Universidad de Concepción
Facultad de Ingeniería
Dpto. de Ing. Informática y Cs. de la Computación

Proyecto 2: Compresión Re-Pair

Fecha: 26/07/2020

Profesor: Diego Seco Naveiras.

Ayudantes: Alexis Espinoza Rebolledo.

Catalina Pezo Vergara.

Integrantes: Benjamin Fernandez Vera

Claudio Rain Levicán.

Descripción del Proyecto

El presente informe tiene por objetivos definir y caracterizar la implementación del algoritmo de compresión Re-pair, uno de los más importantes dentro de la familia de compresión basada en gramáticas. Primero, se determinará el problema a atacar, para luego presentar dos soluciones, una directa y otra avanzada, presentando sus respectivos detalles de implementación. Finalmente, se presentarán y analizarán los resultados de una serie de experimentos hechos con ambas soluciones.

Desarrollo de la Actividad

1. ¿Cuál es el problema a atacar?

El problema a atacar es la creación de un algoritmo que logre comprimir una cadena de números mediante el reemplazo del par que más se repite. Este par será cambiado por un parámetro **σ+1**, que irá aumentando a medida que se llame. Los números de la secuencia se encontrarán en un rango de [1,σ], por lo que ningún parámetro **σ+1** será menor a algún número que se encuentre en la secuencia inicial. El algoritmo deberá iterar hasta que ningún par se repita más de una vez, de esta manera se asegura que la compresión de la cadena sea máxima. A modo de ejemplo, si tenemos una cadena de números "9 1 2 3 1 2 3 8 4 5 1 2 3", con **σ** igual 9, podemos notar que el par más frecuente es el "1 2", por lo que este se reemplazará por el número "10" (**σ+1**), quedando así la siguiente cadena "9 10 3 10 3 8 4 5 10 3". Ahora el par más repetido es el "10 3", por lo que cambiará por el número "11", resultando la cadena "9 11 11 8 4 5 11". Podemos observar que ningún par de la cadena tiene frecuencia mayor a uno, por lo tanto, esta finalmente se encuentra comprimida. Si por algún motivo existe un par con frecuencia mayor a uno, se sigue iterando, hasta cumplir la condición antes mencionada.

2. Solución propuesta

Para solucionar el problema propuesto, se utilizaron dos métodos que comparten componentes similares. Ambos utilizan una lista doblemente enlazada para almacenar la cadena de números a comprimir, estas se diferencian principalmente en su procesamiento, debido a que implementan diferentes funciones y algoritmos para poder realizar su tarea. Además de esto cuentan con una clase Iterador encargada de recorrer la lista doblemente enlazada, que para ambos métodos es idéntica. A continuación, se presentan ambas soluciones.

2.1 Solución directa

2.1.1. Descripción: Esta solución cuenta con dos clases principales: DoublyLinkedLists y Direct_Solution, las cuales llevarán a cabo el funcionamiento del programa. La primera clase mencionada (DoublyLinkedLists) es la encargada de almacenar la cadena de números a comprimir. También deberá cambiar el par más frecuente por el parámetro **(σ+1**). A la segunda clase (Direct_Solution) se le delegó la gestión del compresor, puesto que: creará el mapa que contendrá los pares con sus respectivas frecuencias, al mismo tiempo que obtiene el par más frecuente, para posteriormente comunicarse con la DoublyLinkedLists y realizar los respectivos cambios.

2.1.2. Funcionamiento: Inicialmente se almacenará la cadena en la lista doblemente enlazada y se creará el map con la información de dicha cadena, luego de esto la solución obtendrá el par con mayor frecuencia almacenado en el map para de esta manera iterar en la lista doblemente enlazada haciendo los cambios correspondientes con el parámetro (**σ+1**). Esto se realiza mientras que en la cadena haya algún par con frecuencia mayor a uno. En cada iteración se debe vaciar el map, para así volver a rellenar el map con la cadena actualizada.

2.1.3. Descripción de clases

Para un mejor entendimiento de la solución es necesario un estudio más detallado de las características de las clases que la componen. Dicho esto, a continuación se presenta una descripción detallada de cada clase.

Clase Direct_Solution

Esta clase como antes fue mencionado es la encargada de gestionar el compresor, trabaja utilizando la estructura de datos map e instancias a la clase Iterator y DoublyLinkedLists. Los métodos que posee esta clase son los siguientes:

- **add_Sequence:** Utiliza la instancia a la clase **DoublyLinkedLists** para así ingresar el vector de enteros que recibe por parámetro a la lista doblemente enlazada.

- **frequency:** Llena el mapa con sus respectivos pares y frecuencias, y entrega el par que más se repite. Para realizar esta tarea, la función hace uso de la clase **Iterator,** debido a que debe recorrer toda la lista almacenando cada par que en ella encuentre, si alguno de estos pares ya fue ingresado, aumenta su frecuencia, si el par a ingresar no existe, se crea en el mapa y se sigue buscando hasta recorrer toda la lista.

- **change_List:** Recibe como parámetro el par que más se repite, para así ser buscado en la lista y cambiado con el parámetro correspondiente. Esta función hace uso de la instancia de la clase **Iterator** y **DoublyLinkedLists**. Su funcionamiento es el siguiente: Primero crea un iterador que permita recorrer la lista, luego en cada iteración que encuentre el par, lo reemplazará por el parámetro correspondiente, llamando a la función **change** de la clase **DoublyLinkedLists** que se hará cargo de realizar el cambio pertinente (Esta función se explica en la clase DoublyLinkedLists).

- **re_Pair_Compression:** Se encarga de hacer los respectivos llamados a las funciones anteriormente mencionadas. Su funcionamiento es el siguiente: Primeramente, se llama a la función **frequency** para de esta manera llenar el mapa y obtener el par que más se repite, luego de esto iteramos en un while llamando a la función **change_List** para así realizar el cambio correspondiente del par. Después vaciamos el mapa y volvemos a rellenarlo con la nueva información que fue actualizada en la lista, y así repetimos el proceso hasta que no exista un par con frecuencia mayor a uno.

Clase DoublyLinkedLists

La función principal de esta clase es el manejo y el almacenamiento de la cadena a comprimir. Está basada en una lista doblemente enlazada implementada con punteros utilizando un struct llamado **Node**, en este se almacena el valor y los punteros al número anterior y siguiente. La lista cuenta con un puntero al principio de esta y al final. Ahora se explican sus diferentes métodos:

- **insert:** Inserta el valor entregado por parámetro a la lista enlazada. Para esto se tiene el cuidado que los punteros **tail** y **head** queden apuntando a las direcciones correctas.

- **change:** Recibe como parámetros la dirección del par que se cambiara y el valor de (σ+1) por el cual tiene que ser cambiado. Para poder realizar este cambio tomamos en cuenta 3 operaciones base:
    - **Primera operación:** Creamos nuestro reemplazo y hacemos que apunte a las direcciones que apunta el antiguo par.
    - **Segunda operación:** Redireccionamos los punteros del siguiente y el anterior para que apunten a nuestro reemplazo
    - **Tercera operación:** Eliminamos los nodos inservibles y disminuimos el tamaño de la lista doblemente enlazada.

  Aparte de estos tres casos, deben ser considerados otros tres: Cuando el par se encuentra al principio de la lista, al medio y al final. De esta manera aseguramos que cada uno de los punteros queden apuntando a las posiciones correctas.

- **begin:** Simplemente está encargada de entregar un Iterator con los puteros head y tail.

2.2 Solución avanzada

2.2.1 Descripción: implementa métodos y procedimientos mucho más sofisticados que la solución directa. Primero, la solución avanzada además de contener un **mapa** y una lista enlazada doble posee un **Max heap** cuya clave es la frecuencia y, por cada nodo, el valor será el par correspondiente. El mapa aparte de contener cada par con su respectiva frecuencia almacenará un puntero a la siguiente y última ocurrencia de cada par. Por último, los nodos de la lista doblemente enlazada además de las ocurrencias **prev** y **next,** también contendrán punteros **prev_oc** y **next_oc,** los cuales apuntan a la ocurrencia anterior y siguiente de ese par respectivamente.

Inicialmente, se creará el mapa y el max Heap una sola vez, es decir, la información original de la lista doblemente enlazada (cantidad de pares, frecuencias y los punteros inicial y final de cada par dentro de la lista enlazada) se almacena sólo una vez, para luego ir **cuidadosamente** actualizando la información hasta que no quede ningún par de números que se repita dos veces.

2.2.2 Funcionamiento: una vez que hemos declarado toda la información preliminar dentro del **mapa** y el **Max_heap**, continuamente se pedirá al **Max_heap** el par con mayor frecuencia dentro de la lista, con el objetivo de reemplazar ese par por un parámetro **sigma + 1.** Uno por uno, iremos reemplazando las ocurrencias de dicho par dentro de la lista y, por cada reemplazo, debemos tener en cuenta lo siguiente:

1. Los pares de números que fueron afectados una vez hecho el reemplazo, y cómo su eliminación influye tanto en la lista como en las estructuras de datos. Por ejemplo, si tenemos la secuencia "1 2 3 4 5" y reemplazamos el par "3 4" por 28, los pares afectados serían "2 3" y "4 5". En consecuencia, si la **lista** contiene pares anteriores o siguientes de los elementos afectados, éstos deben ser notificados oportunamente de que dichos pares afectados han sido eliminados para que el programa no termine generando errores. También se establecerá en el **mapa** si el par afectado corresponde a la primera o a la última ocurrencia de dicho par. Finalmente, se le notificará a **Max_heap** que la frecuencia de los pares afectados ha disminuido en una unidad.

2. Los nuevos pares que se han creado una vez hecho el reemplazo, y cómo su creación influye en la lista y las estructuras de datos. Por ejemplo, tomando nuevamente la secuencia "1 2 3 4 5", si reemplazamos el par " 3 4" por 28, la cadena resultante sería "1 2 28 5", por lo tanto, los nuevos pares creados son "2 28" y "28 5". En consecuencia, si la **lista** ya contenía ocurrencias de los pares creados, como por ejemplo "2 28(anterior) 3 2 28(nuevo)", la ocurrencia anterior debe conectarse con la nueva ocurrencia creada mediante los punteros **prev_oc** y **next_oc**. Si el par ya se encontraba dentro del **mapa,** debemos establecer la nueva ocurrencia como última ocurrencia de ese par y aumentar su frecuencia dentro del **Max_heap**. Si no, debemos crear una nueva instancia en el **mapa** con el par nuevo, además de insertar el par dentro del **Max_heap** con la clave 1.

Una vez reemplazado el par con mayor frecuencia y se haya actualizado la información en el **mapa** y el **Max_heap,** se procederá a aumentar el parámetro **sigma** en una unidad y nuevamente, se solicitará al **Max_heap** el par con mayor frecuencia dentro de la lista, repitiendo el proceso de reemplazo. El algoritmo terminará cuando no haya ningún par de números que se repita dos veces, es decir, cuando la clave que otorga el **Max_heap** al solicitar el par con mayor frecuencia sea igual a 1.

2.2.3 Implementación

Clase Advanced_Solution: Es la que interactúa con la aplicación main y la encargada de realizar el algoritmo repair.

Métodos principales:

- **frequency and node:** mediante la clase iterador, recorre toda la **lista** contando todos los pares existentes dentro de ésta y los inserta dentro del mapa. Si se encuentra un par que ya estaba dentro del **mapa** se declarará al par encontrado como última ocurrencia. También se aprovecha esta ocasión para conectar doblemente la dirección de la última ocurrencia del par con la dirección del par recién encontrado mediante los punteros **prev_oc** y **next_oc.** Por ejemplo, para la secuencia **"1 2(último) 3 1 2(encontrado)"**, el par **"1 2(encontrado)"** sería la última ocurrencia y dirección de **"1 2(último)"**. **next_oc** correspondería a la dirección de "1 2(encontrado)". Análogamente, dirección de **"1 2(encontrado". prev_oc** ahora es la dirección de "**1 2(último)"**.

- **wasInMap:** función booleana encargada de insertar un **nuevo par** creado mediante el previo reemplazo de un par, con su respectiva dirección. En el caso de que el par ya se encuentre dentro del mapa debemos conectar las ocurrencias de los pares mediante las direcciones: **next_oc** y **previous_oc**, además de establecer el nuevo par como última ocurrencia, para finalmente retornar **true**, puesto que se ha encontrado una ocurrencia en el mapa. Caso contrario, se insertará el **nuevo par** dentro del mapa y se retornará **false**, ya que no existía previamente ese par dentro del mapa.

- **update ocs map:** restablece la primera y última ocurrencia de los pares afectados al realizar un reemplazo, con el objetivo de que siempre se almacene correctamente dicha información.

- **change list:** Uno a uno, va cambiando las ocurrencias del par con mayor frecuencia dentro de la **lista**. Se solicita al **mapa** la dirección de la primera ocurrencia del par para cambiar todas las ocurrencias del par dentro de la lista mediante el siguiente procedimiento:
    - recolectamos toda la información relevante al realizar el reemplazo de un par por un parámetro **sigma** esto es, los pares de la derecha e izquierda que fueron afectados al realizar el cambio, junto con la dirección de la **primera** ocurrencia de los pares, más los nuevos pares que fueron creados con sus respectivas direcciones.
    - para los pares derecho e izquierdo que fueron afectados al realizar el cambio, debemos actualizar la información de las primeras y últimas ocurrencias dentro del **mapa** mediante la función **mod ocs map,** junto con disminuir en una unidad la clave de esos pares en **Max_heap.**

- para los pares derecho e izquierdo que fueron creados al realizar el cambio, debemos verificar si ese par ya se encontraba dentro de **mapa** mediante la función **wasInMap.** En caso afirmativo, se aumentará la clave en una unidad dentro de **Max_heap,** y si no, se precederá insertar los nuevos pares dentro de éste.

Max_Heap:

Principales métodos públicos:

- **Insert:** inserta los elementos dentro del vector, es decir, el par junto con su respectiva frecuencia, más la clave, la que servirá para ordenar los elementos dentro del heap.

- **Remove Max:** elimina el par con la mayor clave dentro del mapa, provocando la reorganización del heap mediante la función downheap.

- **Mod Key:** modifica la clave del par solicitado dentro del max_heap provocando que suba o baje de nivel dentro del árbol.

- **Up heap:** intercambia el contenido del nodo hijo por el del padre en el caso de que la clave del hijo sea mayor que la del padre. Este proceso termina cuando la clave del padre es mayor que la del hijo
- **Down heap:** Función análoga a **Up heap** con la diferencia que el nodo padre baje de nivel en el caso de que su clave sea menor a la de algún hijo.

Clase DoublyLinkedLists: Encargada de almacenar la cadena de entrada de entrada y las posteriores modificaciones que se le realizan.

métodos principales:

- **Change and get Info:** Recibe por parámetros la dirección del par que se cambiará y un valor **(sigma+1)** por el cual tiene que ser cambiado. Además, esta debe recolectar toda la información relevante al realizar una modificación, es decir, los nodos Right-left afectados y los nodos right-left nuevos (**si es que existen**). Para poder realizar este procedimiento tomamos en cuenta 5 operaciones base:

  - **Primera operación:** Crear un reemplazo y hacer que apunte a las direcciones que apunta el antiguo par.
  - **Segunda operación:** Redireccionar los punteros del siguiente y el anterior para que apunten al reemplazo.
  - **Tercera operación:** mediante la variable collector, se almacenará toda la información relevante especificada anteriormente.
  - **Cuarta operación:** Solicitar la ayuda de la función **fixpointers** para que ordene los punteros siguiente y anterior de los pares afectados.

  - **Quinta operación:** Eliminar los nodos inservibles y disminuimos el tamaño de la lista doblemente enlazada.

Aparte de las operaciones, se debe tener otros tres aspectos en cuenta: Cuando el par se encuentra al principio de la lista, al medio y al final. De esta manera aseguramos que cada uno de los punteros queden apuntando a las posiciones correctas.

- **Fix pointers:** Ordena los punteros de las ocurrencias siguiente y anterior (si es que existen) de los pares que fueron afectados una vez hecho el reemplazo del par por el parámetro sigma. También esta función se encarga de entregar la información correcta de la primera ocurrencia de estos pares, de esta manera el mapa siempre se mantiene actualizado con la primera ocurrencia del par. El funcionamiento de este método es el siguiente.

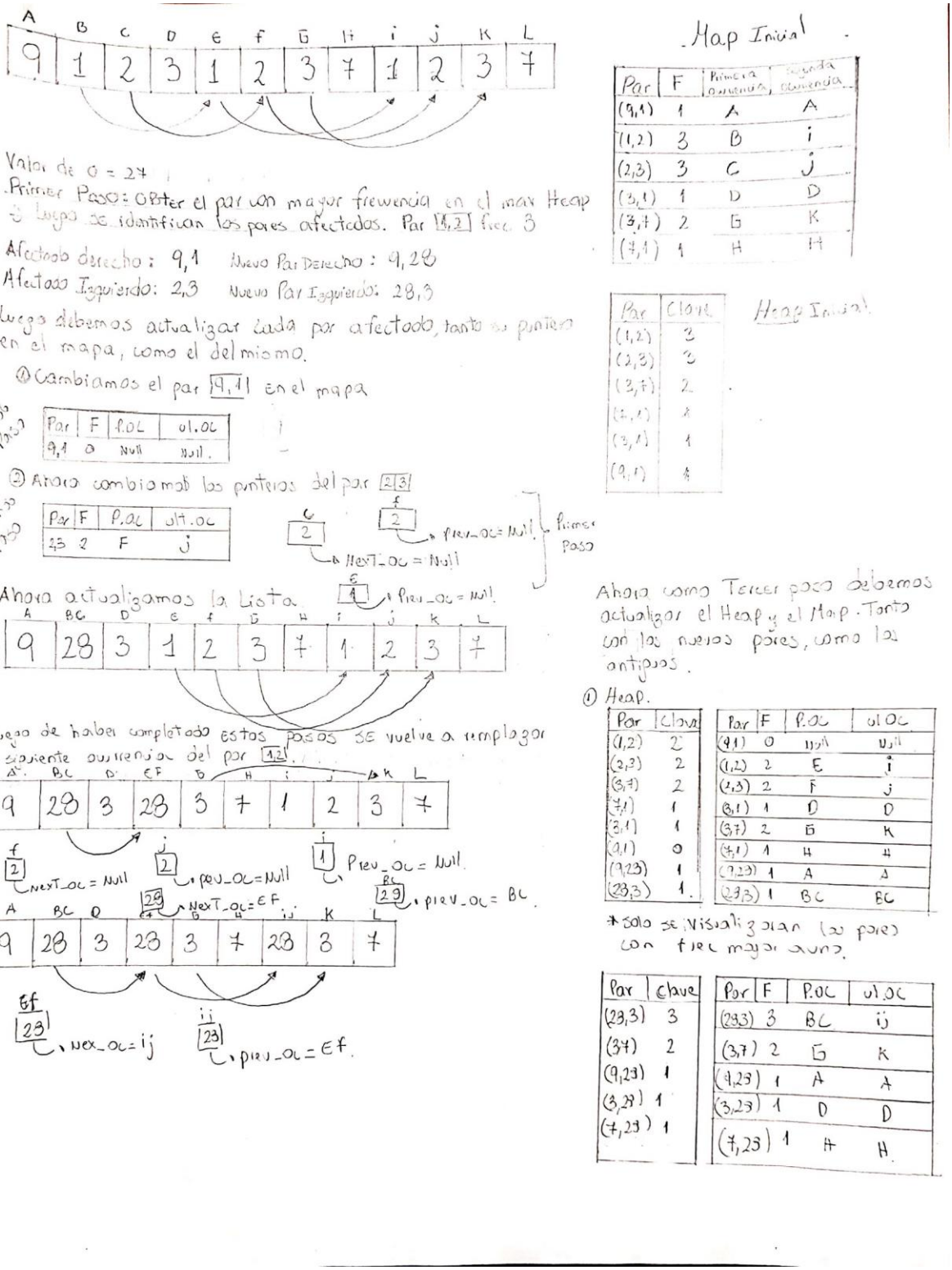
    - **Primera operación:** Se busca la primera ocurrencia de los punteros utilizando un while, si se encuentra una ocurrencia previa, esta será almacenada y entregada al mapa para actualizar su respectivo puntero. Existe un condicional para el par formado por el segundo elemento del par eliminado y su siguiente, éste solo entregará su ocurrencia anterior si es diferente al primer elemento del par eliminado y su anterior, para así no tener un puntero inexistente.

    - **Segunda operación:** Se conectan la ocurrencia anterior y siguiente de cada par afectado, esto es solo si ninguna de esta ocurrencia sea algún par afectado o eliminado. Esta conexión se realiza debido a que el par que fue afectado ya no corresponderá a alguna ocurrencia de estos pares a unir.

    - **Tercera operación:** Se vuelven nullptr la ocurrencia siguiente y anterior del par afectado que se compone por el previo al primer elemento del par a eliminar y el mismo primer elemento. Esto evita fallos como “segmentation fault”, debido a que se mantiene en orden cada puntero, y se cambia null aquellos que ya no existen.

.

Ejemplo solución avanzada.



Map Inicial

Par	F	Primera ocurrencia	Segunda ocurrencia
(9,1)	1	A	A
(1,2)	3	B	i
(2,3)	3	C	j
(3,1)	1	D	D
(3,7)	2	G	K
(7,1)	1	H	H

Par	Clave
(1,2)	2
(2,3)	2
(3,7)	2
(7,1)	1
(3,1)	1
(9,1)	0

Heap Inicial

Ahora como Tercer paso debemos actualizar el Heap y el Map. Tanto con los nuevos pares, como los antiguos.

① Heap.

Par	Clave
(1,2)	2
(2,3)	2
(3,7)	2
(7,1)	1
(3,1)	1
(9,1)	0
(9,23)	1
(23,3)	1

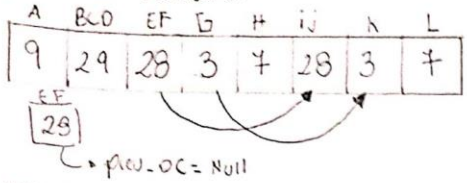
Par	F	P.OC	ul.OC
(9,1)	0	Null	Null
(1,2)	2	E	i
(2,3)	2	F	j
(3,1)	1	D	D
(3,7)	2	G	K
(7,1)	1	H	H
(9,23)	1	A	A
(23,3)	1	BC	BC

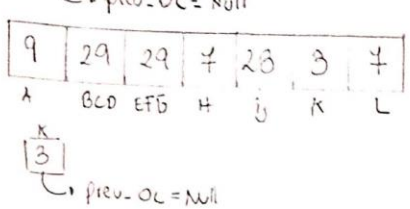
* Solo se visualizan los pares con frec mayor cero.

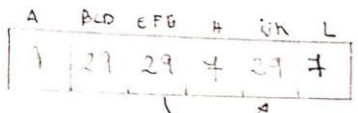
Par	Clave
(23,3)	3
(3,7)	2
(9,23)	1
(3,23)	1
(7,23)	1

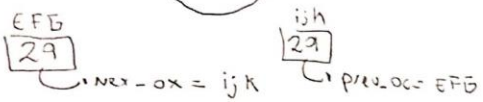
Par	F	P.OC	ul.OC
(23,3)	3	BC	ij
(3,7)	2	G	K
(9,23)	1	A	A
(3,23)	1	D	D
(7,23)	1	H	H

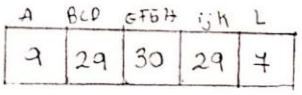
Luego de hacer el cambio respectivo del primer par, pasaremos hacer el remplazo del segundo par mas frecuente.

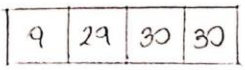












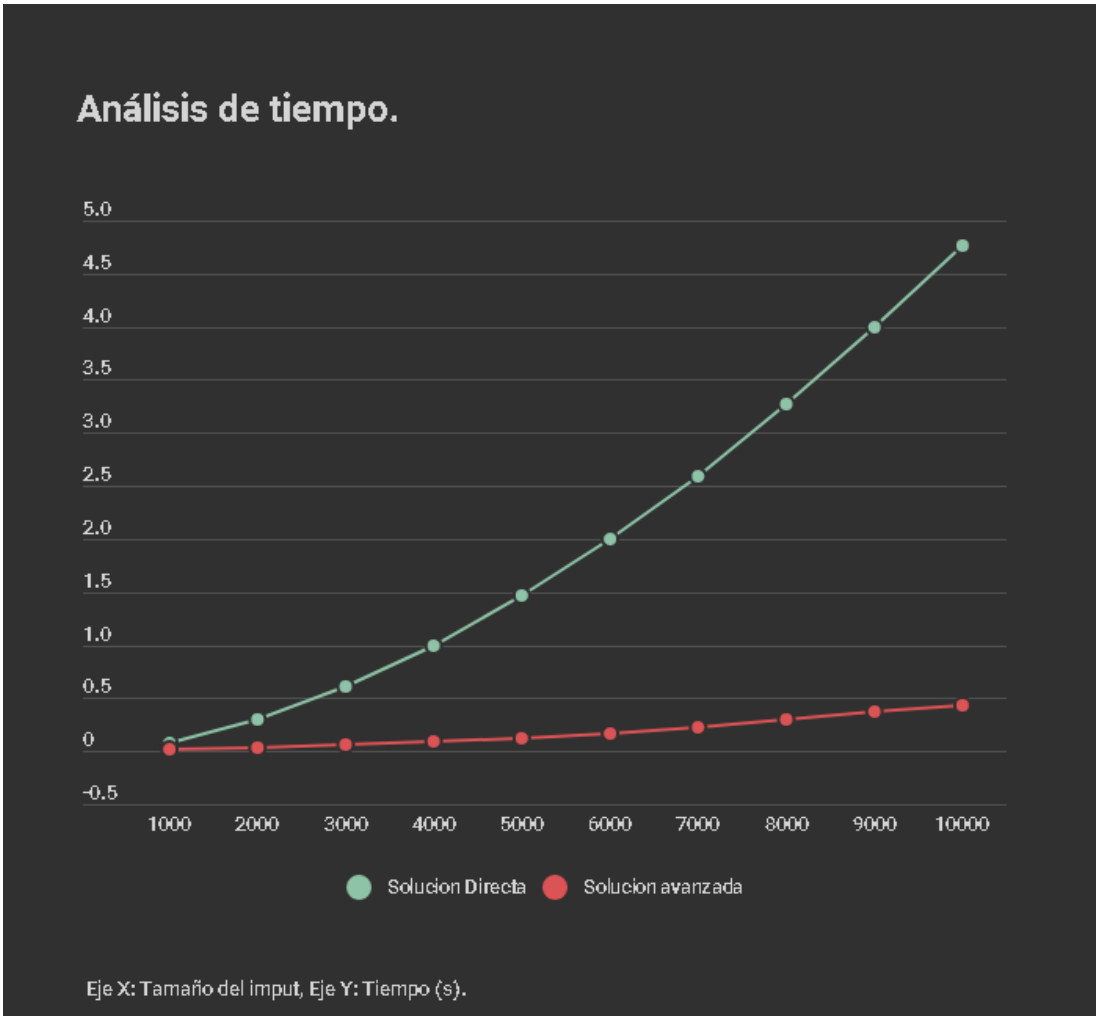
Fin del proceso

* El par |9,28| en el map tendra punteros Null y su frecuencia sera 0, de igual manera que el Heap.
  Lo mismo ocurre con el par |3,28|.
* Tambien son creados los pares |9,29| y |29,28|, estos se almacenan en el Heap y en el mapa
* El par |29|28| en el map queda con punteros a null y frecuencia 0, de igual manera que en el Heap
* El par |3|7| actualiza su primera ocurrencia siendo |3| su primera y ultima ocurrencia. Disminuye su frec
* Se crean y agregan los nuevos nodos |29|29| y |29,7|.
* Los pares |7,28| y |3|7| son actualizados en el mapa apuntando a null y su frecuencia se vuelve 0 en el mismo y en el Heap
* Se crea el nuevo par |7|29| y se actualizan los punteros del par |29|7| en ellos mismos y en el mapa.
* Los pares |29,28| y |7,29| cambian sus punteros a null y su frecuencia a 0. Tanto en el Heap como en el mapa.
* Se crean los nuevos pares |29|30| y |30|27|.
* Los punteros del par |30|29| se vuelven null, y su frec disminuye a 0.
* Se crea el nuevo par |30|30| en el mapa y el Heap.

Fin del Proceso.

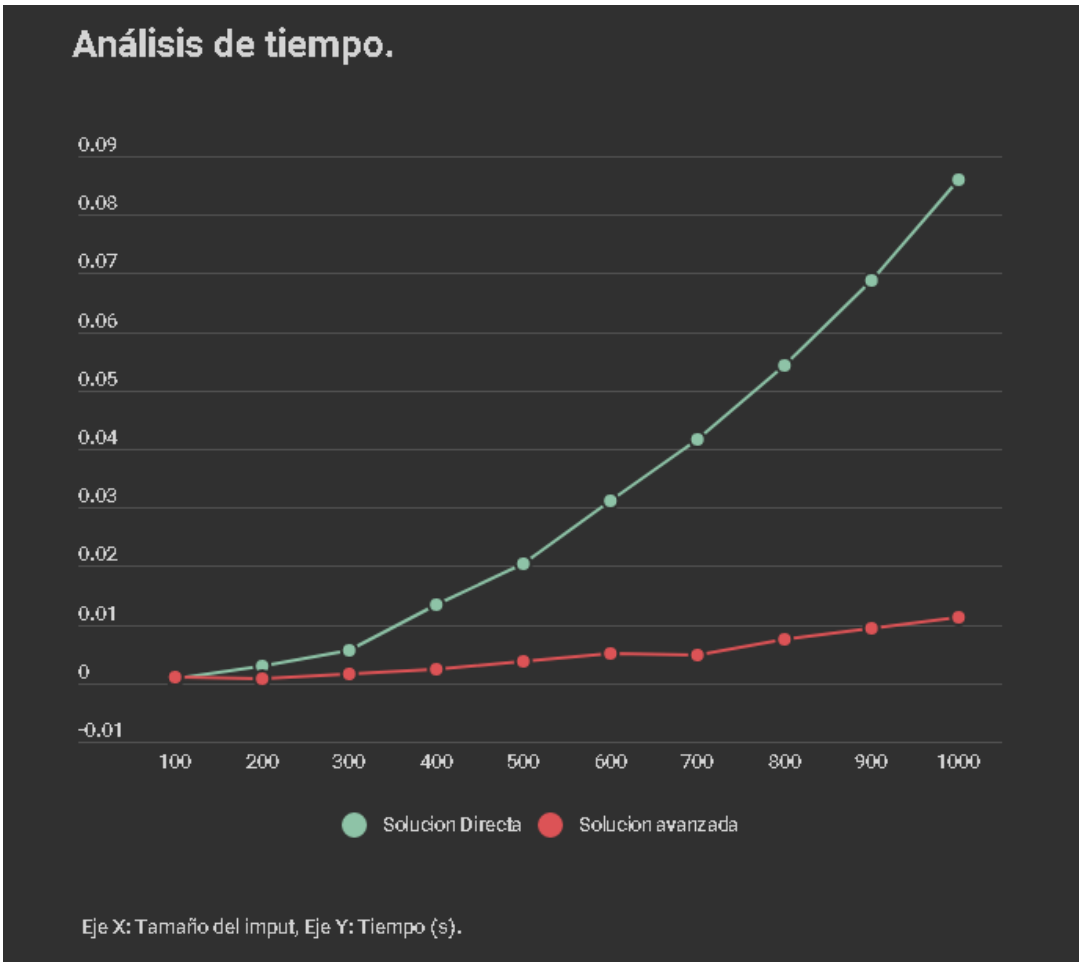
Experimentación

La parte de experimentación se realizó con el objetivo de medir el tiempo de ejecución que le tomó a la clase a ambas soluciones comprimir las cadenas de entrada, este tiempo fue calculado sin considerar el tiempo que se demora en eliminar todos los nodos para volver a insertarlos. Aquí se puede observar el gráfico con su respectiva tabla resultante:



Conclusiones: Con los datos mostrados anteriormente, podemos destacar la eficiencia de la Solución avanzada por encima de la Solución directa, este aumento de eficiencia se debe a la cantidad de punteros agregados en esta misma, ya que al poseer las direcciones de la primera y última ocurrencia en el map, y a su vez poseer en cada par el puntero a su ocurrencia siguiente y anterior, facilita el pronto acceso a dichos pares, con esto podemos modificar de forma veloz y eficiente cada par. Al tener estos punteros "extras" el algoritmo se vuelve más complejo, ya que se debe cuidar la integridad de cada puntero, ya que estos deben apuntar en la dirección correcta de cada par.

La solución avanzada como podemos apreciar es muy ineficiente, ya que a mayor cantidad de datos se vuelve más lenta, esto concluye que su curva de crecimiento es bastante pronunciada en comparación a la solución avanzada. Todo lo dicho anteriormente se basa en una cantidad de datos equidistante, estos datos aumentan de mil en mil hasta llegar a diez mil, lo que genera la siguiente pregunta **¿La solución directa siempre es ineficiente?¿Qué ocurre en un dominio más pequeño?** Con estas preguntas a continuación se muestra el siguiente gráfico.



Con el grafico recien mostrado, podemos concluir que la solución avanzada siempre será más eficiente que la solución directa, aun cuando el dominio de datos sea pequeño.

Conclusión

A partir de lo desarrollado en este informe, se logró definir y caracterizar ambas soluciones de una manera eficiente, por lo que los objetivos fueron cumplidos en su totalidad.

Además, el equipo ha aprendido bastante sobre el uso de la clase Map, Heaps y listas doblemente enlazadas, para hacer implementaciones mucho más eficientes en tiempo que las demás implementaciones basadas en arreglos.

Finalmente, se valora la realización de este proyecto, comprendiendo que, sea cual sea el futuro profesional de cada integrante, siempre es necesario conocer y analizar el problema para resolverlo de manera adecuada, además de analizar sus respectivos tiempos de ejecución, con los que se familiarizan desde hoy y para siempre.