Compresión de datos en redes sociales

Marc Rodríguez Rodríguez

Resumen– Actualmente, las redes sociales son un aspecto vital en nuestro día día, y gran parte de su instauración en nuestra sociedad es debido a que permiten que nos comuniquemos con otras personas, independientemente de la localización geográfica de estas, así como de la eficiencia y la velocidad con la que se realizan las operaciones básicas de dichas redes sociales. Y es que, para que una red social se mantenga en lo más alto, tiene que ser una plataforma con aspectos muy pulidos tanto en el apartado visual, como en el apartado de prestaciones. En este ultimo es en el que entran en acción las diferentes técnicas de compresión que podemos encontrar a día de hoy. Entre estás técnicas destaca una en concreto, Array of binary tree, la cual permite realizar operaciones de manera 'on-the-fly', es decir, trabajar directamente desde los datos comprimidos y poder alterar simplemente los datos necesarios.

Palabras clave– Redes sociales, Array of binary tree, BitString, Dinámico, Compresión, Descompresión, Nodos, Aristas.

Abstract– Nowadays, social networks are something that we live within our day-to-day life, and that has occurred basically, because social networks brings us the chance to communicate with other people, regardless of where we are, as well as the efficiency and the speedness of the basic operations that those social networks can give us. So to achieve the top of the social networks ranking a good interface and an optimized structure is needed. But to accomplish this optimization we need to implement some compression techniques, like Array of Binary Tree, to realize certain operations in an 'on-the-fly' way. Which means that we need to work directly on the data that has been compressed, so we can only modify the data that is strictly necessary.

Keywords– Social networks, Array of binary tree, BitString, Dynamic, Compression, Decompression, Nodes, Edges.

1 Introducción

En la actualidad más de un 50 % de la población tiene acceso a Internet, hecho que comporta que, junto al continuo crecimiento del número de Smartphones, las redes sociales tengan un constante crecimiento en el número de usuarios.

A día de hoy, según datos extraídos de la última publicación de Bond Capital Internet Trends [8], hasta un 30 % del total de personas que tienen acceso a Internet, utilizan a diario Facebook, un 27 % en el caso de Youtube y un 25 % en el caso de Whatsapp, entre otros.

Pero para poder utilizar este tipo de aplicaciones de manera eficiente, con tiempos de respuesta del orden de milisegundos y ocupando tamaños aceptables en cuanto a las capacidades de almacenaje de las memorias RAM, se deben comprimir el mayor número de datos posibles, utilizando algoritmos y metodologías eficientes para que en aplicaciones como podría ser Twitter, la cual es utilizada por millones y millones de personas, que el simple hecho de eliminar un vínculo entre dos perfiles, como podría ser, dejar de seguir a otro usuario, tarde del orden de milisegundos, a pesar de tener que acceder hasta la parte de memoria en que se guarda esa información de forma comprimida, y una vez descomprimida y actualizada la información, volver a indexar dichos datos en su posición anterior cambiando el menor número de datos posibles en posiciones contiguas a la zona donde se han actualizado los valores.

Por estos motivos, junto al constante crecimiento del volumen de datos que se utilizan en las principales redes sociales a día de hoy, produce que la compresión de datos en dichas plataformas tenga un peso cada vez mayor, debido en parte, a la constante exigencia del público de un producto, a que este funcione perfectamente sin esperas de más de pocos segundos.

Por lo tanto, para cumplir estas expectativas es necesario desarrollar algoritmos que trabajen de manera 'on-the-fly',

• E-mail de contacto: marc.rodriguezro@e-campus.uab.cat
• Mención realizada: Tecnologías de la Información
• Trabajo tutelado por: Joan Serrà-Sagrista, Departamento de Ingeniería de la Información y de les Comunicaciones
• Curso 2019/20

es decir, que realicen operaciones sobre los datos comprimidos y simplemente se descomprima o se modifique los datos que sean esencialmente indispensables para realizar la operación concreta que se lleva a cabo.

Hecho que produce un gran aumento en el rendimiento de la aplicación y por tanto de la competitividad que puede ofrecer la red social en el mercado respecto a otras plataformas.

2 Estado del arte

Las tres principales técnicas de compresión más eficientes a día de hoy son, Backlinks Compression [6], la cual está diseñada para comprimir los diferentes nodos y aristas que se utilizan en las representaciones de las redes sociales, en base a las conexiones entrantes a un nodo (backlinks), las cuales permiten ir saltando de nodo en nodo. Estos nodos representan los usuarios de la red social en cuestión sobre la que se esté tratando, y las aristas son los vínculos entre dichos usuarios.

Esta técnica de compresión permite a las aristas que muestran localidad, codificarlas a través de pequeños integers, reduciendo considerablemente el espacio que ocupan.

Sin embargo, por otra parte, esta técnica puede tener un mayor tiempo de respuesta del que podría esperarse para las adjacency queries, debido a la no limitación de la longitud de las cadenas que se analizan, esto desemboca en la problemática de un mayor tiempo de espera para la realización de estas operaciones.

Otra de las tres técnicas de compresión es SlashBurn [7], la cual se basa en eliminar los principales hubs, que en redes sociales como Twitter vendrían a ser personas con muchos seguidores, para así crear pequeñas comunidades de usuarios aislados, y obtener una representación muy compacta de la matriz de adyacencia, que se utiliza para acceder a los diferentes nodos y poder realizar una buena compresión de los datos.

Slashburn es una técnica de compresión la cual contra mayor sea la cantidad de nodos conectados a estos denominados hubs, mayor será la compresión y por tanto esto desemboca en un menor tiempo para realizar operaciones en la matriz.

Pero si se da el caso que hay un gran número de nodos sin conexiones con otros nodos, los también conocidos como Caveman communities, esto propicia que la compresión sea bastante improductiva, ya que habrá un gran número de nodos a los cuales no se puede acceder a través de la matriz de adyacencia.

Finalmente la tercera técnica se trata de Array of Compressed Binary Trees [4], la cual consiste en agrupar un conjunto de nodos vecinos en uno solo, para así comprimir solo este único nodo que representa al conjunto, ocupando muy poco espacio, pero esto implica que cuando se quiere acceder a un nodo 'x' el cual se encuentra en una posición (u,v), se debe comenzar desde el primer elemento del grafo e ir leyendo y descomprimiendo secuencialmente hasta encontrar el nodo u.

Por este motivo, cuando se realizan operaciones sobre un nodo que se encuentra en el lado opuesto al primer nodo, siguiendo una estructura BFS, es necesario reconstruir todo el grafo para que a medida que se avanza en dicho grafo, ir reconstruyendo el BitString y de esta manera poder llegar al nodo en cuestión que se quiere modificar.

Sin embargo, esta técnica de compresión obtiene resultados pésimos para grafos o árboles de tamaños muy reducidos, llegando a generar BitString de mayor tamaño, que la secuencia de bits que se obtiene del grafo sin comprimir. Un ejemplo de esto sería el grafo mostrado a continuación, ya que para un conjunto de bits 1100 al comprimir dicho grafo se obtendría un BitString 11011 (estructurado bajo el algoritmo de ordenación BFS desde el nodo raíz).

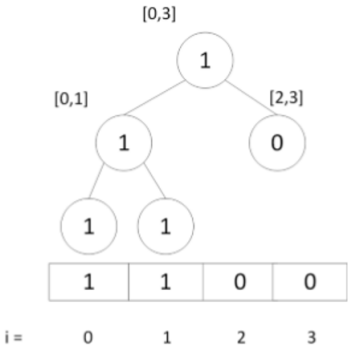

Fig. 1: Ejemplo de grafo de una mala compresión

Finalmente, debido a que esta técnica es la única de las tres que permite realizar compresión dinámica, el trabajo se basará en esta técnica para así intentar obtener resultados sobre la viabilidad de implementar algoritmos que utilicen técnicas 'on-the-fly'.

3 Objetivos

El objetivo principal de este trabajo es llegar a realizar una compresión de una red social, intentando incorporar modificaciones 'on-the-fly', es decir, realizar una compresión y modificación de un elemento concreto de la red social evitando tener que tratar con la plenitud de los elementos que conforman el grafo de la estructura que implementa la red social, formada por nodos y aristas como se ha comentado en el apartado anterior.

Es por eso que el trabajo se centra en ABT, debido a que al ser la única técnica de compresión que permite agregar o eliminar aristas directamente sobre la estructura comprimida, sin tener que volver a comprimir completamente el grafo. Y como puede observarse en el trabajo realizado por C. N. Sekharan, S. Radhakrishnan, B. Nelson y A. Chatterjee en [4], se expone cómo simplemente con añadir o eliminar los bits apropiados se puede descomprimir o comprimir el grafo que se está tratando en cuestión.

Por ello, en este trabajo de final de grado he desarrollado, a partir de la información recopilada por el ex-alumno Ashwin Kumar Gururajan [1] y de los resultados obtenidos en el desarrollo de diferentes algoritmos de compresión de datos referentes a redes sociales, un estudio que profundiza en el dinamismo de la técnica de compresión de Array of Compressed Binary Trees (ABT), la cual es la única de estas técnicas mencionadas anteriormente que permite agregar nuevas aristas y nodos a un grafo comprimido.

Para poder llevar a cabo este proyecto se han definido 5 objetivos, que se muestran a continuación, los cuales pue-

den considerarse como las 5 principales etapas que han sido completadas para lograr el principal objetivo del trabajo, el cual es realizar una compresión y posterior modificación exclusivamente de los elementos imprescindibles para así lograr modificaciones 'on-the-fly'.

Etapas:

- Analizar el código proporcionado por Ashwin para estudiar la viabilidad de modificar el algoritmo de compresión para que soporte compresión dinámica.
- Realizar un algoritmo que mediante el BitString obtenido de su previa compresión se puedan realizar búsquedas de nodos.
- Realizar un algoritmo que permita añadir nodos y aristas a un grafo, donde solo se descomprima y se compriman posteriormente los nodos que sean indispensables para realizar estas operaciones.
- Obtener diferentes datos para calcular viabilidad de este algoritmo en función de diferentes métricas.
- Plantear las conclusiones finales del trabajo sobre el impacto que puede tener esta técnica en un futuro, en función de si se trata de una técnica viable o no.

4 METODOLOGÍA

Para la realización de este proyecto se ha utilizado una metodología ágil como es Kanban lo cual permite llevar a cabo un control del desarrollo del proyecto de manera visual y a través del Work In Progress (WIP), que limita el número de tareas que se realizan en cada fase, aunque en el caso de este proyecto, las fases corresponden a las diferentes entregas de los informes de seguimiento y posteriormente la entrega del informe final. Esta metodología permite que pueda desarrollar el trabajo de manera continua e incremental, pudiendo contemplar en todo momento el progreso y si este progreso se adecua al ritmo de trabajo esperado.

Durante el desarrollo del trabajo, la cual viene marcada por la planificación que se muestra en el siguiente apartado de este informe, se establecieron unas tareas que se debían realizar durante el proyecto como TO DOs, las cuales fueron cambiando de estado según se finalizaban exitosamente otras tareas anteriores, en caso de haber, a un estado de *In Progress*, cuando estaban siendo realizadas. Y finalmente un estado de DONE, cuando dicha tarea ha sido completada con éxito.

Después de haber realizado el trabajo por completo, la metodología Kanban se ajusta a la perfección a la estructura mayormente secuencial que sigue el trabajo pudiendo observar en cada momento si la carga de trabajo avanza adecuadamente o si por ende, el ritmo de trabajo no es el adecuado y por tanto no se cumplen las fechas previstas y se acumula dicha carga de trabajo.

En cuanto a lo referente al lenguaje de programación utilizado para el desarrollo de este proyecto, el cual es C++, he de remarcar que a pesar de ser un lenguaje con el cual tengo experiencia, se han ocasionado una serie de problemas en cuanto a la comprensión y adaptación al código de Ashwin debido a la gran cantidad de utilizaciones de funciones definidas en los estándares de C++, las cuales podemos encontrar a través de std::, que eran desconocidas para mí previamente, dónde a través de la página web [2] se ha obtenido la información necesaria para entender las diferentes funcionalidades del código proporcionado.

También han surgido problemáticas con la utilización de la herramienta de compilación Makefile, cuya herramienta es la que Ashwin utiliza para la compilación del programa según se encuentra en su GitHub [1], la cual también desconocía hasta la actualidad, hecho que ha producido que no haya cumplido con exactitud todos los intervalos de tiempo asignados para cada tarea, puesto que se ha tenido que modificar el archivo Makefile que realizó Ashwin para su trabajo y a su vez he tenido que aprender y documentarme sobre este tema a través de [3].

5 PLANIFICACIÓN

TTED: Tiempo de trabajo estimado en días.
FFE: Fecha de finalización esperada.
RR: Realización de un algoritmo de descompresión del árbol binario junto a un algoritmo de búsqueda de nodos.

Planificación					
Nombre de la actividad	Fecha inicio esperada	Fecha inicio real	TTED*	FFE*	Fecha de finalización real
Obtener documentación	14-02-20	14-02-20	114	07-06-20	21-06-20
Entrega informe inicial	02-03-20	02-03-20	6	08-03-20	08-03-20
FASE 1					
Analizar código proporcionado	09-03-20	09-03-20	8	17-03-20	22-03-20
Realizar algoritmo para encontrar nodos en el grafo	18-03-20	23-03-20	14	01-04-20	05-04-20
Test del código realizado hasta el momento	01-04-20	06-04-20	5	06-04-20	08-04-20
Modificar algoritmo para poder añadir nodos y aristas	06-04-20	09-04-20	7	13-04-20	16-04-20
Test del código realizado hasta el momento	13-04-20	16-04-20	6	19-04-20	19-04-20
Entrega del informe de progreso I	13-04-20	13-04-20	6	19-04-20	19-04-20
FASE 2					
RR*	20-04-20	20-04-20	25	17-05-20	19-05-20
Test del código realizado hasta el momento	20-04-20	20-04-20	25	17-05-20	19-05-20
Entrega del informe de progreso II	18-05-20	20-05-20	6	24-05-20	24-05-20
FASE 3					
Realizar algoritmo para añadir nodos	25-05-20	25-05-20	6	31-05-20	31-05-20
Obtener datos para calcular viabilidad del algoritmo	30-05-20		6	05-06-20	
Test del algoritmo completo	20-04-20	20-04-20	48	07-06-20	
Plantear conclusiones finales	05-06-20	21-06-20	2	07-06-20	21-06-20
Entrega propuesta informe final	08-06-20	08-06-20	6	14-06-20	28-06-20
Entrega propuesta de presentación	22-06-20		6	28-06-20	
Entrega dosier	22-06-20	21-06-20	6	28-06-20	28-06-20
Entrega póster	29-06-20		6	05-07-20	

TABLA 1: PLANIFICACIÓN LLEVADA A CABO DURANTE EL PROYECTO.

6 DESARROLLO

Previamente a comentar las diferentes funcionalidades que han sido implementadas en el proyecto, es necesario explicar en profundidad el funcionamiento de la técnica Array of Binary Tree. ABT es una técnica node-centric, debido a que las redes sociales y sus operaciones de consultas están basadas en nodos. Es por ello que esta técnica de compresión consiste en representar todos los nodos vecinos de un nodo de una forma adecuada.

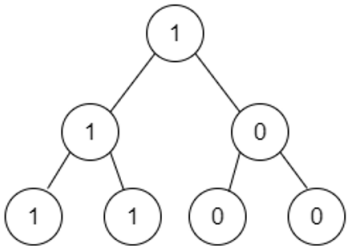

Fig. 2: Representación de un árbol binario sin comprimir, a partir del cual se obtendría el BitString 1101100.

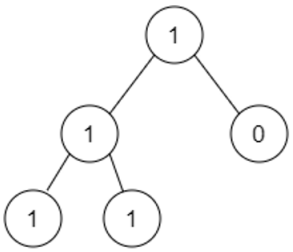

Fig. 3: Representación del árbol binario una vez se ha comprimido mediante la técnica ABT, a partir del cual se obtendría el BitString 11011.

Debido a que cuando se comprime un nodo, este pasa a ser una conjunto de bits, concretamente un único bit, que se añaden al final de un BitString. Un nodo toma un valor 1 (true), si no se trata de un nodo hoja o si es un nodo hoja correspondiente a una arista, siendo todos los nodos restantes marcados como 0 (false). Resultando que este esquema de codificación pode todas las secciones de las filas de la matriz que estén vacías.

De igual manera que con los nodos, las aristas también se comprimen generando una matriz de adyacencia, donde cada arista toma como valor 1, si el nodo que tiene como destino existe y por tanto puede realizarse una conexión entre dos puntos del grafo. Permitiendo así generar un vector que contiene las diferentes rutas que se pueden recorrer para llegar de un punto a otro del grafo, siempre y cuando sea posible.

El programa consta de dos algoritmos, Node addition y Node query. Este ultimo consiste en realizar una búsqueda de un nodo directamente desde el BitString, que se obtiene de la compresión de un grafo mediante la técnica de compresión ABT. Dicho algoritmo consiste en ir calculando en cada iteración el rango en el que se encuentra el nodo sobre el cual se está realizando la búsqueda en cuestión.

Para ello primero cabe destacar los siguientes aspectos, y es que, si tenemos un árbol de $2^{k+1} - 1$ nodos, podemos encontrar $2^{k+1}/2$ nodos en su máxima profundidad, donde k hace referencia a la profundidad del grafo. Y finalmente el nodo inicial de este grafo toma el valor 0.

Para poder llevar a cabo la búsqueda del nodo en cuestión que queremos encontrar, se calculan las variables *minCol*, *maxCol* y *midCol*, todo en función de los nodos que encontramos en la máxima profundidad del grafo, para así poder seleccionar el camino más óptimo en todo momento.

La variable *minCol* hace referencia al nodo que delimita la posición mínima, en la que el nodo que se está buscando puede encontrarse. Una representación gráfica de esto puede encontrarse en la figura 4, donde para una primera iteración, *minCol* haría referencia al nodo que se encuentra en la posición 0 de i. Mientras que *maxCol* hace referencia al nodo que delimita la posición máxima en la que puede encontrarse el nodo que se está buscando.

Si observamos la figura 4, *maxCol* haría referencia al nodo que se encuentra en la posición 7 de i. Finalmente *midCol* hace referencia al nodo central del grafo, calculado a través de realizar la media aritmética entre los valores de *minCol* y *maxCol*, siguiendo una estructura BFS.

Es decir, si nos fijamos en la figura 5, *minCol* toma como valor, el número 7, mientras que *maxCol* toma como valor el número 14. De esta manera al realizar la media aritmética entre estas dos variables se obtiene que *midCol* toma como valor, el resultado de esta media redondeada hacia abajo. En este caso tomaría el valor 10.

Acto seguido se compara el valor de *midCol* con el valor de y, donde y hace referencia a la posición en la que se encuentra el nodo que buscamos cuando el árbol está reconstruido por completo, que por ejemplo, podría ser el valor 9, que encontramos en la figura 5. Haciendo referencia al nodo que se encuentra en la posición 2 del vector y de la figura 4, si el grafo de dicha figura estuviera reconstruido por completo.

Al hacer la comparación obtendríamos que y tiene un valor menor a *midCol* y por tanto avanzaría en el grafo a través del nodo 1, siguiendo el criterio de la figura 5.

Una vez llegados hasta este punto se volverían a calcular *minCol*, *maxCol* y *midCol* obteniendo como valor de *midCol*, la posición número 8, y como valores de *minCol* y *maxCol*, se obtendrían 7 para *minCol*, y 10 para *maxCol*, siguiendo la estructura de la figura 5. Y al realizar la comparación con el valor de y, obtendríamos que dicha variable contiene un valor mayor al de *midCol*, y por tanto avanzaríamos en el grafo hacia el nodo número 4, siguiendo la estructura de la figura 5.

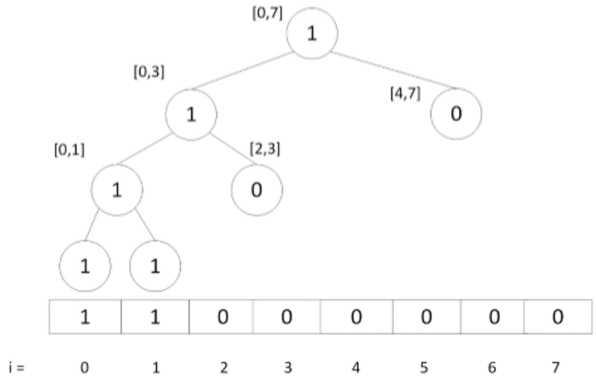

Fig. 4: Representación del árbol binario comprimido mediante la técnica ABT.

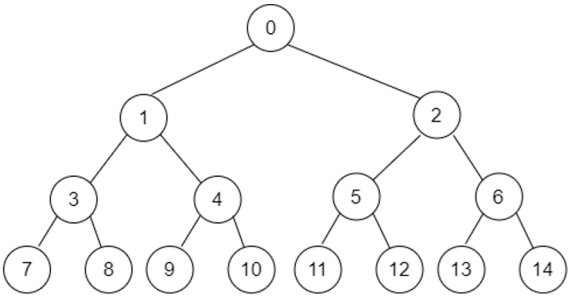

Fig. 5: Representación del árbol binario ordenado de manera BFS.

Justo cuando se inicia este algoritmo y cada vez que se avanza en el grafo, se lee directamente del BitString el valor del nodo correspondiente a la posición en la que nos encontramos en cada iteración, siguiendo la representación mostrada en la figura 5.

Por lo tanto para llevar a cabo este procedimiento es necesario ir reconstruyendo el BitString según se recorre, ya que de esta manera se realiza una descompresión de este.

Para explicar esta parte del algoritmo tomaré como referencia un grafo con la estructura seguida en la figura 5, el cual puede se representa en la figura 6.

Para llevar a cabo esta descompresión, según se avanza en el grafo, se van añadiendo bits con valor *"0"* al BitString directamente. Donde para un grafo como el de la figura 6, obtenemos el BitString *11110011001* una vez se ha comprimido bajo la técnica de compresión ABT.

Si no se realizará una reconstrucción del BitString al realizar la descompresión, el algoritmo al leer dicho BitString entendería que el árbol es como el mostrado en la figura 7. Debido a que el algoritmo va leyendo el BitString interpretando que cada valor que lee corresponde al siguiente nodo, leyendo en formato BFS, independientemente de los valores que tiene el nodo padre en el árbol original.

Por tanto para solucionar esta problemática el algoritmo a medida que va recorriendo el BitString, va reconstruyendo dicho BitString, teniendo tiempos de descompresión mejores, normalmente, para valores más cercanos. Y tiempos peores cuando se realizan operaciones con nodos más alejados del primer valor que se lee del BitString cada 2^k nodos, siendo k la profundidad en la que nos encontramos en cada momento dentro del grafo.

Esta reconstrucción viene propiciada en parte debido a cómo devuelve el resultado el algoritmo de compresión realizado por Ashwin, ya que este algoritmo devuelve un BitString con una cantidad de caracteres múltiples de 8. Donde para el caso expuesto, en el ejemplo previamente explicado, devuelve el BitString: *1111001100100000*. Esta es la única manera de reconstruir el BitString para que pueda leer cualquier valor independientemente de la posición o del BitString inicial, el cual es devuelto por el algoritmo de compresión.

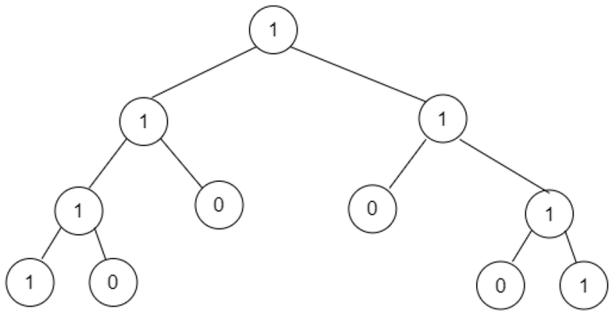

Fig. 6: Representación del resultado de la compresión de un árbol binario comprimido bajo la técnica ABT.

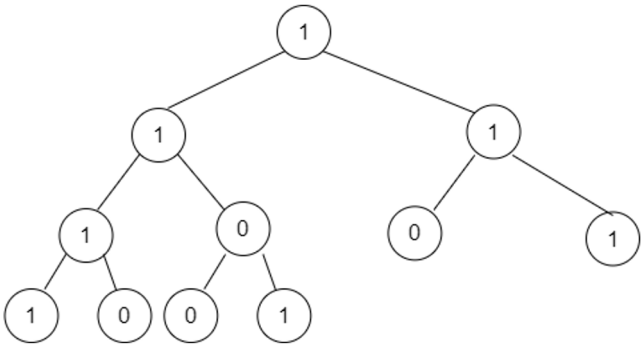

Fig. 7: Representación de cómo interpretaría el algoritmo que es el árbol a partir del BitString sin reconstruir.

En cuanto a lo referente al algoritmo de Node addition, este se encarga de añadir toda una serie de nodos y aristas en función del nodo introducido que se quiera añadir, por ejemplo, en caso de querer añadirse el nodo 9 siguiendo la estructura de la figura 5. Este algoritmo descenderá desde el nodo raíz hasta llegar al nodo 9 convirtiendo en 1 todos los valores de los nodos que tengan un valor de 0 en el momento inicial, como se muestra en la figura 8. Permitiendo así que al añadirse satisfactoriamente los valores correspondientes de los nodos que han cambiado su valor al BitString, se pueda dar paso al algoritmo de recompresión reCompress, el cual comprime el BitString que contiene los nuevos valores añadidos y los valores de los nodos comprimidos previamente.

Para descender por el árbol de manera correcta hacia el nodo que el usuario introduce por teclado se realiza de una manera similar a la del algoritmo de Node query, a través del cálculo de la columna mínima, máxima y la columna central de cada iteración del algoritmo para así evitar descomprimir más que lo necesario para poder llegar a añadir

el nodo en la posición pertinente dentro del BitString. Este proceso se muestra de manera gráfica en la siguiente figura:

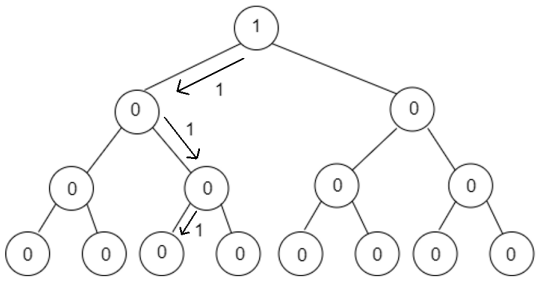

Fig. 8: Representación de las alteraciones que se realizan al añadir el nodo 9, correspondiente a la figura 5.

Una vez añadidos los diferentes '1' en las correspondientes posiciones dentro del BitString, el árbol quedará de la siguiente forma:

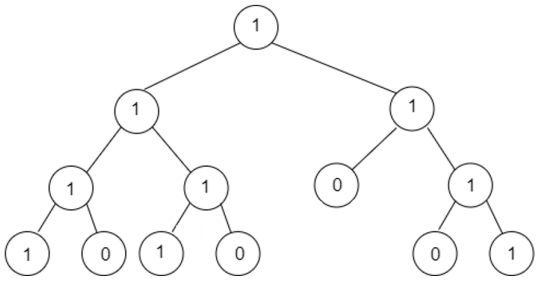

Fig. 9: Representación final de la estructura del árbol, una vez añadido todos los valores necesarios en el BitString.

7 Resultados

Para realizar la comprobación de que los resultados son los esperados, se han realizado dos funciones similares a las de Node query y Node addition donde en vez de reconstruir solo los elementos esenciales del BitString, se reconstruyen por completo todos los nodos y aristas que constituyen dicho BitString, simulando como sería tener que descomprimir en su totalidad el BitString que estamos tratando para que una vez se realicen las modificaciones, se vuelva a comprimir por completo dicho BitString.

Estos algoritmos y todos los nombrados anteriormente pueden encontrarse en el GitHub que aparece en el apéndice.

Cabe destacar que el programa que he desarrollado, solo obtiene resultados correctamente para árboles donde todos los nodos tienen la misma profundidad, por este motivo sería necesario trabajar un poco más en el código y los algoritmos de este, para que así pueda obtener resultados correctos independientemente de la profundidad de los nodos sobre los que se trabaja.

Como era de esperar, los resultados obtenidos para los algoritmos que utilizan técnicas 'on-the-fly', es decir, realizan modificaciones mínimas en el BitString para así evitar tener que descomprimir y comprimir el grafo en su plenitud, al realizar dichas modificaciones, obtienen tiempos menores, que los algoritmos que descomprimen y comprimen el BitString en su plenitud.

Prueba de ello son las figuras que se muestran a continuación, donde tanto en la tabla 2, como en la tabla 3, podemos observar los diferentes tiempos obtenidos al realizar las operaciones de búsqueda y adición de nodos.

Donde por lo general se obtiene tiempos mayores al realizar operaciones para los nodos que se encuentran a una mayor distancia del primer nodo hoja del grafo. Esto es debido a como trabajan los diferentes algoritmos reconstruyendo el grafo, para que, a través del BitString podamos acceder a la posición pertinente.

También, si observamos dichas tablas se puede observar una tendencia en la que, por lo general, contra mayor es la profundidad mayor es el tiempo necesario para realizar la operación de búsqueda, tanto para el método que utiliza técnicas 'on-the-fly', como el que trabaja comprimiendo y descomprimiendo el grafo por completo.

BDD: Búsqueda a través de descompresión dinámica.
BDC: Búsqueda a través de descompresión completa.
IOTF: Inserción de manera 'on-the-fly'.
IDCT: Inserción tras descompresión y compresión total.

Profundidad	BDD*	BDC*	IOTF*	IDCT*
4	1,3758 ms	1,4810 ms	1,7388 ms	1,9274 ms
5	0,8450 ms	0,9440 ms	0,9189 ms	1,0150 ms
10	1,1246 ms	9,1594 ms	1,1540 ms	1,2838 ms
13	7,7347 ms	70,7057 ms	1,3907 ms	1,5351 ms

TABLA 2: Resultados obtenidos para las operaciones de búsqueda e inserción del mayor nodo hoja posible dentro del grafo.

Profundidad	BDD*	BDC*	IOTF*	IDCT*
4	0,9031 ms	0,9163 ms	0,9050 ms	1,0044 ms
5	0,8610 ms	0,9990 ms	0,7551 ms	0,9288 ms
10	0,8804 ms	9,5444 ms	0,9690 ms	1,0171 ms

TABLA 3: Resultados obtenidos para las operaciones de búsqueda e inserción del menor nodo hoja posible dentro del grafo.

De las diferentes tablas mostradas previamente podemos obtener datos de como los algoritmos que utilizan técnicas 'on-the-fly' obtienen una reducción del tiempo necesario para ejecutarse, siendo esta reducción mayor cuanto mayor es la profundidad del grafo.

Sobre todo, se puede observar en las tablas 4 y 5 como hay un notorio decremento en el tiempo necesario a la hora de realizar una búsqueda, debiéndose principalmente a como el método *Node query* realiza las diferentes operaciones en cálculo del nodo que debe recorrer en cada momento.

Si comparamos los resultados entre los métodos de adición de nodos de manera 'on-the-fly' y los métodos que des-

PRMB: Porcentaje de reducción del tiempo necesario para realizar la operación de búsqueda entre el método con mecanismos 'on-the-fly' respecto el método de descompresión y compresión completa.
PRMA: Porcentaje de reducción del tiempo necesario para realizar la operación de adición entre el método con mecanismos 'on-the-fly' respecto el método de descompresión y compresión completa.

Profundidad	PRMB*	PRMA*
4	1,14 %	9,81 %
5	13,81 %	18,70 %
10	90,70 %	4,72 %

TABLA 4: PORCENTAJES DEL DECREMENTO DEL TIEMPO PARA LLEVAR A CABO LAS DIFERENTES OPERACIONES AL UTILIZAR EL PROGRAMA DESARROLLADO PARA EL MENOR NODO HOJA POSIBLE DENTRO DEL GRAFO.

Profundidad	PRMB*	PRMA*
4	7,11 %	9,78 %
5	10,48 %	9,46 %
10	89,72 %	10,11 %
13	89,06 %	9,40 %

TABLA 5: PORCENTAJES DEL DECREMENTO DEL TIEMPO PARA LLEVAR A CABO LAS DIFERENTES OPERACIONES AL UTILIZAR EL PROGRAMA DESARROLLADO PARA EL MAYOR NODO HOJA POSIBLE DENTRO DEL GRAFO.

comprimen y comprimen el grafo, podemos observar como obtienen resultados similares, en la mayoría de los casos, independientemente de cuanto se incremente la profundidad del grafo. Este fenómeno puede ser debido a que el método que realiza la adición descomprimiendo y comprimiendo el árbol por completo es una refactorización del método de que trabaja de manera 'on-the-fly', pero eliminando toda la algoritmia referente a la recompresión y el cálculo de la posición en cada momento.

También es necesario destacar que no todos los grafos con los que se han realizado las pruebas tienen el mismo número de nodos, hecho que provoca que los tiempos puedan ser similares dentro de los bancos de pruebas, independientemente de la profundidad máxima de cada grafo.

8 CONCLUSIONES

En un sector como es el de la compresión y manejo de datos en redes sociales, es vital minimizar tanto como sea posible los tiempos de ejecución, debido a la constante exigencia que los usuarios requieren a cualquiera de las diferentes redes sociales que podemos encontrar hoy en día.

Este hecho, sumado a que anualmente la mayoría de redes sociales aumentan el número de usuarios e interacciones entre estos, son un motivo de porque es necesario realizar algoritmos y técnicas óptimas, para minimizar los tiempos necesarios a la hora realizar ciertas operaciones básicas.

Por ello, y como se ha expuesto en este trabajo Array of binary tree, es la técnica perfecta para manejar estos datos de una manera óptima, debido a que gracias a que permite manejar datos de manera 'on-the-fly', es decir, permite modificar datos directamente desde el BitString, el cual es la estructura comprimida de un árbol inicial, permite así que tras modificar los datos, con tan solo comprimir los nuevos datos añadidos al BitString ahorremos cierto tiempo en realizar operaciones de búsqueda o inserción de usuarios o conexiones entre estos. El motivo principal por el que ocurre este fenómeno, es que al no ser necesario descomprimir el grafo por completo se necesita una menor cantidad de recursos de cómputo y por tanto un menor tiempo de ejecución. Hechos que comportan que ABT sea una técnica idónea para realizar dichas operaciones de manera eficiente.

AGRADECIMIENTOS

Finalmente me gustaría agradecer a Joan Serrà-Sagrista el apoyo constante que me ha brindado para que este trabajo haya podido ser una realidad, así como a Jordi Pons Aróztegui, por facilitar el trabajo a los estudiantes con todo lo relacionado a información respecto dicho trabajo y finalmente a Ashwin Kumar Gururajan, ya que sin su trabajo previo de las diferentes técnicas de compresión que podemos encontrar a día de hoy para las redes sociales, este trabajo no podría haber llegado hasta el punto en el que se encuentra a día de hoy.

REFERENCIAS

[1] A. Kumar. "Social media compression", Universitat Autònoma de Barcelona, Jul 2019, pp. 44. Disponible: https://github.com/G-AshwinKumar/Social-Network-Compression.

[2] CPPReference, cppreference.com, 2019. [Online]. Disponible: https://es.cppreference.com/w/.

[3] R. Mecklenburg. "Managing Projects with GNU Make", pp. 300, Published: O'Reilly Media, Nov 2004.

[4] C. N. Sekharan, S. Radhakrishnan, B. Nelson y A. Chatterjee. "Queryable Compression for Massively Streaming Social Networks", IEEE Transactions Conference on Big Data, Boston, MA, pp. 988-993, Dic 2017.

[5] N. R. Brisaboa, G. de Bernardo, G. Navarro. "Compressed Dynamic Binary Relations", en Information Systems, Volume 69, pp. 106-123, Published: Elsevier Sep 2017.

[6] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A.Panconesi, and P. Raghavan "On compressing social networks," en "Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining", KDD Jun 2009, pp. 219–228, New York, NY, USA, 2009. ACM.

[7] Yongsub Lim, U. Kang, y C. Faloutsos. "SlashBurn: Graph Compression and Mining beyond Caveman Communities". Knowledge and Data Engineering, IEEE Transactions, Knowl. Data Eng., vol. 26, no. 12, pp. 3077–3089, Dic. 2014.

[8] Bond Capital Internet Trends, Bondcap.com, 2019. [Online]. Disponible: https://www.bondcap.com/.

APÉNDICE

1.1. GitHub del código realizado

Código disponible en:
https://github.com/marcrodriguezro/ABTDynamicalOperations