

Práctica 4

Buscador de información

Fecha de entrega: 14 de mayo de 2017

1 Buscadores de información

¿Alguna vez te has preguntado por qué los buscadores modernos de información en Internet funcionan tan rápido? El contenido en la red crece cada día, y sería inviable encontrar información simplemente accediendo, una a una, a todas las páginas que existen para ver en cuál se encuentra la información buscada.

En realidad, lo que hacen los buscadores es tener un *índice* de las páginas que existen. En este índice, las palabras más importantes de una página y las palabras que aparecen en ellas están emparejadas, en un sistema *clave-valor*. Así, un buscador podría crear un índice que tuviera este aspecto (aproximadamente):

URL	Palabras que aparecen en la página
http://www.agenciatributaria.es	contribuyente, desgravar, declaración, renta
https://www.ucm.es	ciencia, aprender, ucm, conocimiento
https://es.wikipedia.org	enciclopedia, libre, discusión, conocimiento
...	...

En este ejemplo, cuando un usuario busca la palabra “conocimiento”, el buscador accedería al índice de la tabla, y devolvería todos los *valores* cuya *clave* contenga la palabra “conocimiento”: <https://www.ucm.es/> y <https://es.wikipedia.org>.

Buscar información en tablas clave-valor es mucho, mucho más eficiente que buscar por el contenido de cada página. Pero la condición es obvia: es necesario crear, desde las páginas, una tabla que represente bien la información. Los buscadores hacen esto: tienen grandes

equipos de proceso automático que literalmente se “descargan Internet”, crean la tabla, y buscan en ella cuando los usuarios escribimos las palabras claves que esperamos que estén en las páginas que buscamos.

Las tablas, sin embargo, son enormes. Si buscamos la palabra “universidad” tendremos tantos resultados que el usuario no sabría cómo navegar por esa información. Para hacer la vida del usuario más sencilla, los sistemas que crean y buscan en índices también hacen una estimación de qué páginas son potencialmente más interesantes, y ordenan el resultado de la búsqueda haciendo que veamos en primer lugar las más prometedoras.

Hay muchos algoritmos para hacer esto. Uno de los más famosos es el [PageRank](#), que no sólo considera que una página es más importante cuantas más sean las páginas que enlazan a ella, sino que los enlaces desde páginas más importantes tienen aún más peso. Así, se considera que enlazar a una página la hace más relevante para una búsqueda.

Los mejores buscadores de Internet son, en general, aquellos que son capaces de asignar el mejor orden de resultados. Hoy en día, los buscadores son capaces incluso de adaptarse a cada usuario para ofrecerle las búsquedas más relacionadas con su perfil.

2 Descripción de la práctica

En esta práctica vamos a implementar un pequeño sistema análogo a los buscadores de Internet: un sistema que analice una serie de ficheros de entrada (en el fondo, las páginas web no son más que ficheros de texto, con mucha información extra aparte del contenido), cree una serie de tablas con índices y relaciones entre los documentos, y permita al usuario encontrar los archivos en los que aparece una palabra concreta. Un ejemplo de ejecución sería éste:

<p>Buscador FdI-FP</p> <p>Por favor, introduzca el nombre del fichero raíz a partir del que se creará el índice: indice.txt</p> <p>Cargando... [CORRECTO]</p> <p>Creando tablas... [CORRECTO]</p> <p>La información ha sido cargada. Puede empezar a hacer búsquedas.</p> <p>Introduzca una palabra (“fin” para terminar): edificio</p> <p>Encontrada en “indice.txt” (relevancia 1.0)</p> <p>Encontrada en “arquitectura.txt” (relevancia 0.8)</p> <p>Encontrada en “facultades.txt” (relevancia 0.63).</p>	<p>Introduzca una palabra (“fin” para terminar): facultad</p> <p>Encontrada en “facultades.txt” (relevancia 0.94)</p> <p>Introduzca una palabra (“fin” para terminar): fin</p> <p>Gracias por usar el buscador.</p>
--	---

3 Primera versión: creación de índices y búsqueda directa

Para la primera versión de la práctica, vamos a hacer un buscador simple. Este buscador analizará una serie de archivos de texto, creará tablas *palabra-archivos en los que aparece*, y permitirá hacer una búsqueda simple, sin prioridad de resultados.

3.1 Módulos

En relación a las prácticas anteriores, esta práctica es más larga, y tiene apartados concretos que se pueden implementar por separado. Por esto, en esta práctica estructuraremos el código en módulos que nos ayudarán a encapsular cada parte de la solución. Para esta primera versión, los módulos serán los siguientes:

- Módulo de listas de cadenas (`listacadenas.h`, `listacadenas.cpp`).
- Módulo de tablas clave-valor (`clavevalor.h`, `clavevalor.cpp`)
- Módulo de creador de índices (`indices.h`, `indices.cpp`)
- Módulo principal (`main.cpp`)

Recuerda que, para cada módulo, tiene que haber una pareja de archivos: un `.h` para las declaraciones y las estructuras de datos, y un `.cpp` para la implementación de las funciones.

Toda la entrada y salida por consola del programa se hará **exclusivamente en el módulo principal**. Algunos módulos tienen funciones para imprimir contenido, pero éstas sólo podrán llamarse desde el módulo principal.

3.2 Detalles de la versión simple

Desde el módulo principal se llamarán a las funciones de entrada y salida, y se usarán las funciones de los otros módulos.

En primer lugar, se pedirá el nombre del archivo origen para la creación del índice. Este archivo será un archivo de texto normal. Se considerará que todos los elementos son palabras (cadenas) sin importar si son números o secuencias de caracteres. A estos efectos consideraremos como una palabra a las sucesiones de caracteres separados por símbolos de puntuación o espacios en blanco, para saber si un carácter es uno de ellos puedes usar las funciones `bool ispunct(char c)` y `bool isspace(char c)`. Sólo hay una excepción: algunas de estas palabras serán **enlaces**, que harán referencia a otros archivos. Los enlaces son las cadenas de caracteres entre los caracteres especiales '`<`' y '`>`' por ejemplo:

`<arquitectura.txt>`

Supondremos que no hay otras apariciones de esos dos caracteres especiales.

Cuando se encuentre un enlace, no se tiene que considerar como una palabra para el índice, sino una referencia a un nuevo archivo, que también tendrá que ser indexado (lo explicamos con más detalle en la sección 3.2.3).

Una vez que se haya creado la tabla, el módulo principal preguntará al usuario una palabra. Mientras esta palabra no sea “fin”, se seguirá preguntando. Para cada pregunta, se llamará a una función del módulo de búsqueda que devolverá una lista de archivos que contienen dicha palabra, y se mostrará por pantalla.

Como ejemplo, imaginemos que tenemos el siguiente fichero inicial (`indice.txt`):

Archivo 1: `indice.txt`

El mejor edificio del mundo es la Facultad de Informática
<arquitectura.txt>, aunque los de las otras facultades <facultades.txt> no
están nada mal.

Cuando el programa carga el archivo, va incluyendo en la tabla cada una de las palabras presentes, y las enlaza con el archivo actual, ignorando signos de puntuación, los enlaces, e incluyéndolo todo en minúsculas¹:

aunque	[<code>indice.txt</code>]
el	[<code>indice.txt</code>]
edificio	[<code>indice.txt</code> , <code>arquitectura.txt</code>]
informática	[<code>indice.txt</code> , <code>facultades.txt</code>]
...	...
mejor	[<code>indice.txt</code>]
...	...

¹ Para transformar un string a minúsculas, puedes usar: `std::transform(myString.begin(), myString.end(), myString.begin(), std::tolower);` tras haber incluido la librería `algorithm`: `#include<algorithm>`, además de la librería `cctype`: `#include<cctype>`, en la que está la función `tolower`.

Si te fijas, verás que esta tabla es igual que la de la introducción, pero invirtiendo clave y valores.

Cada vez que encuentres un enlace, tienes que añadir la información del archivo correspondiente (tienes el nombre entre <...>) a la misma tabla. Para eso, en la función que crea la tabla, tienes que tener una lista de *visitados* y *no visitados*, que serán dos listas de cadenas (puedes usar el módulo de listas de cadenas que tienes que implementar).

Cuando leas un enlace, tienes que incluirlo en la lista de *no visitados*, a no ser que se encuentre previamente en la lista de *visitados* (en cuyo caso, el archivo ya se habrá procesado y no hará falta hacerlo de nuevo). Cuando hayas terminado con el archivo actual, saca el siguiente de *no visitados*, mételo en *visitados*, y procésalo. Cuando no queden archivos en *no visitados*, la creación de la tabla habrá terminado. El algoritmo se describe con más detalle en la sección 3.2.3.

A continuación, detallamos cada uno de los módulos.

3.2.1 Módulo de lista de cadenas

En este módulo se implementará un tipo estructurado que describa una lista de tamaño variable en la que los elementos serán cadenas de caracteres, lo usaremos para representar las listas de archivos donde podemos encontrar cada palabra. El tipo estructurado tendrá un campo para el tamaño variable, y un array de `string` para almacenar las cadenas. El tamaño máximo del array será una constante (100, por ejemplo). Tendrás que implementar, al menos, las siguientes funciones:

- `int getTamanno(const tListaCadenas & l)`. Devuelve el tamaño de la lista.
- `string getCadena(const tListaCadenas & l, int pos)`. Devuelve una cadena en la posición `pos`.
- `void insertar(tListaCadenas & l, const string & s)`. Inserta un elemento al final de la lista.
- `bool buscar(const tListaCadenas & l, const string & s)`. Devuelve `true` si la cadena está en la lista, y `false` si no.
- `void eliminar(tListaCadenas & l, int pos)`. Elimina el elemento en la posición `pos`.
- `void imprimir(const tListaCadenas & l)`. Imprime los elementos de la lista.

3.2.2 Módulo de tablas clave-valor.

En este módulo se implementará un tipo de datos estructurado pares clave-valor. Para ello tienes que implementar un tipo estructurado para cada registro (cada par clave-valor): la clave será una cadena de caracteres, y el valor una lista de cadenas de caracteres (según módulo descrito en la sección 3.2.1).

Después, tendrás que hacer una lista de tamaño variable en la que cada elemento sea del tipo estructurado que acabas de definir. Análogamente a la anterior, el array tendrá un tamaño máximo según una constante (200 elementos). Usaremos un elemento de este tipo para representar el índice de palabras.

Tendrás que implementar, al menos, las siguientes funciones:

- `int getTamanno(const tIndicePalabras & l)`. Devuelve el tamaño de la tabla.
- `tRegistroIndicePalabras getRegistro(const tIndicePalabras & l, int pos)`. Devuelve el registro en la posición `pos`.
- `bool buscar(const tIndicePalabras & l, const & string s)`. Devuelve `true` si la palabra es una clave en la tabla, y `false` en caso contrario.
- `int getPosicion(const tIndicePalabras & l, const & string s)`. Devuelve la posición del registro cuya clave es `s`, y `-1` si no existe.
- `void insertar(tIndicePalabras & idx, const & string palabra, const & string nombreArchivo)`. Si `palabra` es ya una clave de la tabla, añade `nombreArchivo` a la lista de archivos correspondiente. Si no, crea un registro nuevo con esa información.
- `void imprimir(const tIndicePalabras & idx)`. Imprime la tabla.

Para que las búsquedas sean lo más eficientes posibles, mantendremos las entradas de la tabla ordenadas por la clave. Aprovecharemos este orden para implementar las búsquedas usando el método de búsqueda binaria.

3.2.3 Módulo de creador de índices

En este módulo, dada una tabla clave-valor vacía (según la implementación de la sección 3.2.2) y un archivo origen, se calculará el valor (lista de páginas) para cada una de las entradas (palabra), quedando almacenado en la tabla correspondiente.

Para ello, el algoritmo es el siguiente:

1. Inicializar visitados y no-visitados a listas vacías
2. Meter el archivo inicial en no visitados
3. Mientras haya algún archivo en no-visitados
 - a. Sacarlo y meterlo en visitados
 - b. Abrir el archivo:
 - i. Por cada palabra que no sea enlace o signo de puntuación, insertar en la tabla esa palabra y el archivo actual (con el procedimiento `insertar`)
 - ii. Por cada enlace que no esté en visitados o en no-visitados, meterlo en no visitados

Necesitarás la siguiente función (al menos, puedes usar funciones auxiliares):

- `Void crearTabla(tIndicePalabras & tabla, const string & archivoInicial)`. A partir de `archivoInicial`, rellena la tabla `tabla`, tras haberla inicializado (a vacía).

Puedes asumir que todos los enlaces (<...>) apuntan a archivos que existen, no hace falta que compruebes esto en el código. Asegúrate, eso sí, de que realmente existen los ficheros referenciados en tus datos de prueba.

3.2.4 Módulo principal

El módulo principal simplemente tiene la función `main`, que llama a las funciones auxiliares que se han implementado anteriormente, imprime por pantalla el estado de la creación de las tablas, y pregunta al usuario por las palabras que quiera buscar en el índice, tal como se muestra en el ejemplo de la sección 2.

4 Segunda versión: ordenar los resultados por relevancia

En esta segunda versión de la práctica vamos a mejorar nuestro buscador, haciendo que los resultados no sólo sean correctos, sino que estén **ordenados por importancia**. Para esto, vamos a implementar una **versión sencilla de PageRank**, en la que los **archivos más enlazados por otros archivos** sean considerados más relevantes.

En fórmulas, si llamamos:

- $PR(A)$ al PageRank de la página A y
- $N(A)$ al número de enlaces que la página A contiene,

queremos que se cumpla:

$$PR(A) = (1 - c) \cdot \frac{1}{N} + c \cdot \sum_{E \text{ tiene un enlace hacia } A} \frac{PR(E)}{N(E)}$$

Dónde c es un número real entre 0 y 1, que introduce la posibilidad de que un usuario decida no seguir ningún enlace del archivo en el que se encuentra y elija al azar un archivo entre el resto, los expertos aseguran que un famoso buscador usa el valor **0.85** para la constante c .

Así numeramos todas las páginas A_1, A_2, \dots, A_N llamamos N_i al número de enlaces desde la página A_i y definimos la matriz M' dada por $m'_{ij} = \frac{1}{N_j}$ si A_j tiene un enlace hacia A_i , $m'_{ij} = \frac{1}{N}$ si la página A_j no tiene enlaces hacia ninguna página y $m'_{ij} = 0$ en otro caso.

La condición anterior se puede escribir matricialmente como:



$$MX = X$$

con X un vector columna de forma que cada componente X_i es el PageRank de la página A_i y M la matriz $(1 - c)C_N + cM'$, donde C_N es la matriz con todas las entradas iguales a $\frac{1}{N}$.

Un teorema de álgebra lineal nos asegura que, debido a la estructura de la matriz M , la ecuación anterior tiene solución. Es fácil convencerse de que si X es solución AX también lo es. Como lo que queremos es usar el vector X para ordenar nuestras páginas, nos da igual un X que otro proporcional. Otra propiedad interesante es que un método iterativo nos permite aproximarnos a una solución: empezamos con un vector de unos

calculamos $w_1 = Mw$, si Mw_1 está muy próximo a w_1 paramos si no continuamos aplicando M a w_1 obteniendo w_2, \dots , así hasta que Mw_n esté muy próximo a w_n . Una buena opción para decidir qué significa que dos vectores están muy próximos es que los cocientes entre sus componentes sean cercanas a 1.0 (por ejemplo que la diferencia entre estos cocientes y 1.0 sea menor que 10^{-5}).



Si interpretamos m_{ij} como la probabilidad de que un “navegante” pase de la página A_j a la A_i , $1 - c$ es la probabilidad de que el usuario no siga ningún enlace y acceda a una página cualquiera.

Para implementar esto, vamos a añadir un módulo nuevo, y vamos a modificar el programa principal para que lo use.

4.1 Módulo de trabajo con matrices

Para empezar, necesitamos recopilar la información sobre los enlaces entre las páginas, construiremos una matriz $L = l_{ij}$ de forma que $l_{ij} = 1.0$ si hay un enlace desde la página A_j a la página A_i y $l_{ij} = 0.0$ en caso contrario. La numeración de las páginas corresponde al orden de aparición en alguno de los archivos. Para generar la matriz L mientras generamos el índice de palabras tenemos que modificar un poco el algoritmo anterior. Para empezar, necesitamos otra lista de cadenas que llamaremos totales y una matriz L que irá creciendo de tamaño. Así, el algoritmo quedará como sigue:

- 1) Inicializar visitados, no-visitados y totales a listas vacías, totales será siempre la unión de visitados y no-visitados.
- 2) Meter el archivo inicial en no visitados y en totales e iniciar la matriz L a tamaño 1 con un cero
- 3) Mientras haya algún archivo en no-visitados
 - a) Sacarlo y meterlo en visitados, guardar su posición j en totales.
 - b) Abrir el archivo:
 - i) Por cada palabra que no sea enlace o signo de puntuación, insertar en la tabla esa palabra y el archivo actual (con el procedimiento insertar)
 - ii) Por cada enlace del archivo abierto:
 - (1) Si está en totales en la posición i poner l_{ij} a 1.0

- (2) Si no está en totales añadirlo a totales y no-visitados, ampliar el tamaño de L en 1 (de forma que las nuevas entradas sean nulas) y poner $l_{t-1,j}$ a 1.0 siendo t el nuevo tamaño

Así tendremos que modificar el procedimiento crearTabla:

```
void crearTabla(tIndicePalabras & tabla, tListaCadenas & totales, const string & archivoInicial, tMatriz & L)
```

para obtener la lista de enlaces totales y la matriz L .

tMatriz es el tipo de datos que se describe en el siguiente apartado.

4.1.1 Implementación del módulo para el trabajo con matrices.

Debes declarar el tipo **tMatriz** que contenga un array bidimensional cuadrado de números reales con tamaño MAX_TAM y una variable de tipo `int` para guardar la dimensión de la matriz en cada momento.

Declararemos también el tipo **tVector** de forma adecuada.

Necesitaremos al menos **funciones** con las siguientes cabeceras:

- `tMatriz matrizConstante(double x, int n)`. Crea una matriz de $n \cdot n$, en la que todos los valores iniciales son x .
- `tVector operator*(const tMatriz & m, const tVector & v)`. Multiplica una matriz por un vector.
- `tMatriz operator*(double x, const tMatriz & m)`. Obtiene la matriz que se obtiene multiplicando cada elemento de m por x .
- `tMatriz operator+(const tMatriz & m1, const tMatriz & m2)`. Suma dos matrices.
- `tMatriz desdeEnlacesAMatriz(const tMatriz & L)`. Devuelve la matriz M obtenida de L como en la introducción de la sección 4.
- `tVector normaliza(const tVector & v)`. Devuelve un vector proporcional a v con la primera componente igual a 1.0.
- `tVector vectorPropio(const tMatriz & M)`. M es una matriz tal que todas sus columnas suman 1, devuelve un vector v con la primera componente igual a 1 tal que $M \cdot v \simeq v$.

4.2 Modificaciones en el módulo principal

Debes modificar crearTabla como se ha indicado para obtener la matriz L y la lista de archivos **1. Aplicando las funciones del módulo matriz obtenemos un v de tipo tVector**





que contiene el PageRank de cada archivo en 1. Con esa información debes ordenar las listas de archivos que obtengas en las consultas.

5 Entrega de la práctica

La práctica se entregará a través del Campus Virtual. Se habilitará una nueva tarea **Entrega de la Práctica 4** que permitirá subir un archivo comprimido con el código fuente (todos los módulos: cada uno de los archivos de cabecera y cada uno de los archivos fuente).

Fin del plazo de entrega: **14 de Mayo a las 23:55.**