767$
n95	n50	$0{,}001918$	$0{,}007845$
n66	n03	$0{,}001970$	$0{,}006711$
n69	n05	$0{,}001844$	$0{,}006879$
n89	n85	$0{,}001903$	$0{,}007721$
n19	n17	$0{,}001890$	$0{,}006740$
n51	n70	$0{,}001737$	$0{,}008129$
n01	n81	$0{,}002044$	$0{,}008365$
n07	n13	$0{,}001924$	$0{,}007758$
n09	n39	$0{,}001989$	$0{,}007833$
n58	n31	$0{,}002088$	$0{,}006705$

Tabla 2: Últimas 25 sumas comparativas
Fuente: Elaboración propia.

Matriz 1	Matriz 2	Tiempo de ejecución suma k^2-tree	Tiempo de ejecución suma convencional
n29	n90	$0{,}001951$	$0{,}006713$
n79	n11	$0{,}001790$	$0{,}006983$
n44	n57	$0{,}001933$	$0{,}007496$
n80	n45	$0{,}002004$	$0{,}007367$
n76	n67	$0{,}001955$	$0{,}007546$
n24	n12	$0{,}001775$	$0{,}007479$
n42	n53	$0{,}001943$	$0{,}006853$
n33	n08	$0{,}001985$	$0{,}007815$
n86	n59	$0{,}002012$	$0{,}007508$
n65	n60	$0{,}001874$	$0{,}007104$
n16	n99	$0{,}001801$	$0{,}008327$
n73	n40	$0{,}001942$	$0{,}007210$
n34	n26	$0{,}001788$	$0{,}006897$
n93	n62	$0{,}001790$	$0{,}007154$
n30	n04	$0{,}001709$	$0{,}007697$
n83	n00	$0{,}001866$	$0{,}007281$
n63	n64	$0{,}001780$	$0{,}006779$
n77	n72	$0{,}001842$	$0{,}006902$
n10	n71	$0{,}001812$	$0{,}007834$
n78	n02	$0{,}001682$	$0{,}007620$
n74	n55	$0{,}001855$	$0{,}007201$
n15	n27	$0{,}001881$	$0{,}007850$
n14	n54	$0{,}002034$	$0{,}007075$
n41	n22	$0{,}001877$	$0{,}008015$
n18	n47	$0{,}001703$	$0{,}006776$
n84	n68	$0{,}001733$	$0{,}007696$

Si comparamos ambos datos de las tablas 1 y 2, podemos ver que los tiempos de ejecución de la suma k^2-tree son menores que la convencional cuando se tratan de matrices dispersas. El siguiente gráfico resume cómo se comportan ambas sumas con todas las características anteriores.

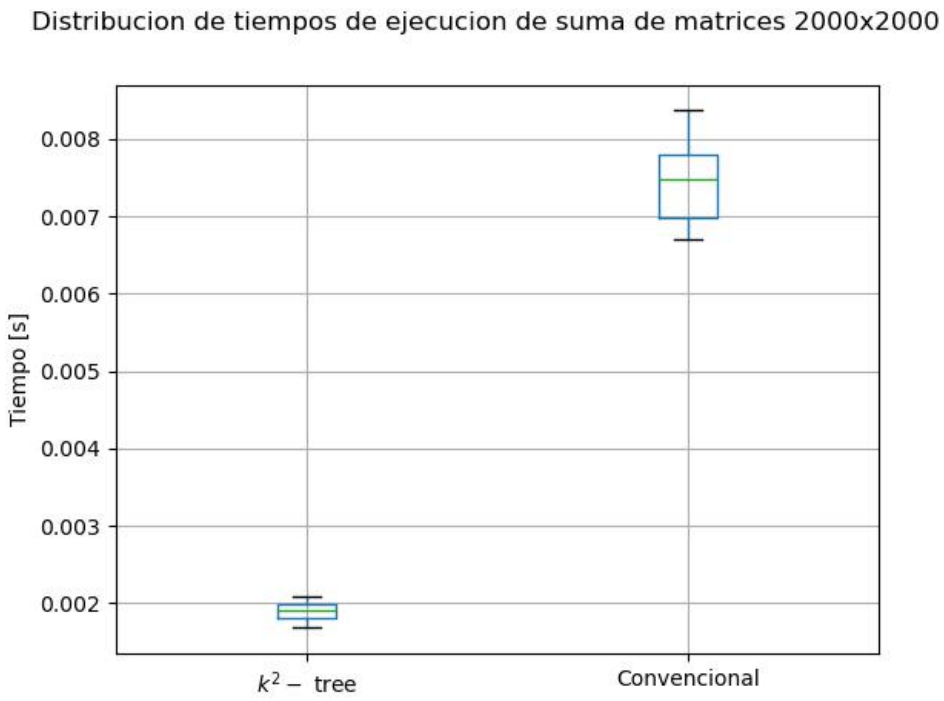


Figura 8: Distribución de tiempos de ejecución de las 50 sumas
Fuente: Elaboración propia

En el cual, se puede comprobar que, en todos los casos, es posible determinar con certeza que la suma con la estructura k^2-tree tiene un promedio de $0,001$ segundos mientras que la convencional tiene un promedio de $0,007$ segundos haciendo que la suma con la estructura planteada es 7 segundos más rápida.

Analizando el gráfico, podemos ver claramente su desempeño destaca cuando las matrices son dispersas, su tiempo de ejecución se concentra alrededor de su promedio $0,001$ segundos, lo que refleja su capacidad de manejar grandes matrices dispersas de manera eficiente, y la baja dispersión en los tiempos de ejecución sugiere que la solución propuesta es consistente y estable, independientemente de la distribución de los elementos no nulos de la matriz. Por otro lado, la solución actual tiene tiempos de ejecución considerablemente mas altos, los cuales varían entre los $0,007$ segundos y los $0,008$ segundos.

Por otra parte, la mayor dispersión de los tiempos de ejecución podría deberse a factores como la ineficiencia en la gestión de elementos vacíos y el costo adicional de recorrer toda la matriz, independiente de su densidad, por ende, este comportamiento refleja que el método no está optimizado para estructuras dispersas, ya que realiza cálculos innecesarios sobre las posiciones vacías.

4.3. Resultados del rendimiento

Para evaluar el rendimiento de la solución propuesta, se realizaron aproximadamente 1050 sumas aleatorias entre 2101 matrices dispersas provenientes del *Wikidata Graph Pattern Benchmark* (WGPB) introducido por [Hogan *et al.*, 2019]. Este conjunto corresponde a un subgrafo de *Wikidata* compuesto por $81{,}426{,}573$ tripletas RDF (sujeto, predicado, objeto) que involucran 2101 predicados distintos. Las tripletas se almacenaron como relaciones binarias según sus predicados: para cada tripleta (s, p, o), el par (s, o) se guardó en la relación binaria asociada al predicado p, añadiendo además un valor que convierte la matriz en una no binaria. Los primeros 100 resultados obtenidos se incluyen en el anexo 5.2.5. Como métrica principal de comparación, se utilizó el número total de elementos no nulos en ambas matrices.

Para el análisis, se ha hecho el siguiente grafico mostrando la distribución, en orden de magnitud, el número de elementos no nulos de las matrices.

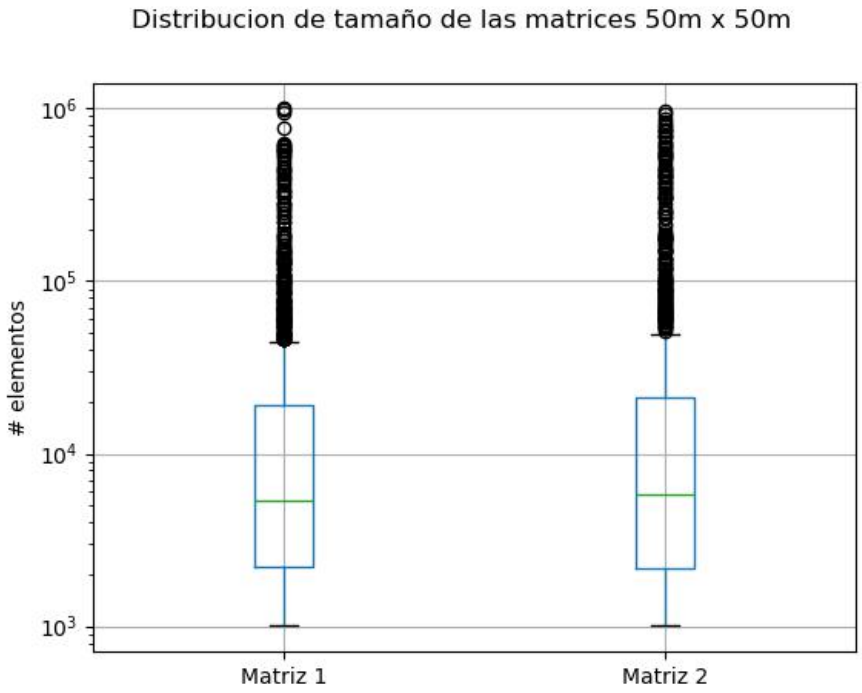


Figura 9: Orden de magnitud de la cantidad de elementos no nulos de las matrices de estudio
Fuente: Elaboración propia

Por ende, podemos comprobar que existen matrices cuyo número de elementos salen de la media esperada, por lo que, durante esta sección, se analizará lo que ocurre con las matrices que están dentro del rango esperado de dispersión, y a su vez, matrices que estén fuera de este rango de dispersión, a las que llamaremos como poco dispersas.

4.3.1. Matrices dispersas

Para matrices grandes y dispersas, se realizó un análisis con referente al tiempo de ejecución y el número de elementos dentro del rango de los 1006 y $260{,}000$ elementos aproximadamente. El resultado del rendimiento general de la solución propuesta es el siguiente:

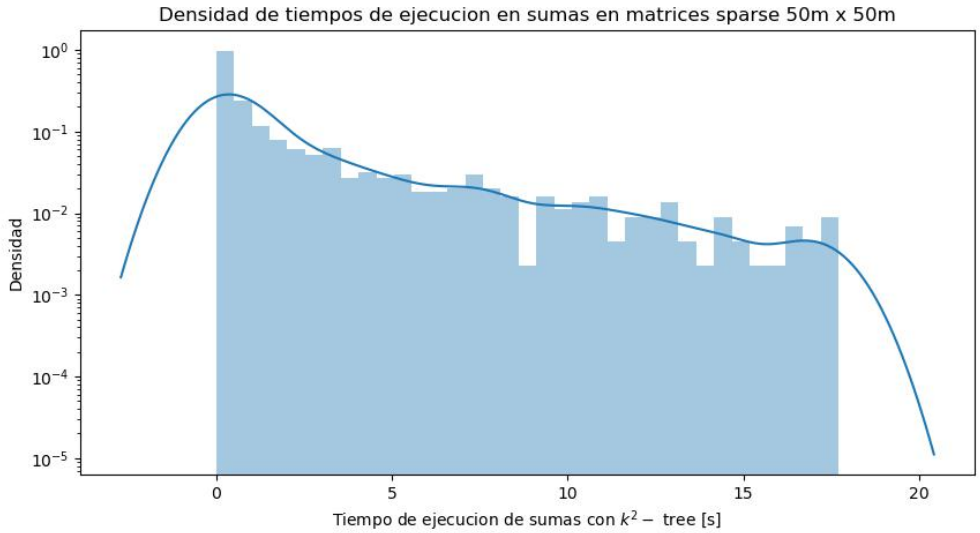


Figura 10: Densidad de tiempos de ejecución resultantes para matrices dispersas.
Fuente: Elaboración propia

El cual, relacionándolos con los tamaños correspondientes podemos observar el siguiente gráfico:

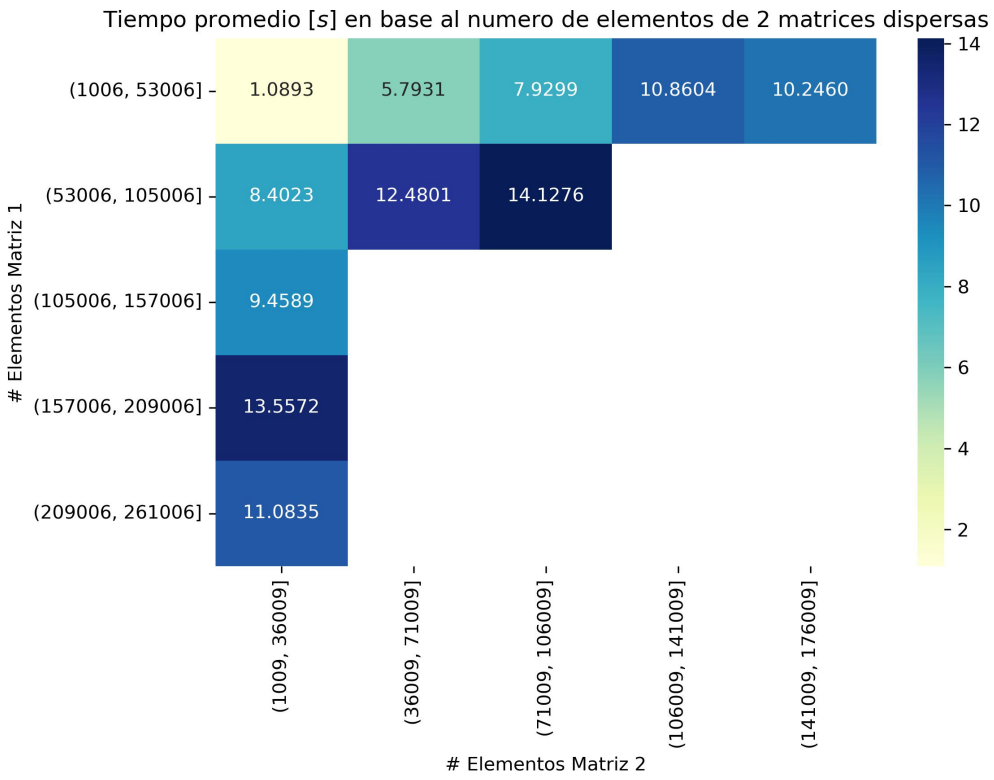


Figura 11: Tiempos de ejecución promedio de la suma k^2-tree
Fuente: Elaboración propia

Esto nos indica que, para matrices muy grandes, la solución propuesta nos entrega tiempos de ejecución decentes, con un promedio general de $3,49$ segundos. Del gráfico, se puede apreciar que existe una leve diferencia de los tiempos promedios de ejecución en base al número de ambos elementos, haciendo que mientras más elementos no nulos existan dentro de las matrices, el tiempo de ejecución aumente en unos solos segundos.

Esto puede deberse a que el número de accesos que las 2 matrices deben de someterse influyan en los tiempos de ejecución, sin embargo, la diferencia no parece ser muy grande, ya que el promedio no supera los 14 segundos cuando existen más de $110{,}000$ elementos no nulos.

Sin embargo, también existen algunos casos en los cuales, el número de elementos no nulos de solo una matriz no afecta al rendimiento de la solución propuesta, como por ejemplo es el caso del intervalo $(209{,}006, 261{,}006)$ para la primera matriz y en $(1009, 36{,}009)$ para la segunda matriz.

Por otra parte, es posible indicar que, si bien el tiempo aumenta con el tamaño de ambas matrices, este incremento es controlado, por lo que sugiere que la solución propuesta está optimizada para trabajar con estas matrices dispersas, mostrando que, existe un rango mas

uniforme en sus tiempos.

4.3.2. Matrices poco dispersas

Para matrices que son poco dispersas, se realizó un análisis aparte de lo general, haciendo énfasis cuando el número de elementos es mayor a los $750{,}000$ elementos no nulos de la matriz. El grafico es el siguiente:

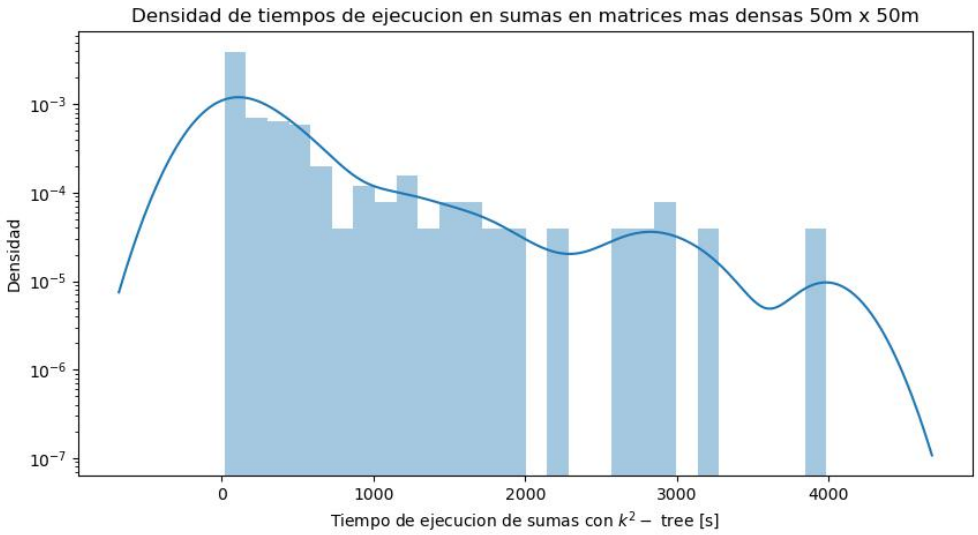


Figura 12: Densidad de tiempos de ejecución resultantes para matrices poco dispersas. Fuente: Elaboración propia

El cual, relacionándolo con las matrices correspondientes, obtenemos los siguientes resultados.

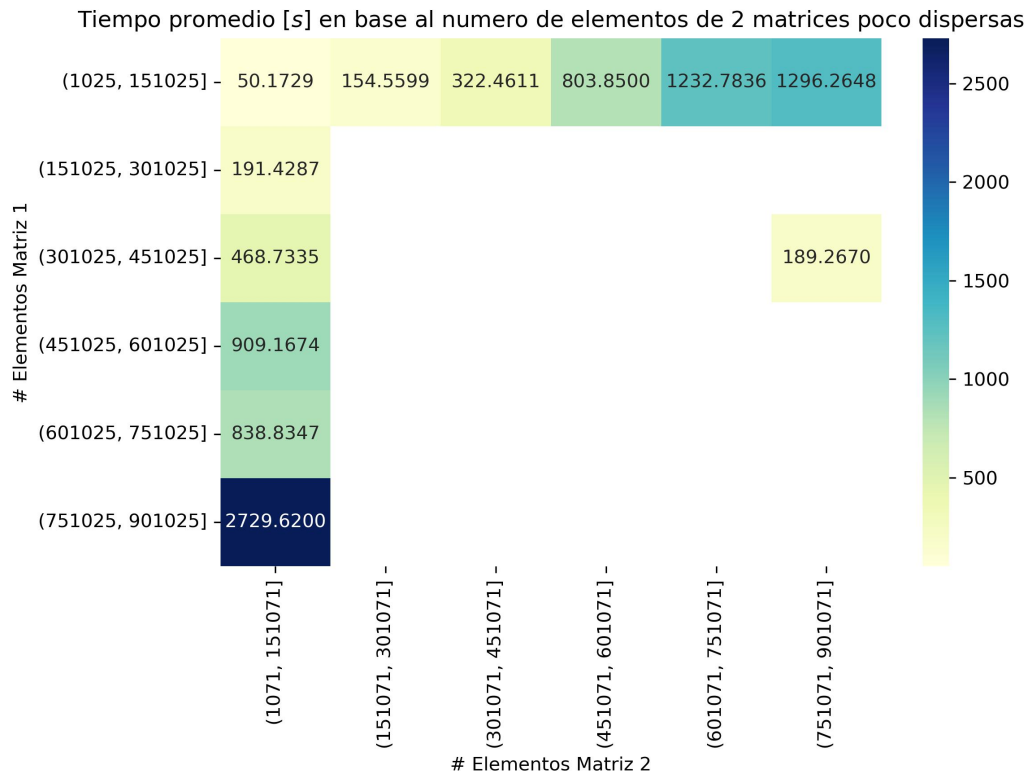


Figura 13: Tiempos de ejecución promedio de la suma k^2-tree para matrices poco dispersas
Fuente: Elaboración propia

A diferencia de los resultados anteriores, podemos apreciar de mejor manera que el rendimiento de la solución propuesta ya no entrega buenos tiempos de ejecución, debido que el promedio general es de 384 segundos, por lo que el rendimiento disminuyó bastante.

Se puede afirmar, es que en este caso, es aún más evidente que el número de elementos no nulos de las 2 matrices tienen una influencia sobre el rendimiento de la solución propuesta, especialmente cuando ambas matrices tienen un gran numero de elementos no nulos.

Aunque, comparándolo con el caso anterior, también existen casos donde el comportamiento es inusual, como lo es el caso de las matrices dentro del intervalo $(151{,}025, 301{,}025)$ para la primera matriz y $(451{,}071, 601{,}071)$ donde se puede apreciar que el tiempo medio de ejecución es de 189 segundos, el cual es notablemente bajo comparándolo con otras combinaciones similares.

4.4. Programa para sumar

Como elemento final, se ha desarrollado un programa básico en C++ que permite realizar esta operación de forma expedita para matrices dispersas. Este programa se ha diseñado para que cualquier usuario pueda sumar las matrices que necesiten. El cual se encuentra disponible en el siguiente enlace.

4.4.1. Entrada de datos

Para que el usuario pueda sumar 2 matrices usando este programa, debe de tener dos archivos con las matrices, llamados 'matriz1.txt' y 'matriz2.txt'. Ambos archivos deben de tener todas las entradas no nulas de la matriz usando la estructura CSR, como se muestra en la siguiente figura:

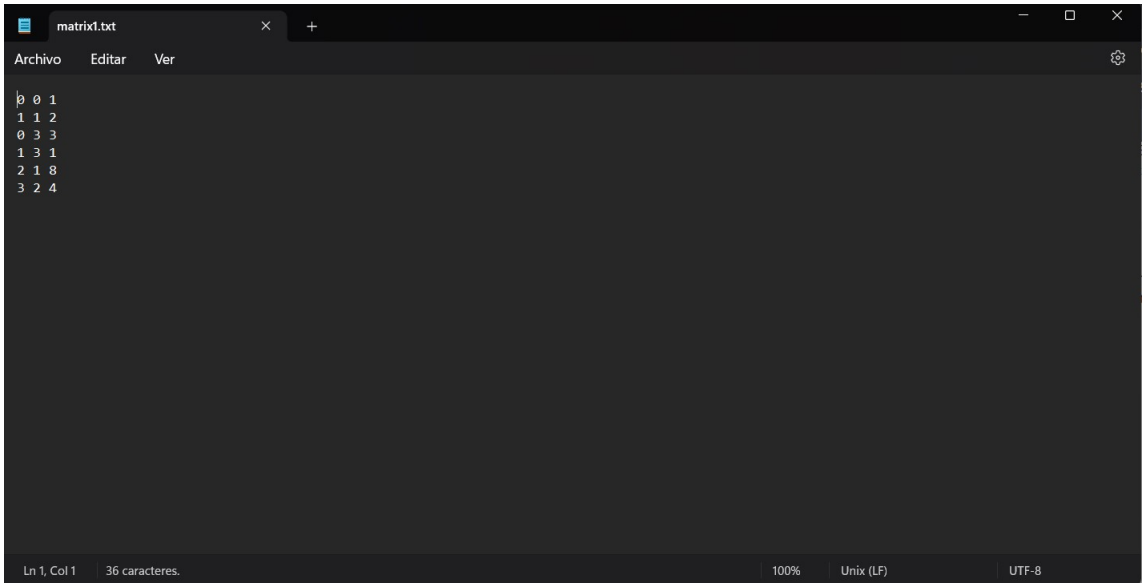


Figura 14: Matriz de ejemplo para el programa
Fuente: Elaboración propia

Cabe destacar que, además de que la carga de datos es automática, los archivos no necesariamente pueden ser iguales en cuanto al tamaño de elementos no nulos, ya que la construcción del árbol no discrimina el número de elementos del archivo.

4.4.2. Ejecución del programa

Para ejecutar este programa, se debe de compilar y ejecutar mediante el comando `make`, de esta manera, el usuario solo debe de correr el comando `make run` para que el programa corra. Una vez hecho esto el usuario verá lo siguiente.

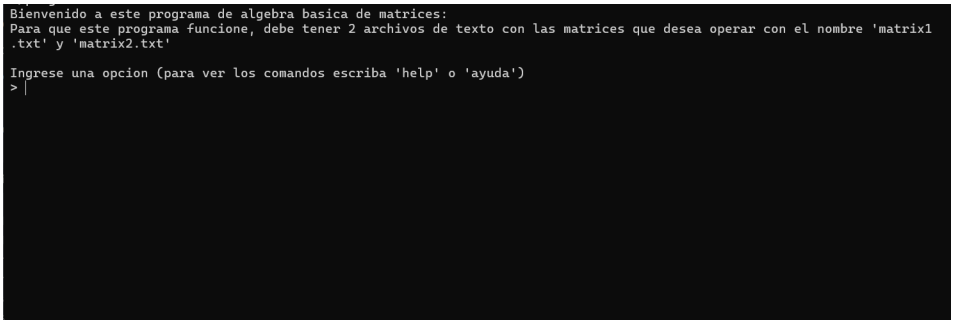


Figura 15: Programa básico para sumar matrices
Fuente: Elaboración Propia

Una vez ejecutado, el usuario dispone de las siguientes opciones:

- **Sumar**: El programa suma las 2 matrices usando la solución propuesta, donde, además de indicar el resultado, también se indica el tiempo de ejecución que le tomó llegar a ese resultado.
- **Ayuda**: Dentro del propio programa existe un pequeño manual de instrucciones para el usuario.
- **Salir**: Este comando te permite salir del programa.

4.4.3. Salida de los datos

Cuando el programa suma las dos matrices usando la solución propuesta, estos resultados se guardan en formato CSR para una mejor comprensión de la propia matriz. Por ende, para tener un cierto orden de entrega de los datos, en primer lugar, se guarda el tiempo de ejecución del algoritmo, y seguido de cada tupla obtenida en la suma, guardando todo en un archivo llamado 'out.txt', como se muestra a continuación:

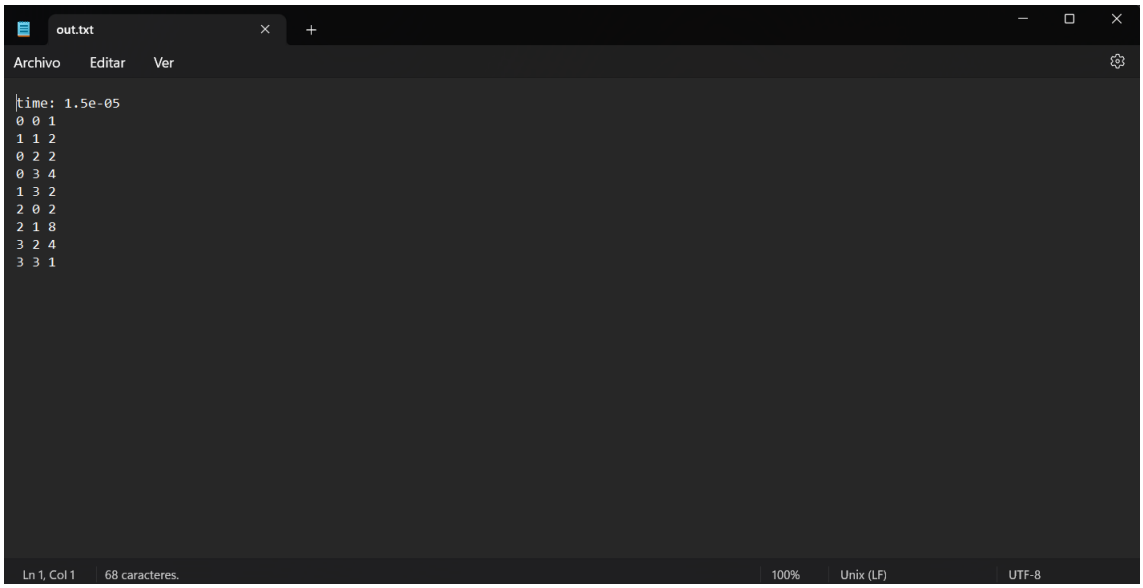


Figura 16: Salida de la suma realizada por el programa
Fuente: Elaboración propia

De esta forma, no se pierde la información entregada por el programa luego de dejarse de ejecutar.

CAPÍTULO 5
CONCLUSIONES

5.1. Resultados

De lo que se puede analizar, es posible concluir que el algoritmo propuesto es extremadamente dependiente de las dos matrices que se le presenten, haciendo evidente que los tiempos de ejecución asintóticos presentados anteriormente se respaldan con los datos resultantes de las operaciones realizadas, sin embargo, esta propuesta tiene tiempos de ejecución mejores a lo que se conoce actualmente.

Además, esta nueva propuesta mejora el proceso descrito en la figura 1 del capítulo 1.3.1, porque el acceso a estos datos ya se maneja de forma óptima, por lo que, el siguiente proceso se muestra a continuación:

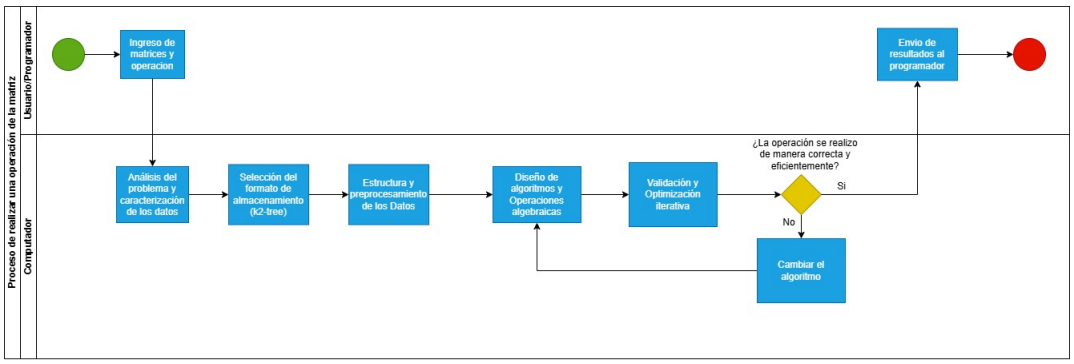


Figura 17: Proceso mejorado con la nueva propuesta
Fuente: Elaboración propia

Por otro lado, para matrices poco dispersas, los tiempos de ejecución sugieren que podría ser útil explorar otras técnicas específicas para manejar matrices con más elementos, como puede ser otro enfoque de implementación. Por lo tanto, para este tipo de matrices, no es muy recomendable utilizar esta estructura.

Para las matrices que son dispersas, este enfoque permite beneficiarse de esta estructura propuesta dado que permite un procesamiento de datos más eficiente que con otras estructuras, haciendo que el rendimiento de la solución propuesta sea predecible y consistente.

En cuanto al algoritmo propuesto en sí, este todavía presenta margen de mejora, en términos de optimización, haciendo posible optimizar aún más los tiempos de ejecución resultantes de la matriz. Sin embargo, el enfoque actual es prometedor en cuanto a resultados obtenidos comparados con el algoritmo convencional.

En cuanto a la representación propuesta, se ha logrado unificar una estructura que, en un principio, fue diseñada para representar grafos y relaciones entre páginas web, a una matriz con varios elementos no nulo en el cual se pueden realizar operaciones.

Por último, en cuanto al programa, este cumple con el objetivo de que cualquier usuario tenga una herramienta disponible que le permita usar esta estructura para sumar las dos matrices que estime conveniente. Por lo que, para este caso, es posible continuar con el desarrollo para futuras operaciones.

5.2. Trabajo futuro

El siguiente paso para este trabajo, es continuar implementando operaciones en base a esta estructura como lo es la multiplicación matriz-escalar, la multiplicación matriz vector, la multiplicación matriz-matriz, la transposición de una matriz, etc., además de optimizar las funciones ya existentes para un mejor procesamiento de los datos en matrices poco densas.

5.2.1. Multiplicación matriz escalar

Para este trabajo, se debe optimizar el número de llamadas recursivas al momento de decender por el árbol y solo multiplicar el escalar por los elementos vacíos. En este caso, se define A como una matriz en el espacio $\mathcal{M}(\mathbb{R})_{n\times n}$ teniendo en cuenta que $k = 2$, λ como un escalar teniendo en cuenta que $\lambda \in \mathbb{R}$, la multiplicación se puede resolver como:

$$C = A \cdot B = \lambda \cdot \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$$

Por lo que cada hijo de la matriz se puede calcular como:

- $C_1 = \lambda \cdot A_1$
- $C_2 = \lambda \cdot A_2$
- $C_3 = \lambda \cdot A_3$
- $C_4 = \lambda \cdot A_4$

Donde A_i y C_i son las submatrices o hijos de los árboles A y C correspondientes.

5.2.2. Multiplicación matriz-vector y matriz-matriz

Para este trabajo, se puede utilizar el mismo enfoque que la suma, donde también se asume que una submatriz es un hijo de un árbol. Para esta operación, se definen A y B como matrices en el espacio $\mathcal{M}(\mathbb{R})_{n\times n}$ además de $k = 2$, y por último se define:

$$C = A \cdot B = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \cdot \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$$

Por lo que cada componente se calcula como:

- $C_1 = A_1 \cdot B_1 + A_2 \cdot B_3$
- $C_2 = A_1 \cdot B_2 + A_2 \cdot B_4$
- $C_3 = A_3 \cdot B_1 + A_4 \cdot B_3$
- $C_4 = A_3 \cdot B_2 + A_4 \cdot B_4$

Donde A_i, B_i, C_i son submatrices o hijos de los árboles A, B y C correspondientes. En esta ocasión, el algoritmo propuesto es indispensable para que la multiplicación funcione, dado que ocupa las mismas características que resuelven el problema de la suma, como lo es la definición de una submatriz en el árbol. Por último, este algoritmo se basa en el algoritmo *z-order* discutido previamente, con la diferencia de que *z-order* se aplica directamente sobre una matriz mientras que, en este caso, se aplica sobre un k^2-tree.

Para el caso de la operación matriz-vector, al propio vector se le pueden ampliar el número necesario de columnas vacías para transformar el vector en una matriz donde solo la columna principal es la columna original del vector, por ejemplo, en el caso de este vector:

$$\vec{v} = \begin{pmatrix} 1 \\ 4 \\ 7 \\ 2 \end{pmatrix}$$

Si se le añaden 3 columnas vacías al vector, podemos obtener la siguiente matriz:

$$V = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

Haciendo posible que se pueda construir un k^2-tree y pudiendo aplicarse este nuevo algoritmo propuesto.

Por último, en este trabajo se deben de comparar el algoritmo propuesto con los ya existentes, como lo es el de dividir y conquistar, el de strassen, etc. y comprobar si esta estructura disminuye el tiempo de multiplicación.

5.2.3. Transpuesta de una matriz

Para este trabajo, se propone ir recorriendo e ir agrupando todos los hijos hoja del árbol en base a la posición que se encuentre en el vector de bits de las hojas, discriminando entre el primer, segundo, tercer y cuarto bit dentro del recorrido. Para ello, el recorrido debe de ir almacenando las coordenadas conforme avance la recursión del árbol para cuando se llegue a la hoja, almacenar los valores intercambiados, de esta forma, solo se procesan los valores relevantes no nulos.

En lo que hay que tener cuidado, es que, para matrices no cuadradas, la ampliación resultante a matrices que cumplan con el tamaño de la estructura, debe de tomarse en consideración que esta ampliación también se verá afectada por la operación, es decir, en caso de añadirse alguna fila o columna vacía, la traspuesta también puede generar una columna o fila resultante a la trasposición.

5.2.4. Diagonal de una matriz

Para este trabajo, se propone ir recorriendo el árbol almacenando coordenada conforme se descienda por éste, para cuando se lleguen a las hojas, comparar si los elementos de las coordenadas son iguales, se guarden en la matriz de retorno. Al igual que con la traspuesta de la matriz, tanto el acceso como esta comparación deben ser eficientes para que el costo sea mínimo.

Por suerte, esta operación solo es posible aplicarse en matrices cuadradas, así que, en caso de que no se cumplan los rangos de tamaños mínimos para el árbol, solo se debe de aumentar un igual número de filas y columnas, ignorando los últimos valores de la diagonal, ya que se sabe que serán vacíos.

5.2.5. Otras operaciones

Para este trabajo, se debe de tomar en cuenta estas y otras nuevas operaciones por describir:

- Factorización LU: En este caso, se debe de proponer dos algoritmos que puedan trian-

gular matrices mientras se realicen operaciones elementales sobre ellas. En este caso, la factorización LU se puede aprovechar de la multiplicación con matrices elementales.

- Determinante e inversa de una matriz: En este caso, estas operaciones deben de tomar en cuenta que la ampliación de la matriz invalidará el resultado debido a la naturaleza de ambas operaciones. Por lo que se sugiere trabajar solo con matrices dentro del tamaño de 2^n.

Por ende, para este trabajo, se debe de tomar en cuenta nuevas formas de cálculo para varias de las operaciones descritas, en el sentido de pensar un nuevo método matemático que pueda adaptar esta estructura.

ANEXOS

A. Resultados sumas de matrices de rendimiento

Debido al tamaño de los datos, se presentarán 100 de las sumas de las matrices realizadas en la sección 4.3.

Tabla 3: Primera tanda de 25 resultados
Fuente: Elaboración propia.

Matriz 1	Matriz 2	Tiempo de ejecución suma k^2-tree
m1006.txt	m1914.txt	$5{,}15962$
m1683.txt	m1143.txt	$52{,}9479$
m425.txt	m364.txt	$1{,}54455$
m38.txt	m1118.txt	$0{,}688327$
m1145.txt	m1815.txt	$0{,}118724$
m728.txt	m1193.txt	$0{,}061459$
m332.txt	m577.txt	$3{,}96139$
m1785.txt	m712.txt	$4{,}19774$
m138.txt	m205.txt	$0{,}046454$
m1265.txt	m169.txt	$0{,}036312$
m1244.txt	m1150.txt	$0{,}102171$
m1269.txt	m1622.txt	$1{,}58708$
m1133.txt	m1398.txt	$0{,}057164$
m1506.txt	m984.txt	$0{,}812173$
m1468.txt	m1447.txt	$8{,}56436$
m1137.txt	m1813.txt	$0{,}212145$
m1890.txt	m1421.txt	$9{,}76815$
m90.txt	m862.txt	$20{,}3228$
m731.txt	m668.txt	$0{,}018818$
m1668.txt	m113.txt	$0{,}806663$
m878.txt	m780.txt	$0{,}316337$
m1985.txt	m1397.txt	$0{,}037591$
m250.txt	m1572.txt	$0{,}058966$
m381.txt	m487.txt	$1{,}6644$
m861.txt	m1663.txt	$0{,}024929$

Tabla 4: Segunda tanda de 25 resultados
Fuente: Elaboración propia.

Matriz 1	Matriz 2	Tiempo de ejecución suma k^2-tree
m2002.txt	m130.txt	$0{,}008661$
m190.txt	m1699.txt	$8{,}36511$
m1503.txt	m1724.txt	$0{,}031098$
m411.txt	m269.txt	$0{,}131347$
m1783.txt	m1523.txt	$0{,}339002$
m1611.txt	m1685.txt	$0{,}197101$
m906.txt	m438.txt	$0{,}179141$
m883.txt	m317.txt	$0{,}809281$
m2062.txt	m1555.txt	$2{,}85764$
m1399.txt	m1116.txt	$0{,}885348$
m696.txt	m1602.txt	$0{,}037588$
m585.txt	m889.txt	$0{,}036681$
m1476.txt	m202.txt	$1{,}09804$
m1769.txt	m1853.txt	$0{,}017254$
m408.txt	m1423.txt	$0{,}035765$
m187.txt	m1361.txt	$7{,}00396$
m1214.txt	m633.txt	$0{,}854795$
m1966.txt	m1226.txt	$1{,}95517$
m660.txt	m1809.txt	$1{,}0902$
m583.txt	m377.txt	$0{,}565722$
m478.txt	m228.txt	$6{,}77162$
m390.txt	m295.txt	$132{,}175$
m1465.txt	m801.txt	$0{,}100418$
m640.txt	m1057.txt	$0{,}20619$
m2014.txt	m1098.txt	$0{,}039668$

Tabla 5: Tercera tanda de 25 resultados
Fuente: Elaboración propia.

Matriz 1	Matriz 2	Tiempo de ejecución suma k^2-tree
m1464.txt	m1099.txt	$0{,}769708$
m1636.txt	m1987.txt	$0{,}029145$
m1920.txt	m1267.txt	$0{,}037088$
m1983.txt	m1709.txt	$1{,}73011$
m277.txt	m953.txt	$0{,}560705$
m1886.txt	m1467.txt	$1{,}45973$
m336.txt	m81.txt	$12{,}7312$
m1630.txt	m245.txt	$0{,}032775$
m1607.txt	m1367.txt	$0{,}223943$
m79.txt	m1659.txt	$1{,}37251$
m809.txt	m584.txt	$1{,}57374$
m242.txt	m1069.txt	$3983{,}69$
m1351.txt	m1402.txt	$4{,}02023$
m536.txt	m1179.txt	$3{,}27209$
m1958.txt	m1511.txt	$0{,}095467$
m1255.txt	m157.txt	$0{,}420156$
m1772.txt	m754.txt	$4{,}64716$
m13.txt	m2030.txt	$2{,}3287$
m1835.txt	m335.txt	$7{,}1881$
m1160.txt	m1846.txt	$460{,}914$
m704.txt	m437.txt	$351{,}041$
m766.txt	m549.txt	$0{,}409417$
m1578.txt	m1531.txt	$0{,}981149$
m2026.txt	m1394.txt	$14{,}5309$

Tabla 6: Última tanda de 25 resultados
Fuente: Elaboración propia.

Matriz 1	Matriz 2	Tiempo de ejecución suma k^2-tree
m605.txt	m1688.txt	$2{,}60695$
m1856.txt	m1463.txt	$0{,}024906$
m1321.txt	m412.txt	$0{,}016708$
m877.txt	m1117.txt	$6{,}96841$
m1583.txt	m282.txt	$1{,}4597$
m311.txt	m544.txt	$6{,}35129$
m5.txt	m1020.txt	$0{,}042523$
m2096.txt	m666.txt	$1{,}36309$
m63.txt	m595.txt	$5{,}73607$
m486.txt	m1470.txt	$1{,}08168$
m1681.txt	m529.txt	$85{,}2575$
m1246.txt	m547.txt	$1{,}79475$
m788.txt	m1959.txt	$23{,}1847$
m286.txt	m413.txt	$4{,}59171$
m186.txt	m362.txt	$55{,}6304$
m533.txt	m1652.txt	$0{,}009408$
m956.txt	m40.txt	$6{,}57351$
m400.txt	m473.txt	$6{,}48705$
m96.txt	m172.txt	$22{,}3012$
m146.txt	m1195.txt	$0{,}25434$
m1534.txt	m1682.txt	$0{,}225243$
m299.txt	m890.txt	$0{,}314311$
m109.txt	m265.txt	$1{,}24573$
m710.txt	m280.txt	$1963{,}05$
m1359.txt	m977.txt	$0{,}253047$
m1471.txt	m1931.txt	$5{,}18369$

Para ver todos los resultados, ingrese al siguiente enlace.

REFERENCIAS BIBLIOGRÁFICAS

[Aggarwal, 2020] Aggarwal, C. C. (2020). *Linear Algebra and Optimization for Machine Learning*. Springer.

[Brisaboa *et al.*, 2015] Brisaboa, N. R., De Bernardo, G., Gutierrez, G., Ladra, S., Penabad, M. R., y Troncoso, B. A. (2015). Efficient set operations over k2-trees. *2015 Data Compression Conference, IEEE*.

[Brisaboa *et al.*, 2013] Brisaboa, N. R., Ladra, S., y Navarro, G. (2013). Compact representation of web graphs with extended functionality. *Elsevier*.

[Buluç y Gilbert, 2018] Buluç, A. y Gilbert, J. R. (2018). Optimizing sparse matrix-vector multiplication on multi-core and many-core architectures. *SIAM Journal on Scientific Computing*.

[Dongarra, 2020] Dongarra, J. (2020). Challenges in sparse linear algebra for high-performance computing. *Acta Numerica*.

[Dongarra *et al.*, 2000] Dongarra, J., Koev, P., Li, X., Demmel, J., y van der Vorst, H. (2000). Templates for the solution of algebraic eigenvalue problems: A practical guide. *SIAM Review*.

[Hogan *et al.*, 2019] Hogan, A., Riveros, C., Rojas, C., y Soto, A. (2019). A worst-case optimal join algorithm for sparql. *18th International Semantic Web Conference (ISWC)*.

[Samet, 2006] Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

[Schoor, 1982] Schoor, A. (1982). Fast algorithm for sparse matrix multiplication. *Information Processing Letters*.

[Simongog, 2015] Simongog (2015). Succinct data structure library 2.0, https://github.com/simongog/sdsl-lite. *Github*.