

Compresión de datos distribuida basada en clustering

Guillermo Andrés Navarro Giglio

Memoria para optar al título de Ingeniero Civil en Informática y
Telecomunicaciones

Profesor guía
Francisco Claude-Faust

Comité
Roberto Konow
Jonathan Frez

Julio, 2015

UNIVERSIDAD DIEGO PORTALES
Facultad de Ingeniería
Escuela Ingeniería Informática

Compresión de datos distribuida basada en clustering

Guillermo Andrés Navarro Giglio

Memoria para optar al título de Ingeniero Civil en Informática y
Telecomunicaciones

Francisco Claude-Faust
Profesor guía

Roberto Konow
Comité

Jonathan Frez
Comité

Julio, 2015

Dedico esta memoria a todas las personas.

Contenido

Lista de figuras	v
Lista de tablas	vii
Resumen	ix
Capítulo 1 Introducción	1
1.1. Antecedentes generales	1
1.2. Conceptos Básicos	2
1.2.1. String o cadena de caracteres	2
1.2.2. Colección de documentos	2
1.2.3. Entropía de un texto	3
1.2.4. Punto	3
1.2.5. Medida de similitud	3
1.2.6. Agrupamiento	3
1.3. Objetivos	4
1.3.1. Objetivo específicos	4
1.4. Organización de la memoria	4
Capítulo 2 Estado del arte	5
2.1. Algoritmo de agrupamiento	5
2.2. Medida de similitud para Cadenas de caracteres	7
2.3. Metodos de compresión de datos	10
Capítulo 3 Algoritmo de agrupamiento propuesto	11
3.1. Proceso de agrupamiento	11
3.2. Algoritmo de agrupamiento implementado	11
3.3. Mejoras del Algoritmo de agrupamiento implementado	15
3.3.1. Paralelización	15
3.3.2. Estrategia Greedy	18
3.3.3. Análisis	20
Capítulo 4 Experimento	21
4.1. Diseño	21
4.2. Resultados	23
4.2.1. Resultados iniciales	23
4.2.2. Resultados Finales	25

Capítulo 5 Conclusiones	29
Referencias bibliográficas	31
Anexo A CODIGO ALGORITMO DE AGRUPAMIENTO BASE	35
Anexo B CODIGO ALGORITMO DE AGRUPAMIENTO ALEATORIO	39

Lista de figuras

4.1. Tamaño de los resultados cambiando la cantidad de muestras.	26
4.2. Legos.	27

Lista de tablas

2.1. Ejemplo LZ78.	10
4.1. Colecciones.	22
4.2. Resultado algoritmo de agrupamiento aleatorio con 10 Grupos.	24
4.3. Distribución grupos.	25
4.4. Tabla de resultados cambiando las muestras.	27

Resumen

Hoy en día la información crece rápidamente con nuevos contenidos provenientes de páginas Web, redes sociales, aplicaciones móviles entre otros. Cada vez es más difícil manejar esta gran cantidad de datos, demandando mayores recursos para las empresas y transformándose en un importante desafío en el futuro. Para esto, se debe buscar nuevos mecanismos que permitan de manera eficiente almacenar esta gran cantidad de datos.

Esta memoria intenta buscar un mecanismo eficiente de distribuir una colección de datos para maximizar la compresibilidad y mantener un balance en lo que respecta al uso de recursos en múltiples servidores. Primero, se explica la importancia de los documentos versionados en la actualidad y por que se hace necesario distribuir los datos, luego se implementa un mecanismo de distribución de la colección de datos y finalmente se forma una conclusión de los resultados obtenidos.

Capítulo 1

Introducción

1.1. Antecedentes generales

¿La información tiene límites? ¿Somos capaces de guardar toda esta información? Actualmente la información crece a pasos agigantados; cada día aparecen nuevos contenidos provenientes de páginas Web, redes sociales, aplicaciones móviles y de nuevas tecnologías como Internet de las cosas, que son capaces de generar una gran cantidad de información.

La información alojada en los sitios, aplicaciones, redes sociales u otros puede cambiar con el paso del tiempo y, en algunos casos, es necesario guardar el historial de cambios de esta. Ejemplos de aplicaciones que tienen estos requerimientos son: Wikipedia, una enciclopedia online colaborativa [1], y Git, un manejador de versiones utilizado principalmente para almacenar código de fuente [2]. Herramientas tradicionales de almacenamiento y versionamiento no son capaces de manejar una gran cantidad de datos de manera eficiente o, incluso, práctica. Además que generan costos elevados, transformándose en un verdadero desafío para las empresas.

Frente a este escenario, almacenar los datos de una aplicación masiva es cada vez menos viable usando un solo computador. Empiezan a aparecer nuevas soluciones a este problema, por ejemplo, la empresa Backblaze desarrolla una aplicación que genera una copia de seguridad en la nube a muy bajo costo; usa una granja de servidores para almacenar los datos, repartiendo la información entre un conjunto de computadores que forman un sistema distribuido [3]. Se deben buscar nuevos mecanismos que logren de manera eficiente almacenar los datos aprovechando los recursos, lo que a la larga se traduce en una disminución en los costos de las empresas.

Esta memoria tiene como objetivo estudiar una alternativa basada en clustering para repartir la información de manera inteligente entre varios computadores, haciendo uso de una métrica de distancia en base al contenido de la información para luego comprimirla, y así, de esta forma, mejorar la compresión, idealmente manteniendo un balance en la carga de almacenamiento [4]. Clustering es una técnica que genera agrupaciones de objetos según un criterio, con esto se pretende separar los datos de tal manera que en cada agrupación se tenga un espacio parecido con respecto a los demás y que al utilizar cierto compresor sea mucho más eficiente en término de espacio que solamente separar los datos de manera aleatoria.

1.2. Conceptos Básicos

En este apartado se explican conceptos básicos que se utilizarán más adelante y que ayudará a la comprensión del trabajo realizado.

1.2.1. String o cadena de caracteres

Una cadena de caracteres $S = \{s_1, s_2 \dots, s_l\}$ es una secuencia de símbolos de algún alfabeto Σ en particular de un tamaño l .

1.2.2. Colección de documentos

Una colección de documentos es un conjunto de ϱ documentos, representado como $C = \{d_1, \dots, d_\varrho\}$, donde los documentos son cadenas de caracteres S .

1.2.3. Entropía de un texto

La compresión de datos es la reducción del volumen de datos, sin embargo, comprimir datos tiene un límite definido por la entropía, que se refiere a la información nueva y esta se define como la cantidad total de datos menos su redundancia.

La entropía empírica de orden cero H_0 es el número promedio de bits necesarios para representar cada símbolo del texto T de tamaño n , está definida como[5]:

$$H_0(S) = \sum_{i=0}^{\sigma-1} \frac{n_i}{n} \log \frac{n}{n_i} \quad (1.1)$$

donde el alfabeto $\Sigma = \{c_0, \dots, c_\sigma\}$ de tamaño σ y n_i es el número de ocurrencias de caracteres c_i en T .

1.2.4. Punto

Un Punto se define como un vector de n dimensiones, $P = \{v_1, v_2, \dots, v_n\}$. Un punto $P \in R^n$ se dice que es un punto de cluster para un subconjunto A si para cada $\delta > 0$, δ vecindad de P , tenemos $B(P; \delta) \cap A \neq \emptyset$, donde $B(P; \delta) = \{x \in R^n \mid \|x - a\| < \delta\}$.

1.2.5. Medida de similitud

La medida de similitud es una función real que cuantifica la similitud entre dos puntos, es lo opuesto a la medida de distancia. Se define la medida de similitud para un punto x_i e y_j como $s(i, j) = -\|i - j\|^2$ donde $\|x - y\|^2$ es la distancia euclidiana al cuadrado [6]. Ver capítulo 2 para una explicación más detallada.

1.2.6. Agrupamiento

Agrupamiento es el acto de formar k grupos de puntos $M = \{P_1, \dots, P_\vartheta\}$ de ϑ puntos, siguiendo alguna métrica como puede ser la medida de similitud o de distancia, ver capítulo 2 para una explicación más detallada.

1.3. Objetivos

El objetivo general de esta memoria es maximizar la compresibilidad de la colección agrupándolos en función de su similitud de contenido, para esto se quiere implementar y evaluar un mecanismo que represente colecciones versionadas de datos de forma distribuida.

1.3.1. Objetivo específicos

- Realizar un estudio de mecanismos de clustering para agrupar conjuntos de cadenas de caracteres.
- Generar un repositorio de datos distribuido basado en clustering simulando un ambiente de varios computadores.
- Medir la efectividad de usar clustering previo a la compresión para el almacenamiento distribuido de datos, en términos de la compresibilidad de la colección.

1.4. Organización de la memoria

La organización de la memoria se define en los siguientes capítulos:

1. En el primer capitulo, Capitulo 2, se realiza un estudio del estado del arte de los algoritmo de agrupamiento más utilizados y las medidas de distancias para poder calcular la similitud entre cadenas de caracteres.
2. En el siguiente Capitulo, Capitulo 3, se muestra el análisis y diseño de una propuesta de un algoritmo de agrupamiento frente al problema que se quiere abordar en la memoria.
3. En el Capitulo 4 se presenta el diseño de los experimentos y los resultados de la ejecución del algoritmo propuesto. Además se realiza una comparación entre las distintas versiones del algoritmo propuesto y un algoritmo de agrupamiento aleatorio.
4. Por ultimo, en el Capitulo 5 se realiza las conclusiones que se obtuvo a partir de los resultados del Capitulo 4.

Capítulo 2

Estado del arte

2.1. Algoritmo de agrupamiento

El algoritmo de agrupamiento es un proceso a través del cual se tiene un conjunto de puntos y se crean grupos de puntos a partir de una medida de similitud, en la mayoría de los algoritmos de agrupamiento se asume un espacio euclidiano.

El espacio euclidiano es un espacio geométrico que se cumplen los axiomas de Euclides, la cual define tres reglas entre las distancias de dos puntos en el espacio euclidiano:

- La distancia entre los puntos nunca es negativa y solamente es 0 consigo mismo.
- La distancia es simétrica, es decir, no importa si se calcula la distancia del punto x a y o de y a x .
- La distancia obedece a la desigualdad del triángulo; la distancia de x a y a z no puede ser menor a la distancia de x a z .

Los algoritmos de agrupamiento se pueden dividir en dos grupos: jerárquicos y no jerárquicos. Los primeros pueden ser aglomerativos o divisivos.

Cuando es aglomerativo, cada punto en el espacio euclidiano representa un grupo y en cada iteración se unen los grupos hasta llegar a los números de grupos deseados; en cambio cuando es divisivo todos los puntos se encuentran en un solo grupo y en cada iteración se dividen los grupos. El siguiente algoritmo 1 muestra el algoritmo de agrupamiento aglomerativo. Primero se debe determinar cuándo detener el algoritmo, una opción puede ser hasta tener el número deseados de grupos.

Algoritmo 1 Algoritmo Jerárquico aglomerativo

```
while No es tiempo para detenerse do  
    Elegir los dos grupos más cercanos;  
    Unir ambos grupos en uno sólo;  
end while
```

Este algoritmo de agrupamiento es muy lento en caso de que la colección de datos sea muy grande, existen algoritmo de agrupamiento para colecciones grandes, como CURE [7].

CURE asume un espacio euclidiano, el algoritmo de CURE empieza tomando una pequeña muestra de la colección principal y usa cualquier algoritmo de agrupamiento que asuma el espacio euclidiano sobre la muestra, por ejemplo, los algoritmo jerárquicos serían una buena opción. Luego, selecciona en cada grupo formado en la muestra, una pequeña cantidad de puntos que se llamarán “puntos representativos”, estos serán los que estén más alejados del grupo y entre ellos mismos, la cantidad de puntos representativos es libre de elegirse. Después los puntos representativos se mueven al centro del grupo un porcentaje, por ejemplo, 10 %.

La siguiente etapa consiste en unir los grupos que tengan un punto representativo cerca de un punto representativo de otro grupo, la distancia entre ambos puntos representativos para unir ambos grupos es libre de elegirse. Por último los puntos de la colección se comparan con los puntos representativos y se asigna al grupo que tenga la menor distancia.

En los algoritmo de agrupamiento no jerárquicos, *k-means* [7], es uno de los más utilizado y simple de entender.

K-means, como la mayoría de los algoritmos de agrupamiento, asume un espacio euclidiano y también asume el número de grupos conocidos. El número de grupos se puede determinar de distintas maneras, como por ejemplo, por prueba y error o con conocimiento previo de las características de las observaciones.

Dado k grupos se generan k centros de grupos o centroides iniciales, los centroides son vectores de las medias de las características de todas las observaciones dentro de cada grupo. Los centroides se pueden asignar de distintas maneras, una opción es asignar observaciones aleatoriamente de un conjunto de observaciones. Luego se itera los siguientes pasos:

Algoritmo 2 Algoritmo K-means

```
C conjunto de  $k$  centroides;
while true do
  Para cada observación calcular la distancia a todos los  $C_i$ ;           #
   $0 < i < k$ 
  Las observaciones se asignan al  $C_i$  con la media más cercana;
   $C_i$  se recalculan las medias con las nuevas observaciones agregadas;
  if Todos los  $C_i$  no cambian then
    Termina
  end if
end while
```

Se itera hasta que converja, es decir, ya no se asignan nuevas observaciones a los grupos o los centroides ya no se mueven.

La mayoría de los algoritmos de agrupamiento asumen un espacio euclidiano para medir la similitud entre los vectores, pero en el caso de las cadenas de caracteres no son puntos que se puedan representar en el espacio euclidiano. Para poder solucionar este problema se debe buscar un mecanismo para medir la similitud entre las cadenas de caracteres. Existen varias medidas de similitud para determinar la similitud entre cadenas de caracteres, en la siguiente sección se nombran algunas de las más utilizadas junto con la medida de similitud que se utilizó en la implementación del algoritmo de agrupamiento.

2.2. Medida de similitud para Cadenas de caracteres

Uno de los puntos más importante al momento de implementar un algoritmo de agrupamiento es la medida de similitud entre los puntos, en este caso los puntos representan las cadenas de caracteres y se debe buscar una medida de similitud que logre una buena calidad, esto significa que se puedan representar las cadenas de caracteres similares con una distancia pequeña y las cadenas de caracteres que no son similares en distancias mayores.

Existen varios métodos para calcular la similitud entre una cadena de caracteres y otra como por ejemplo:

- Distancia de Edición: Se tiene dos cadenas de caracteres A y B . La distancia de edición es la cantidad mínima de insertar, sustituir o eliminar necesarios para transformar A en B . Con distancia de edición se logra una buena calidad en la similitud, pero presenta una desventaja: el algoritmo de edición de distancia requiere de tiempo de $O(n \times m)$, donde n y m son el largo de ambas secuencias de strings. Por ejemplo para las cadenas “abracadabra” y “alabaralabarda”, se necesitan 7 operaciones. La fórmula 2.1 calcula la distancia de edición d_{mn} [8]:

$$\begin{aligned}
d_{i0} &= \sum_{k=1}^i w_{\text{del}}(b_k), & \text{for } 1 \leq i \leq m \\
d_{0j} &= \sum_{k=1}^j w_{\text{ins}}(a_k), & \text{for } 1 \leq j \leq n \\
d_{ij} &= \begin{cases} d_{i-1,j-1} & \text{for } a_j = b_i \\ \min \begin{cases} d_{i-1,j} + w_{\text{del}}(b_i) \\ d_{i,j-1} + w_{\text{ins}}(a_j) \\ d_{i-1,j-1} + w_{\text{sub}}(a_j, b_i) \end{cases} & \text{for } a_j \neq b_i \end{cases} & \text{for } 1 \leq i \leq m, 1 \leq j \leq n.
\end{aligned} \tag{2.1}$$

donde $a = a_1 \dots a_n$ y $b = b_1 \dots b_m$.

- Jaccard: Es una medida de similitud que está definida por el tamaño de la intersección de dos secuencias dividido por el tamaño de la unión de ambas secuencias, un ejemplo en la medida de similitud entre las secuencias “night” y “nacht” es de 0.3. Se tiene el conjunto S y T la fórmula para determinar el similitud jaccard es 2.2 [7]

$$Jaccard = \frac{(S \cap T)}{(S \cup T)}, \tag{2.2}$$

- Distancia Hamming: Se tiene dos cadenas de caracteres de igual tamaño y se calcula cantidad de sustituciones necesarias para transformar una cadena de caracteres en otra. Por ejemplo “night” y “nacht” se necesita 2 sustituciones. La fórmula de Distancia Hamming es 2.3 [7]

$$D_h(s_i, s_j) = \sum_{k=1}^m \delta(s_{ik}, s_{jk}) \quad (2.3)$$

donde $\delta(x, y) = \begin{cases} 0 & \text{si } x = y \\ 1 & \text{si } x \neq y \end{cases}$

s_i y s_j son cadenas de caracteres y m es el largo de las cadenas de caracteres.

En esta memoria se quiere una medida de distancia basada en compresión, la medida de similitud implementada para el algoritmo de agrupación es una distancia de compresión utilizando algún método de compresión como lzma, gzip o bzip, ver 2.3. Se aplica la siguiente fórmula para obtener la *Similitud* 2.4

$$Similitud = \frac{(d_{1+2} - d_2)}{d_1}, \quad (2.4)$$

donde d_1 es el tamaño comprimido del documento más grande, d_2 es el tamaño comprimido del documento más pequeño, y d_{1+2} es el tamaño comprimido de la unión de d_1 y d_2 sin comprimir.

El valor de la variable d_{1+2} va a depender de que tan parecidas son las cadenas de caracteres d_1 y d_2 . Si las cadenas de caracteres tienen un grado de similitud la compresión será mucho más efectiva como puede ocurrir en el caso de documentos versionados, ya que se necesita un diccionario mucho menor para comprimir, al contrario ocurre cuando las cadenas de caracteres son muy distintas entre sí.

Entre más pequeño el valor de la variable *Similitud*, significa que ambas cadenas de caracteres son muy similares, y entre más grande los valores significa que las cadenas de caracteres son diferentes. En comparación con la distancia edición está obtiene una buena calidad de similitud, pero el tiempo de ejecución es lineal. Mejorando el tiempo $O(n \times m)$ de la distancia de edición. Esto hace una buena opción al momento de seleccionar una medida de similitud para grandes colecciones de datos. Esta medida de similitud como entrega solamente un valor R^n es de una sola dimensión.

Por ejemplo, si se utiliza el compresor LZ78 [5] en la secuencia $S_1 = 'abracadabra'$, $S_2 = 'abracadadah'$ y la suma $S_{1+2} = 'abracadabraabracadadah'$, al aplicar la fórmula 2.4 se obtiene el valor 0,71.

Ahora si cambiamos la segunda secuencia a $S_3 = 'casasyperro'$, la cual no tiene similitud con la primera secuencia, se obtiene el valor 0.77. De esta forma se observa que con menor similitud entre los strings, mayor es el valor, entonces al tener strings que son similares el valor se acerca al 0. Otros estudios, como el de Ming et al ya utiliza un metodo de distancia basado en la compresion para secuencias de ADN [9].

2.3. Metodos de compresión de datos

La compresión de datos es la reducción del volumen de datos. Existe dos tipos de compresores: la compresión sin pérdida y la compresión con pérdida.

En nuestro caso estudiaremos el caso particular LZ78 [10], el cual es un compresor sin pérdida basado en diccionario. Es un algoritmo greedy adaptativo. En el diccionario guarda un índice y un carácter, el índice entrega la posición de su prefijo de una secuencia y el carácter es el último de la subcadena. El algoritmo empieza recorriendo la cadena de texto desde el principio, carácter por carácter, revisa si el carácter nuevo ya se encuentra en el diccionario o si pertenece a una subcadena del diccionario, si ya se encuentra sigue con el siguiente carácter, en caso de que no se encuentre, ingresa al diccionario ese nuevo carácter seguido con el índice de su prefijo.

Por ejemplo, se tiene la cadena de texto $S = "abracadabra"$, el resultado de la compresión con LZ78 se muestra en la tabla 2.1.

Nº	S1
1	<0,a>
2	<0,b>
3	<0,r>
4	<1,c>
5	<1,d>
6	<1,b>
7	<3,a>

Tabla 2.1: Ejemplo LZ78.

Capítulo 3

Algoritmo de agrupamiento propuesto

3.1. Proceso de agrupamiento

El Proceso que se utilizó para agrupar una colección de datos se describe en los siguientes pasos:

1. Primero se recibe la colección de datos que puede ser cualquier cadena de caracteres, por ejemplo, secuencias de ADN o información de Wikipedia.
2. Se elige un algoritmo de agrupamiento y se ejecuta sobre la colección. Cuando termina de ejecutar, la colección se encontrará repartida en diferentes directorios que representan los grupos formados por el algoritmo de agrupamiento.
3. Por último, se comprime cada directorio utilizando algún método de compresión.

3.2. Algoritmo de agrupamiento implementado

El algoritmo de agrupación implementado para la distribución de las cadenas de caracteres es una variante del algoritmo CURE 2.1, que utiliza un algoritmo jerárquicos para formar los grupos iniciales.

A continuación se muestran las etapas que sigue el algoritmo implementado:

1. El algoritmo empieza con la selección de un algoritmo de agrupamiento jerárquico aglomerativo o diviso, ambas opciones son válidas, en este caso se elige el aglomerativo. Luego, se obtiene una muestra pequeña de la colección de datos, en lo posible la muestra debe ser lo suficientemente representativa de la colección de datos. Posteriormente, se calcula la distancia, utilizando alguna medida de distancia para cadenas de caracteres entre todas las muestras (ver 2.2). Si las muestras se mantienen el algoritmo es determinista, es decir, siempre entregará los mismos resultados, lo que implica una ventaja ya que puede replicarse en varias máquinas.

2. Con las distancias de todas las muestras se ejecuta el algoritmo de agrupamiento aglomerativo hasta obtener los grupos deseados. Por último se elige en cada grupo los puntos representativos, a diferencia de CURE que selecciona algunos puntos, en este caso los puntos representativos son todos los puntos del grupo. La ventaja de elegir todos los puntos como representativos es asegurar que al momento de asignar una cadena de caracteres a un grupo se elija la mejor opción pagando un costo en el tiempo de ejecución. En cambio, si se seleccionan algunos como puntos representativos puede existir la posibilidad de eliminar cadenas de caracteres con potencial para ser un punto representativo lo que puede significar que un documento se asigna a otro grupo, en consecuencia la calidad de los grupos formados por los puntos representativos puede empeorar.

3. Con los grupos contruidos, se asigna cada cadena de caracteres de la colección de datos al grupo que tenga el punto representativo con la mejor medida de distancia. Uno de los objetivos de la memoria es el balance en términos de la carga de almacenamiento de los grupos, para esto la cadena de caracteres, antes de ser asignado al grupo, se comprueba que 3.1

$$Grupo_i < \frac{C}{k}, \quad (3.1)$$

donde $Grupo_i$ es el tamaño en disco del grupo más cercano a la cadena de caracteres seleccionada, C es el tamaño en disco de la Colección de cadenas de caracteres y k es el número de grupos.

Con esto, al momento de asignar una cadena de caracteres a un grupo, se busca generar un balance en cada grupo, si el $Grupo_i$ es mayor, entonces se comprueba el siguiente grupo más cercano a la cadena de caracteres seleccionada, hasta encontrar un grupo que sea menor. Si bien se genera un balance en el espacio de memoria la comprobación se realiza antes de la compresión y puede ocurrir que al momento de la compresión no asegura un balance en todas las agrupaciones. También puede pasar que un grupo contenga documentos que fueron asignados no por la medida de distancia sino por la falta de espacio en los demás grupos, lo que podría provocar que el resultado de la compresión sea mayor al resto de los grupos.

A continuación se observa los pseudocódigo del algoritmo implementado, el algoritmo 3 muestra la función distancia que representa el resultado de la medida de distancia entre dos cadenas de caracteres y el algoritmo 4 muestra el algoritmo de agrupamiento propuesto en la memoria.

Algoritmo 3 Funcion DISTANCIA

Require: *String1*

Require: *String2*

- 1: $s1 \leftarrow COMPRESS(String1)$
 - 2: $s2 \leftarrow COMPRESS(String2)$
 - 3: $s12 \leftarrow COMPRESS(String1 + String2)$
 - 4: **return** $size(s12) - size(s2)/size(s1)$
-

Algoritmo 4 Algoritmo de agrupamiento propuesto

Require: *Sampling* = $\{d_1, \dots, d_n\}$ #

Muestra, donde n es la cantidad de cadenas de caracteres.

Require: *Collection* = $\{d_1, \dots, d_k\}$ #

Colección de datos, donde k es numero de cadenas de caracteres.

Require: *C* #

Número de grupos deseados.

- 1: $S \leftarrow \langle \rangle$
 - 2: **for each** $s1$ in *Sampling* **do**
 - 3: **for each** $s2$ in *Sampling* **do**
 - 4: $S \cup (DISTANCIA(Sampling[s1], Sampling[s2]), s1, s2)$
 - 5: **end for**
 - 6: **end for**
 - 7: Sort(S)
 - 8: $i = 0$
 - 9: **while** *Sampling* > C **do**
 - 10: $Sampling[S[i][1]] \cup Sampling[S[i][2]]$
 - 11: $i = i + 1$
 - 12: **end while**
 - 13: *Grupo* $\leftarrow \langle \rangle$ #
 - 14: Grupo lista de tamaño C
 - 15: **for each** d in *Collection* **do**
 - 16: $i = 1$
 - 17: **for** $s = 0$ to n **do**
 - 18: **if** $DISTANCIA(Sampling[s], d) < i$ **then**
 - 19: $i = DISTANCIA(Sampling[s], d)$
 - 20: **end if**
 - 21: **end for**
 - 22: $Grupo[i] \cup d$
 - 23: **end for**
-

3.3. Mejoras del Algoritmo de agrupamiento implementado

3.3.1. Paralelización

El tiempo de ejecución del algoritmo se hace importante cuando se manejan grandes volúmenes de datos. Una de las ventajas del algoritmo de agrupamiento implementado permite la paralelización, es decir, ejecutar varios procesos paralelos en los que se consume el mayor tiempo de ejecución. Esto ocurre cuando se calculan las distancias entre las muestras para generar los grupos iniciales y al calcular las distancias entre los documentos que serán asignados con las muestras. En el algoritmo 5 muestra las modificaciones necesarias para poder implementar el algoritmo con paralelización.

Algoritmo 5	Algoritmo de agrupamiento propuesto Paralelización	
Require:	$Sampling = \{d_1, \dots, d_n\}$	#
	Muestra, donde n es la cantidad de cadenas de caracteres.	
Require:	$Collection = \{d_1, \dots, d_k\}$	#
	Colección de datos, donde k es numero de cadenas de caracteres.	
Require:	C	#
	Número de grupos deseados.	
1:	$S \leftarrow \langle \rangle$	
2:	<i>PARALELIZAR</i> {	
3:	for each $s1$ in $Sampling$ do	
4:	for each $s2$ in $Sampling$ do	
5:	$S \cup (DISTANCIA(Sampling[s1], Sampling[s2]), s1, s2)$	
6:	end for	
7:	end for	
8:	}	
9:	Sort(S)	
10:	$i = 0$	
11:	while $Sampling > C$ do	
12:	$Sampling[S[i][1]] \cup Sampling[S[i][2]]$	
13:	$i = i + 1$	
14:	end while	
15:	$Grupo \leftarrow \langle \rangle$	#
	Grupo lista de tamaño C	
16:	<i>PARALELIZAR</i> {	
17:	$i = 0$	
18:	for $s = 0$ to n do	
19:	if $DISTANCIA(Sampling[s], d) > i$ then	
20:	$i = DISTANCIA(Sampling[s], d)$	
21:	end if	
22:	end for	
23:	$Grupo[i] \cup d$	
24:	}	

La mejora en los tiempos de ejecución está directamente relacionado con la cantidad de procesos que se quiera ejecutar, estos deben ser preferiblemente menor o igual a la cantidad de nucleos que cuenta el procesador de la máquina en la que se ejecutará el algoritmo. El tiempo de ejecución se divide por cada proceso adicional.

Otra manera para mejorar el tiempo de ejecución es seleccionando algunas cadenas de caracteres como puntos representativos, pero se debe buscar el modo que los puntos representativos realmente representen todas las cadenas de caracteres del grupo, para ver más en detalle las desventajas y ventajas de este método ver Capítulo 3.2. En el caso del espacio euclidiano es fácil de lograr, pero para cadenas de caracteres es muy difícil, ya que no asegura que representen todas las cadenas. Podría pasar el caso de que todas las cadenas de caracteres estén muy alejadas entre ellas, lo cual significa que todos deberían ser puntos representativos, por eso el algoritmo propuesto asegura todas las cadenas de caracteres del grupo como puntos representativos sacrificando tiempo de ejecución.

3.3.2. Estrategia Greedy

Una posible mejora que se puede emplear al algoritmo de agrupamiento propuesto es al problema que se genera cuando se asignan los documentos a un grupo con la restricción de un máximo de documentos asignados a cada grupo, utilizado para balancear la carga de estos. El algoritmo selecciona un documento y se asigna a un grupo, sin tomar en cuenta los documentos que serán asignados posteriormente. En consecuencia, el documento asignado a un grupo podría no ser la mejor opción frente a un posible candidato con una mejor distancia para ese mismo grupo.

Para solucionar este problema se puede utilizar la estrategia greedy, lo que significa que siempre toma la mejor opción local para lograr la mejor solución, aunque no implica que siempre llegue a la mejor solución. La mejora se implementó ocupando una cola de prioridad, basado en la distancias de cada documento a todos los grupos, para luego asignar los documentos en el orden que la cola de prioridad entrega.

El algoritmo 6 muestra las modificaciones necesarias para poder implementar esta mejora.

Algoritmo 6 Algoritmo de agrupamiento propuesto Estrategia Greedy

Require: $Sampling = \{d_1, \dots, d_n\}$ #

Muestra, donde n es la cantidad de cadenas de caracteres.

Require: $Collection = \{d_1, \dots, d_k\}$ #

Colección de datos, donde k es numero de cadenas de caracteres.

Require: C #

Número de grupos deseados.

1: $S \leftarrow \langle \rangle$

2: **for each** $s1$ in $Sampling$ **do**

3: **for each** $s2$ in $Sampling$ **do**

4: $S \cup (DISTANCIA(Sampling[s1], Sampling[s2]), s1, s2)$

5: **end for**

6: **end for**

7: $Sort(S)$

8: $i = 0$

9: **while** $Sampling > C$ **do**

10: $Sampling[S[i][1]] \cup Sampling[S[i][2]]$

11: $i = i + 1$

12: **end while**

13: $Grupo \leftarrow \langle \rangle$ #

Grupo lista de tamaño C

14: $Cola_{prioridad} \leftarrow \langle \rangle$

15: **for each** d in $Collection$ **do**

16: **for** $s = 0$ to n **do**

17: $Cola_{prioridad}.add(DISTANCIA(Sampling[s], d), d, s)$

18: **end for**

19: **end for**

20: **while** $Cola_{prioridad}$ **do**

21: $i = Cola_{prioridad}.pop()$

22: $Grupo[i[2]] \cup i$

23: **end while**

3.3.3. Análisis

Unos de los problemas del algoritmo propuesto es al tiempo de ejecución cuando las muestras son demasiadas. Si se tiene n documentos de largo m cada uno y s muestras, el tiempo de ejecución del algoritmo se divide en los siguientes pasos:

1. Pritmero se calcula las distancias entre las muestras, su tiempo de ejecución es $O(s^2 \cdot m)$.
2. Con la lista de las distancias entre las muestras se forman los grupos y su tiempo de ejecución es $O(s)$.
3. Luego se asignan los documentos calculandolos con todas las muestras, su tiempo de ejecución es $O(n \cdot s \cdot m)$.
4. Por ultimo se comprime, como el metodo de compresión puede cambiar su tiempo de ejecución sera de $O(k)$

Como resultado entre todos los pasos del algoritmo se obtiene el tiempo de ejecución de $O(s \cdot m \cdot (s + n) + s + k)$. Como el objetivo de la memoria está orientado al manejo de datos de gran volumen, es importante que el algoritmo propuesto pueda ejecutarse en tiempos razonables. La mayor parte del tiempo de ejecución del algoritmo propuesto se utiliza para calcular las distancias entre las cadenas de caracteres.

Con el algoritmo Greedy cambiaria el tiempo de ejecución en el paso 3 a $O(s \cdot m \cdot n \cdot \log n)$. Finalmente se obtiene un tiempo de ejecución de $O(s \cdot m \cdot (n \cdot \log n + s) + k)$.

Capítulo 4

Experimento

4.1. Diseño

Las pruebas se realizaron comparando dos tipos de algoritmos de agrupamiento:

- Algoritmo de agrupamiento propuesto
- Algoritmo de agrupamiento random

El algoritmo de agrupamiento random crea n grupos y asigna de manera uniforme y distribuida aleatoriamente cada cadena de caracteres a un grupo, a diferencia del algoritmo propuesto que luego de obtener las muestras es determinista. También mantiene el balance de espacio en memoria en cada Grupo.

Con ambos algoritmos se pretende demostrar que realizando los agrupamientos de manera inteligente se pueden obtener mejores resultados en términos de compresión, que agrupándolos aleatoriamente y mantener cierto balance en cada agrupación.

En el algoritmo de agrupamiento propuesto existen distintas variables que pueden determinar un buen agrupamiento de la colección, estas son:

- Cantidad de Grupos: es difícil determinar la cantidad exacta de grupos que se necesitan para lograr la mejor compresión. En este caso el número representa la cantidad de máquinas disponibles.

- **Tamaño de la muestra:** si la muestra es muy pequeña, es probable que no represente todos los tipos de grupos que se encuentra en la colección. En caso contrario, si la muestra es muy grande el tiempo de ejecución crece.
- **Medida de distancia:** la función implementada para medir la distancia permite cambiar el método de comprimir, en este caso se utiliza ZIP. También la librería de ZIP utilizada permite determinar el nivel de compresión de una cadena de caracteres, cambiando los valores en la medida de distancia, entre mayor sea el nivel de la compresión se obtiene una mejor calidad de la distancia pero más lento poder calcularla.

Las pruebas se realizaron en una colección versión en español de Wikipedia, las muestras se tomaron de manera uniformemente al azar, en UTF-8. Para cada documento seleccionado se obtienen todas sus versiones. Esto fue hecho usando la librería go-wikiparse y limpiado con Tika para obtener sólo el texto de los artículos. Se concatenan todos los documentos en uno sólo y se divide por bloques.

Colecciones	Tamaño Total (GiB)	Entropia(bits)	Nº Documentos	Tamaño docu- mento (MiB)
Wiki-ES	16.384	5.0831497879	16384	1

Tabla 4.1: Colecciones.

4.2. Resultados

4.2.1. Resultados iniciales

En las pruebas se tomó una muestra de 30 documentos y en esta primera etapa las pruebas se ejecutaron solamente una vez en cada caso, a excepción del algoritmo de agrupamiento random que se ejecutaron diez veces. La muestra se obtuvo aleatoriamente de la colección de documentos, manteniéndose estas para todos los casos. Como la muestra es insignificante, en comparación al tamaño de la colección, es muy probable que ningún documento pertenezca a uno de la misma versión. Este problema origina que la mayoría de los documentos pertenezca a un solo grupo y el resto solamente es representado por un documento, porque si un grupo crece es muy probable que al intentar unir dos grupos, el grupo más grande se una a otro grupo. También cabe mencionar que la elección de la cantidad de agrupaciones es arbitraria, pero la cantidad de agrupaciones es una variable importante al momento de obtener buenos resultados en las agrupaciones. En este caso las pruebas se realizaron con un número fijo de agrupaciones para observar el comportamiento de otras variables que afectan a las agrupaciones.

En la tabla 4.2 muestra los resultados de cada método con una cantidad de 10 agrupaciones. El *Método 1* utiliza el algoritmo de agrupación aleatoria, que en cada grupo comprimido y no comprimido se mantiene una carga de almacenamiento balanceada que es uno de los objetivos deseados en la memoria. En los métodos siguientes se utiliza el algoritmo de agrupación propuesto pero modificando algunas variables para observar su comportamiento.

El *Método 2* es el algoritmo de agrupamiento implementado sin la implementación de un balance en la cantidad de puntos representativos en cada grupo y sin el balance en la cantidad de documentos asignados a cada grupo. Para el caso del *Método 2* se observa una mejora de la compresión equivalente al 45 % del tamaño total del resultado en el algoritmo de agrupamiento aleatorio, aquí la muestra es de 30 documentos. En términos de balance en la carga de almacenamiento que se representa en el *Error* de la tabla 4.4, este método es ineficiente, ya que la mayor parte de la carga se concentra solamente en un grupo. Esto se debe a que en el momento de crear los grupos con las muestras, la mayor parte de las muestra quedan solamente en un grupo, dejando a las demás con pocas muestras de representación.

En el *Método 3* es la implementación del *Método 2* agregando un balance en la cantidad de puntos representativos por grupos. Para esto, cada grupo no tendrá una muestra superior a tres en un universo de 30 documentos. Con esto se busca balancear la cantidad de documentos en cada agrupación para solucionar el problema que ocurrió en el *Método 2* donde se concentraba la mayor parte de los puntos representativos en un solo grupo. El resultado de la compresión, utilizando el *Método 3*, es equivalente al 55 % del tamaño total del resultado con el método del algoritmo de agrupamiento aleatorio, que sigue siendo una mejor alternativa, pero comparando con los resultado del *Método 2* se paga un costo al balancear las muestras en los grupos, de un 24 % más del tamaño total del resultado en el *Método 2*.

En el *Método 4*, se hace la misma prueba que en el método anterior, pero se agrega la condición de que el tamaño de los grupos no supere determinado límite, el que en este caso es el tamaño de la colección de datos dividido por la cantidad de agrupaciones. Con esta medida se asegura que en todos los grupos tengan, aproximadamente, la misma cantidad de cadenas de caracteres. El resultado del *Método 4* es el equivalente al 70 % del tamaño total del resultado en el algoritmo de agrupamiento aleatorio, que sigue siendo una mejora, pero nuevamente pagando un costo, con respecto al *Método 2* aumenta 55 % más de tamaño, incluso mayor que en el *Método 3*, pero con mejores resultados en el balance de la carga de almacenamiento. Se observan en los resultados que al agregar el balance en la distribución de documentos se paga un costo en la compresión con respecto al *Método 2* que no cuenta con ninguna condición.

Grupos	Método 1(KiB)	Método 2(KiB)	Método 3(KiB)	Método 4(KiB)
Total	65.560	29.371 (45 %)	36.036 (55 %)	45.639 (70 %)

Tabla 4.2: Resultado algoritmo de agrupamiento aleatorio con 10 Grupos.

En la tabla 4.3 se observa el resultado de la distribución de los datos de cada grupo utilizando los métodos mencionados.

Para determinar si la carga de almacenamiento en todos los grupos se encuentra balanceada en términos del número de documentos, se calcula el promedio del error absoluto \bar{E}_a , que se define como 4.1:

$$\bar{E}_a = \frac{1}{k} \sum_{i=1}^k |V_{verdadero} - V_i| \quad (4.1)$$

donde $V_{verdadero} = \frac{\varrho}{k}$ con ϱ el número de documentos en la Colección, k es el número de grupos formados por el algoritmo de agrupamiento y V_i la cantidad de documentos en un grupo. \bar{T} representa el promedio de la carga de almacenamiento que ocupan los grupos. Por último, \bar{S} es el promedio de la distancia entre los documentos de las muestras de un mismo grupo, y se define como 4.2:

$$\bar{S} = \frac{1}{k} \sum_{i=1}^k Distancia(G_i) \quad (4.2)$$

donde $Distancia(G_i)$ es el promedio de la distancia entre los documentos del mismo grupo. Se puede observar en la tabla que en el método 3 y 4 se tienen los mismos valores en \bar{S} , esto se debe a que las muestras son las mismas y \bar{S} se calcula sobre las muestras.

Grupos	\bar{E}_a	$\bar{T}(KiB)$	\bar{S}
Método 1	0.2	6553.6	-
Método 2	1910.4	2937.2	0.196718
Método 3	509	3603.7	0.994174
Método 4	1.4	4563.9	0.994174

Tabla 4.3: Distribución grupos.

4.2.2. Resultados Finales

En esta segunda etapa de la experimentación se ejecutó el algoritmo de agrupamiento propuesto, con el objetivo de observar qué tan eficiente es el algoritmo con las mejoras implementadas, ver 3.3, comparándolos con el algoritmo propuesto inicial. Además de ver qué tan determinante es la elección de las muestras de manera aleatoria en el resultado de la compresión y, determinar qué ocurre cuando se aumentan las muestras.

Las muestras que se utilizaron fueron de 10, 20, 30, 40, 50. Para los casos del algoritmo con las mejoras implementadas se ejecutaron pruebas con muestras de 10, 30, 50 y 100. En cada caso se ejecutaron 10 veces, a excepción del caso con 100 muestras el que se ejecutaron cinco veces.

En el gráfico 4.1 muestra el total del tamaño en cada prueba realizada, las pruebas realizadas se dividen en dos grupos: sin cola de prioridad(s_n) y con la cola de prioridad($S_{n\text{cola}}$), donde n representa la cantidad de muestras utilizadas. En él se puede observar que aumentando las muestras mejoran los resultados de la compresión. Además, con las mejoras implementadas se puede observar que se tienen mejores resultados en la compresión que en los resultados sin las mejoras implementadas, por ejemplo, si se compara con las pruebas con 30 muestras se observa que el resultado con la mejora implementada representa un 86 % del resultado del algoritmo sin la mejora implementada. Otro punto que se puede observar del gráfico es cómo varían los resultados con el cambio de muestras, con esto se puede deducir que las muestras pueden influir en algún grado en el resultado de la compresión.

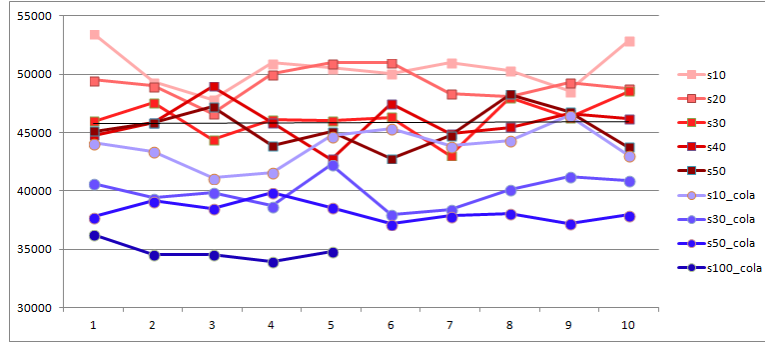


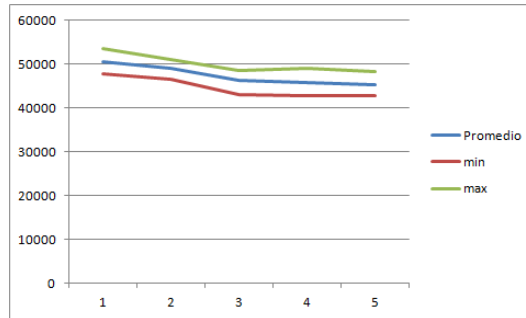
Figura 4.1: Tamaño de los resultados cambiando la cantidad de muestras.

En el gráfico 4.2(a) muestra el promedio, el mínimo y el máximo tamaño entre las 10 pruebas para cada caso realizado con el algoritmo sin las mejoras implementadas. Se observa que aumentando las muestras, hasta llegar a las 30, la compresión de los datos baja y luego se mantienen estables. En cambio en el gráfico 4.2(b) la compresión sigue mejorando luego de las 30 muestras y el rango entre el máximo y el mínimo cada vez es menor, lo que significa que las muestras tomadas aleatoriamente influyen menos en la compresión.

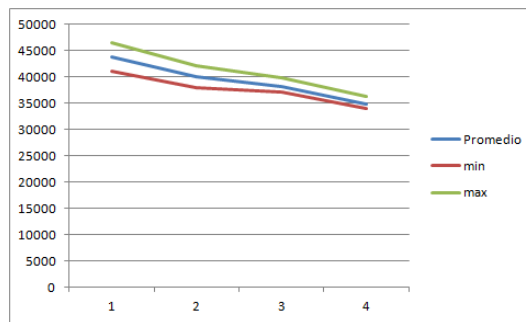
Comparando con los resultados iniciales 4.2.1, el algoritmo con las mejoras implementadas, si se ejecuta con 150 muestras se puede conseguir casi el mismo resultado que se obtuvo del *Método 2* (versión del algoritmo sin ninguna limitación y mejora) con tan solo 30 muestras.

Pruebas	S_{10}	S_{20}	S_{30}	S_{40}	S_{50}	S_{10cola}	S_{30cola}	S_{50cola}	$S_{100cola}$
Promedio	50506,9	49171,6	46216,5	45879,7	45343,1	43800,7	39956,4	38220,2	34820,6
Mínimo	47852	46626	43017	42713	42783	41112	38003	37165	33943
Máximo	53468	50990	48537	48950	48274	46449	42242	39861	36237
Random	77,04 %	75,00 %	70,49 %	69,98 %	69,16 %	66,81 %	60,95 %	58,30 %	53,11 %

Tabla 4.4: Tabla de resultados cambiando las muestras.



(a) Sin Cola



(b) Con cola

Figura 4.2: Legos.

Capítulo 5

Conclusiones

Se tuvo un proceso de estudio de los distintos algoritmos de agrupamiento más utilizados, mecanismos que permiten medir la similitud entre cadenas de caracteres y mecanismos para comprimir datos. Luego se implementó un algoritmo de agrupamiento, aplicando lo estudiado y obteniendo los primeros resultados.

Evaluando los resultados se puede concluir varias cosas, agrupando las cadenas de caracteres de manera inteligente, se obtienen mejores resultados que agrupándolos aleatoriamente, pero al intentar balancear la carga de espacio en cada agrupación se paga un costo al comprimir. Parte importante para obtener una buena agrupación es la medida de distancia, para agrupar grandes cantidades de cadenas de caracteres es necesario que la medida de distancia entre dos cadenas de caracteres sea rápida, ya que cada cadena de texto debe compararse con todos los del sampling. La ventaja de este algoritmo es que los resultados son determinista, siempre que se mantengan las mismas muestras, es decir, cuantas veces se ejecuta el algoritmo para una misma colección siempre entrega el mismo resultado, entonces al momento de obtener las agrupaciones con las muestras es posible asignar cadenas de caracteres en varios procesos, rebajando el tiempo de ejecución. Es importante aclarar que el algoritmo todavía puede seguir mejorando, en el análisis de los resultados se observó que la compresión cambia dependiendo de las muestras iniciales y entre más muestras se tomen, estas mejoran los resultados, pero con un mayor costo en el tiempo de ejecución.

Referencias bibliográficas

- [1] Wikipedia. <https://en.wikipedia.org/>.
- [2] Git. <https://git-scm.com/>.
- [3] Backblaze. <https://www.backblaze.com/blog/storage-pod-4-5-tweaking-a-proven-design/>.
- [4] David Salomon and Giovanni Motta. *Handbook of Data Compression*. Springer London, 2012.
- [5] Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. 12:180–192, 2012.
- [6] Brendan J. Frey and Delbert Dueck. Clustering by passing messages between data points. pages 972–976, 2007.
- [7] Anand Rajaraman and Jeffrey David Ullman. Mining of massive datasets. pages 221–260, 2011.
- [8] Daniel Jurafsky and James H. Martin. Speech and language processing pearson. education international. pages 107–111, 1999.
- [9] Ming Li Jonathan H. Badge Xin Chen Sam Kwong Paul Kearney and HaoYong Zhang. An information-based sequence distance and its application to whole mitochondrial genome phylogeny. 2000.
- [10] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences,theoretical computer science. 483:115–133, 2011.

Compresión de datos distribuida basada en clustering

Guillermo Andrés Navarro Giglio

ANEXOS

Profesor guía
Francisco Claude-Faust

Comité
Roberto Konow
Jonathan Frez

Julio, 2015

Anexo A

CODIGO ALGORITMO DE AGRUPAMIENTO BASE

```
#!/usr/bin/python

""" argv[1]= carpeta de documtnos para generar clusters(sampling)
    argv[2]= Numero de clusters deseado
    argv[3]= carpeta guardar clusters
    argv[4]= carpeta de documentos
    argv[5]= max cluster size
    argv[6]= calidad distancia documento 0-9
    argv[7]= Numero de core
    argv[8]= carpeta de resultados comprimidos
"""
import os, sys ,getopt
import editdist
import shutil
import distance
import zlib
from multiprocessing import Pool
from datetime import datetime
from heapq import heapify, heappush, heappop

data=[]
combinaciones=[]

def get_size(start_path = '.'):
    total_size = 0
    for dirpath, dirnames, filenames in os.walk(start_path):
        for f in filenames:
            fp = os.path.join(dirpath, f)
            total_size += os.path.getsize(fp)
    return total_size

def comprimir( ):
    for i in os.listdir(str(sys.argv[3])):
        os.system('7z a ' +str(sys.argv[8])+i+'.7z '+str(sys.argv[3])+i+'/'
        )
```



```

def distancia_zip(s1, s2, nivel=6):
    compressed1 = zlib.compress(s1,nivel)
    compressed2 = zlib.compress(s2,nivel)
    compressed12 = zlib.compress(s1+s2,nivel)
    if len(compressed1) > len(compressed2):
        n = (len(compressed12) - len(compressed2)) / float(len(compressed1))
    else:
        n = (len(compressed12) - len (compressed1)) / float(len(
            compressed2))
    return n

def listadistancia(comb):
    k=combinaciones[comb][0]
    j=combinaciones[comb][1]
    comp=distancia_zip(data[k][0],data[j][0],int(sys.argv[6]))
    print "t1:"+str(k)+" contra t2:"+str(j)+"="+str(comp)
    return [comp,data[k][0],data[j][0]]

def asignar_documento(document):
    test2=[]
    f = open(str(sys.argv[4])+str(document))
    texto=f.read()
    doc_z = len(texto)
    for j in data:
        for k in range(len(j)):
            #comp=editdist.distance(j[k],texto)
            #comp=distance.hamming(j[k],texto)
            comp=distancia_zip(j[k],texto,int(sys.argv[6]))
            test2.append([comp,data.index(j)])
    test2.sort()
    f.close()
    return [str(document),test2,doc_z]

if __name__ == "__main__":
    #numero de clusters
    clusters = int(sys.argv[2])
    #num cores
    numprocesos=int(sys.argv[7])

    #guardar bloque de datos
    for document in os.listdir(str(sys.argv[1])):
        f = open(str(sys.argv[1])+str(document))
        data.append([f.read()])
        f.close()

    print "sampling cargado, buscando las distancias minimas..."

    for k in range(len(data)):
        for j in range(k+1,len(data)):
            combinaciones.append((k,j))
    pool = Pool(processes=numprocesos)
    test = pool.map(listadistancia, range(len(combinaciones)), len(
        combinaciones)/numprocesos)
    pool.close()
    pool.join()

    test.sort()
    print len(test)

    print "creando clusters"
    i=0
    while len(data)>1 and len(data) > clusters and i < len(test):
        cluster1=[]
        cluster2=[]
        t1=False
        t2=False
        for j in data:
            if t1 == True and t2 == True :
                break
            for k in range(len(j)):

```

```

        if test[i][1] == j[k]:
            t1=True
            cluster1=data.index(j)
        if test[i][2] == j[k]:
            t2= True
            cluster2=data.index(j)

    i=i+1
    if cluster2 ==cluster1 or (len(data[cluster1])+len(data[cluster2])
        ) > int(sys.argv[5]) :
        continue
    test3=data.pop(cluster2)
    if cluster2<cluster1:
        cluster1-=1

    test4=data.pop(cluster1)
    data.append(test3+test4)

    densidad=[]
    print "densidad sampling clster:"+str(test[i][0])
    for t in data:
        densidad.append(len(t))
    print densidad

densidad=[]
print "IMPRIMIR CLUSTER"
for t in data:
    densidad.append(len(t))
print densidad

print "Creando carpetas de clusters..."

clust_size=[]
for t in range(len(data)):
    os.mkdir( sys.argv[3]+str(t));
    clust_size.append(0)

i=0

print "Asignando documentos a los clusters"

pool2 = Pool(processes=numprocesos)
todo=pool2.map(asignar_documento, os.listdir(str(sys.argv[4])), len(
    combinaciones)/numprocesos)
pool2.close()
pool2.join()

# [ str(document),test2,doc_z ] -> nombre_documento, lista_distancias,
#   tamaño_documento
# lista_distancia -> [distancia , numero_clustguillermo99er]

div=get_size(str(sys.argv[4]))/len(data)
cola_prioridad=[]
diccionario={}

for t in todo:
    for s in t[1]:
        heappush(cola_prioridad, [s[0],t[0],s[1],t[2]])
        #s[0] -> distancia      t[0]-> nombre_documento s[1]->
        numero_cluster t[2] ->tamaño_documento

while len(cola_prioridad):
    archivo=heappop(cola_prioridad)
    print archivo
    if archivo[1] in diccionario:
        continue
    else:
        if clust_size[archivo[2]] < div:
            shutil.copyfile(str(sys.argv[4])+archivo[1], sys.
                argv[3]+str(archivo[2])+'/' +archivo[1])
            clust_size[archivo[2]]+= archivo[3]
            diccionario[archivo[1]]= archivo[2]
        else:

```

`continue`

`comprimir()`

Anexo B

CODIGO ALGORITMO DE AGRUPAMIENTO ALEATORIO

```
#!/usr/bin/python
import glob
import os,sys
import random
import shutil

""" argv[1]= carpeta de documentos
    argv[2]= n clusters
    argv[3]= carpeta resultados
"""

print sys.argv[1]
lista=os.listdir(str(sys.argv[1]))
random.seed()
print len(lista)

print "Creando carpetas de clusters..."

for t in range(int(sys.argv[2])):
    os.mkdir( sys.argv[3]+str(t));

div=len( lista)/int(sys.argv[2])
print div

for i in range(int(sys.argv[2])) :
    for j in range(div) :
        rand = random.randint(0,len( lista)-1)
        texto= lista.pop(rand)
        shutil.copyfile( sys.argv[1]+texto , sys.argv[3]+str(i)+"/"+texto)
```
