Ingeniería en Computación

Proyecto Terminal

Sistema de recuperación de información de textos de investigación de la web

Elaborado por:
Luis Yamil García Jurado

Asesorado por:
Maricela Claudia Bravo Contreras
Y
María Lizbeth Gallardo López

Tabla de Contenido

1. Resumen 3
2. Introducción 3
   2.1 Arañas web 3
   2.2 Minería de datos 4
3. Objetivo 4
4. Análisis y diseño del sistema 5
   4.1 Clustering 5
   4.2 Diagrama de clases 6
   4.3 Implementación 8
      4.3.1 Araña web 8
      4.3.2 Clúster 14
5. Pruebas 17
6. Conclusiones 19
7. Bibliografía 20

1. Resumen

El presente documento proporciona información sobre el desarrollo del Proyecto Terminal "Sistema de recuperación de información de textos de investigación de la web" para la División de Ciencias Básicas e Ingeniería (DCBI) de la UAM-Azcapotzalco.

Este proyecto se enfoca en desarrollar un sistema usando arañas web focalizadas que pueda encontrar documentos (específicamente en formato PDF) de tipo científico o de investigación en la web, aplicándoles minado de datos para obtener documentos más relevantes de entre los encontrados; esto para evitar revisar contenido irrelevante o de poca ayuda para una búsqueda con propósitos científicos.

2. Introducción

La búsqueda de información en la web ha sido de vital importancia en los últimos años y esto no es diferente para el ámbito científico, pues con la creciente expansión de la web y de la información que se puede adquirir en ella, es más importante que nunca conseguir resultados más precisos en una búsqueda especifica, ganando de esta manera valioso tiempo de trabajo.

Por estas razones se decidió crear un sistema que permita conseguir documentos con alta relevancia para una búsqueda dada, y específicamente para búsquedas de propósito científico. Para realizar esta tarea se necesitó una manera de encontrar documentos en la web y un mecanismo para elegir los mejores de ellos; este sistema fue creado bajo estos términos utilizando como buscador una araña web y como mecanismo de filtrado un minador de datos que use *clusters*.

2.1 Arañas web

Una araña web (conocida en inglés como *crawler*) es un programa que recorre el entramado de distintas páginas web dentro del Internet de forma automática y sistemática. Una araña web es un tipo especializado de robot de la web que se encarga de llevar a cabo un tipo concreto de tareas, en particular: analizar su información y hacer exploraciones de las páginas que encuentre referenciadas mediante una URL para procesarlas. Pueden ser utilizadas con fines diversos, aunque su uso más conocido es el de agente software en los motores de búsqueda, donde su función básica es proporcionar al indizador el contenido apropiado para ser indizado, según distintos criterios e intereses.

Se tienen 2 tipos básicos de arañas web que son:

1. Arañas web generales

   Son la forma más básica de una araña web y se caracterizan por visitar todos los sitios web, sin hacer distinción entre ellos; es decir, dado un conjunto de URLs en una cola de descarga, por cada una obtiene las URL que ésta contenga y las añade a la misma cola de descarga para después procesarlas, y sigue con este procedimiento hasta un determinado punto.

Pero al no hacer distinción entre los documentos, se tienen ciertas desventajas, como son: el saturar el ancho de banda con demasiadas descargas y obtener documentos con muy poca o nula relevancia para la búsqueda deseada.

2. Arañas web focalizadas

   Estas funcionan de manera distinta, pues hacen su cola de descargas de acuerdo a una serie de prioridades; es decir, asignan prioridad a cada enlace no visitado estimando un valor de importancia a la página enlazada. Estas prioridades se calculan según su relación con un conjunto de palabras clave ingresadas como búsqueda.

   En este tipo de arañas web se busca evitar descargas innecesarias mediante la relevancia de una página en la búsqueda, pero al mismo tiempo tienen como desventaja el tiempo de procesamiento, que resulta ser mucho más costoso. [1]

2.2 Minería de datos

La minería de datos (conocida como *Knowledge Discovery in Database* o *KDD*) se puede definir como el proceso de descubrimiento de patrones útiles, correlaciones significativas, tendencias, además de obtener conocimiento de fuentes de datos mediante Estadística, Inteligencia Artificial, entre otros.

El análisis de los datos que se hace mediante esta disciplina aprovecha la información que se encuentra en las bases de datos, permitiendo hacer predicciones y ayudando en la toma de decisiones. El minado de datos comprende 3 etapas:

1. Pre-procesado: Puede ser que los datos no estén en condiciones adecuadas para minarse, así que esta etapa se encarga de hacer una reducción de los datos para quedarse con los atributos necesarios y remover el ruido que pudieran tener.

2. Minado de datos: Aquí se encuentra el algoritmo encargado de extraer los patrones de los datos pre-procesados.

3. Post-procesado: Aquí se determina que patrones son útiles, según la aplicación que los requiera.

El proceso de minería de datos se realiza para conseguir resultados útiles.

3. Objetivo

Diseñar un sistema basado en arañas web focalizadas que recuperen información de contenidos de investigación públicos en la web, el cual reciba como entrada varias palabras clave para realizar la búsqueda y así recuperar los documentos relacionados más relevantes, para su posterior procesado dentro de un minador de datos usando algoritmos de clústers y clasificación.

4. Análisis y diseño del sistema

Por restricciones de hardware, el sistema se dividirá en 2 módulos principales: un módulo para la araña web y uno para el minador de datos. El minador de datos utilizara la estrategia de *clustering*, para agilizar su procesado y más adelante sus pruebas.

4.1 Clustering

Esta fue la estrategia específica que usamos para el módulo minador de datos. El *clustering* o "aprendizaje no supervisado" es un área del aprendizaje de máquina y reconocimiento de patrones y es conocido por usar algoritmos que descubren conjuntos de documentos que mantienen un grado de similitud suficiente [2].

Un problema básico del *clustering* es que no es fácil decidir la manera en cómo se escogen tales documentos para estar en cierto conjunto. Se pueden usar diferentes medidas de similitud y es difícil saber cuál escoger pues se tienen muchos atributos que medir en un documento, tales como: el tamaño del documento, la frecuencia de ocurrencia de las palabras claves y las palabras importantes.

Para el caso del presente proyecto, se estudiaron 2 algoritmos muy famosos: el HAC (*Hierachical Agglomerative Clustering*, por sus siglas en ingles) y el GQF (*Global Quality Function*, por sus siglas en ingles). Ambos son algoritmos de clasificación, ambos miden la cantidad y la relevancia de las palabras buscadas en los documentos encontrados por nuestra araña web.

Para encontrar cual es mejor, tomaremos en cuenta 2 requerimientos: primero, el algoritmo debe producir *clusters* fáciles de navegar y segundo, debe ser rápido aunque se aplique a un gran número de documentos. Por lo anterior se decidió usar el algoritmo GQF que se basa en funciones heurísticas que evalúan lo bueno del set de *clusters* que se tenga. Ambos algoritmos (HAC y GQF) empiezan igual: se tiene cada documento en un *cluster* solo y se van juntando según un cierto criterio, pero se diferencian en la manera en cómo se van emparejando los *clusters*.

El algoritmo GQF se queda con la pareja de *clusters* que maximicen su función heurística en cada iteración, y lo regresa a la piscina de *clusters* (el conjunto de *clusters*), continuando con su proceso hasta que la función ya no se eleve más. Por sus propiedades glotonas y su bien definido final, es un algoritmo más recomendable que el HAC [3] [4].

La manera en que funciona la función heurística es de la siguiente manera:

Lo que se buscará será maximizar una función heurística, que a su vez, maximizará la cantidad de clusters con varios documentos, acercando los que se parezcan entre sí, según su "cohesión". La cohesión, se mide como el logaritmo del número de términos en común entre 2 documentos, se usa el logaritmo pues este número se vuelve muy grande y así se ahorra tiempo de ejecución.

La función se calcula así:

$$GQF(ClusterSet) = \frac{f(ClusterSet)}{g(|ClusterSet|)} \sum_{c \in ClusterSet} s(c)$$

F (*ClusterSet*) = número de documentos en *clusters* con más de 1 documento, entre el número total de documentos.

G (|*ClusterSet*|) = raíz cuadrada del número de *clusters* con más de 1 documento.

Sumatoria de s(c) = sumatoria de: número de documentos en el *cluster*, multiplicado por la cohesión del *cluster* (a este término se le conoce como *score*).

4.2 Diagrama de clases

A continuación, se mostrarán los diagramas de las estructuras de clases del sistema.

Tenemos primero la araña web (fig. 2), que consta de 4 clases:

- *Crawler* base: esta clase es la araña web básica, de la cual se heredaran los atributos base para una araña web focalizada.
- *Crawler* focalizado: esta clase hereda las propiedades del *Crawler* base. Contiene todos los métodos necesarios para el filtrado de páginas web, el filtrado de los documentos encontrados y la descarga de aquellos que cumplan con los requerimientos mínimos para pasar al módulo del *cluster.*
- Controlador de docs: Esta clase lleva el recuento de los documentos descargados por el *Crawler* focalizado. Usa un documento en formato txt para llevar el control de los documentos descargados.
- *Crawler* principal: Esta clase simplemente crea un objeto de un *Crawler* focalizado, y le da las palabras clave que el usuario quiere buscar.

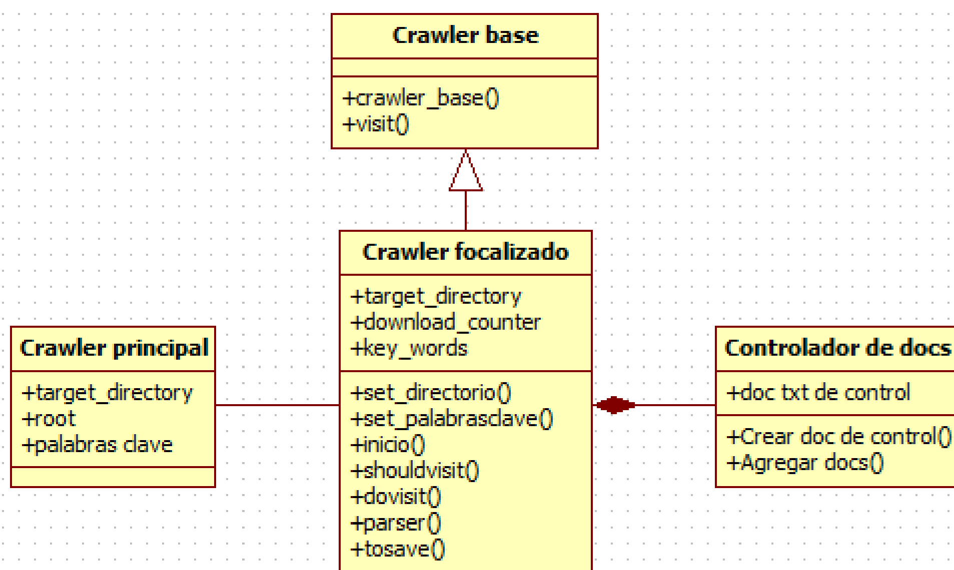

Fig. 2. Diagrama de clases del módulo: araña web

Ahora tenemos el diagrama de clases para el minador de datos (figura 3), que consta de 3 clases:

- *Cluster*: Esta clase contiene la abstracción de un *cluster*, con esta clase se crearán objetos para calcular el algoritmo GQF.
- Op *cluster*: Esta clase contiene todos los métodos requeridos para el cálculo del algoritmo y la creación de los *clusters*.
- *Cluster* panel: Esta clase crea un Op *cluster* y recupera los documentos que el algoritmo haya arrojado como los mejores, para poder presentárselos al usuario, mediante un Visualizador.
- Visualizador: Esta ventana despliega los resultados obtenidos al usuario.

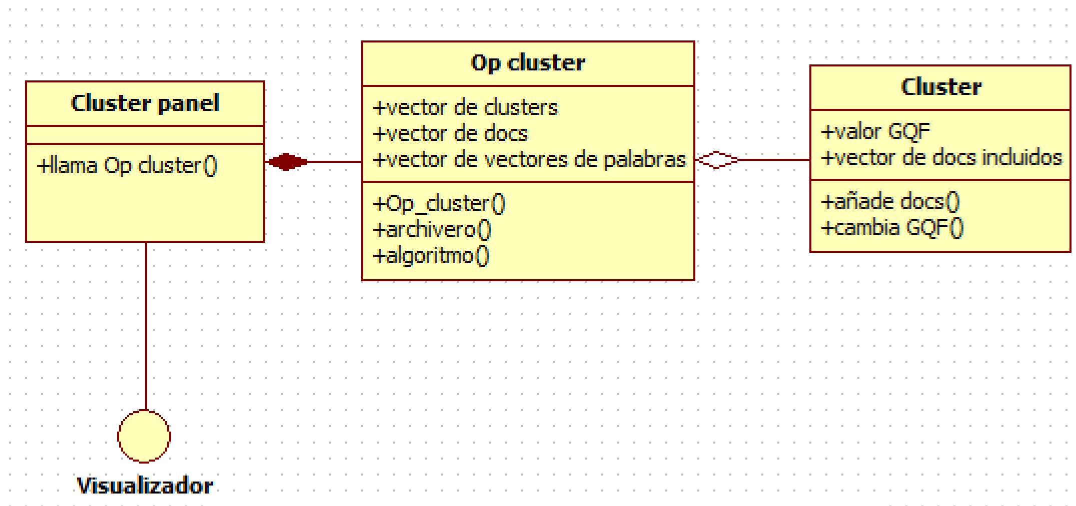

Fig. 3. Diagrama de clases del módulo: minador de datos

4.3 Implementación

En este apartado se explicarán todos los componentes de los que consta el proyecto, apoyándonos en el código. El lenguaje de programación utilizado es Java, usando el Netbeans IDE 7.2.1.

4.3.1 Araña web

La araña web es el primero de nuestros 2 módulos principales. Este, se encarga de buscar y descargar documentos relevantes según las palabras clave a buscar por el usuario, para ello se usó la biblioteca Websphinx, con la cual podremos crear un crawler a partir de un código base.

```
import websphinx.Crawler;
import websphinx.DownloadParameters;
import websphinx.Page;


//clase crawler primaria que usare para crear CRAWLERS FOCALIZADOS
//se extiende de websphinx
public abstract class Crawler_base extends Crawler{
    private static final long serialVersionUID = 2383514014091378008L;
    protected final Log log = LogFactory.getLog(getClass());

    public Crawler_base(){
        super();
        DownloadParameters dp = new DownloadParameters();
        dp.changeObeyRobotExclusion(true);
        dp.changeUserAgent("MyCrawler Google Chrome (X11; U; Windows x86; es-MX; rv:1.3) "
                + "WebSPHINX 0.5 contact Metropolitan Autonomus University");
        this.setDownloadParameters(dp);
        //visitara todas las subpaginas de la raiz
        this.setDomain(Crawler.SUBTREE);
        this.setLinkType(Crawler.SUBTREE);
        //maxima profundidad
        this.setMaxDepth(1000);
    }

    @Override
    public void visit(Page page){
        dovisit(page);
```

Fig. 4. Clase Crawler base

En la fig. 4 se ve la clase "Crawler base" con la cual se construye una araña web básica. Esta clase nos servirá más adelante para crear una araña web focalizada. A continuación se describe el contenido de este "Crawler base":

- Se necesita especificar la manera en que la araña actuará en la web, por lo que en el apartado "*DownloadParameters*" se describe que esta araña web, es un robot con propósitos solamente de investigación no lucrativos.
- Se especifica el nombre del servidor donde se buscarán páginas dentro del campo "*SetDomain*", en este caso, solo se buscará en el subárbol de la raíz para evitar visitar páginas que no pertenezcan a ésta. En el campo "*SetLinkType*" se ajusta como subárbol también de la raíz, para mantenernos en páginas deseadas.
- Por último se construyó un nuevo método que prevalece sobre el método de la clase padre (esta clase se construyó utilizando la biblioteca Websphinx). Tal método fue el método "*visit*", esto debido a que se necesita usar tal método

heredado pero de manera distinta. Este procedimiento se usa en otro método más adelante.

Ahora tenemos la siguiente clase: la araña focalizada. A esta clase se le nombro "Focalized crawler" y hereda los métodos y atributos de la clase "*Crawler* base". Este método recibirá las palabras clave para la búsqueda, y comenzara en la página raíz buscando páginas, apilándolas en una cola automáticamente. Lo hace así pues se utiliza la biblioteca Websphinx que ya tiene definido como apilar las páginas.

Además de los métodos heredados tiene los suyos, en la fig. 5 se observan los 3 primeros:

```
30      public void setpatron(List<String> bus){
31          palabras = new ArrayList<>();
32          for(int i=0;i<bus.size();i++){
33              palabras.add(bus.get(i));
34          }
35      }
36
37      public void settargetdir(String targetdir){
38          this.targetdir = targetdir;
39      }
40
41      public void setclusterdir(String clusterdir){
42          this.clusterdir = clusterdir;
43      }
44
45      public void init() throws Exception{
46          File targetdirfile = new File(targetdir);
47          if(!targetdirfile.exists()){
48              FileUtils.forceMkdir(targetdirfile);
49          }
50          //vector = new ArrayList<>();
51          controlador = new Doc_manager(clusterdir);
```

Fig. 5. método init

- El método "*setpatron*" asigna las palabras de la búsqueda.
- El método "*settargetdir*" asigna el nombre de la carpeta donde se guardarán los documentos.
- El método "*setclusterdir*" asigna el nombre de la carpeta donde se guardara el documento cluster.txt. Se hablara de este, más adelante.
- El método "*init*" verifica la existencia de la carpeta antes mencionada y crea un objeto del tipo "doc manager", que llevará control de los documentos descargados como se verá más adelante.

En la fig. 6 podemos ver el método "*shouldvisit*" (este método es otro de los que prevalecen por sobre los métodos de la clase padre, como en el caso de "visit" en la clase "Crawler base") el cual se encarga de restringir el acceso al método "*visit*". La función de este método es descartar cualquier página que parezca innecesaria de revisar, como lo puede ser una imagen, un acceso a alguna cuenta, el ordenamiento de la misma página, etc.

```
47        //esta funcion determina si se visita o no un link
          @Override
49        public boolean shouldVisit(Link link){
50            URL linkurl = link.getURL();
51            boolean visita;
52            if(linkurl.toString().contains("sort")||linkurl.toString().contains("membership")||
53                    linkurl.toString().contains("kdd.org")||linkurl.toString().contains("login")||
54                    linkurl.toString().contains("previous")||linkurl.toString().contains("png")||
55                    linkurl.toString().contains("gif")||linkurl.toString().contains("session")||
56                    linkurl.toString().contains("cacm.acm")||linkurl.toString().contains("access")
57                    ||linkurl.toString().contains("ico")||linkurl.toString().contains("jpg")||
58                    linkurl.toString().contains("awards.acm")){
59                visita = false;
60            }
61            else{
62                visita = true;
63            }
64            return(visita);
65        }
```

Fig. 6 método shouldvisit

Si "*shouldvisit*" considera que la pagina vale la pena, manda una bandera al método "*visit*" para que la revise. Esta última clase manda a llamar a otra más: "*dovisit*", la cual podemos observar en la fig. 7.

```
          @Override
69        protected void dovisit(Page page){
70            String pageurl = page.getURL().toString();
71
72            if(pageurl.contains("url=http")){
73                System.out.println("Detecto que la pagina contiene url=http ");
74                try {
75                    //Dividir la cadena en dos a partir del s�mbolo de =
76                    String []urlBuffer = pageurl.split("=");
77                    //Tomar s�lo la segunda parte
78                    String urlName = urlBuffer[urlBuffer.length-1];
79                    String urlEncoded = URLDecoder.decode(urlName, "UTF-8");
80                    //Crear una nueva url a partir de este urlName
81                    URL urlSec = new URL(urlEncoded);
82                    System.out.println("URL secundaria "+ urlSec.toString());
83                    pageurl = urlSec.toString();
84                    //Crear un nuevo objeto Page para guardar
85                    Link link = new Link(urlSec);
86                    Page myPage = new Page(link);
87                    //Re-apuntar el objeto page al nuevo
88                    page = myPage;
89                }
90                catch(Exception ex){
91                    System.out.println("error al pasar url=http");
92                }
93            }
94
95
96            System.out.println("Visitando "+pageurl);
97            //solo usaremos paginas que contengan archivos PDF
98            if(pageurl.endsWith(".pdf")||pageurl.endsWith("PDF")){
99                System.out.println("\n    Revisando "+pageurl);
100               URL url = page.getURL();
101               //se revisara que el PDF cumpla con los requerimientos
102               boolean save = parser(url);
103               if(save){
104                   tosave(page);
105               }
106               //vector.clear();
107           }
108           page.discardContent();
109           page.getOrigin().setPage(null);
```

Fig. 7 método dovisit

Este método también prevalece sobre otro de mismo nombre de la clase padre y es donde se discriminan las páginas a descargar.

- Primero se asegura de entender el URL recibido, decodificándolo usando UTF-8 si es necesario (esto es porque así procesa las paginas la biblioteca Websphinx);
- Después debe verificar que se trate de un documento en formato pdf, ya que solo nos interesa ese tipo de documentos.
- Si lo anterior se cumple, manda a llamar al método "*parser*", el cual decidirá si es un documento útil o no y así determinar si se descargará o no el documento.

Al final simplemente se evita repetir la visita de la misma página más adelante, marcándola como ya visitada con "*discardContent*" y "*getOrigin*".

```java
        try {
            //creamos un stream donde se guardara el contenido
            InputStream stream = url.openStream();
            PdfReader reader = new PdfReader(stream);
            int num_pag = reader.getNumberOfPages();
            String linea, buffer_token;
            int i, k;
            buffer_token = PdfTextExtractor.getTextFromPage(reader, 1);
            for(i=2;i<=num_pag;i++){
                buffer_token = buffer_token + PdfTextExtractor.getTextFromPage(reader, i);
            }
            //ya teniendo todo nuestro texto, se tokeniza para parsear
            reader.close();
            stream.close();
            System.out.println("   tokenizando "+ url.toString());
            StringTokenizer parser = new StringTokenizer(buffer_token);
            buffer_token = null;
            while(parser.hasMoreTokens()){
                linea = parser.nextToken();
                System.out.println(linea);
                if(linea.contains(c1)||linea.contains(c2)||
                        linea.contains(c3)||linea.contains(c4)||
                        linea.contains(c5)||linea.contains(c7)){
                    coincidencias++;
                }
                for(k=0;k<palabras.size();k++){
                    if(linea.contains(palabras.get(k))){
                        coin_patron++;
                    }
                }
            }
            if(coincidencias>=1&&coin_patron>=1){
                System.out.println("   Se guardara el  contenido de: "+url.toString());
                //controlador.push_doc(parser); //se guardara el doc en el vector de docs
                resultado = true;
            }
            else{
                System.out.println("   Documento no util");
            }
            parser = null;
            linea = null;
        } catch (IOException
        return resultado;
```

Fig. 8 método parser

En el método "*parser*" que se puede ver en la fig. 8, se utiliza otra biblioteca extra: *itextpdf*. Esta biblioteca nos permite revisar palabra por palabra un documento en formato pdf. Con esta biblioteca se lee el documento que encontró la clase "*dovisit*" y se revisan 2 cosas:

- Que el documento contenga las palabras de búsqueda
- Que el documento contenga palabras propias de un documento de investigación (estas palabras se denotan por c1, c2, etc.).

Si el documento resulta ser útil, el método "parser" le dirá al método "dovisit" que lo descargue, y este mandará llamar al método "*tosave*" que se encargará de descargar el documento (ver fig. 9) pasándole el URL de la página.

```
172     public void tosave(Page page){
173         URL url = page.getURL();
174         String fileName;
175         System.out.println("\tGuardando URL: "+url.toString());
176         try{
177             if(url.getPath()==null){
178                 fileName = url.toString();
179             }
180             else{
181                 String[] buffer = url.toString().split("/");
182                 //Caso HOST/.../Recurso.xyz
183                 fileName = buffer[buffer.length-1];
184                 //Caso HOST/Recurso/
185                 if(fileName==null){
186                     fileName = buffer[buffer.length-2];
187                 }
188             }
189
190             fileName = fileName.replace("*","(asterisco)").replace("|","(barra)")
191                     .replace("\\", "(dinversa)").replace(":", "(dospuntos)")
192                     .replace("\"", "(comillas)").replace("<", "(menorque)")
193                     .replace(">", "(mayorque)").replace("?", "(interrogacion)")
194                     .replace("/", "(barrad)");
195             System.out.println("\tArchivo: "+this.targetdir+"/"+fileName);
196             File targetFile=new File(this.targetdir+"/"+fileName);
197             FileUtils.writeByteArrayToFile(targetFile, page.getContentBytes());
198
199
200             try {
201                 File nuevo_doc=new File("C:/Users/Grevious/Documents/cluster",fileName+".txt");
202                 if(!nuevo_doc.exists()){
203
204                     controlador.agrega_doc(fileName);
205                 }
206             } catch (Exception e) {
207                 System.out.println("Error al guardar txt del pdf");}
208
209
210         }
211         catch(Exception e){}
212     }
```

Fig. 9 método tosave

En este método se descargará el documento seleccionado, con los siguientes pasos:

- Arregla el nombre del documento, separando el nombre del resto de la URL y después cambiando los caracteres especiales (\\, /, *, etc.) para evitar conflictos de escritura.
- Se guarda el documento como un arreglo de bytes en la carpeta que especificamos en el método "*init*" para guardar los documentos en pdf.
- Después se usa el objeto de tipo "*Doc manager*" que se había creado en el método "*init*" al principio. Con este controlaremos un pequeño documento en formato txt que llamaremos: controlador. Este documento será usado para llevar un control de todos los documentos descargados. En este punto se agrega a nuestro controlador el documento que se acaba de descargar.

En la clase "*Doc manager*" se hace el manejo del documento en formato txt (fig. 10) mencionado antes. Este documento contendrá todos los documentos descargados y para ello se vale de 2 métodos:

- "*Doc_manager*". Este método crea una carpeta donde poner nuestro txt y lo crea en caso de que no exista uno u otro. El documento se nombra cluster.txt.
- "*agrega_doc*". Este método agrega al final de cluster.txt, el documento pdf nuevo descargado.

```
10  public class Doc_manager {
11
12      public Doc_manager(String dir){
13          File targetdirfile = new File(dir);
14          if(!targetdirfile.exists()){
15              try {
16                  FileUtils.forceMkdir(targetdirfile);
17              } catch (IOException ex) {
18                  System.out.println("error al crear carpeta cluster");
19              }
20          }
21          File doc_ctrl = new File(dir,"cluster.txt");
22          if(!doc_ctrl.exists()){
23              try {
24                  doc_ctrl.createNewFile();
25              } catch (IOException ex) {
26                  System.out.println("error al crear cluster.txt");
27              }
28          }
29      }
30
31      public void agrega_doc(String nombre, String dir){
32          try {
33              File doc_ctrl = new File(dir,"cluster.txt");
34              FileWriter escribe_actrl;
35              escribe_actrl = new FileWriter(doc_ctrl,true);
36              escribe_actrl.write(nombre+" ");
37              escribe_actrl.close();
38              System.out.println("\tDocumento agregado a cluster.txt");
39          } catch (IOException ex) {
40              System.out.println("Error al guardar txt del pdf");}
41      }
42
43  }
```

Fig. 10. método Doc_manager

4.3.2 Clúster

En este módulo esta implementado el algoritmo GQF, del que se habló en la sección 4.1 *clustering*. Este módulo consta de 3 clases: "*Cluster* panel", "*op cluster"* y "*cluster"*.

Primero tenemos la clase "*cluster"* que es vital para el cálculo de la función heurística, pues todo el algoritmo se basa en clusters y por tanto se necesita representarlos. En nuestra clase tenemos una lista de documentos que están incluidos en este cluster; si el clúster tiene o no solo un documento (lo usaremos más adelante en nuestra función heurística) y su último valor GQF calculado para denotar su relevancia.

```
10  public class cluster {
11      List<Integer> docs_incluidos;
12      boolean single;
13      double GQF;
14
15      public cluster(int doc){
            docs_incluidos = new ArrayList<Integer>();
17          docs_incluidos.add(doc);
18          single = true;
19          GQF = 0;
20      }
21
31
32      public void añade(List<Integer> lista){
33          int i;
34          for(i=0;i<lista.size();i++){
35              docs_incluidos.add(lista.get(i));
36          }
37          single = false;
38      }
39
40      public void coloca_gqf(double valor){
41          GQF = valor;
42      }
43
44  }
```

Fig. 11. clase cluster

Después tenemos la clase "*op cluster". *Esta clase contiene los métodos necesarios para el cálculo de la función heurística GQF:

- "op_cluster". Este método leerá cuantos documentos se procesaran y guardará sus nombres en un vector, leyéndolos del cluster.txt que se creó en el módulo anterior.
- "archivero". Este método leerá todos y cada uno de los documentos pdf que se procesaran, gracias al vector de nombres obtenido en "op_cluster". Guardará cada documento en un vector de vectores de palabras, de tal manera que se tengan todas las palabras de cada uno de los documentos. Además de guardar todas las palabras de cada documento, este método se asegura de no incluir palabras repetidas ni palabras irrelevantes (como es el caso de artículos o pronombres). Este método también hace uso de la biblioteca itextpdf para leer pdfs así como lo hizo el método "*parser*" en el módulo anterior.
- "algoritmo". Este método calcula la función GQF haciendo uso de clusters de nuestros documentos y devuelve una lista de los mejores documentos encontrados.

```
17  public class op_cluster {
18      int no_docs;
19      List<String> vector;
20      List<List<String>> lista_doc;
21      //Hashtable<Integer,List<String>> tabla;
22
23      public op_cluster(){
24
25          String temp;
26          StringTokenizer token = null;
27          try {
28              File doc_ctrl = new File("C:/Users/Grevious/Documents/cluster","cluster.txt");
29              if(doc_ctrl.exists()){
                    FileReader lector = new FileReader(doc_ctrl);
31                  BufferedReader br = new BufferedReader(lector);
32                  temp = br.readLine();
33
34                  br.close();
35                  lector.close();
36                  token = new StringTokenizer(temp);
37              }
38          } catch (IOException ex) {
39              System.out.println("Error al leer el cluster.txt");}
40
41          vector = new ArrayList<>();
42          while(token.hasMoreElements()){
43              vector.add(token.nextToken());
44          }
45          no_docs = vector.size();
            lista_doc = new ArrayList<List<String>>();
47          System.out.println("se tienen "+no_docs+" documentos");
48      }
```

Fig. 12 método op cluster

En la fig. 12 se ve el método “op_cluster” de la clase de su mismo nombre.

En la fig. 13 se ve el método “archivero”.

```
55          while(j<no_docs){
56              linea = vector.get(j);
57              try {
58                  String linea2 = "C:/Users/Grevious/Documents/ServiciosWeb/"+linea;
59                  PdfReader lector = new PdfReader(linea2);
60                  System.out.println("se leera el documento: "+linea);
61                  int i;
62                  System.out.println("tiene "+lector.getNumberOfPages()+" paginas");
63                  String buffer_token, temp;
                    List<String> doc = new ArrayList<String>();
65                  for(i=1;i<=lector.getNumberOfPages();i++){
66                      buffer_token = PdfTextExtractor.getTextFromPage(lector, i);
67                      StringTokenizer tokenizador = new StringTokenizer(buffer_token);
68                      //System.out.println("se tienen "+tokenizador.countTokens()+" palabras en
69                      while(tokenizador.hasMoreTokens()){
70                          temp = tokenizador.nextToken();
71                          if(!"and".equals(temp)&&!"a".equals(temp)&&!"for".equals(temp)&&
72                                  !"the".equals(temp)&&!"that".equals(temp)&&
73                                  !"is".equals(temp)&&!"or".equals(temp)&&!"as".equals(temp)&&
74                                  !"this".equals(temp)&&!"to".equals(temp)&&
75                                  !"in".equals(temp)&&!"are".equals(temp)&&
76                                  !"we".equals(temp)&&!"be".equals(temp)&&!"of".equals(temp)){
77                              if(!doc.contains(temp)){
78                                  doc.add(temp);
```

Fig. 13. método archivero

A continuación en la fig. 14 está el método “algoritmo”.

- En el código se ve como se comparan los “k” documentos del clúster “i” contra los “m” documentos del clúster “j”, mientras se itera “i” y “j” veces.
- Se calcula el score de cada pareja de clusters para poder elegir a la mejor.
- Se calcula el GQF y se mantiene a la mejor pareja en cada iteración para mezclarla en un solo cluster y se vuele a iterar.

El proceso continua hasta que el valor GQF ya no se eleva más.

```
106  while(actual_gqf<max_gqf){
107      actual_gqf = max_gqf;
108      //compararemos los "k docs del cluster i" contra los "m docs del cluster j"
109      for(i=0;i<lista_cluster.size()-1;i++){
110          for(j=i+1;j<lista_cluster.size();j++){
111              score = 0;
112              //cuantos docs habria en el cluster
113              posible_tam_cluster = lista_cluster.get(i).docs_incluidos.size() + lista_cluster.ge
114              for(k=0;k<lista_cluster.get(i).docs_incluidos.size();k++){
115                  doc1 = lista_cluster.get(i).docs_incluidos.get(k);
116                  tamk = lista_doc.get(doc1).size();
117                  for(m=0;m<lista_cluster.get(j).docs_incluidos.size();m++){
118                      //guardamos el score haciendole raiz y mult por el tam_posible
119                      doc2 = lista_cluster.get(j).docs_incluidos.get(m);
120                      tamm = lista_doc.get(doc2).size();
121                      //calculo de la cohesion
122                      for(r=0;r<tamk;r++){
123                          for(s=0;s<tamm;s++){
124                              if(lista_doc.get(doc1).get(r).equals(lista_doc.get(doc2).get(s))){
125                                  temp_sc = temp_sc+1;
126                              }
127                          }
128                      }
129                      temp_sc = temp_sc+(posible_tam_cluster*(Math.log(temp_sc)));
130                      score = score + temp_sc;
131                      temp_sc = 0;
132                  }
133              }
134              //calculo de toda la ecuacion GlobalQualityFunction entre i y j
135              for(x=0;x<lista_cluster.size();x++){
136                  if((!lista_cluster.get(x).single)||(lista_cluster.get(x)==lista_cluster.get(i))||(lista_
137                      nosingle_cluster = nosingle_cluster + 1;
138                      docs_nosingle_cluster = docs_nosingle_cluster+lista_cluster.get(x).docs_incluidos.si
139                  }
140              }
141              temp_gqf = (docs_nosingle_cluster/posible_tam_cluster)/(Math.sqrt(nosingle_cluster))*score;
142              if(temp_gqf>max_gqf){
143                  max_gqf = temp_gqf;
144                  posible_cluster_merged1 = i;
145                  posible_cluster_merged2 = j;
146              }
147              nosingle_cluster = 0;
148              docs_nosingle_cluster = 0;
149          }
150      }
151      //asignacion de los nuevos clusters segun el maximo GQF
152      if(max_gqf>actual_gqf){
153          //se incluyen los docs de los 2 clusters y despues se borran
154          cluster no2 = lista_cluster.get(posible_cluster_merged2);
155          lista_cluster.get(posible_cluster_merged1).añade(no2.docs_incluidos);
156          lista_cluster.get(posible_cluster_merged1).coloca_gqf(max_gqf);
157          lista_cluster.remove(posible_cluster_merged2);
158      }
159  }//fin del while del algoritmo
```

Fig. 14. método algoritmo

Por último, tenemos a la clase "*cluster* panel" que manda a llamar a "op cluster" y se encarga de lanzar una ventana con los resultados del cluster con el mayor valor GQF y presentara a manera de lista los documentos que contenga (fig. 15).

```
94     private void but_inicioActionPerformed(java.awt.event.ActionEvent evt) {
95         //accion cuando se clikee inicio
96         op_cluster cluster;
97         cluster = new op_cluster();
98         cluster.archivero();
           List<String> resultado = new ArrayList<>();
.00        resultado = cluster.algoritmo();
.01        DefaultListModel modelo = new DefaultListModel();
.02        for(String x : resultado){
.03            modelo.addElement(x);
.04        }
.05        lista_docs.setModel(modelo);
.06    }
.11    public static void main(String args[]) {
.12        /* Set the Nimbus look and feel */
.13        Look and feel setting code (optional)
.34
.35        /* Create and display the form */
.36        java.awt.EventQueue.invokeLater(new Runnable() {
          public void run() {
.38                new cluster_panel().setVisible(true);
.39            }
.40        });
.41    }
```

Fig. 15 clase cluster panel

5. Pruebas

En este apartado se pueden encontrar los resultados obtenidos con la implementación del sistema.

Se realizaron las pruebas con las palabras de búsqueda: "computer technology". A lo que el crawler nos devolvió 11 documentos en total (fig. 16 y 17)

```
Output - PT_Crawler (run)
run:
Que quieres buscar?
computer technology
Crawling .. http://www.acm.org/
Guardando archivos en .. C:/Users/Grevious/Documents/ServiciosWeb

Visitando http://www.acm.org/
Visitando http://www.acm.org/portal_css/Plone%20Default/ploneStyles7664.css
Visitando http://www.acm.org/portal_css/Plone%20Default/ploneStyles1381.css
Visitando http://www.acm.org/search_form
Visitando http://www.acm.org/portal_javascripts/Plone%20Default/ploneScripts3514.js
Visitando http://www.acm.org/sitemap
Visitando http://www.acm.org/search?SearchableText=&Title=&Subject_usage%3Aignore_empt
Visitando http://www.acm.org/author/
Visitando http://www.acm.org/author/admin
Visitando http://www.acm.org/google_styles.css
Visitando http://www.acm.org/publications/alacarte
Visitando http://www.acm.org/Members/admin
Visitando http://www.acm.org/dashboard
```

Fig. 16 pruebas de la araña web

```
Output - PT_Crawler (run)
Visitando http://www.acm.org/sigs/elections/2013-sig-election-results/sigcse-2013-results
Visitando http://www.acm.org/sigs/elections/2013-sig-election-results/sigsam-2013-results
Visitando http://www.acm.org/sigs/elections/sigir-2013/M_de_Rijke_Chair.pdf

   Revisando http://www.acm.org/sigs/elections/sigir-2013/M_de_Rijke_Chair.pdf
Visitando http://www.acm.org/sigs/elections/sigir-2013/P_Bennett_Treasurer.pdf

   Revisando http://www.acm.org/sigs/elections/sigir-2013/P_Bennett_Treasurer.pdf
Visitando http://www.acm.org/sigs/elections/sigir-2013/I_Soboroff_Treasurer.pdf

   Revisando http://www.acm.org/sigs/elections/sigir-2013/I_Soboroff_Treasurer.pdf
Visitando http://www.acm.org/sigs/elections/sigir-2013/D_Kelly_Treasurer.pdf

   Revisando http://www.acm.org/sigs/elections/sigir-2013/D_Kelly_Treasurer.pdf
   tokenizando http://www.acm.org/sigs/elections/sigir-2013/I_Soboroff_Treasurer.pdf
Candidate
for
Treasurer
Ian
Soboroff
```

Fig. 17 pruebas de la araña web

Para la prueba del módulo de clustering, se utilizaron las medidas de precisión y recuperación (precision and recall). Estos, son 2 estándares para conocer el grado de relevancia de un resultado; en este caso, de los resultados obtenidos por el algoritmo GQF.

La precisión es la fracción de instancias recibidas que son relevantes, es decir, nos dice cuántas de las páginas recuperadas son realmente importantes; mientras que la recuperación se refiere a la fracción de instancias relevantes que son recibidas, es decir, de todas las páginas importantes, cuantas se recuperaron

```
Output - PT_Crawler (run)
8
se leera el documento: p095.pdf
tiene 7 paginas
1234
9
se leera el documento: race_to_the_top_comments_final.pdf
tiene 13 paginas
1176
10
se leera el documento: pdf.php(interrogacion)id=nst-0106.pdf
tiene 10 paginas
1119
11
El nuevo valor GQF es: 233.7202746457397
el nuevo cluster tiene 2 documentos
El nuevo valor GQF es: 406.30464565301855
el nuevo cluster tiene 3 documentos
El nuevo valor GQF es: 538.3632086699846
el nuevo cluster tiene 4 documentos
El nuevo valor GQF es: 702.114687519681
el nuevo cluster tiene 5 documentos
El nuevo valor GQF es: 750.8951286373949
el nuevo cluster tiene 6 documentos
Output
```

```
Cluster - Proyecto Terminal
Busqueda del crawler:   computer technology
Iniciar
resultados
race_to_the_top_comments_final.pdf
pdf.php(interrogacion)id=nst-0106.pdf
p034.pdf
p042.pdf
p095.pdf
ccsup.pdf
```

Fig. 18 pruebas del minador de datos

Las pruebas del minador de datos (fig. 18) arrojaron que 5 documentos de los 11 fueron seleccionados como importantes, pero de esos, solo 3 lo fueron realmente. Y uno de los documentos importantes, fue omitido. Dados las fórmulas de precision y recuperación:

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

Tenemos así: documentos relevantes = 4, documentos recuperados = 5

De esta manera, tenemos:

- Precision: 3/5 = 60%
- Recuperación: 3/4 = 75%

Con esto, podemos ver que se obtuvieron buenos resultados de nuestras pruebas, aunque obviamente no las óptimas.

6. Conclusiones

En este proyecto se desarrolló un sistema basado en arañas web capaz de buscar documentos en la web y de revisarlos, aun antes de descargarlos, lo que ahorra tiempo y espacio en disco duro; además de que se acopló con un módulo minador de datos, que mediante el uso de *clusters*, es capaz de otorgar al usuario los más prometedores documentos de entre los elegidos por la araña web.

Este sistema es capaz de realizar varias búsquedas y puede realizarlas en distintos puntos de partida en la web (raíces), además de que no importa el número de documentos descargados, se encargará de elegir solo los mejores para el usuario.

De las pruebas realizadas se puede concluir que el sistema es capaz de presentar al usuario los mejores documentos, sin importan con cuantos haya empezado el módulo del *cluster* o cuales hayan sido las palabras clave, aunque el encontrar documentos de tipo científico gratuitos en la web no es tan común, por lo que la cantidad de documentos encontrados no es muy alta.

Los retos durante el desarrollo de este proyecto fueron:

- Definir la manera en que la araña discrimina las páginas que visita.
- Decidir que algoritmo se debía usar en el módulo minador de datos.
- Definir como se habrían de separar los módulos, pues por restricciones de hardware, fue necesario.

Al final este es un proyecto terminado satisfactoriamente con unos pocos cambios al esquema original: se separó el proyecto en 2 módulos. El sistema diseñado e

implementado en este proyecto podrá ser extendido para buscar otro tipo de archivos (no solo en formato pdf); incluso se podrían integrar algoritmos paralelos con el hardware apropiado.

7. Bibliografía

[1] Soumen Chakrabarti, “Mining the web, discovering knowledge from hypertext data”, The Morgan Kaufmann series in data management systems, pp. 8-9, 18-22, 47-57, 67-71, 84-89, 2003.

[2] R. J. Howlett, N. S. Ichalkaranje, L. C. Jain, G. Tonfoni, “Internet-Based Intelligent Information Processing Systems”, Series of Innovative Intelligence, vol. 3, pp. 33-37, 2003.

[3] Oren Zamir, “Fast and intuitive clustering of web documents”, reporte técnico, pp. 1-6, Abr., 1997.

[4] Kenrick Mock, “A comparison of three clustering algorithms: Treecluster, Word Intersection GQF, and Word Intersection Hierarchical Agglomerative Clustering”, reporte técnico, pp. 1-7, Sep., 1998.