Memoria Tokenizador (Práctica 1)

El desarrollo de la clase Tokenizador entregada ha sido guiado principalmente por la reutilización de código. Si bien la eficiencia temporal ha sido tenida en cuenta para algunas decisiones, el objetivo posible era tener un código lo más corto y maleable posible para facilitar la codificación y posterior corrección de errores.

OutputIF, OutputString y OutputList

Con esto en mente, una vez establecido un entorno de trabajo y acabadas las funciones constructor, *getter* y *setter*, el siguiente paso fue trazar un árbol de dependencias entre las funciones Tokenizar, de manera que el algoritmo de tokenización estuviese implementado una única vez. El problema principal para lograr esto radicaba en los dos tipos de salida diferentes que el programa utiliza: almacenar en un string para escribirlo en memoria permanente y almacenar en una lista de strings. Para abstraer ambos comportamientos se creó la clase *OutputIF*, una interfaz que implementa una única función: la utilizada para añadir una palabra a la colección (*OutputIF::add(const string&))*.

Por cada método de almacenamiento se creó una implementación que encapsulase una referencia al tipo de almacenamiento utilizado (*OutputString y OutputList*), con lo que la utilización de una clase *Output* consistiría en crear el almacenamiento, enviar la referencia al constructor, y una vez ejecutado el algoritmo de tokenización, utilizar directamente el almacenamiento creado. Esto evita que *Output* necesite implementar funciones más allá de la de añadir palabras.

Conversión a minúsculas y sin acentos

Tal y como se recomendó en la asignatura, la implementación de conversión fue implementada en tiempo constante utilizando un array a modo de función precalculada. El array (*char conversion[256]*) se rellena al crearse la clase.

Cómo el uso de la conversión depende del estado de la clase (en concreto de la variable *bool pasarAminuscSinAcentos*), se decidió que al cambiar el estado de la clase, se cambiase automáticamente el contenido del array, para evitar comprobaciones constantes del estado que resultarían en un código menos legible y posiblemente menos eficiente. Para implementar, se emplea otro array (*char normal[256]*) que actuaría como función de no conversión. El acceso a los arrays se hace a través de un puntero (*char* addChar*) que se hace apuntar al array pertinente según el estado de la instancia.

Aunque pueda parecer un desperdicio de recursos utilizar un array para no hacer ningún cambio al carácter, la decisión se tomó en base a legibilidad del código, y el coste de la operación se mantiene constante, con lo que es improbable que haya impacto negativo significativo, aunque esta hipótesis **no ha sido probada**.

Reorganización de acceso a delimitadores

Una de las técnicas empleadas para intentar mejorar la eficiencia del tokenizador fue reorganizar el acceso a los delimitadores. La idea era utilizar el texto ya tokenizado para estimar la frecuencia de aparición de los delimitadores, de manera que los más frecuentes se comprobasen antes. Para esto se utilizan tres arrays:

- *idxDelims* contiene los delimitadores en el orden proporcionado, además del carácter ‘ ’ si los casos especiales están activados.
- *idxCount* contiene el conteo de apariciones por el que se ordenan los delimitadores. Utiliza números en coma flotante para evitar el riesgo de desbordamiento utilizando su representación interna: a partir de un cierto punto (2^{24}) sumar uno no cambia el número almacenado[1], actuando como un límite superior que no reordena los delimitadores (al contrario de lo que ocurriría en un overflow)
- *idx* contiene los índices de los delimitadores ordenados según su frecuencia.

Cuando un delimitador es encontrado, se llama a la función *encontradoDelimitador*, que aumenta el número de apariciones del delimitador encontrado y reordena si es necesario.

Este método obtuvo un aumento de eficiencia significativo (de 14 segundos a 9 segundos, una mejora del 150%), muy probablemente por la predominancia del espacio como delimitador. Al ser tan frecuente en comparación al resto, se evita gran parte de los recorridos por el vector de delimitadores. De esta manera, aunque el método empeora el peor caso sin mejorar el mejor, consigue que la mayoría de casos sean más cercanos a este último.

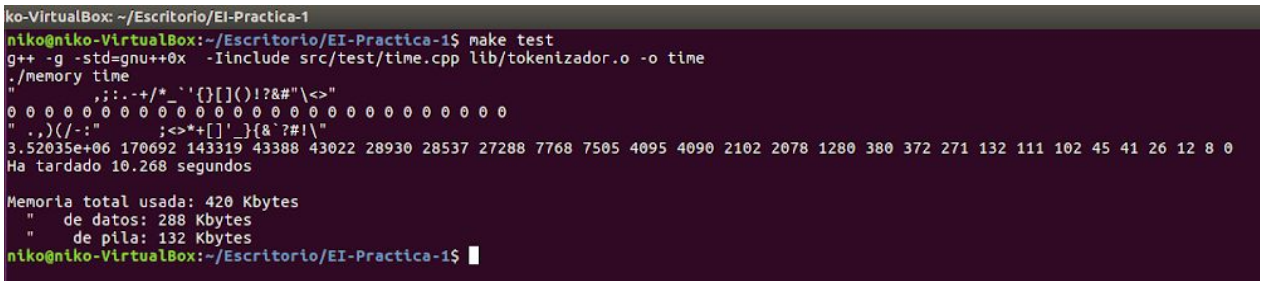

Figura 1: ordenación de los delimitadores tras tokenizar el corpus, junto con el número de apariciones.

[1] https://stackoverflow.com/questions/30943156/numpy-sum-function-returns-1-67772e07

Implementación de casos especiales

Para la implementación de los casos especiales, se utilizó un autómata de estados basado en punteros a función. El objetivo de esta decisión era mantener el algoritmo de Tokenizar ajeno a los cambios de funcionamiento de los casos especiales (no se pudo conseguir completamente por falta de tiempo, pero sería trivial implementar un nuevo estado con el código de comprobación de última palabra al acabar el string).

El autómata está formado por dos vectores de punteros a función: *funcionesDelim* contiene el código del estado en el caso de que se encuentre un delimitador, *funcionesNoDelim* contiene el código del estado en el caso de que se encuentre un no delimitador.

De nuevo se optó por evitar las comprobaciones y en su lugar ligar el estado de la clase a su funcionamiento, de manera que cambiar los casosEspeciales también cambia el puntero en la posición 0 de los vectores que representan el autómata de estados. Si los casos especiales están desactivados, el autómata permanece en el estado 0 con el código sin casos especiales.

Aunque los estados para implementar multipalabra, acrónimos y URL son relativamente sencillos, la manera de implementar los decimales acabó complicandose, ya que por falta de tiempo, en lugar de diseñar un autómata de estados convencional, la implementación se basó en heurísticas experimentales y la comprobación de algunos casos concretos (la mayoría de ellos los presentes en las pruebas automáticas de la práctica) para asegurar el funcionamiento correcto. La idea de los estados 5 al 10 (encargados de la detección de decimales) se basa en que el delimitador coma (estados 7, 9 y 10) debe interpretarse de manera distinta según los caracteres posteriores, mientras que el delimitador punto (estados 6 y 8) siempre se interpreta como no delimitador (a excepción del primer punto en expresiones como .3). Es por esto que existe un string *especulativoNum*, para almacenar los caracteres que aun no sabemos si pertenecen al número detectado o si son un acrónimo o palabra independiente que ha sido separada por una coma. El caso especial de correo electrónico **no está implementado**.

Otros detalles

Se probaron más métodos para conseguir mejoras de eficiencia, aunque fueron infructuosos. Entre ellos se encuentran:

- Intento de mejora de eficiencia de la lectura de ficheros a través de distintas funciones. La función utilizada en el enunciado de la práctica (*getline*), obtenía mejores resultados que las demás funciones, incluidas funciones de lectura por *chunks* de tamaño estático.

- Eliminación de llamadas a funciones virtuales y punteros a función. Una de las desventajas de utilizar estos métodos es que las llamadas a las funciones son más costosas, con lo que se comprobó la eficiencia obtenida de utilizar métodos alternativos, aunque no se apreció diferencia alguna.