Sparse Set

1st Diego Quispe
Computer Science
UTEC
Lima, Perú
diego.quispe.ap@utec.edu.pe

2nd Christian Frisancho
Data Science
UTEC
Lima, Perú
christian.frisancho@utec.edu.pe

3rd Camilo Soto
Data Science
UTEC
Lima, Perú
camilo.soto@utec.edu.pe

***Abstract*—This study examines the representation of sets, focusing on the sparse set as a highly efficient data structure for operations such as insertion, deletion, and search, all performed in constant time. The project aims to evaluate the sparse set's efficiency against other structures like bit vectors and hash tables by formal and empirical analysis. Sparse sets provide significant advantages by performing operations independently of the number of stored elements, often surpassing alternatives in applications with a large universe of elements but a small set of active members. The comparative analysis, implemented in C++, confirms the sparse set's superior performance, though it acknowledges potential drawbacks in complex scenarios.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introducción

El presente trabajo se refiere a la representación de un set; más específicamente, nos centraremos en el sparse set. Esta estructura es de gran importancia para representar un conjunto de valores de manera eficiente. Su valor radica en que operaciones como la inserción, eliminación, búsqueda y clear son realizadas con una complejidad constante.

Dada la existencia de diversas estructuras de datos que pueden representar un set, es de suma importancia elegir la más eficiente y conveniente según el proyecto en que se utilice. Por ello, es necesario realizar un análisis comparativo entre el sparse set y otras estructuras de propósito similar.

Es así, que este trabajo tiene el fin de confirmar la eficiencia del sparse set frente a otras alternativas y adicionalmente, reproducir los resultados comparativos obtenidos por Briggs [1].

II. Objetivo del proyecto

El objetivo de este proyecto es determinar si el sparse set es superior en la eficiencia de sus operaciones respecto a otras representaciones del set. Las estructuras de datos con las que se realizará la comparación son: bit vector y hash table.

Este objetivo se logrará mediante la demostración formal de la complejidad espacial y computacional de las operaciones del sparse set. Asimismo, se realizará una demostración empírica de la eficiencia de las estructuras de datos. Para ello, se implementarán estas estructuras y se analizarán los tiempos de ejecución de sus operaciones a fin de compararlos entre sí.

Este análisis comparativo se basará principalmente en operaciones como buscar, añadir y eliminar un elemento, así como en operaciones entre sets como unión e intersección. De esta manera, se determinará si el sparse set es más eficiente que sus alternativas.

III. Fundamento teórico

A. Sparse Set

El sparse set es una estructura de datos diseñada como una alternativa a otras representaciones del set, tales como el bit vector, la hash table y el árbol binario balanceado, entre otros. Su ventaja radica en que sus operaciones de acceso a elementos son independientes de la cantidad de datos almacenados en el set. Es decir, la inserción y eliminación de elementos se realizan en O(1) [1]. Además, de acuerdo con Cox, el sparse set presenta una característica importante: la de tener valores no inicializados y que las operaciones no se vean afectadas [2]. Esta característica permite que la operación de limpieza/eliminación de todos los elementos se ejecute en tiempo constante, a diferencia de sus alternativas que requieren O(n) [1], [2].

De acuerdo con Manenko [4], los sparse sets son comúnmente utilizados en el patrón arquitectónico Entity Component System. Este es usado en videojuegos para representar los objetos que se pueden encontrar en los mundos. Estos objetos están compuestos por componentes de datos como el sparse set, lo que minimiza el uso de recursos entre los elementos del juego, ya que al existir un universo de datos en un juego, el sparse se limita a recorrer por los datos que pertenecen a cada usuario.

B. Propósito y alcance del algoritmo

Gracias al sparse set, se ha descrito una representación adecuada para conjuntos con un universo de tamaño fijo, que ofrece implementaciones en tiempo constante O(1) de operaciones como: pertenencia, insercion y eliminacion. En virtud de la eficiencia demostrada por estas operaciones, esta nueva representación frecuentemente supera a alternativas tales como vectores de bits, árboles binarios balanceados y tablas hash. Además, la nueva representación permite la enumeración de los miembros en tiempo lineal, lo que la convierte en una elección competitiva para conjuntos relativamente escasos que requieren operaciones como copia de conjunto, unión de conjuntos y diferencia de conjuntos.

C. Desventajas

Finalmente, aunque el sparse set es una estructura muy eficiente, también presenta algunas limitaciones y desventajas. Por ejemplo, su rendimiento podría verse afectado en conjuntos de datos muy grandes en los que se requiera una reorganización constante de los elementos. También hay que tomar en cuenta la complejidad de implementación y mantenimiento del sparse set, ya que puede ser más difícil en comparación con estructuras más establecidas como los vectores de bits. Sumado a ello tambien es una deventaja el espacio del que dispones, ya que comparado con el bit vector, el sparse set utiliza 2 veces el espacio en memoria que la otra estructura.

D. Restricciones de tipos de datos

El sparse set maneja datos numéricos enteros positivos o, como se puede ver en el código la implementación , el tipo de dato int. Esto se debe a que la eficiencia de sus operaciones radica en el uso de índices de arrays, que a su vez actúan como elementos del conjunto, de manera que solo es posible hacer con numeros.

E. Ejemplos de Uso

El Sparse Set gracias a su eficiencia al manejar grandes volúmenes de datos en los cuales abundan los elementos nulos es usado frecuentemente en diversas implementaciones varias donde se aprovecha al maximo el potencial de sus cualidades:

1) *Diccionarios para Datos Dispersos*: Una implementación práctica de los Sparse Sets es a través de diccionarios, donde las llaves representan las posiciones y los valores almacenan los elementos no nulos. Esta técnica es particularmente útil en aplicaciones como índices invertidos en motores de búsqueda, donde se manejan grandes cantidades de datos esparcidos a través de un amplio rango de índices.
2) *Listas de Coordenadas en Bases de Datos*: El método de listas de coordenadas consiste en almacenar una lista de tuplas, donde cada tupla contiene las coordenadas (índice de fila y columna) y el valor no nulo. Este enfoque es eficiente para gestionar matrices muy grandes con pocos valores no nulos, optimizando tanto el almacenamiento como la recuperación de datos.
3) *Aplicaciones en Videojuegos y Simulaciones*: En el desarrollo de videojuegos y simulaciones de mundos virtuales, los Sparse Sets son utilizados para manejar grandes mundos donde solo unas pocas ubicaciones contienen objetos relevantes. Esta técnica permite un uso más eficiente de la memoria y mejora el rendimiento del sistema al evitar el almacenamiento de grandes cantidades de datos nulos.

F. Estructura del Sparse Set

1) Construcción del sparse set: Para la representacion del sparse set, se tiene 4 componentes principales, 2 arrays de longitud u, donde u es la capacidad maxima de elementos del set y un entero que indica la cantidad de elementos que pertenecen al set.

- **Dense array**: Este es un vector que contiene los elementos actuales del conjunto.
- **Sparse array**: Este vector se utiliza para mapear los índices de los elementos que están en el Dense Vector, así como sus posiciones.
- **Count**: Un entero que registra el número de miembros en el conjunto, es decir, la cantidad de elementos activos en el Dense Vector.

2) Pertenece: Determina si un elemento pertenece al conjunto. Este chequeo es rápido porque utiliza acceso directo por índice. Para determinar si un elemento x esta incluido en un set, se accede al valor de sparse[x] verificando que este sea menor a count, luego se consulta si dense[sparse[x]] es igual a x, de ser asi, el elemento esta incluido. En la figura 1, si se quiere verificar que 5 esta incluido, se busca sparse[5] que tiene un valor de 0, ademas es menor a count (3), luego se consulta si dense[0] es 5, esto es cierto, por lo que 5 pertenece al set.

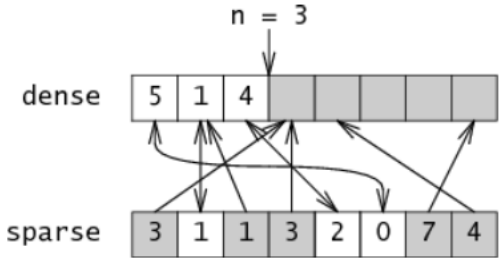

Fig. 1. Search in Sparse Set

3) Insert: Agregar un elemento al conjunto es eficiente porque se inserta en una posición conocida y actualiza dos arrays. Para la insercion de un elemento x, a sparse[x] se le da el valor de count, luego a dense[count] se le da el valor de x, por ultimo se incrementa count en 1. En la figura 2, se inserta el numero 4, por lo que sparse[4] se iguala a 4 dado que count vale tambien 4. Luego, dense[4] se iguala a 4 y por ultimo count toma el valor de 5.

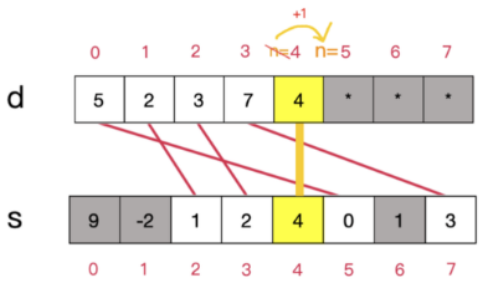

Fig. 2. Insert in Sparse Set

4) Delete: Eliminar un elemento es directo y rápido, similar a añadir, debido a la estructura de los arrays, ya que son posiciones conocidas. En este caso, primero se verifica que

el elemento pertenece al set, de ser asi, se intercambia el ultimo elemento por el elemento eliminado. En la figura 3, se elimina el 3 del dense y su referencia del sparse, en este caso 2. Luego, dense[2] toma el valor de su ultimo elemento, 4. Para completar las referencias entre arrays, sparse[4] toma el valor de la referencia de 3, es decir, 2. Por ultimo, count reduce en 1.

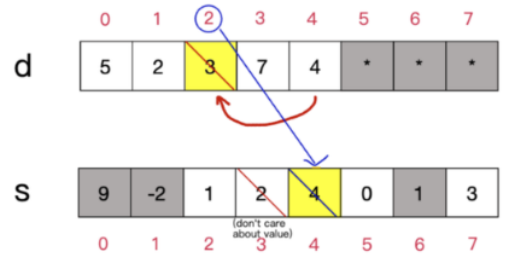

Fig. 3. Delete in Sparse Set

5) Clear: Limpiar el conjunto es rápido porque simplemente se cambia el valor de count a 0 gracias a la caracteristica de no inicializacion del sparse set[2]. En la Figura 4, se puede ver que no se requiere ninguna accion sobre los valores debido a que si se deja con valores basura, esto no afectara el funcionamiento de la estructura.

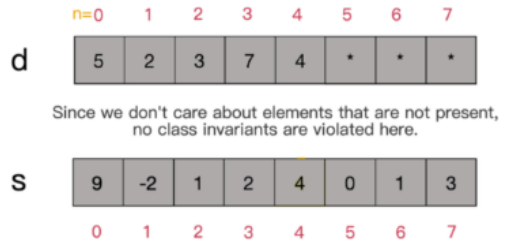

Fig. 4. Clear in Sparse Set

6) Operaciones de conjunto: Las operaciones de conjunto, como unión, intersección, diferencia, copia y verificación de igualdad, se realizan en tiempo O(n) [1].

- **Set-Union**: Combinar dos conjuntos. Cada elemento de ambos conjuntos debe procesarse.
- **Set-Intersection**: Encontrar elementos comunes entre dos conjuntos. Requiere iterar sobre cada elemento.
- **Set-Difference**: Obtener los elementos que están en un conjunto pero no en otro. Debe iterar y verificar cada elemento de ambos conjuntos.
- **Set-Copy**: Copiar todos los elementos de un conjunto a otro. Requiere iterar sobre todos los elementos.
- **Set-Equality**: Verificar si dos conjuntos son iguales comparando cada elemento.

G. Demostración formal de complejidad

1) Espacial: El SparseSet requiere de dos vectores de elementos de tamaño u, donde u es el tamaño del universo de elementos o, en la práctica, el rango de elementos a insertar. Además, utiliza dos enteros, por lo que para cada set se requieren 4u + 2 bytes. Es decir, tiene una complejidad espacial de O(u), la cual es independiente de la cantidad de elementos del set [1].

Algorithm 1 Estructura del SparseSet

```
Class SparseSet
    Variables:
        dense: array de enteros
        sparse: array de enteros
        size: entero
        maxElements: entero
```

2) Computacional: En cuanto a la complejidad computacional de las operaciones del sparse set. Sea un elemento x, donde 0 ¡= x y x¡= u - 1 x pertenece a sparse y dense[sparse[x]] = x. Para la inserción se verifica que xno pertenezca al set, de ser asi, se agrega a dense y se apunta a esta posición desde sparse. Como se puede ver, la inserción es independiente de la cantidad de elementos, lo cual hace de esta operacion O(1). La eliminación verifica la pertenencia de x al set, de ser así, sobreescribe dense[x] con el último elemento de dense y se actualiza el sparse para apuntar a esta posición. De manera similar a la inserción, las operaciones son independientes de los elementos del set, por lo que también tiene un O(1). Por último, la existencia de x en el set, se realiza en O(1) ya que solo se realizan comparación y acceso aleatorio a memoria.

Algorithm 2 Función Insertar

```
Function Insertar(elemento)
    Input: elemento (entero)
if elemento ≥ maxElements o elemento < 0 then
    lanzar "Elemento inválido"
end if
if sparse[elemento] ≥ size o dense[sparse[elemento]] ≠ elemento then
    sparse[elemento] ← size
    dense[size] ← elemento
    size ← size + 1
end if
end Function
```

Algorithm 3 Función Eliminar

```
Function Eliminar(elemento)
    Input: elemento (entero)
if elemento ≥ maxElements o elemento < 0 then
    lanzar "Elemento inválido"
end if
índice ← sparse[elemento]
if Existe(elemento) then
    últimoElemento ← dense[size - 1]
    dense[índice] ← últimoElemento
    sparse[últimoElemento] ← índice
    size ← size - 1
end if
end Function
```

Algorithm 4 Función Existe

```
Function Existe(elemento) return booleano
    Input: elemento (entero)
    Output: booleano indicando si el elemento existe
return  elemento < maxElements y elemento ≥ 0 y
sparse[elemento] < size y dense[sparse[elemento]] ==
elemento
end Function
```

En adición, operaciones entre sets como unión, intersección y diferencia se realizan en O(n). Esto ocurre debido al uso del dense set que mantiene los elementos juntos y permite la iteración eficiente por todos los elementos del set.

H. Estructuras Similares

1) Bit Vector: Un bit vector, también conocido como bit set, es una representación compacta de un conjunto de valores usando una secuencia de bits, donde cada bit indica la presencia o ausencia de un elemento en el conjunto. Los valores que puede almacenar esta estructura está limitado su tamaño, un bit vector es muy eficiente en términos de memoria y permite realizar operaciones a una velocidad muy alta.Sin embargo el Bit vector para el análisis de flujo de datos, no es eficiente en este caso. Ya que se requiere O(u) tiempo para borrarse y O(u) tiempo para iterar sobre todos los miembros, donde u representa el tamaño del universo. Estos requisitos son especialmente preocupantes en las aplicaciones , donde el número de elementos del conjunto es pequeño en relación con el tamaño del universo [1].

2) Hash Table (Chaining): Otra de las representaciones de un set, es el hashtable con la variación de que la key y el value almacenadas tienen el mismo valor. Esta estructura proporciona la inserción, búsqueda y eliminación con una complejidad constante. Respecto a operaciones entre sets como la unión, intersección y diferencia, proporciona una complejidad lineal. Asimismo, la operación de eliminación de todos los elementos tiene una complejidad lineal.

IV. Metodología

A. Implementación de estructuras de datos

Las estructuras de datos se implementarán en C++ 17 con los métodos básicos de un conjunto, como la inserción, eliminación y búsqueda de elementos. Además, se desarrollarán operaciones entre conjuntos, tales como la unión, la intersección y la diferencia.

B. Análisis comparativo

La comparación entre las estructuras de datos, se hará mediante el tiempo de ejecución de cada operación de las diferentes representaciones del set. Se utilizan sets de 500, 5000, 50000 y 500 000 elementos. Para cada conjunto de datos se tomarán 10 mediciones por cada operación y se usara el resultado promedio para hacer graficas que ayuden a su entendimiento.

C. Experimentos

Se haran en total 3 experimentos comparativos. El primero se basa en la comparacion de operaciones fundamentales del set tales como insercion, eliminacion y pertenencia. Para garantizar una reproduccion de los resultados de Briggs, se utilizara el siguiente codigo para los test:

Algorithm 5 Operaciones con estructura

```
for i ← 0 to size - 1 do
    estructura.Insertar(num)
    estructura.Eliminar(num)
    estructura.Existe(num)
end for
```

Este codigo se repetira 10 veces por cada tamaño de set para sparse set, bit vector y hash table.

El segundo experimento consta en la comparacion de las operaciones union e interseccion. En este se pondran a prueba el bit vector y sparse set dado que ambas estrcturas se incializan con una capacidad maxima y se mantienen asi independientemente de la cantidad de elementos que contienen. En cambio, el hash table puede redimensionarse basado en la cantidad de elementos, por lo que no se ajusta al experimento. En ese sentido solo se compara el tiempo computacional de las operaciones del bit vector y sparse set.

Como tercer y ultimo experimento, se hara una comparacion para la operacion clear entre el sparse set y el bit vector. Este se hara para determinar la presunta eficiencia constante del sparse set.

V. Evaluación experimental

Los resultados de los experimentos 1 y 2 se muestran en la Tabla I, donde puede apreciar el tiempo en microsegundos que le tomo a cada estructura ejecutar una operación según el tamaño del set. Las 3 primeras filas corresponden al primer experimento, en el que se ejecutan las operacion fundamentales de un set. Las siguientes filas corresponde al segundo experimento, el cual se centra en las comparacion de las operaciones unión e intersección sparse set y el bit

vector.Estos experimentos se realizaron con una laptop que tiene un procesador Ryzen 5 4500u y 20 GB de RAM

TABLE I
PROMEDIO DE LOS TIEMPOS DE EJECUCIÓN(US) DE OPERACIONES EN REPRESENTACIONES DEL SET

Operaciones	500	5000	50000	500000
basicsSparseSet	24	277	2283	29081
basicsBitVector	18	166	1685	15187
basicsHashSet	35	332	3272	29592
testUnionSparseSet	5	9	16	19
testUnionBitVector	8	34	327	3262
testIntersectSparseSet	3	6	6	6
testIntersectBitVector	8	33	325	3158
testClearSparseSet	3	3	4	6
testClearBitVector	7	32	311	3040

La Figura 5 muestra los tiempos de ejecucion de las operaciones fundamentales del set(inserción, eliminación y pertenencia). En el caso de las 3 representaciones de set, el tiempo de ejecución sigue un comportamiento lineal en cada caso. Esto se debe a que las operaciones fundamentales incluidas tienen una complejidad constante. Por lo tanto, al realizar n iteraciones de estas operaciones, donde n es el tamaño del set, la complejidad se debe mantener lineal, exactamente lo que la Figura 5 evidencia. Asimismo, Briggs menciona que estas operaciones se realizan más rápido en el bit vector que en el sparse set, dado que la implementacion del bit vector es mas simple y, en adicion, sus operaciones no involucran tantos pasos como los del sparse set, lo cual se confirma con la Figura 5 [1].

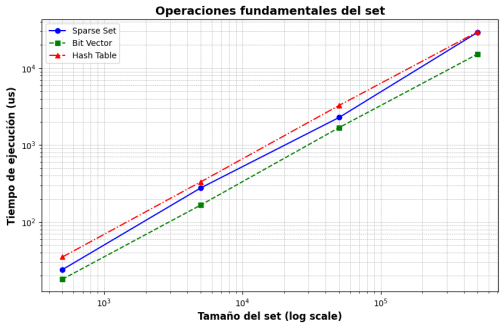

Fig. 5. Operaciones fundamentales

La Figura 6 muestra la operación de unión de conjuntos utilizando el bit vector y el sparse set. Como se puede apreciar, el tiempo de ejecución del bit vector es mayor que el del sparse set en cada medición. Lo mismo ocurre en la Figura 7, los tiempos del bit vector se alejan del sparse set a medida que el tamaño del set incremente. Esto se puede explicar por la complejidad de las operaciones: la unión e intersección son O(u) para el bit vector, donde u es la capacidad del set, mientras que para el sparse set son O(n), donde n es la cantidad de elementos en el conjunto [1].

Las complejidades mencionadas se dan ya que el bit vector tiene que iterar por todos los elementos de su universo para encontrar cuales de ellos pertenecen al set. Es asi que, dado que en el experiemento la capacidad del set incrementa progresivamente desde 500 hasta 500 000, su complejidad para la operacion union e interseccion tambien incrementa.

En cambio, el sparse set, tiene un array dense en el que tiene a todos sus elementos contiguos uno tras otro, lo cual le evita recorrer todo el universo de elementos y solo itera por los que pertenecen al set. De esta manera, dado que el sparse set es independiente de la capacidad del set, el incremento en esta variable no representa un incremento en la complejidad computacional de las operaciones. De hecho, en la Figura 6, el sparse set muestra un comportamiento cercano al constante. Esto ocurre, ya que se insertaron 100 elementos en ambas estructuras y esta variable se mantuvo fija en todo el experimento. De esta manera, la complejidad de la union e interseccion usando un sparse set, tambien se mantuvo fija y segun el experimeto, como se ve en la Tabla 2, el tiempo de ejecución tuvo ligeras variaciones en comparacion con los tiempos del bit vector.

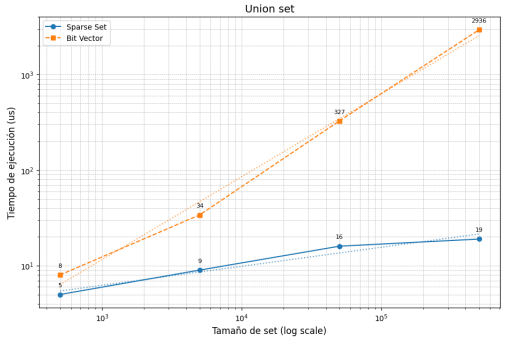

Fig. 6. Union set

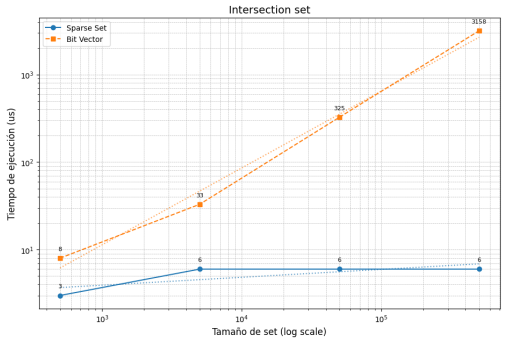

Fig. 7. Intersection set

La Tabla II muestra los resultados del tercer experimento. En el cual se puso a prueba la operacion clear para el sparse set y bit vector. En este caso los tiempos se calcularon en nanosegundo. Esto se hizo para poder hacer una comparacion mas detallada donde las diferencias se vean claramente.

TABLE II
PROMEDIO DE LOS TIEMPOS DE EJECUCIÓN(US) DE LA OPERACION CLEAR

	500	5000	50000	500000
testClearSparseSet	1320	1292	1348	1292
testClearBitVector	1264	1438	3003	17489

En la Figura 8 se puede ver la grafica de la Tabla II. En este caso los tiempos de ejecucion fueron casi los mismos al inicio, con un set de capacidad 500. Sin embargo, a medida que el tamaño del set aumuentaba el tiempo para el bit vector tambien aumuentaba.

Esto se debe a que, de la misma manera que con la union e interseccion, el bit vector tuvo que iterar sobre todo su universo de elementos para limpiar el set independientemente de la cantidad de elementos que pertenecen a el. Por ello, la operacion clear en el bit vector es O(u).

Por otro lado, los tiempos de ejecucion de clear para el sparse set, son cercanos entre si. De manera que se asemeja a una complejidad constante. Esta diferencia de eficiencia entre estructuras de datos, es lo que Cox refiere como un truco que permite llevar operaciones de una complejidad lineal a una constante [2]. De hecho, de acuerdo con Cox, el usar 2 arrays que se apuntan entre ellos como lo hacer el sparse set, permite a esta estructura diferenciarse de otras, ya que no necesita ser inicializada y puede funcionar con normalidad aun con valores basura [2]. Bajo este mismo razonamiento, la operacion clear tampoco requiere iterar por todo el universo de elementos como lo hace el bit vector, ni siquiera por los elementos que pertencen al conjunto. De hecho, su capacidad de no ser afectada por valores basura, hace que el costo de clear sea constante. Lo cual se evidencia en la Figura 8, donde los tiempos del sparse set son cercanos entre si independientes del tamaño del set.

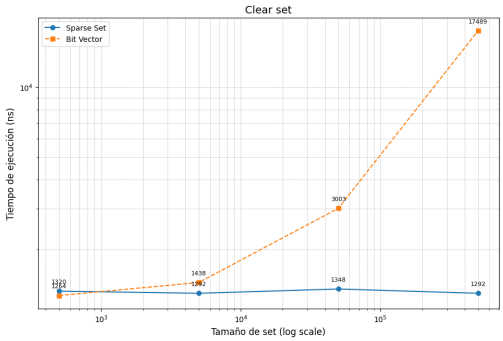

Fig. 8. Clear set

VI. CONCLUSIONES

Este trabajo hizo una comparacion entre diferente representaciones del set. La representacion que mostro una mejor complejidad computacional fue la del sparse set. Si bien el bit vector proporciona insercion, eliminacion y pertenecia con una complejidad constante, sus operaciones entre conjuntos tiene un costo del tamaño del universo de variables. Por su parte, el hash table reduce las operaciones entre conjuntos a una complejidad lineal de los elementos que contiene, sin embargo la operacion clear tambien tiene un costo lineal. Por ello, si se quiere eficiencia en operaciones entre conjuntos y un clear de costo constante, el sparse set es superior a las otras representaciones.

En conclusión, el sparse set se presenta como una opción preferible para aplicaciones que requieren alta eficiencia en operaciones entre conjuntos y manejo de elementos en un espacio de memoria relativamente grande pero con pocos elementos activos. Su estructura permite un desempeño robusto y escalable, destacándose por su capacidad para mantener operaciones de complejidad constante incluso en tareas de limpieza y reorganización del conjunto. Esta investigación respalda la recomendación de utilizar sparse sets en sistemas donde la eficiencia y rapidez en la manipulación de conjuntos sean críticas.

REFERENCES

[1] P. Briggs and L. Torczon, “An efficient representation for sparse sets,” vol. 2, no. 1–4, pp. 59–69, Mar. 1993, doi: https://doi.org/10.1145/176454.176484.
[2] R. Cox, “research!rsc: Using Uninitialized Memory for Fun and Profit,” Swtch.com, 2008. https://research.swtch.com/sparse (accessed Jun. 17, 2024).
[3] R. Belwariar, “Sparse Set,” GeeksforGeeks, May 12, 2016. https://www.geeksforgeeks.org/sparse-set/ (accessed Jun. 17, 2024).
[4] Oleksandr Manenko’s Blog, “Sparse sets,” Oleksandr Manenko’s Blog, May 23, 2021. https://manenko.com/2021/05/23/sparse-sets.html (accessed Jun. 17, 2024).

El codigo utilizado para el trabajo se encuentra en este link.