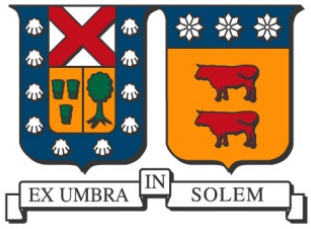
2022-08

IMPLEMENTACIÓN DE UN ALGORITMO APROXIMADO PARA LA ESTRUCTURA PARTITIONED ELIAS-FANO E-OPTIMAL

CARMONA TABJA, GABRIEL ALFREDO

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



"IMPLEMENTACIÓN DE UN ALGORITMO APROXIMADO PARA LA ESTRUCTURA *PARTITIONED ELIAS-FANO* ϵ*-OPTIMAL*"

GABRIEL CARMONA TABJA

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

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

Agosto - 2022

AGRADECIMIENTOS

Quisiera comenzar agradeciendo a mi madre y padre, gracias a ellos tuve la oportunidad de concentrarme en mis estudios y dar lo mejor de mí.

A mis hermanos los cuales me apoyaron durante mi crecimiento y enseñaron lo que ellos iban aprendiendo.

Al resto de mi familia que siempre estuvo ahí para quererme, apoyarme y cuidarme.

A mis amigos del colegio, dónde a pesar de que el tiempo pasa seguimos igual de cercanos. A mis amigos de la universidad, ellos me escucharon, acompañaron, alegraron, aconsejaron e hicieron que mi estadía en la universidad fuera de los mejores años que he tenido a pesar de las distintas dificultades que pasé.

A mis amigos que constan fuera de los grupos anteriormente mencionados, los que conocí realizando talleres y a mis amigos de la comunidad de speedrun. Ellos me ayudaron a pasar los malos momentos, formando una amistad durante estos años.

Especial agradecimiento a todos los que me ayudaron cuando les pedía consejos de cómo redactar correctamente y leyeron este documento con el fin de ayudarme.

A mi pareja que me apoyó en cada decisión que tomaba.

Por último, quisiera agradecer a los profesores:

- Diego Arroyuelo, debido a que fue mi mentor en esta área que me enganchó desde que pasé por estructuras de datos, ha sabido guiarme, enseñarme y darme oportunidades.
- José Luis Martí, dado a que me dio las oportunidades de ser su ayudante y confiar siempre en mí.
- Edson Carquín, dado que gracias a él y su apoyo incondicional pude adentrarme al mundo de la investigación en la vida real, obteniendo una colaboración en el proyecto *ACTS* del *CERN*.

Definitivamente si no fuera por todos ellos no estaría aquí.

RESUMEN

Las representaciones que permiten comprimir información son útiles en el día de hoy, esto debido a la masividad de la información, por lo que se busca poder comprimir correctamente la información original de tal forma de mantenerla correctamente, pero ocupando menos espacio. Entre las representaciones existen, Partitioned Elias-Fano corresponde a una estructura que permite comprimir la información correctamente, esta estructura tiene dos implementaciones: uniform que consiste en dividir la información en bloques con una cantidad uniforme de 1's y ϵ-optimal que consiste en dividir la información en bloques de bits con una cantidad variable de 1's utilizando un algoritmo aproximado. En ambas implementaciones, cada bloque será dividida por separado. Pero, además de comprimir el poder realizar operaciones sobre este conjunto comprimido es importante para no tener que descomprimir la información cada vez que se quiera realizar una consulta. Por esto, se implementaron las operaciones rank y select en la representación Partitioned Elias-Fano y los resultados de las pruebas de estas operaciones indican que si bien esta representación comprime correctamente, los tiempos de estas operaciones son notablemente mayores a otras representaciones.

Palabras clave: Partioned Elias-Fano, rank, select, algoritmo aproximado

ABSTRACT

The representations that allow information to be compressed are useful today, due to the massive amount of available data. For this reason, we seek to compress the original information in such a way as to maintain it while occupying less space. Among the existing representations, Partitioned Elias-Fano corresponds to a structure that allows the information to be compressed correctly. This structure has two implementations: uniform which consists of dividing information into blocks with a uniform amount of 1's and ϵ−optimal which consist of dividing the information into arrays of bits with a variable quantity of 1's using an approximate algorithm. In both implementations, each block will be partitioned separately. But, in addition to compressing the ability to perform operations on this set, it is important not to have to decompress the information every time you want to perform a query. For this reason, rank and select were implemented in the Partitioned Elias-Fano representation and the results of the operation tests indicate that although this representation compresses correctly, the execution times of these tests are greater than other representations

Keywords: Partioned Elias-Fano, rank, select, approximation algorithm

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

Hoy en día, los datos se trabajan de manera masiva, provocando que el almacenamiento comience a ser un factor a considerar. Con el fin de poder almacenar más información es relevante el poder representarlos y almacenarlos, los métodos de compresión llaman la atención al momento de trabajar los datos, esto se debe a que al momento de comprimir uno puede almacenar más información que la factible originalmente. Este caso es importante, especialmente cuando uno posee conjunto de enteros dentro un intervalo 0 hasta el universo del conjunto. Pero, comprimir y descomprimir son operaciones no necesariamente eficientes, por lo tanto uno debería ser capaz de realizar operaciones sobre la información comprimida sin tener que descomprimirla. Dentro de las operaciones que se pueden aplicar, las dos más conocidas son las operaciones *rank* y *select*.

Este trabajo se enfoca en la implementación de una estructura de compresión llamada *Partitioned Elias-Fano* [Ottaviano y Venturini, 2014], realizando un algoritmo aproximado para elegir una respuesta de tal forma sea $1 + \epsilon$ óptima. Además, se implementaran las operaciones *rank* y *select*, las cuales son dos operaciones que no se encuentran implementadas hoy en día.

Este documento consiste de cinco capítulos. En el capítulo 1 se introduce el contexto de este trabajo, la definición de las operaciones a implementar y los objetivos de este trabajo. En el capítulo 2 se detalla sobre algoritmos que serán utilizados para la implementación y representaciones que permiten comprimir información. En el capítulo 3 se especifica la implementación de la estructura, los algoritmos diseñados de las operaciones anteriormente mencionadas y las diversas formas en las cuales se calculo el espacio utilizado para las representaciones utilizadas. En el capítulo 4 se presenta los experimentos diseñados y los resultados obtenidos por estos mismos. En el capítulo 5 se concluye y discute sobre el trabajo futuro.

CAPÍTULO 1
DEFINICIÓN DEL PROBLEMA

1.1. Contextualización

Hoy en día, hay diversas aplicaciones para comprimir información con el fin de ahorrarse espacio y generar envíos de menor tamaño, pero que también permitan realizar operaciones sin tener que descomprimir con bajo consumo de tiempo. Este ha sido un tema que lleva años siendo trabajado, generando nuevas estructuras, algoritmos y representaciones, para comprimir de mejor manera la información. Es por esto que, al momento de generar un aporte al mundo de la compresión de texto, se deben generar comparaciones con las mejores propuestas publicadas hasta el momento, para poder comparar el trabajo en términos de tiempo de compresión, cantidad de espacio utilizar y tiempo en realizar operaciones dentro del conjunto comprimido.

1.2. Situación actual

Actualmente, existe una representación que tiene buenos resultados a nivel de compresión, con la que resulta imprescindible hacer una comparación de resultados. Esta se llama *Partioned Elias-Fano* (*PEF*) y fue publicada el 2014 por G. Ottaviano y R. Venturini [Ottaviano y Venturini, 2014]. PEF se basa en la idea de procesar la información antes de comprimirla: propone dividir la información en varios bloques, para que luego, dependiendo de la densidad de bits 1, se escojan distintas representaciones. De esta forma, cada bloque será comprimido utilizando la mejor representación para ese bloque en específico, y así ocupar menos espacio. Este trabajo, además, utiliza la base de la representación Elias-Fano [Elias, 1974], una representación clásica y estudiada en el área.

Por otro lado, *PEF* ha sido citada en diversos artículos, los que la mencionan, profundizan y/o se comparan con esta. Ejemplos de estos artículos son: *Clustered Elias-Fano Indexes* por G. Ottaviano y R. Venturini [Ottaviano y Venturini, 2017], donde, a base de *PEF*, profundizaron y generaron el *Clustered Elias-Fano; Dynamic Elias-Fano Representation* por R. Venturini and G. Ermanno [Venturini y Ermanno, 2017], donde se menciona la estrategia de *PEF*, al ser una estructura eficiente en espacio; y *Compressed Suffix Arrays to Self-Indexes Based on Partitioned Elias-Fano* por Guo Wenyu y Qu Youli [Wenyu y Youli, 2017], donde nuevamente en base a *PEF*, generan una nueva estructura para permitir mejores niveles de compresión.

Si bien la representación *PEF* presenta buenos resultados a nivel de compresión, no incluye las operaciones *rank* y *select*[Ottaviano, 2014], dos de las operaciones clásicas que se implementan en esta área, por lo que se podría mejorar. Ambas operaciones se definen a continuación:

- La operación *rank* se define como: dado un arreglo de bits y dada una posición i, se quiere determinar cuantos 1's hay entre la posición 0 e i.
- La operación *select* se define como: dado un arreglo de bits y dado un número i entre 1 y la cantidad de 1's en el arreglo, determinar en que posición del arreglo está el i-ésimo 1.

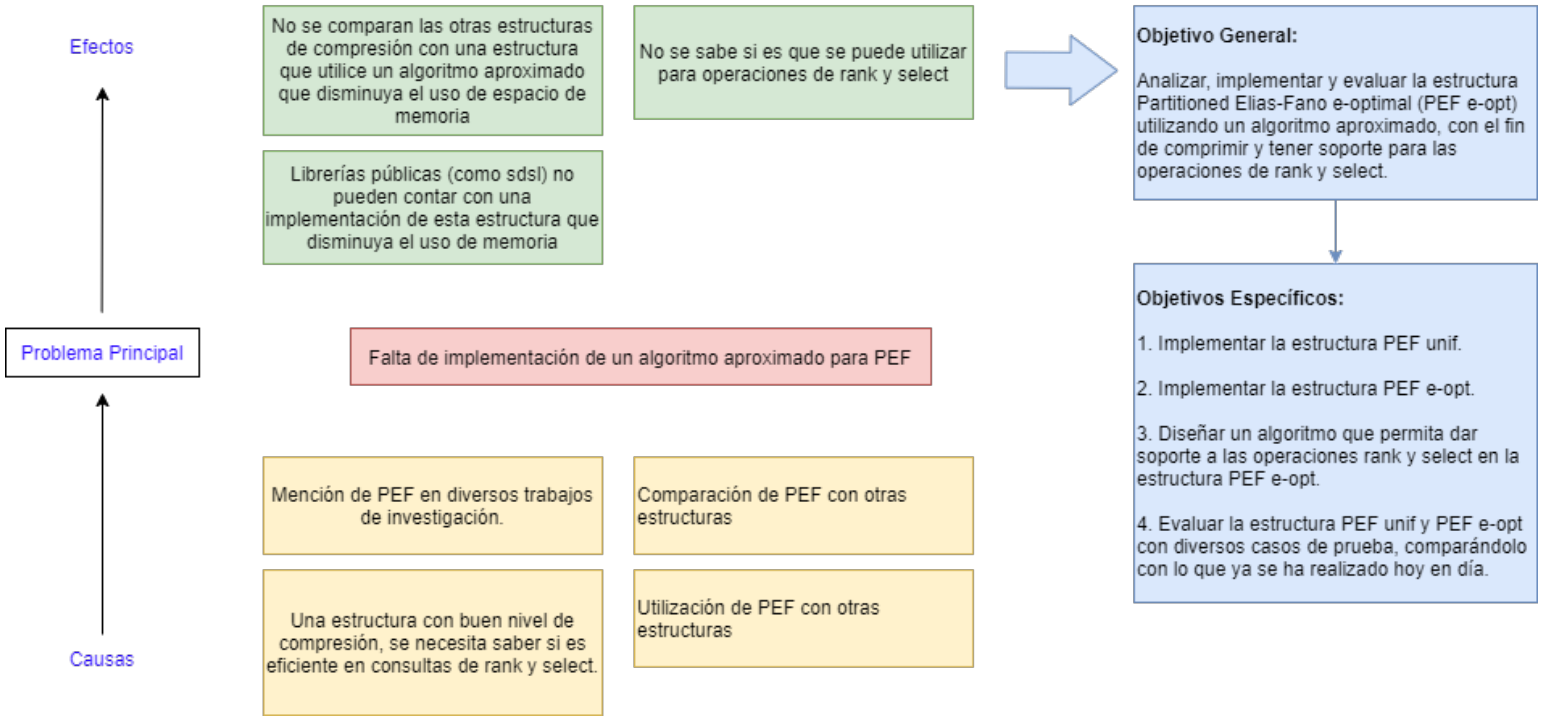


Figura 1: Árbol del problema

Por lo tanto, dada la importancia y relevancia que ha tenido *PEF* dentro del área, es de especial interés poder mejorar el código actual de esta estructura, enfocándose en la estructura *PEF ϵ-opt*, dado que presenta los mejores resultados en términos de espacio usado después de la compresión. Además, se busca también poder implementar las dos operaciones explicadas anteriormente, para así analizar y comparar esta estructura con las otras del área, pudiendo así concluir sobre cómo se comporta esta estructura para operaciones de ese estilo.

Parte del código actual de *PEF* se encuentra en uso con la librería sdsl [Gog y et al., 2019], una librería que contiene diversas estructuras, algoritmos y operaciones de compresión, por lo que los resultados obtenidos durante este trabajo podrán finalizar en una colaboración dentro de esta librería, la cual significaría aportar al área con una estructura de compresión.

1.3. Objetivos

1.3.1. Objetivo General

Analizar, implementar y evaluar la estructura *Partitioned Elias-Fano ϵ-optimal* (*PEF ϵ-opt*) utilizando un algoritmo aproximado, con el fin de comprimir y tener soporte para las operaciones de *rank* y *select*.

1.3.2. Objetivos Específicos

- Implementar la estructura *Partitioned Elias-Fano Uniform* con soporte a las operaciones *rank* y *select*.
- Implementar la estructura *Partitioned Elias-Fano ϵ-optimal* utilizando un algoritmo aproximado.
- Diseñar un algoritmo que permita dar soporte a las operaciones *rank* y *select* en la estructura *Partitioned Elias-Fano ϵ-optimal*.
- Evaluar la estructura *Partitioned Elias-Fano Uniform* y *Partitioned Elias-Fano ϵ-optimal* con diversos casos de prueba, comparándolo con lo que ya se ha realizado hoy en día.

CAPÍTULO 2
MARCO CONCEPTUAL

2.1. Algoritmos Aproximados

Los problemas de optimización son problemas en los que se quiere encontrar la respuesta óptima dentro de un contexto definido. Esto es posible de realizar en diversos problemas, pero existen otros en los cuales no se posee una solución en tiempo polinomial, provocando que, aunque en teoría se pueda llegar al valor óptimo del problema con un tiempo de ejecución considerablemente grande, ese tiempo no sea viable de aplicar en la práctica. Este grupo de problemas es conocido como problemas *NP-Hard* [Knuth, 1974].

Para solventar que no existan soluciones en tiempo polinomial para los problemas *NP-Hard*, se definieron los algoritmos aproximados. Estos algoritmos consisten en encontrar una solución no óptima en tiempo polinomial que cumpla ciertos estándares de calidad, es decir, cercano al valor óptimo. Pero no es suficiente con desarrollar una solución cualquiera, sino que también las respuestas que entregue el algoritmo aproximado deben ser una respuesta de calidad, o sea cercanas al valor óptimo [Vazirani, 2001].

La principal característica de los algoritmos aproximados es que tienen definido un valor ϵ, donde este valor indica cuán peor será la respuesta obtenida con respecto a la respuesta óptima. De esta forma, esta técnica puede asegurar respuestas de calidad en tiempo polinomial que, si bien no es la solución óptima, es suficiente para diversos problemas de optimización.

Debido a esto último, al usar un algoritmo aproximado se debe realizar un análisis previo, con el objetivo de demostrar que se puedan obtener soluciones de calidad. Si se demuestra que la mejor aproximación para un problema dará como resultado soluciones de baja calidad, esta técnica debería ser descartada. En cambio, si se demuestra que el algoritmo aproximado posee un ϵ que permita entregar soluciones de calidad, entonces se puede continuar por este camino.

Este tipo de algoritmos se han aplicado para resolver problemas clásicos como *Set Cover*, *Multiway Cut* y *k-Cut* [Vazirani, 2001].

2.2. Bit vector plano

Un bit vector plano es una representación de un conjunto de números, utilizando una secuencia de 0's y 1's, donde si en la posición i se encuentra un 1 significa que el número i se encuentra en el conjunto, si en la posición i se encuentra un 0 significa que el número i no se encuentra en el conjunto.

Para almacenar una secuencia de u bits, utiliza palabras de largo w, de esa forma guarda w bits en una palabra, por lo que permite almacenar esta secuencia utilizando $\lceil \frac{u}{w} \rceil$ palabras. Obteniendo de esta manera una representación que utilizaría $w \cdot \lceil \frac{u}{w} \rceil$ bits.

Por ejemplo, si se tiene el conjunto siguiente $S = 1, 3, 5, 6, 8, 12$, con un universo $u = 24$, este conjunto se representaría con la siguiente secuencia de números $BS = 101011010001$. Ahora, se almacena esta secuencia en palabras de largo 3, obteniendo las divisiones siguientes $w_1 = 101$, $w_2 = 011$, $w_3 = 010$ y $w_4 = 001$. Generando el bit vector plano $BV_S = 101, 011, 010, 001$.

Existen diversas librerías que implementan esta representación, con acceso a las operaciones *rank* y *select*: *Sux library* [Vigna, 2008], la librería *sdsl* [Gog y et al., 2019] y *Succint Library* [Grossi y Ottaviano, 2013].

2.2.1. Bit vector plano en la *sdsl*

La implementación de un bit vector plano en la librería *sdsl* consiste en una clase la cual usa palabras de largo 64. Además, si es que uno quiere hacer uso de las operaciones *rank* y *select* sobre el bit vector, uno debe hacer uso de clases extras para poder tener la información generada y así permitir que las operaciones sean eficientes.

Para poder realizar la operación *rank*, la librería otorga diversas clases, entre las cuales se encuentran: *rank_support_v* y *rank_support_v5*, pero al ser usadas generan un espacio extra de $25\,\%$ y $6{,}25\,\%$ del universo del bit vector respectivamente. Las complejidades de la operación *rank* para ambas clases es $O(1)$[Gog y et al., 2019].

Para poder realizar la operación *select*, la librería otorga diversas clases, entre las cuales se encuentra: *select_support_mcl* que al ser usada genera un espacio extra de a lo más $20\,\%$ del espacio generado. La complejidad de la operación *select* en esta clases es $O(1)$ [Gog y et al., 2019].

2.3. Elias-Fano

La representación Elias-Fano es una codificación para secuencias monótonas, la cual posee una buena compresión de la información, acceso eficiente y operaciones de búsqueda dentro de la información comprimida [Elias, 1974, Fano, 1972].

Para explicar la representación, considere una secuencia monótona $S[0, m-1]$ de m enteros no negativos, esto quiere decir que $S[i] \leq S[i+1]$ para cualquier $0 \leq i \leq m-1$. Y estos elementos son extraídos de un universo u ($u = \{0, 1, ..., u-1\}$).

Dado un entero l, los elementos de S son agrupados según los $\lceil \log u \rceil - l$ bits más significa-

tivos. Luego de crear los grupos, se escriben las cardinalidades de cada grupo en un vector de bits, donde se escribirá en código unario, escribiendo un 0 para indicar que hasta ahí llega la cardinalidad de ese grupo, de esta forma el arreglo actual tendrá a lo más $m + \frac{u}{2^l}$ bits.

Los restantes l bits inferiores serán concatenados en otro vector de bits. Obteniendo así un arreglo que requiere ml bits.

Por ejemplo, si se tiene el arreglo siguiente $S = \{2, 3, 5, 7, 11, 13, 24\}$, con un universo $u = 24$ y el valor $l = 2$. Primero, se debe que escribir cada número en binario para tener los $\lceil \log u \rceil - l$ bits más significativos separados del resto de bits. Teniendo en consideración que el largo de cada número en binario será de $\log u$.

Tabla 1: Ejemplo de arreglo S con sus valores descompuestos por sus $\lceil \log u \rceil - l$ bits más significativos y l bits menos significativos

Números	2		3		5		7		11		13		24	
Binario	000	10	000	11	001	01	001	11	010	11	011	01	110	00

En la tabla 1, se puede ver los valores por sus $\lceil \log u \rceil - l$ bits más significativos y l bits menos significativos. Ahora, se agruparán los $\lceil \log u \rceil - l$ bits más significativos de cada número, obteniendo así cuántas veces aparece ese grupo en los $\lceil \log u \rceil - l$ bits más significativos de cada número del conjunto S.

Tabla 2: Cardinalidades de cada grupo de $\lceil \log u \rceil - l$ bits según lo observado en la tabla 1

$\lceil \log u \rceil - l$ más significativos	000	001	010	011	100	101	110	111
Cardinalidades	2	2	1	1	0	0	1	0
Unario	11	11	1	1			1	

Se puede observar en la tabla 2, las cardinalidades de cada grupo y su codificación en unario, luego a cada cardinalidad en unario se debe añadir un 0 al final y agregarlo al primer vector de bits, dando como resultado el siguiente vector de bits 110110101000100. Posteriormente, se añade los l bits menos significativos de cada número en otro vector de bits, dando el siguiente vector de bits 10110111110100. Finalmente, se junta el primer vector de bits con el segundo vector de bits, dando como resultado el siguiente vector de bits $11011010100010010110111110100$, correspondiente a la representación final del conjunto S.

Se puede observar que en esta representación, el valor de l es un valor que se debe buscar para poder minimizar el uso del espacio de la representación. Y el valor que minimiza el uso del espacio, cumpliendo así el objetivo de la compresión, corresponde a $\lceil \log \frac{u}{m} \rceil$.

Debido a lo explicado anteriormente se puede sumar el espacio ocupado por los dos arreglos mencionados. Al sumarlos se puede observar que esta representación requiere a lo más $m\lceil \log \frac{u}{m} \rceil + 2m$ bits para poder generarla [Ottaviano y Venturini, 2014].

Existen diversas implementaciones eficientes de *Elias-Fano* que tienen soporte a operaciones *rank* y *select*, unas de esas representaciones, la cuál será utilizada, corresponde a la implementada en la librería sdsl [Gog y et al., 2019] y la implementada en la *Sux library*[Vigna, 2008].

2.4. Problema del camino más corto en un grafo dirigido sin ciclos

El problema del camino más corto en un grafo consiste en un tópico común dentro del área de la algoritmia teniendo así diferentes algoritmos clásicos para poder resolver este problema, como lo son *Breadth-first search* [Bauer y Wössner, 1972] que sirve para grafos sin peso o el algoritmo de Dijkstra [Dijkstra, 1959] que sirve para un grafo con pesos.

Aunque existe una gran cantidad de variaciones para este problema, en este casp se resolverá el problema para un grafo dirigido con pesos y sin ciclos. Esto debido a ser un caso limitado de los mencionados anteriormente.

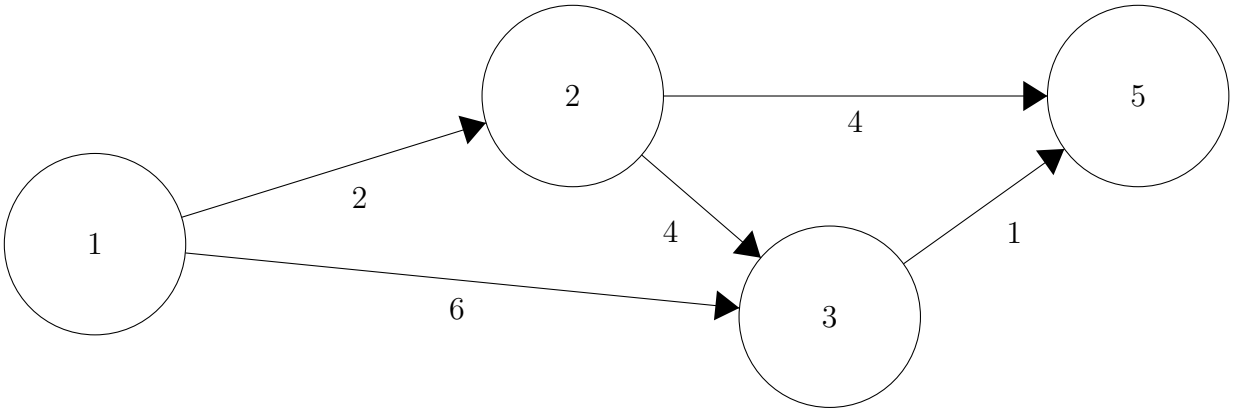


Figura 2: Ejemplo de un grafo dirigido con pesos y sin ciclos

En la figura 2, se puede observar cómo las conexiones que se presentan no permiten la aparición de ciclos. Entonces, para este caso la solución del problema, de ir desde el nodo 1 hasta el nodo 5, sería ir del nodo 1 hasta el nodo 2 y del nodo 2 dirigirse al nodo 5.

Si bien esto es similar a usar el algoritmo de Dijkstra, al tener la limitación que no puede tener ciclos permite la generación de diferentes algoritmos para resolver el problema. Además que este tipo de grafos posee diferentes aplicaciones como: planificación de tareas ordenados según restricciones [Skiene, 2008] o procesamiento de información en redes [Touati y Dupont, 2014].

CAPÍTULO 3
PROPUESTA DE SOLUCIÓN

3.1. Partitioned Elias-Fano

PEF consiste en una técnica de compresión que toma la información y la divide en bloques. Comprimiendo la información en dos niveles.

- Primer nivel: corresponde a una representación de *Elias-Fano* de un arreglo L que contiene el último elemento de cada bloque.
- Segundo nivel: corresponde a una representación por cada bloque generado, entre estas representaciones se encuentran: *Elias-Fano*, *bit_vector plano* y *runs* de únicamente 1's. Más adelante en el documento se elige la mejor representación para cada bloque.

Existen dos diferentes versiones: *PEF unif* y *PEF* ϵ*-opt* [Ottaviano y Venturini, 2014]. La diferencia principal que tienen estas dos se da en cómo construyen los bloques.

Ambas versionen tienen en común lo siguiente: un arreglo para guardar un puntero hacia cada representación (*blocks*), un arreglo para guardar un puntero hacia la operación *rank* (*blocks_rank*) y un arreglo para guardar un puntero hacia la operación *select* (*blocks_select*).

Por último, aparte del arreglo *blocks*, se necesita un *bit_vector* plano que tendrá un universo equivalente a la cantidad de bloques y donde si en la posición i se almacena un 1 es debido a que el bloque i utiliza una representación *Elias-Fano*, sino se almacenará un 0.

Gracias a este *bit_vector* y al arreglo *blocks*, se puede determinar a qué representación corresponde cada bloque:

- Si la posición i del *bit_vector* se almacena un 1, entonces corresponde a la representación *Elias-Fano*.
- Si la posición i del *bit_vector* se almacena un 0 y la misma posición del arreglo de punteros hacia la representación no es nulo, entonces corresponde a la representación de un *bit_vector* plano.
- En otro caso, significa que el bloque corresponde a una *run* de únicamente 1's.

3.1.1. Partitioned Elias-Fano Uniform

PEF unif consiste en mantener el tamaño de los bloques (b) constante, de tal forma que cuando recibe un arreglo de bits, genera bloques con la misma cantidad de 1's. Luego, por cada bloque debe decidir qué compresión realizar, diferenciando tres casos:

1. Primero, si el bloque contiene una cantidad b de bits y, por lo tanto, el bloque contiene solamente 1's, solo se debe recordar esa uniformidad.
2. Segundo, si el bloque tiene un universo u_j, siendo j el número del bloque, donde $b > \frac{u_j}{4}$, entonces se utilizará la representación *Elias-Fano* [Elias, 1974].
3. Tercero, si no se cumple ninguna de las dos condiciones anteriores, entonces se utilizará la representación de un *bit_vector* plano.

Además, por cada bloque se almacenará en un arreglo L la última posición en la cual hubo un 1 dentro del arreglo de bits dado.

Por ejemplo, si se tiene el siguiente arreglo de bits $B = \{0,0,1,1,0,1,0,1,0,1,0,1,0,0,0,1,0,0,0,1,1,1,1,1\}$, con universo $u = 24$, con tamaño de bloques $b = 4$, se realizarán los siguientes pasos:

1. El primer bloque que contendrá b 1's, será el que parte en la posición 1 del arreglo y termina en la posición 8 del arreglo, generando un bloque con universo $u_1 = 8$, $b_1 = \{0,0,1,1,0,1,0,1\}$. Este bloque se puede ver que no cumple la primera condición explicada anteriormente, ya que tiene bits que son 0. La segunda condición explicada sí la cumple, esto debido a que $b = 4$ y $u_1 = 8$, entonces $4 > 8/4$, por lo que b_1 tendrá una codificación utilizando un *bit_vector*. Por último, se almacenara en el arreglo L un 7, ya que este corresponde a la posición del último 1 del bloque.
2. El segundo bloque que contendrá b 1's, será el que parte en la posición 8 y termina en la posición 19, generando así un bloque con universo $u_2 = 11$, $b_2 = \{0,1,0,1,0,0,0,1,0,0,0,1\}$. Al igual que el bloque anterior, $b > u_2/4$, por lo que nuevamente se utilizará la codificación utilizando un *bit_vector*. Y por último se guardará en L un 19, correspondiente a la posición del último 1 del bloque. Quedando hasta ahora un arreglo $L = \{7, 19\}$
3. Finalmente, el último bloque que se puede generar que contiene b 1's, corresponde a los últimos cuatro bits del arreglo, quedando así un bloque con universo $u_3 = 4$, $b_3 = \{1,1,1,1\}$. Como se puede observar, este bloque contiene solamente 1's, por lo tanto se cumple la primera condición explicada anteriormente, provocando que no se utilice ninguna codificación y solo se tenga que recordar que este bloque contiene solamente 1's. Finalmente, al arreglo L se agrega un 23, quedando finalmente de la siguiente manera: $L = \{7, 19, 23\}$.

Finalmente, dado el procedimiento anterior, se puede observar en las tablas 3 y 4 cuáles fueron los bloques generados y el arreglo L obtenido. Cada bloque ocupará la codificación descrita en el procedimiento anterior.

Tabla 3: División del arreglo B utilizando *PEF Uniform*.

Bloque 1								Bloque 2											Bloque 3			
0	0	1	1	0	1	0	1	0	1	0	1	0	0	0	1	0	0	0	1	1	1	1

Tabla 4: Arreglo L obtenido a partir de división aplicada al arreglo B.

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

El principal problema de esta técnica corresponde a la elección del tamaño de los bloques b. Esto debido a que puede darse el caso en que para un bloque el tamaño b puede ser correcto, pero para otro bloque sería mejor tener un tamaño mayor o menor dependiendo del caso. Por estas razones, es que surge la siguiente versión para solventar este problema, permitiendo decidir de qué tamaño será cada partición, ajustando la codificación a cada partición.

3.1.2. Partitioned Elias-Fano ϵ-optimal

PEF ϵ-opt consiste en tener bloques de tamaño variables. Esto porque, si se tiene un tamaño fijo b, la división no será óptima, dado que pueden existir bloques a los que, si se les agrega o quita una mayor cantidad de 1's, pueden comprimirse mejor.

Sin embargo, buscar la partición que minimice el espacio utilizado total no es trivial, esto debido a que: por un lado, los bloques deberían ser lo más grande posible para minimizar el número de bloques a crear (primer nivel); y, por otro lado, los bloques deben ser de igual manera pequeños para minimizar la distancia entre los bits 1 dentro de cada bloque, disminuyendo así la ocupación del espacio a nivel de bloque (segundo nivel). Se define un algoritmo de programación dinámica [Buchsbaum *et al.*, 2003] que podría computar el resultado en $\theta(n^2)$, tanto en tiempo como en espacio, pero esta complejidad es inviable para entradas mayores que unos pocos miles de enteros.

Este algoritmo responde al problema del camino más corto en grafo dirigido sin ciclos (DAG), donde este grafo tendrá n vértices, uno por cada valor en el conjunto a comprimir, y $\theta(n^2)$ aristas, donde cada arista $e(i, j)$ tiene un peso $w(i, j)$ que representaría el número de bits que se necesita para representar el conjunto a comprimir de la posición i hasta la j [Ottaviano y Venturini, 2014] [Ottaviano y Venturini, 2017].

La solución entregada a partir de este algoritmo debe, por cada partición, almacenar el universo, su tamaño y un puntero al bloque codificado. Entonces, todo esto resulta en una representación que utilizaría $O(2 \log u + \log n)$ bits.

Para lograr esto, se deben podar las aristas con el objetivo de reducir el número de aristas, donde se tienen que cumplir dos condiciones:

- Si existe un entero $h \geq 0$ tal que $w(i,j) \leq (1+\epsilon)^h F < w(i,j+1)$, donde F representa el costo de almacenar la codificación de la secuencia original.
- (i,j) es la última arista que sale del nodo i, esto quiere decir que $j = n$.

Con esto se generará un grafo G_ϵ el cual sus aristas permitirían aproximar el valor, esto debido a que los costos de las aristas en este problema corresponden a una secuencia monótona. Estas aristas son llamadas aristas $(1+\epsilon)$-*maximal*.

Finalmente, gracias a lo anterior, el grafo G_ϵ tendría a lo más $\log_{1+\epsilon} \frac{u}{F}$ aristas, permitiendo así reducir la complejidad del algoritmo de programación dinámica de $\theta(n^2)$ a $O(n \log_{1+\epsilon} \frac{u}{F})$, aunque posteriormente esto fue reducido a $O(m \log_{1+\epsilon} \frac{1}{\epsilon})$ en tiempo y complejidad lineal en espacio [Ottaviano y Venturini, 2014] [Ottaviano y Venturini, 2017]. Finalmente se obtiene un algoritmo aproximado que permite asegurar que la respuesta será a lo más $1+\epsilon$ ($\epsilon \in (0,1)$) más grande que la solución óptima.

Ahora, para poder generar este grafo G_ϵ se decide aplicar una cantidad de $O(\log_{1+\epsilon} \frac{1}{\epsilon})$ ventanas $W_1, W_2, ...W_q$. Cada ventana permitiría cubrir una diferente porción del arreglo original. Entonces, se generan máximo q aristas que salen del nodo i a medida que el algoritmo, para encontrar el camino más corto, visite ese nodo. Inicialmente, todas las ventanas parten y terminan en la posición 0. Luego, cuando el algoritmo visita el siguiente nodo i, se avanza la posición inicial por cada ventana de una posición y la posición final j, hasta que se cumpla que $w(i,j) > (1+\epsilon)^j F$. Añadido a este algoritmo, se debe definir un parámetro que consiste en el costo mínimo que ocuparía un bloque, a parte de la representación respectiva, este parámetro se denominará *fixed cost*

Por último, gracias a la complejidad de construcción de la representación *PEF ϵ-optimal*, se tiene dos parámetros a fijar $\epsilon_1 \in [0,1)$ y $\epsilon_2 \in [0,1)$. Los valores de estos parámetros afectan directamente al espacio y el tiempo para la codificación del arreglo original, por ende buscar los valores correctos es esencial para una buena codificación y en un tiempo correcto.

En una primera instancia Ottaviano y Venturini fijan $\epsilon_1 = 0$, esto para no tener de límite el costo de la construcción de bloques, y hacen variar ϵ_2 entre $0{,}025$ y $0{,}5$, los experimentos realizados muestran que ocurre una disminución rápida cuando $0{,}025 \leq \epsilon_2 \leq 0{,}1$, luego mientras más grande el valor ϵ_2 el espacio ocupado por la codificación no disminuye considerablemente. Ahora, mientras se aumente el valor de ϵ_2 el tiempo empleado para la construcción es mayor. Posterior a estos experimentos, ellos decidieron fijar $\epsilon_2 = 0{,}3$ para poder balancear el espacio y tiempo utilizado. Luego, al tener fijado ϵ_2 se puede hacer variar ϵ_1, los experimentos realizados hacen variar ϵ_1 entre 0 y $0{,}1$. Los resultados de este último experimento permite observar un comportamiento similar al anterior. Donde a partir de este los autores fijaron el valor de ϵ_1 en $0{,}03$ [Ottaviano y Venturini, 2014].

A partir de la realización de este algoritmo uno obtiene las particiones, usando estas particiones uno generará los bloques con la mejor representación para cada uno. Finalmente, para mantener la representación correcta uno no solo necesita el arreglo L mencionado anteriormente y un puntero a la representación, sino que se debe agregar un arreglo E donde el i-ésimo 1 debe corresponder a la cantidad de 1's acumulada hasta el bloque i. La representación de este arreglo será una representación *Elias-Fano*.

Para ejemplificar lo anteriormente explicado, si se tiene el siguiente arreglo de bits $B = \{0,0,1,1,0,1,0,1,0,1,0,1,0,0,0,1,0,0,0,1,1,1,1,1\}$, con universo $u = 24$ y las particiones obtenidas son $B_1 = \{0,0,1,1,0,1,0,1,0,1\}$, $B_2 = \{0,1,0,0,0,1,0,0,0,1\}$ y $B_3 = \{1,1,1,1\}$, cabe destacar que esto es un ejemplo, porque en la práctica el algoritmo no dividiría así este arreglo de bits.

A partir de estos bloques se debe generar los arreglos L y E. L en este caso sería $\{9,19,23\}$, puesto que es el último elemento de cada bloque. E en este caso sería $\{5,8,12\}$: esto debido a que el primer bloque tiene 5 elementos; el segundo bloque tiene 3 elementos y antes del segundo bloque habían 5 elementos, dando un total de 8 elementos hasta ese bloque; por último el tercer bloque tiene 4 elementos, dando entonces 12 elementos hasta ese bloque.

Finalmente, se puede apreciar en las tablas 5, 6 y 7, cuáles fueron los bloques generados, el arreglo L y el arreglo E obtenido.

Tabla 5: División del arreglo B utilizando *PEF ϵ-opt*.

Bloque 1										Bloque 2										Bloque 3			
0	0	1	1	0	1	0	1	0	1	0	1	0	0	0	1	0	0	0	1	1	1	1	1

Tabla 6: Arreglo L obtenido a partir de división aplicada al arreglo B.

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

Tabla 7: Arreglo E obtenido a partir de división aplicada al arreglo B.

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

3.2. Operación Rank en PEF

3.2.1. *PEF uniform*

La operación *rank* recibe una posición i entre 0 y u, donde u es el universo del arreglo de bits original. Al recibir i debe primero determinar en qué bloque se encuentra, el cual será denominado blk. Para ello se realiza la operación *rank* sobre el arreglo L. Si no corresponde al primer bloque, entonces se sabe que hay al menos $b \cdot blk$.

Dado el bloque blk en que se encuentra la posición i, se debe saber cuál es la respectiva posición dentro de él, para eso se realiza la operación *select* sobre L para determinar el último elemento del bloque anterior, el cual será conocido como $ultimo\_elemento$. Con esto, la respectiva posición dentro del bloque es $j = i - 1 - ultimo\_elemento$.

Por último, se realiza la operación *rank* de la representación del bloque con el valor j, obteniendo así la cantidad de 1's antes de la posición j, donde estos serán llamados $unos\_bloque$. Por lo que, el resultado de *rank* en *PEF uniform* será $b \cdot blk + unos\_bloque$.

Por ejemplo, si se tiene los bloques de la tabla 3 y el arreglo L de la tabla 4, se realiza la consulta *rank* sobre la posición 14. Primero, al realizar *rank* sobre L se obtiene que la posición 14 se encuentra en el segundo bloque, entonces se sabe que hay por lo menos 4 1's antes.

Ahora, se debe determinar la posición dentro del segundo bloque aplicando la operación *select* sobre L, gracias a esto se obtiene que el último 1 del bloque anterior fue 7, por lo tanto la posición dentro del bloque equivale a $14 - 7 - 1 = 6$.

Finalmente, se aplica la operación *rank* dentro del segundo bloque utilizando la posición obtenida previamente, dando como resultado 2. Por lo tanto, el resultado de la operación *rank* en *PEF uniform* sería $4 + 2 = 6$.

3.2.2. *PEF ϵ-opt*

La operación *rank* recibe una posición i entre 0 y u, donde u es el universo del arreglo de bits original. Para obtener el valor de esta operación en *PEF ϵ-opt* se deben realizar los siguientes pasos.

Primero, al recibir i, se debe determinar el bloque en que se encuentra esa posición, el cual será denominado blk. Para ello se realiza la operación *rank* sobre el arreglo L.

Segundo, como *PEF ϵ-opt* tiene bloques de tamaño variable se debe realizar la operación *rank* sobre el arreglo E, ya que esto determinará cuantos 1's hay dentro de los bloques anteriores al bloque actual ($unos\_anteriores$).

Obtenido ese valor y teniendo el bloque en que se encuentra la posición i, se debe saber cuál es la posición respectiva dentro de él. Para eso se aplica la operación *select* sobre L con el valor blk para determinar el último elemento del bloque anterior ($ultimo\_elemento$). Por lo tanto, la posición respectiva dentro del bloque es $j = i - 1 - ultimo\_elemento$.

Por último, se realiza la operación *rank* de la representación del bloque con el valor j, obteniendo así la cantidad de 1's antes de la posición j ($unos\_bloque$). Por lo que, el resultado de *rank* será $unos\_anteriores + unos\_bloque$.

Por ejemplo, si se tiene los bloques de la tabla 5, el arreglo L de la tabla 6 y el arreglo E de la

Data: i
if $i \geq u$ **then**
  return n ;
end
$blk \leftarrow rank\_L(i)$;
if $blk \geq cantidad\_de\_bloques$ **then**
  return n ;
end
$rank\_ret \leftarrow 0$;
if $blk > 0$ **then**
  $rank\_ret \leftarrow rank\_ret + b \cdot blk$;
end
if $B[blk]$ **then**
  if $blk == 0$ **then**
    $rank\_ret \leftarrow rank\_ret + block\_rank[blk](i)$;
  else
    $i\_block \leftarrow i - 1 - select\_L(blk)$;
    $rank\_ret \leftarrow rank\_ret + block\_rank[blk](i\_block)$;
  end
else
  if $P[blk]$ **then**
    if $blk == 0$ **then**
      $rank\_ret \leftarrow rank\_ret + block\_rank[blk](i)$;
    else
      $i\_block \leftarrow i - 1 - select\_L(blk)$;
      $rank\_ret \leftarrow rank\_ret + block\_rank[blk](i\_block)$;
    end
  else
    if $blk == 0$ **then**
      $rank\_ret \leftarrow rank\_ret + i$;
    else
      $rank\_ret \leftarrow rank\_ret + i - 1 - select\_L(blk)$;
    end
  end
end
return $rank\_ret$

Algoritmo 1: Operación *rank* en *PEF uniform*

tabla 7, se realiza la operación *rank* sobre la posición 14. Primero, al realizar *rank* sobre L se obtiene que la posición 14 se encuentra en el segundo bloque, entonces se debe determinar cuantos 1's habían antes de ese bloque, para ello se debe realizar la operación *select* sobre E, gracias a esta operación se determina que hay 5 1's antes del segundo bloque.

Ahora, se debe obtener la posición dentro del segundo bloque aplicando la operación *select* sobre L, gracias a esto se tiene que el último 1 del bloque anterior fue 9, por lo tanto la posición equivale a $14 - 9 - 1 = 4$.

Finalmente, se aplica la operación *rank* dentro del segundo bloque, utilizando la posición descrita previamente, obteniendo como resultado 1. Por lo tanto, el resultado de la operación *rank* sobre la posición 14 será $5 + 1 = 6$.

3.3. Operación Select en PEF

3.3.1. *PEF uniform*

La operación *select* recibe i, indicando que se consulta sobre el i-ésimo 1, entre 1 y n, donde n es la cantidad de 1's en el arreglo de bits original. Para obtener la posición del i-ésimo 1, se debe realizar los siguientes pasos.

Primero, se determina en qué bloque se encuentra el i-ésimo 1, realizando la operación $blk = (i-1)/b$, debido a que todos los bloques tienen la misma cantidad de 1's.

Segundo, al saber en qué bloque se encuentra, si corresponde al primero, entonces se retorna la operación *select* sobre este. En caso contrario, se debe determinar a qué 1 corresponde dentro del bloque. Para ello, se realiza la operación $i\_bloque = (i-1)\ \%b + 1$.

Luego, se aplica la operación *select* dentro del bloque con el valor de i_bloque dado para determinar la posición dentro de él, la cual será denominada pos_bloque. Además, se debe obtener cuál es la posición del último 1 del bloque anterior. Para ello hay que realizar la operación *select* sobre L con blk ($ultimo\_elemento$).

Finalmente, el resultado de la operación *select* corresponde a la suma $pos\_bloque + ultimo\_elemento + 1$.

Por ejemplo, si se tiene los bloques de la tabla 3 y el arreglo L de la tabla 4, se quiere determinar la posición del sexto 1.

Primero, se debe determinar el bloque en que se encuentra el sexto 1, $blk = \lfloor (6-1) \rfloor / 4 = 1$. Por lo tanto se encuentra en el segundo bloque, dado que los bloques parten de la posición 0.

Segundo, al ser el segundo bloque, se debe conseguir la posición del último 1 del bloque

Data: i
if $i \geq u$ **then**
  return n ;
end
$blk \leftarrow rank\_L(i)$;
if $blk \geq cantidad\_de\_bloques$ **then**
  return n ;
end
$rank\_ret \leftarrow 0$;
if $blk > 0$ **then**
  $rank\_ret \leftarrow rank\_ret + select\_E(blk)$;
end
if $B[blk]$ **then**
  if $blk == 0$ **then**
    $rank\_ret \leftarrow rank\_ret + block\_rank[blk](i)$;
  else
    $i\_block \leftarrow i - 1 - select\_L(blk)$;
    $rank\_ret \leftarrow rank\_ret + block\_rank[blk](i\_block)$;
  end
else
  if *P[blk]* **then**
    if $blk == 0$ **then**
      $rank\_ret \leftarrow rank\_ret + block\_rank[blk](i)$;
    else
      $i\_block \leftarrow i - 1 - select\_L(blk)$;
      $rank\_ret \leftarrow rank\_ret + block\_rank[blk](i\_block)$;
    end
  else
    if *blk == 0* **then**
      $rank\_ret \leftarrow rank\_ret + i$;
    else
      $rank\_ret \leftarrow rank\_ret + i - 1 - select\_L(blk)$;
    end
  end
end
return $rank\_ret$

Algoritmo 2: Operación *rank* en *PEF ϵ-opt*

Data: i
$blk \leftarrow (i-1)/b$;
if $B[blk]$ **then**
  if $blk == 0$ **then**
    return $block\_select[blk](i)$;
  else
    return $select\_L(blk) + 1 + block\_select[blk]((i-1)\,\%b + 1)$;
  end
else
  if *P[blk]* **then**
    if $blk == 0$ **then**
      return $block\_select[blk](i)$;
    else
      return $select\_L(blk) + 1 + block\_select[blk]((i-1)\,\%b + 1)$;
    end
  else
    if *blk == 0* **then**
      return $i - 1$;
    else
      return $select\_L(blk) + (i-1)\,\%b + 1$;
    end
  end
end

Algoritmo 3: Operación *select* en *PEF unif*

anterior, para lo que se aplica la operación *select* sobre L con el valor blk, obteniendo 7.

Tercero, se debe obtener la posición del 1 dentro del bloque. Para esto, se aplica la operación *select* sobre el bloque con $(6-1)\,\%4+1=2$, entregando la posición 3 debido a que el sexto 1 corresponde al segundo 1 dentro del segundo bloque.

Finalmente, se debe sumar los resultados obtenidos en el segundo y tercer paso más 1, $7+3+1=11$, por lo tanto la posición del sexto 1 es 11.

3.3.2. *PEF ϵ-opt*

La operación *select* recibe i, indicando que se consulta sobre el i-ésimo 1, entre 1 y n, donde n es la cantidad de 1's en el arreglo de bits original. Para obtener la posición del i-ésimo 1. Se debe realizar los siguiente pasos.

Primero, se debe determinar en qué bloque se encuentra el i-ésimo 1 (blk). Para esto se realiza la operación *rank* sobre E con el valor i, debido a que no todos los bloques tienen la misma cantidad de 1's.

Segundo, al saber en qué bloque se encuentra, si corresponde al primero, entonces se debe retornar la operación *select* sobre este. En caso contrario, se debe determinar a qué 1 corresponde dentro de él. Para ello, se resta a i el resultado de aplicar *select* sobre E con el valor blk, esta posición se denominará i_bloque.

Luego, se debe aplicar la operación *select* dentro del bloque dado con el valor i_bloque. Con el fin de obtener la posición dentro de él (pos_bloque). Junto a eso, se debe saber cuál es la posición del último 1 del bloque anterior ($ultimo\_elemento$). Para obtenerlo, hay que realizar la operación *select* sobre L con el valor blk. Finalmente, el resultado de la operación *select* corresponde a la suma $pos\_bloque + ultimo\_elemento + 1$.

Por ejemplo, si se tiene los bloques de la tabla 5, el arreglo L de la tabla 6 y el arreglo E de la tabla 7, se quiere determinar la posición del sexto 1.

Primero, se debe obtener el bloque en que se encuentra el sexto 1, realizando *rank* sobre E con el valor 6. Esto entregará 1, al igual que en *PEF unif*, cuyo valor corresponderá al segundo bloque.

Segundo, al ser el segundo bloque, se debe encontrar la posición del último 1 del bloque anterior, para eso se aplica la operación *select* sobre L con el valor blk, obteniendo 9.

Tercero, se debe determinar a qué 1 corresponde dentro del bloque. Para ello, al valor 6 se le debe restar el valor que entregue *select* sobre E con el valor blk, dando como resultado $6-5=1$, por lo que corresponde al primer 1.

```
Data: i
blk ← rank_E(i);
if blk ≥ cantidad_de_bloques then
    return n ;
end
if B[blk] then
    if blk == 0 then
        return block_select[blk](i);
    else
        rank_block ← i − select_E(blk);
        return select_L(blk) + 1 + block_select[blk](rank_block);
    end
else
    if P[blk] then
        if blk == 0 then
            return block_select[blk](i);
        else
            rank_block ← i − select_E(blk);
            return select_L(blk) + 1 + block_select[blk](rank_block);
        end
    else
        if blk == 0 then
            return i − 1;
        else
            rank_block ← i − select_E(blk);
            return select_L(blk) + rank_block;
        end
    end
end
```

Algoritmo 4: Operación *select* en *PEF* ϵ*-opt*

Cuarto, se debe encontrar la posición del 1 dentro del bloque, por lo tanto, se aplica la operación *select* sobre el bloque con el valor obtenido en el tercer paso, dando como resultado la posición 1.

Finalmente, se debe sumar los resultados obtenidos en el segundo y cuarto paso más 1, $9 + 1 + 1 = 11$, por lo tanto la posición del sexto 1 es 11.

3.4. Implementaciones de *Partioned Elias-Fano*

Para la representación de *Elias-Fano* se realizaron dos implementaciones: una con *sd_vector* de la librería *sdsl* [Gog y et al., 2019] y otra con *EliasFano* de la librería *sux*[Vigna, 2008].

Para la representación de un *bit_vector* plano se utilizó la implementación de la librería *sdsl*, se hará uso de la clase *select_support_mcl* para poder realizar *select* y se utilizará *rank_support_v* o *rank_support_v5* para poder realizar *rank*[Gog y et al., 2019].

Debido a esto, se generaron las siguientes implementaciones de *PEF*:

- PEF unif + SD + rank v: utilizando *PEF unif* con *sd_vector* y *bit_vector* con *rank_support_v*
- PEF unif + SD + rank v5: utilizando *PEF unif* con *sd_vector* y *bit_vector* con *rank_support_v5*
- PEF unif + Vigna + rank v: utilizando *PEF unif* con *EliasFano* y *bit_vector* con *rank_support_v*
- PEF unif + Vigna + rank v5: utilizando *PEF unif* con *EliasFano* y *bit_vector* con *rank_support_v5*
- PEF opt + SD + rank v: utilizando *PEF opt* con *sd_vector* y *bit_vector* con *rank_support_v*
- PEF opt + SD + rank v5: utilizando *PEF opt* con *sd_vector* y *bit_vector* con *rank_support_v5*
- PEF opt + Vigna + rank v: utilizando *PEF opt* con *EliasFano* y *bit_vector* con *rank_support_v*
- PEF opt + Vigna + rank v5: utilizando *PEF opt* con *EliasFano* y *bit_vector* con *rank_support_v5*

3.5. Funciones para calcular espacio de representaciones

Debido a que *PEF ϵ-opt* pregunta qué representación ocupa menos espacio para obtener las particiones óptimas, se deben generar funciones que permitirían calcular el espacio ocupado en bits para cada una de las representaciones dado dos parámetros u y n, el universo y la cantidad de 1's dentro de la secuencia de bits.

3.5.1. *rank_support_v*

En el caso que se utilice la clase *rank_support_v*, se debe calcular el espacio ocupado por un *int_vector* que almacena palabras de largo 64 y el espacio ocupado por un *int_vector* que almacena palabras de largo $\lceil \log(n) \rceil$. La cantidad de elementos que ocupa cada uno de estos *int_vector* corresponde a $size = (\frac{(u+63)}{2^6} \cdot 2^6) \cdot \frac{1}{2^9}) + 1$. Por lo tanto el espacio total ocupado por esta clase es $size \cdot 64 + size \cdot \lceil \log(n) \rceil$ bits.

3.5.2. *rank_support_v5*

En el caso que se utilice la clase *rank_support_v5*, corresponde al mismo calculo de la clase *rank_support_v*, pero con diferencia en el valor de $size$, donde para esta clase corresponde a $size = (\frac{(u+63)}{2^6} \cdot 2^6) \cdot \frac{1}{2^1 1}) + 1$ bits.

3.5.3. *select_support_mcl*

En el caso que se utilice la clase *select_support_mcl*, para calcular su espacio se debe calcular la cantidad de *super_blocks* que genera, donde este corresponde a un *int_vector* que almacena palabras de largo $\lceil \log(u) \rceil$. La cantidad de *super_blocks* corresponde a $size = \frac{n+4096-1}{2^1 2}$. Al espacio de esto último, se le debe sumar 8 enteros de 64 bytes que utiliza esta clase. Por lo tanto el espacio total ocupado por esta clase es $size \cdot \lceil \log(u) \rceil + 64 \cdot 8$ bits.

3.5.4. bit_vector plano de la *sdsl*

En el caso del *bit_vector* plano de la *sdsl*, el espacio utilizado por solo el *bit_vector* plano consiste de $64 \cdot (\frac{u}{64} + 1)$ bits, a esto último se le debe sumar el espacio de las clases *select_support_mcl* y *rank_support* (v o v5, dependiendo de cual fue elegida).

3.5.5. *SD vector*

En el caso del *SD vector* de la *sdsl*, el espacio total utilizado corresponde al espacio utilizado por un *bit_vector* plano, un *int_vector* y dos *select_support_mcl*.

El *bit_vector* tiene un tamaño de $size = n + 2^{(\lceil \log(n) \rceil)}$, por lo tanto su espacio equivale a $64 \cdot (\frac{size}{64} + 1)$ bits.

El *int_vector* tiene un tamaño de n elementos y lo almacena en palabras de largo $\lceil \log(u) \rceil - \lceil \log(n) \rceil$, por lo tanto su espacio equivale a $n \cdot (\lceil \log(u) \rceil - \lceil \log(n) \rceil)$ bits.

Un *select_support_mcl* utiliza el espacio utilizado descrito previamente con u y n, igual al del arreglo de bits original. El segundo *select_support_mcl* utiliza el espacio utilizado descrito previamente con u y $u - n$.

Finalmente, el espacio total corresponde a la suma de lo descrito previamente.

3.5.6. *Elias-Fano* de *sux*

En el caso del *Elias-Fano* de *sux*, el espacio total utilizado corresponde al espacio utilizado por dos objetos de la clase *vector* y dos objetos de clases *select*, todas clases de la misma librería.

Un objeto de la clase *vector* corresponde a los *lower* bits de la representación *Elias-Fano*, el espacio de este objeto corresponde a $(\frac{n \cdot (\lfloor \log(\frac{u}{n}) \rfloor + 63)}{64}) + 2) \cdot 64 \cdot 8$ bits.

El otro objeto de la clase *vector* corresponde a los *upper* bits de la representación *Elias-Fano*, el espacio de este objeto equivale a $(n + \frac{u}{\lfloor \log(\frac{u}{n}) \rfloor} + 1 + 63)/64 \cdot 64 \cdot 8$ bits.

Un objeto de la clase *select* utiliza un espacio correspondiente a $(\frac{(n+2^{10}-1)}{2^{10}} \cdot (2^2 + 1) + 1) \cdot 64 \cdot 8$ bits. Mientras que el otro objeto de la clase *select* utiliza un espacio correspondiente a $((n + \frac{u}{\lfloor \log(\frac{u}{n}) \rfloor} - n + 2^{10} - 1)/2^{10} \cdot (2^2 + 1) + 1) \cdot 64 \cdot 8$ bits.

CAPÍTULO 4
VALIDACIÓN DE LA SOLUCIÓN

4.1. Parámetro *fixed cost* en cada implementación de *PEF opt*

Para todas las implementaciones de *PEF opt* se utilizará los siguientes valores para el parámetro *fixed cost*: 64, 128, 256, 512, 1024 y 2048.

4.2. Estructuras a comparar con *PEF*

Para todos los experimentos descritos posteriormente, se realizaron con las siguientes estructuras para poder compararlas con *PEF*:

- *hyb_vector*
- *sd_vector*
- *rrr_vector<b>*, donde b tendrá lo siguientes valores: 15, 31, 63 y 127
- *s18_vector<b>*, donde b tendrá los siguientes valores: 8, 16, 32, 64, 128 y 256
- *rle_vector<b>*, donde b tendrá los siguientes valores: 32, 64, 128 y 256

4.3. Datos experimentales

Para la realización de los experimentos se utilizó el dataset llamado *IBDA-sorted Gov2 inverted index* [Arroyuelo *et al.*, 2018], este consiste de 7814 listas invertidas, cada una de al menos 100000 elementos. Se representó cada una de las listas como un *bit vector* de 25138630 bits, los cuales corresponden a l números de documentos en la colección *Gov2*, se marcará con 1 los bits correspondientes a la id del documento en la lista invertida. Luego, se concatenan todos los *bit vector* resultados construyendo así un *bit vector* grande de universo 196433254820 y 5055078461 1's.

4.4. Experimentos

4.4.1. Espacio utilizado

Para poder observar el espacio utilizado de las estructuras, se utilizó los datos anteriormente mencionado, al tener la estructura generada se obtuvo el espacio utilizado por esta y se divide por la cantidad de elementos dentro del *bit vector*, en este caso 5055078461.

En el caso de las estructuras de *PEF* para calcular el espacio se harán uso las funciones descritas en la sección $3{,}5$. En el caso del resto de estructuras se hará uso de la función que su respectiva librería otorgue.

4.4.2. Operación *rank*

En el caso de la operación *select* se realizaron 1000000 operaciones con números aleatorios entre 0 y 196433254820, se obtuvo el tiempo total de las 1000000 operaciones, de esta forma poder obtener el tiempo promedio de la operación.

4.4.3. Operación *select*

En el caso de la operación *select* se realizaron 1000000 operaciones con números aleatorios entre 0 y 5055078460, se obtuvo el tiempo total de las 1000000 operaciones, de esta forma poder obtener el tiempo promedio de la operación.

4.5. Entorno de ejecución y compilación

Para la realización de los experimentos, se utilizó un servidor con: una CPU de modelo *Intel(R) Xeon(R) CPU ES-2630 0 @ 2.30GHz*, la versión 4.4.0 de Linux,

Los experimentos llevados a cabo para este trabajo se realizaron en el siguiente entorno de ejecución:

- Procesador: *Intel(R) Xeon(R) CPU ES-2630 0 @ 2.30GHz*
- SO: Linux 4.4.0

Al momento de compilar las pruebas se utilizó la versión 8.4.0 de *g++* y las siguientes *flags* al momento de compilar las pruebas: *-std=c++17*, *-msse4.2* y *-O5*.

4.6. Resultados

4.6.1. Espacio utilizado por *PEF*

En los experimentos realizados, se puede observar según las tablas 9, 13, 17, 25 y 29, que siempre la representación *Elias-Fano* fue la más elegida por sobre el resto y que entre más grande el valor *fixed cost* menor la cantidad de bloques generados en su totalidad. Esto se debe a que a mayor *fixed cost* el algoritmo descrito previamente permite una generación de bloques con un universo más grande. Agregando a lo anterior, las tablas 10, 14, 18, 22, 26 y 30 presentan que entre menor el universo de los bloques generado, no implica un menor uso de espacio, sino todo lo contrario, dónde el mayor espacio ocupado es con los bloques de universo más pequeños.

También, se puede apreciar en las tablas 8, 12, 16, 20, 24 y 28 que el espacio de los arreglos L, E y B no varía notablemente entre diferentes valores de *fixed cost*, pero si entre utilización de la implementación de *Elias-Fano*. Dónde, la implementación de *Vigna* obtiene mejores resultados.

4.6.2. Operación Rank

La figura 3 corresponde a las pruebas para la operación rank. En este gráfico se puede observar que *PEF opt + Vigna + rank v* y *PEF opt + Vigna + rank v5* presenta mejores resultados que *PEF opt + SD + rank v* y *PEF opt + SD + rank v5*, obteniendo un mejor tiempo promedio con un similar uso de espacio. Pero, todas las implementaciones de *PEF* presentan peores resultados en tiempo que el resto de estructuras, exceptuando la estructura *RLE*, dónde solo *PEF opt + Vigna + rank v* y *PEF opt + Vigna + rank v5* presentan mejores resultados que esta última.

Por último, la estructura con mejor resultado, relacionando el uso de espacio con el tiempo promedio, corresponde a *S18* y *RRR*. Estas estructuras presentan un espacio utilizado similar y con tiempos notablemente mejores al resto. Ahora bien, la estructura *SD* obtuvo buen resultado en tiempo, pero su espacio utilizado fue grande en comparación y lo mismo ocurre con la estructura *HYB*.

4.6.3. Operación select

La figura 4 corresponde a las pruebas para la operación *select*. En este gráfico, al igual que la operación *select*, se puede observar que *PEF opt + Vigna + rank v* y *PEF opt + Vigna + rank v5* presenta mejores resultados que *PEF opt + SD + rank v* y *PEF opt + SD + rank v5*, obteniendo un mejor tiempo promedio con un similar uso de espacio. A diferencia de la operación *rank*, las

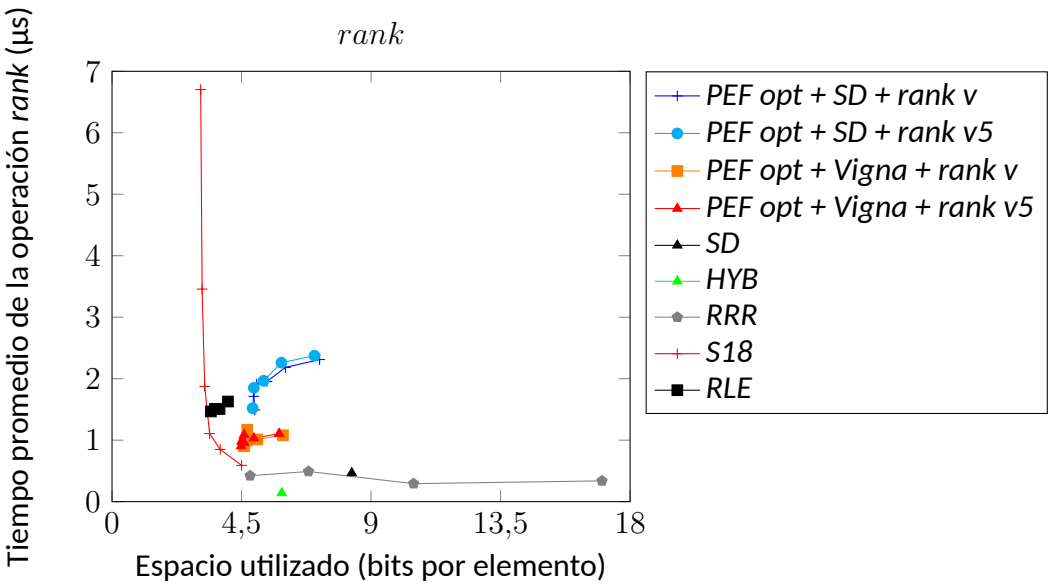


Figura 3: Tiempo promedio de la operación *rank* en función del espacio utilizado (bits por elemento)

implementaciones de *PEF* presentaron mejores resultados que *RRR*. Las implementaciones de *PEF* que hacen uso de Vigna obtuvieron mejores resultados en tiempo que *RLE*, aunque este último consiguió un mejor uso de espacio.

Por último, la estructura con mejor resultado, relacionando el uso de espacio con el tiempo promedio, corresponde a *S18*. Esta estructura presenta un espacio utilizado similar y con tiempos notablemente mejores al resto. Ahora bien, la estructura *SD* obtuvo buen resultado en tiempo, pero su espacio utilizado fue grande en comparación.

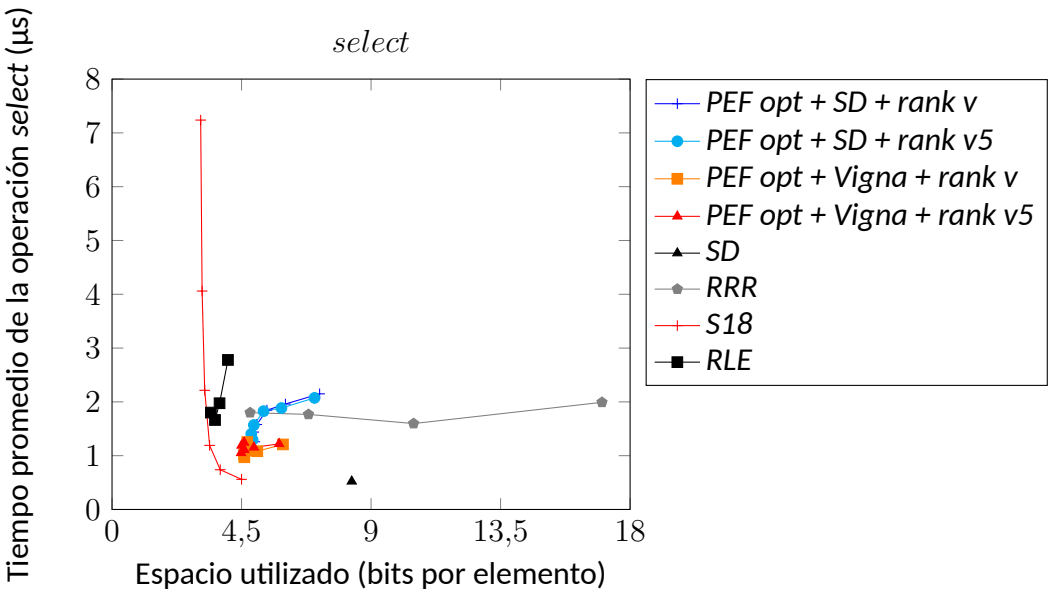


Figura 4: Tiempo promedio de la operación *select* en función del espacio utilizado (bits por elemento)

CAPÍTULO 5
CONCLUSIONES

Este trabajo logra la implementación de un algoritmo aproximado para una compresión de dos niveles que utiliza *PEF* obteniendo una respuesta a lo más $1+\epsilon$ más grande que la solución óptima. Este algoritmo presenta resultados mejores en espacio que otras estructuras. Igualmente, se puede observar que el método de elección de bloques y como se calculan los espacios es importante para poder obtener resultados notables.

Sobre las implementaciones de *PEF* ϵ*-opt*, se pudo observar que cuando esta generaba bloques de universos pequeños era contraproducente, obteniendo un uso de espacio mayor que cuando permitía bloques de universos más grandes. También, se puede concluir que en la mayoría de casos se va a elegir utilizar la representación de *Elias-Fano* por sobre la representación de un *bit_vector* plano. Además, existe una diferencia notable entre las implementaciones que utilizaron la representación de *Elias-Fano* de Vigna y de la librería *sdsl*, donde la representación de Vigna fue superior en tiempo promedio de las operaciones *rank* y *select*.

Con respecto a la operación *rank* implementada, se puede concluir que *PEF* es ineficiente para esta operación con respecto a las otras estructuras, debido a que el tiempo empleado es mayor en comparación al resto. Esto último se debe a la complejidad de poder calcular la respuesta, debido a que debe hacer múltiples operaciones *rank* y *select* sobre los arreglos L

y E.

Con respecto a la operación *select* implementada, se puede concluir que *PEF* es más competitivo con respecto a la operación en otras estructuras. Pero, de todas formas fue superado por la estructura *s18_vector*.

Por último, se puede concluir que *PEF* consiste en una estructura con un algoritmo que si bien puede presentar buenos resultados al momento de comprimir, si es que se implementan dos de las operaciones clásicas, como son *rank* y *select*, presenta malos resultados esto debido a la complejidad que consiste en poder determinar el resultado, ya que se debe hacer varias consultas en los arreglos respectivos de la *PEF*.

Para un trabajo futuro, se podría probar que sucede si es que en vez de generar arreglos de 1's para los bloques, se utilizarán listas, esto permitiría disminuir la cantidad de 0's que existe entre el último 1 de un bloque con el primer 1 del siguiente bloque. De esta forma, se estaría ahorrando una cantidad notable de 0's los cuales podrían mejorar los resultados de espacio.

REFERENCIAS BIBLIOGRÁFICAS

[Arroyuelo *et al.*, 2018] Arroyuelo, D., Oyarzún, M., González, S., y Sepulveda, V. (2018). Hybrid compression of inverted lists for reordered document collections.

[Bauer y Wössner, 1972] Bauer, F. y Wössner, H. (1972). The “plankalkül” of konrad zuse: A forerunner of today’s programming language.

[Buchsbaum *et al.*, 2003] Buchsbaum, A., Fowler, G., y Giancarlo, R. (2003). Improving table compression with combinatorial optimization.

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs.

[Elias, 1974] Elias, P. (1974). Etticient storage and retrieval by content and address of static files.

[Fano, 1972] Fano, R. M. (1972). On binary representations of monotone sequences.

[Gog y et al., 2019] Gog, S. y et al. (2019). Succinct data structure library. `https://github.com/simongog/sdsl-lite`.

[Grossi y Ottaviano, 2013] Grossi, R. y Ottaviano, G. (2013). Design of practical succinct data structures for large data collections. pp. 5–7.

[Knuth, 1974] Knuth, D. E. (1974). Postscript about np-hard problems. 6(2):15–16.

[Ottaviano, 2014] Ottaviano, G. (2014). Partitioned elias-fano. `https://github.com/ot/partitioned_elias_fano`.

[Ottaviano y Venturini, 2014] Ottaviano, G. y Venturini, R. (2014). Partitioned elias-fano indexes.

[Ottaviano y Venturini, 2017] Ottaviano, G. y Venturini, R. (2017). Clustered elias-fano indexes.

[Skiene, 2008] Skiene, S. (2008). The algorithm design manual. pp. 179–180.

[Touati y Dupont, 2014] Touati, S. y Dupont, B. (2014). Advanced backend optimization. p. 123.

[Vazirani, 2001] Vazirani, V. (2001). Approximation algorithms. Springer.

[Venturini y Ermanno, 2017] Venturini, R. y Ermanno, G. (2017). Dynamic Elias-Fano Representation.

[Vigna, 2008] Vigna, S. (2008). Broadword implementation of rank/select queries. pp. 154–168.

[Wenyu y Youli, 2017] Wenyu, G. y Youli, Q. (2017). Compressed suffix arrays to self-indexes based on partitioned elias-fano.

ANEXOS

Estructura	L	E	B
PEF opt + SD + rank v	0.40573500	0.03967140	0.00272321
PEF opt + SD + rank v5	0.40450200	0.03882330	0.00264612
PEF opt + Vigna + rank v	0.04788410	0.03223220	0.00291062
PEF opt + Vigna + rank v5	0.04693480	0.03161450	0.00284429

Tabla 8: Espacio usado en bits por los arreglos L. E y B dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 64

Estructura	# *Elias-Fano*	# *bit_vector*	# *runs* de 1's
PEF opt + SD + rank v	10784677	2658821	322516
PEF opt + SD + rank v5	10465616	2604907	305757
PEF opt + Vigna + rank v	11964660	2476245	272482
PEF opt + Vigna + rank v5	11608178	2511560	258371

Tabla 9: La cantidad de bloques para cada representación posible en la estructura generada utilizando *Gov2* con un *fixed cost* equivalente a 64

Estructura	*Elias-Fano*	*bit_vector*
PEF opt + SD + rank v	5.52250000	1.24179000
PEF opt + SD + rank v5	5.31498000	1.27407000
PEF opt + Vigna + rank v	4.56755000	1.28745000
PEF opt + Vigna + rank v5	4.37401000	1.35017000

Tabla 10: Espacio utilizado en bits por los bloques que utilicen la representación *Elias-Fano* y los bloques que utilicen la representación de un *bit_vector* plano dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 64

Estructura	Espacio Total	Rank promedio	Select promedio
PEF opt + SD + rank v	7.21242000	2.3114400	2.1493100
PEF opt + SD + rank v5	7.03502000	2.3738200	2.0740900
PEF opt + Vigna + rank v	5.93803000	1.0800500	1.2090400
PEF opt + Vigna + rank v5	5.80558000	1.1054500	1.2192700

Tabla 11: Espacio total en bits de la estructura dividido por la cantidad de 1's dentro del vector de bits de *Gov2* y los tiempos promedio de la operación rank y select utilizando un *fixed cost* equivalente a 64

Estructura	L	E	B
PEF opt + SD + rank v	0.38676900	0.02663240	0.00153785
PEF opt + SD + rank v5	0.38686700	0.02690940	0.00150192
PEF opt + Vigna + rank v	0.02817560	0.01944240	0.00163674
PEF opt + Vigna + rank v5	0.02765510	0.01909190	0.00160275

Tabla 12: Espacio usado en bits por los arreglos L. E y B dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 128

Estructura	# *Elias-Fano*	# *bit_vector*	# *runs* de 1's
PEF opt + SD + rank v	6142318	1432264	199313
PEF opt + SD + rank v5	5902358	1500060	189900
PEF opt + Vigna + rank v	6467092	1617668	189090
PEF opt + Vigna + rank v5	6276254	1647457	178292

Tabla 13: La cantidad de bloques para cada representación posible en la estructura generada utilizando *Gov2* con un *fixed cost* equivalente a 128

Estructura	*Elias-Fano*	*bit_vector*
PEF opt + SD + rank v	4.56914000	1.04078000
PEF opt + SD + rank v5	4.37188000	1.09792000
PEF opt + Vigna + rank v	3.80631000	1.19121000
PEF opt + Vigna + rank v5	3.63166000	1.24965000

Tabla 14: Espacio utilizado en bits por los bloques que utilicen la representación *Elias-Fano* y los bloques que utilicen la representación de un *bit_vector* plano dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 128

Estructura	Espacio Total	Rank promedio	Select promedio
PEF opt + SD + rank v	6.02487000	2.1836000	1.9601100
PEF opt + SD + rank v5	5.88508000	2.2606900	1.8849700
PEF opt + Vigna + rank v	5.04678000	1.0118200	1.0829200
PEF opt + Vigna + rank v5	4.92966000	1.0320300	1.1544200

Tabla 15: Espacio total en bits de la estructura dividido por la cantidad de 1's dentro del vector de bits de *Gov2* y los tiempos promedio de la operación rank y select utilizando un *fixed cost* equivalente a 128

Estructura	L	E	B
PEF opt + SD + rank v	0.37651300	0.01958080	0.00089680
PEF opt + SD + rank v5	0.37616800	0.01934360	0.00087524
PEF opt + Vigna + rank v	0.01651810	0.01165740	0.00091719
PEF opt + Vigna + rank v5	0.01632290	0.01152210	0.00090523

Tabla 16: Espacio usado en bits por los arreglos L. E y B dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 256

Estructura	# *Elias-Fano*	# *bit_vector*	# *runs* de 1's
PEF opt + SD + rank v	3614207	794338	124796
PEF opt + SD + rank v5	3454710	851550	118065
PEF opt + Vigna + rank v	3526667	982171	127589
PEF opt + Vigna + rank v5	3430162	1027647	118140

Tabla 17: La cantidad de bloques para cada representación posible en la estructura generada utilizando *Gov2* con un *fixed cost* equivalente a 256

Estructura	*Elias-Fano*	*bit_vector*
PEF opt + SD + rank v	4.07294000	0.91179600
PEF opt + SD + rank v5	3.89086000	0.97715300
PEF opt + Vigna + rank v	3.54060000	1.10535000
PEF opt + Vigna + rank v5	3.39655000	1.16701000

Tabla 18: Espacio utilizado en bits por los bloques que utilicen la representación *Elias-Fano* y los bloques que utilicen la representación de un *bit_vector* plano dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 256

Estructura	Espacio Total	Rank promedio	Select promedio
PEF opt + SD + rank v	5.38173000	1.9546600	1.8457400
PEF opt + SD + rank v5	5.26440000	1.9662600	1.8274500
PEF opt + Vigna + rank v	4.69851000	1.1701800	1.2521900
PEF opt + Vigna + rank v5	4.59231000	0.9592760	1.1097700

Tabla 19: Espacio total en bits de la estructura dividido por la cantidad de 1's dentro del vector de bits de *Gov2* y los tiempos promedio de la operación rank y select utilizando un *fixed cost* equivalente a 256

Estructura	L	E	B
PEF opt + SD + rank v	0.37007400	0.01505560	0.00051411
PEF opt + SD + rank v5	0.36995200	0.01499470	0.00050179
PEF opt + Vigna + rank v	0.00956724	0.00702474	0.00050766
PEF opt + Vigna + rank v5	0.00950109	0.00698151	0.00050385

Tabla 20: Espacio usado en bits por los arreglos L. E y B dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 512

Estructura	# *Elias-Fano*	# *bit_vector*	# *runs* de 1's
PEF opt + SD + rank v	2070368	456360	72093
PEF opt + SD + rank v5	2013419	458463	64676
PEF opt + Vigna + rank v	1939215	552051	74991
PEF opt + Vigna + rank v5	1890563	587290	69093

Tabla 21: La cantidad de bloques para cada representación posible en la estructura generada utilizando *Gov2* con un *fixed cost* equivalente a 512

Estructura	*Elias-Fano*	*bit_vector*
PEF opt + SD + rank v	3.79139000	0.85130800
PEF opt + SD + rank v5	3.64764000	0.89494400
PEF opt + Vigna + rank v	3.53162000	1.03709000
PEF opt + Vigna + rank v5	3.36866000	1.09880000

Tabla 22: Espacio utilizado en bits por los bloques que utilicen la representación *Elias-Fano* y los bloques que utilicen la representación de un *bit_vector* plano dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 512

Estructura	Espacio Total	Rank promedio	Select promedio
PEF opt + SD + rank v	5.02834000	1.9146700	1.5789100
PEF opt + SD + rank v5	4.92803000	1.8509600	1.5702900
PEF opt + Vigna + rank v	4.58581000	0.9025480	1.0215000
PEF opt + Vigna + rank v5	4.48444000	0.9066240	1.0513800

Tabla 23: Espacio total en bits de la estructura dividido por la cantidad de 1's dentro del vector de bits de *Gov2* y los tiempos promedio de la operación rank y select utilizando un *fixed cost* equivalente a 512

Estructura	L	E	B
PEF opt + SD + rank v	0.36611600	0.01224630	0.00028444
PEF opt + SD + rank v5	0.36594900	0.01210410	0.00027943
PEF opt + Vigna + rank v	0.00545117	0.00404441	0.00027643
PEF opt + Vigna + rank v5	0.00542758	0.00402854	0.00027514

Tabla 24: Espacio usado en bits por los arreglos L. E y B dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 1024

Estructura	# *Elias-Fano*	# *bit_vector*	# *runs* de 1's
PEF opt + SD + rank v	1191938	210574	35352
PEF opt + SD + rank v5	1135435	242836	34248
PEF opt + Vigna + rank v	1063514	292575	41242
PEF opt + Vigna + rank v5	1038065	313254	39492

Tabla 25: La cantidad de bloques para cada representación posible en la estructura generada utilizando *Gov2* con un *fixed cost* equivalente a 1024

Estructura	*Elias-Fano*	*bit_vector*
PEF opt + SD + rank v	3.78612000	0.76015100
PEF opt + SD + rank v5	3.59029000	0.86862500
PEF opt + Vigna + rank v	3.60823000	0.97912000
PEF opt + Vigna + rank v5	3.45194000	1.03900000

Tabla 26: Espacio utilizado en bits por los bloques que utilicen la representación *Elias-Fano* y los bloques que utilicen la representación de un *bit_vector* plano dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 1024

Estructura	Espacio Total	Rank promedio	Select promedio
PEF opt + SD + rank v	4.92492000	1.7128200	1.4384400
PEF opt + SD + rank v5	4.83725000	1.6419400	1.4044600
PEF opt + Vigna + rank v	4.59712000	0.9377780	0.9710900
PEF opt + Vigna + rank v5	4.50067000	0.9776780	1.1900000

Tabla 27: Espacio total en bits de la estructura dividido por la cantidad de 1's dentro del vector de bits de *Gov2* y los tiempos promedio de la operación rank y select utilizando un *fixed cost* equivalente a 1024

Estructura	L	E	B
PEF opt + SD + rank v	0.36366900	0.01044320	0.00015565
PEF opt + SD + rank v5	0.36367500	0.01046060	0.00015336
PEF opt + Vigna + rank v	0.00305399	0.00229156	0.00014807
PEF opt + Vigna + rank v5	0.00310540	0.00229511	0.00014833

Tabla 28: Espacio usado en bits por los arreglos L. E y B dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 2048

Estructura	# *Elias-Fano*	# *bit_vector*	# *runs* de 1's
PEF opt + SD + rank v	663939	105151	17725
PEF opt + SD + rank v5	634359	123033	17778
PEF opt + Vigna + rank v	576031	148208	34208
PEF opt + Vigna + rank v5	564061	162115	23607

Tabla 29: La cantidad de bloques para cada representación posible en la estructura generada utilizando *Gov2* con un *fixed cost* equivalente a 2048

Estructura	*Elias-Fano*	*bit_vector*
PEF opt + SD + rank v	3.87573000	0.70977600
PEF opt + SD + rank v5	3.68429000	0.82652700
PEF opt + Vigna + rank v	3.74394000	0.93147800
PEF opt + Vigna + rank v5	3.58970000	0.99378000

Tabla 30: Espacio utilizado en bits por los bloques que utilicen la representación *Elias-Fano* y los bloques que utilicen la representación de un *bit_vector* plano dividido por la cantidad de 1's dentro del vector de bits de *Gov2* utilizando un *fixed cost* equivalente a 2048

Estructura	Espacio Total	Rank promedio	Select promedio
PEF opt + SD + rank v	4.95977000	1.4924000	1.2585200
PEF opt + SD + rank v5	4.88510000	1.5203500	1.2935500
PEF opt + Vigna + rank v	4.68091000	0.9955730	1.0938600
PEF opt + Vigna + rank v5	4.58903000	1.0882900	1.2456900

Tabla 31: Espacio total en bits de la estructura dividido por la cantidad de 1's dentro del vector de bits de *Gov2* y los tiempos promedio de la operación rank y select utilizando un *fixed cost* equivalente a 2048