

# Concurso de EDA

Omar Pera Mira

4 de mayo de 2007

# Índice

<b>1. Funcionamiento del programa</b>	<b>1</b>
1.1. Procesado de datos . . . . .	1
1.1.1. Entrada de datos . . . . .	1
1.1.2. Cuckoo Hashing . . . . .	1
1.1.3. Estructura de datos para el fichero <i>Diccionario</i> . . . . .	2
1.1.4. Estructura de datos para el fichero <i>Bigramas</i> . . . . .	3
1.2. Resolución del problema . . . . .	3
1.2.1. Salida de datos . . . . .	3
1.2.2. Algoritmo de traducción . . . . .	3
<b>2. Optimizaciones realizadas</b>	<b>4</b>
2.1. Quicksort . . . . .	4
2.2. <i>Pooling</i> . . . . .	4
2.3. Tipo de dato para la probabilidad . . . . .	5
2.4. Otros . . . . .	5

## Resumen

Este documento es una memoria del programa realizado para el concurso de EDA 06/07.

La mayor parte de la documentación del programa reside en el directorio *doc*, el cual esta generado por Doxygen<sup>1</sup>. Este software permite extraer toda la estructura de tu código fuente, generando una navegación intuitiva por las distintas partes del programa. La información de cada fichero fuente se encuentra en su correspondiente cabecera.

---

<sup>1</sup>Sistema de documentación para diversos lenguajes de programación. Software con Licencia GPL.

# 1. Funcionamiento del programa

Para poder abordar todos los aspectos del programa realizado de una manera más clara, he dividido el documento en varias secciones según su contenido.

## 1.1. Procesado de datos

El primer paso para resolver el problema planteado es procesar toda la información proveniente de los ficheros *Diccionario* y de *Bigramas*.

### 1.1.1. Entrada de datos

Para cubrir la necesidad de leer datos de fichero de una manera eficiente, he optado por usar archivos proyectados en memoria (del inglés “memory-mapped files”).

Esta forma de acceder a un fichero se realiza con la llamada al sistema de UNIX *mmap*. Y su forma de actuar no es la de la E/S tradicional, trayendo los datos de disco a memoria y copiados a un buffer para después procesarlos. Al proyectar el contenido de un archivo en memoria, los datos pueden ser accedidos mediante referencias a posiciones de memoria con punteros, basándose en el sistema de gestión de memoria virtual y sin necesidad de crear buffers intermedios.

Con este sistema sólo se accedera a cada página del disco cuando necesite ser leída, por lo que la lectura se realiza en bloques del tamaño de página, y de la misma manera sólo se volcaría a disco las páginas modificadas (no es nuestro caso). Éste método facilita el acceso y manipulación de datos, pues podemos pensar que esta cargado completamente en memoria referenciando cualquier parte del fichero, cuando en realidad el sistema se encarga de forma transparente.

Al utilizar dicha forma de leer los ficheros, no he necesitado de buffers auxiliares, sólo he realizado dos funciones que simulan el funcionamiento de *strtok()* adaptado a las necesidades de la entrada de datos.

### 1.1.2. Correspondencia entre *tokens* y *enteros*: Cuckoo Hashing

El siguiente paso es mantener una correspondencia biunívoca entre las palabras o *tokens* con un número entero.

Para ello, la estructura más apropiada es una *tabla hash*. He creado dos tablas distintas, una para todas las palabras del fichero *Diccionario* que fueran del lenguaje origen, y otra para todas las del lenguaje destino (ya sean del fichero de *Bigramas* o *Diccionario*).

Hemos elegido un método para realizar la correspondencia denominado *Cuckoo Hashing*<sup>2</sup>. La idea básica consta usar dos funciones hash diferentes, esto provoca que haya 2 posibles ubicaciones para la misma clave. Además, esta implementada con dos tablas hash, por lo que cada función hash corresponde a una tabla distinta. Proporciona una búsqueda en **tiempo constante** en el peor caso (inspecciona una cubeta en cada tabla).

Entrando en detalle, el funcionamiento general es el siguiente: con cada *clave*(cadena) nueva a insertar, a, aplicando una función hash, se devuelven 2

---

<sup>2</sup>Algoritmo realizado por Rasmus Pagh y Flemming Friche Rodler en 2001.

valores enteros,  $h1(a)$  y  $h2(a)$ . Dicha *clave* se inserta en  $Table1[h1(a)]$ . Si esa cubeta (sólo hay espacio para una *clave*) esta ocupada por otra,  $b$ , se mueve a  $Table2[h2(b)]$ . Las *claves* son insertadas y movidas a las distintas tablas hasta que encuentre una cubeta vacía o el limite se haya alcanzado (si no podríamos entrar en un bucle infinito). Una solución al llegar limite, podría ser hacer un *rehashing* con distintas funciones hash (no lo hemos implementado).

Al no implementar la solución de realizar un rehashing(inviabile para este concurso), si el tamaño de los ficheros del oráculo cambiara periódicamente puede que nuestro método no funcionara. Para solucionar esto, hemos obtenido el tamaño de los ficheros y aproximado el numero de líneas/palabras para realizar el tamaño de las 2 tablas dinámico (aproximandolo además a una potencia de 2), de forma que aunque no hemos logrado del todo solucionar el problema, llegamos a una aproximación. Otra opción sería ir probando hasta que no falle el oráculo, con tamaño estático, pero hemos visto esta solución menos "elegante".

La inserción en este tipo de *tabla hash* puede llegar a parecer "lenta", pero le resta importancia cuando hablamos de la búsqueda. Para buscar una cadena,  $k$ , sólo tendremos que comprobar si esta situada en  $Table1[h1(k)]$  o en  $Table2[h2(k)]$ , por lo que el acceso es constante.

Por otra parte, hemos implementado algunas optimizaciones para hacerla más rápida si cabe. En primer lugar no hemos usado dos funciones hash distintas, sino una función<sup>3</sup> que devuelve dos *hashes* de 32-bit.

En segundo lugar, a parte de guardar en una cubeta la *clave* (cadena) y un valor único, albergo las 2 hashes de 32-bit sin módulo aplicadas a la cadena de caracteres (*bitwise AND* en realidad). Todo ello, para el posterior ahorro de aplicar la función hash de nuevo al mover elementos de una tabla a otra y, más importante si cabe, para no comparar cadenas con *strcmp()* sino comparar 2 enteros de 32-bit<sup>4</sup>.

Este último cambio en el planteamiento de la tabla hash nos permite no guardar ninguna cadena de caracteres del fichero *Diccionario* que pertenezcan al lenguaje origen.

Por último, hemos tenido en cuenta que cualquier palabra del lenguaje destino que no estuviera en el fichero *Diccionario*, nunca va a ser traducida, por lo que al leer el fichero de *Bigramas* nunca inserto en su correspondiente *tabla hash*, simplemente busco, y si no lo encuentra no introduzco dicha terna en la estructura de Datos apropiada para ello.

### 1.1.3. Estructura de datos para el fichero *Diccionario*

Tras realizar la asociación entre *tokens* y enteros, desde este punto sólo trabajamos con números. Ahora procederemos a explicar la estructura creada para albergar cada terna del *Diccionario Estadístico*.

Para ello hemos creado una clase (*ListNode*) que alberga toda la información referente a **una** palabra del lenguaje origen. Tiene una lista de nodos, y cada uno de ellos contiene el *id* de la palabra destino<sup>5</sup> y su probabilidad condicional asociada.

---

<sup>3</sup>Realizada por Bob Jenkins en Mayo 2006

<sup>4</sup>En conjunto estos dos enteros sin acotarlos con el tamaño de la tabla son únicos (probabilidad despreciable)

<sup>5</sup>LLamaremos *id* al numero que identifica a una palabra

En resumen, para guardar todas las ternas, hemos creado un vector de objetos de dicha clase indexado con el *id* de la palabra origen, siendo muy fácil acceder a lista de "traducciones" de cada palabra origen.

#### 1.1.4. Estructura de datos para el fichero *Bigramas*

La estructura utilizada para las ternas del fichero de *Bigramas* no es la misma que la anterior. Para guardar los *id* de las 2 palabras consecutivas se pensó en que dichos enteros de 32-bit formaran una *clave* y la probabilidad condicional su *valor*. Se ve claro que se trata de una *tabla hash*.

Hemos implementado una tabla hash básica en la que el nodo consta de un entero de 64-bit, que es la concatenación de los enteros de 32-bit que constituyen los *id* de los 2 *tokens* y su probabilidad. Por lo que para buscar de probabilidad condicional de 2 cadenas procederemos a concatenar sus identificadores y aplicarles una función hash<sup>6</sup> que da como resultado un valor hash de 32-bit. Tras esto sólo queda recorrer la lista comparando la *clave*.

Por otra parte, a medida que leemos el fichero de *Bigramas* insertando en la anterior hash, también se guarda en un vector la máxima probabilidad asociada a cada palabra anterior, indexada por el *id* de ésta. Se explicará en la sección 1.2.2 su uso.

## 1.2. Resolución del problema

Una solución básica al problema propuesto, se obtendría iterando sobre la lista de "traducciones" de la palabra a traducir e ir buscando en la tabla hash la probabilidad asociada a la palabra anterior y la supuesta traducción actual. Todo ello hasta llegar al final de la lista de la palabra a traducir, para buscar el máximo productor de probabilidades, siendo su *id* la traducción correcta.

A continuación presentamos la forma de resolver el problema con mejoras sobre el esquema básico.

### 1.2.1. Salida de datos

Para volcar por salida estándar la solución del problema, nos limitamos a reservar bloques de memoria, concatenando las distintas traducciones en dichos búffers con una versión mejorada de *strcat()*, para su posterior volcado mediante una llamada al sistema *write()*.

### 1.2.2. Algoritmo de traducción

Para encontrar la traducción correcta de cada palabra, primero vemos si sólo hay un elemento en la lista de nodos de la palabra a traducir, obteniendo directamente la solución, y si no, operamos como mostramos a continuación.

Los pasos a seguir son: ordeno parcialmente por probabilidad la lista de nodos correspondiente a la palabra a traducir del fichero *Diccionario*, mediante un Quicksort adecuado con varias mejoras que se citarán en la sección 2. Tras esto, itero sobre la lista empezando por la probabilidad más alta, y para cada *id* de dicha lista, concateno con el *id* de la palabra anterior y lo busco en la

---

<sup>6</sup>Función *hash6432shift* realizada por Thomas Wang, Jan 1997

tabla hash de *Bigramas*, calculando cada vez su producto y el máximo hasta el momento.

Por otra parte, hay que pensar que no valdría la pena ordenar si no hubiera alguna forma de parar el bucle. Para dicho propósito en cada iteración, aparte de realizar lo anterior, calculamos el máximo producto de probabilidades que se puede llegar a obtener en iteraciones posteriores.

Para obtener ese valor en cada iteración, lo óptimo sería los restantes máximos de la lista de ternas relacionadas con la palabra a traducir y de las ternas relacionadas con la palabra anterior de *Bigramas*, pero sería muy costoso en el caso de las ternas del fichero *Bigramas* al no estar ordenado, por lo que en vez del siguiente máximo, se realiza una estimación con la probabilidad máxima. Ésta máxima probabilidad relacionada con todas las ternas de la palabra anterior, se obtiene del vector indexado por los dichos *id* introducido en la sección 1.1.4. Si el máximo producto calculado es mayor que el esperado en futuras iteraciones, obtenemos la solución.

## 2. Optimizaciones realizadas

A parte de las optimizaciones de alguna estructura de datos y otras mejoras que ya he explicado, quería citar alguna más que no encaja en ningún apartado anterior.

### 2.1. Quicksort

Como ya expusimos en la sección 1.2.2, ordenamos una lista enlazada de nodos por probabilidad. Sólo es útil para tener una mayor probabilidad de romper el bucle de traducción que itera sobre la lista actual mediante la *Probabilidad esperada*.

Las mejoras son las siguientes:

- Para mejorar el caso peor elegimos como pivote del algoritmo *Partición*, la mediana de 3 elementos.
- Al no ser necesario en ocasiones ordenar todos los elementos, sólo ordenamos los elementos mayores del pivote.
- Utilizamos la técnica *tail recursion*, para eliminar una recursión de los dos, pero como hemos eliminado la posibilidad de ordenar los menores del pivote, el algoritmo será iterativo.
- Cuando el número de elementos está por debajo de un umbral, se ordena por **Inserción directa**.

### 2.2. Pooling

El uso de un *pool* de memoria permite la reserva de memoria dinámica del mismo modo que *new* o *malloc*. Pero si el uso de dichas primitivas es muy frecuente y son pequeños los bloques de memoria reservados, la fragmentación y el coste puede llegar a ser un problema. La solución pasa por reservarnos un bloque adecuado de memoria para su posterior uso en cada inserción del tipo de dato.

Su implementación es muy sencilla y sólo se ha de cuidar el tamaño de cada *pool*, ya que en este caso en el momento en que se requiera más memoria de la previamente reservada, producirá un error. Cada tamaño de *pool* creado se ha definido dinámicamente en relación al tamaño del fichero y su tipo de dato. Se han creado una clase *Pooling*, y se ha utilizado para:

- Bloque de nodos para la lista enlazada de cada objeto de la clase *ListNode* (ternas de *Diccionario*)
- Bloque de nodos para la hash de probabilidades (ternas de *Bigramas*)
- Bloque de memoria para las cadenas de caracteres

### 2.3. Tipo de dato para la probabilidad

Para ganar una mayor eficiencia en el producto de probabilidades, no las hemos guardado en *coma flotante* como sería lo normal, sino en base decimal en un entero de 32-bits.

El producto de dos probabilidades podría dar lugar a desbordamiento, por lo que realizo un *casting* al realizar el producto a un entero de 64-bit.

### 2.4. Otros

Todas las estructuras de datos creadas y los *pools* de memoria utilizados han sido creados dinámicamente según un criterio. La mayoría son una aproximación según el número de líneas/palabras, o bien, como en el caso del vector de probabilidades máximas del fichero de *Bigramas*, ya sabemos de antemano al leer el fichero, cuál es el número exacto (Número de palabras no repetidas).

A parte de dicha estimación de los tamaños, hemos redondeado a una potencia de dos todos estos tamaños para ganar una mayor eficiencia en el acceso a una posición de cada estructura de datos, y para realizar *bitwise AND* para ahorrarnos la operación módulo. No es necesario realizar dichos cálculos pero creemos conveniente esta opción de realizarlo que no declarar todas las estructuras estáticas.

Hemos realizado la inserción de nodos en la tabla hash de probabilidades en un bucle a parte, para aprovecharnos de la localidad del código. Para cada palabra de *Bigramas* sólo le asignamos el siguiente hueco libre en el *pool* de dichos nodos.

Mediante una llamada al sistema *advise()* aconsejamos al kernel la forma de tratar la memoria (secuencial), lo utilizamos después de proyectar los ficheros en memoria.

No es una optimización, pero hemos encapsulado todas las variables globales necesarias a todo el programa en la clase *varGlobal*. Todos los atributos y miembros han sido declarados estáticos, para una mayor claridad y legibilidad en el código proporcionado.