



EXPLOTACIÓN DE LA INFORMACIÓN

Práctica 1 - Tokenizador

Raúl Beltrán Marco
23900664F

ÍNDICE

| | |
|---|---|
| Presentación del problema | 2 |
| Hipótesis de partida..... | 2 |
| Análisis de la solución implementada..... | 3 |
| Justificación de la solución elegida..... | 4 |
| Análisis de las mejoras realizadas en la práctica | 4 |
| Análisis de complejidad (teórica y práctica) del tokenizador..... | 6 |
| Complejidad temporal..... | 6 |
| Complejidad espacial | 7 |
| Nuevo Tokenizador..... | 8 |

Presentación del problema

En esta práctica se nos ha pedido crear un Tokenizador el cual deberá ser capaz de **tokenizar** una **cadena** y **añadir** las distintas **palabras** que separe en un **list<string>** pasado por referencia.

A su vez el programa deberá ser capaz de detectar casos especiales en los que tendrá que ignorar en ocasiones los delimitadores definidos por el usuario con tal de obtener la palabra correcta.

Además de clasificar los distintos casos especiales mediante las heurísticas definidas en el pdf de la práctica y siguiendo un orden de prioridad a la hora de detectar los casos especiales:

URL – Números – E-mail – Acrónimos – Guiones

Hipótesis de partida.

Sin haber implementado nada del código lo primero que pienso es que la función más importante del código es la función **void Tokenizar()** la cuál realizará un tipo de tokenización especial dependiendo de la variable **casosEspeciales**.

Por tanto, lo primero que hago es realizar una separación mediante un **if** con la cual obligar al programa a realizar un bucle u otro con tal de que en caso de **casosEspeciales** sea **false**, que en el bucle no se realicen comprobaciones innecesarias, a pesar de repetir el bucle while que se encarga de recorrer la cadena a tokenizar.

Respecto a gestionar el sistema de prioridad y detección de en qué caso especial estamos actualmente, lo primero que pensé fue en una estructura amplia de else-if en el que mediante la función **.find()** que hay en la clase string junto a una lista de string compuesta por los delimitadores especiales de cada caso.

Al encontrar un problema con el objeto iterator y obtener el string que quería en cada comparación, decidí anidar un bucle for en el que itero hasta que encuentro un caso especial, cuando encuentro uno, hago un break y mediante un switch gestiono los distintos casos.

TokenizarH (str, delimiters, pos, lastPos, tokens)

1. **string especiales;**//Variable en la que se guardan los delimitadores con los que se detecta el caso especial
2. **string delimitadores= delimiters;** //Guardamos delimiters en una auxiliar
3. **delimitadores.remove(especiales);** //Se eliminan los delimitadores especiales del string delimitadores
4. **string::size_type fin= str.find_first_of(delimitadores, lastPos);** //Detecta el final del caso especial
5. **string casoEspecial= substr(lastPos, fin – lastPos);** //Guardamos en un string el token completo
6. Comprobamos que cumpla las condiciones mediante **if-else** o lo que sea necesario.
7. Realizamos bucle recorriendo el string generado en el paso 5 en busca que se cumplan las condiciones.
8. En caso de que NO se cumplan las condiciones se modificará pos y se realizará un return false.
9. En caso de que SI se cumplan las condiciones se realizará el paso 10.
10. Se añade el string del paso 5 a tokens además de modificar pos y lastPos para leer el siguiente token y realizará un return true.

Para este algoritmo hemos utilizado las estructuras de datos como list y string (vector), con las cuales cubrimos todas las necesidades del tokenizador.

Justificación de la solución elegida

He decidido optar por este algoritmo porque creo que usar lastPos y pos es la mejor forma posible de “separar” una cadena de palabras además de todas las funciones que dispone string en C++ para facilitar dicha tarea.

También había pensado en realizar una conversión de string a list<string> con todas las palabras de la cadena y después recorrer esta lista elemento a elemento mirando a ver si se trataba de un caso especial, pero descarté esta idea debido a que luego se pedía volver a separar las palabras si no cumplían la condición y eso ya no sabía cómo implementarlo.

Además, que tengo poca experiencia con list<string> y no me sentía cómodo utilizándolo, no sé si usar varios strings o string::size_type es peor que la idea que tenía de hacer una list<string> en cuanto a complejidad espacial, pero al final utilizar lastPos y pos me ha resultado más cómodo a la hora de plantear el algoritmo final.

Análisis de las mejoras realizadas en la práctica

He realizado varias mejoras en varias heurísticas, en la mayoría he cambiado el bucle for que se encargaba de leer los string carácter a carácter por un bucle while en el que de forma parecida al implementado en Tokenizar hay un string::size_type que con ayuda de la función find_first_of se va actualizando y avanzando con “saltos” yendo directamente a la posición en la que hay un delimitador y aplicando las condiciones pertinentes.

El cambio más destacable a la hora de recorrer los bucles lo encontramos en la función Email() la que se encarga de recorrer el email para verificar que efectivamente se trata de uno.

En esta función paso de hacer dos bucles for en los que recorría el lado izquierdo y derecho del email, usando el '@' como separador a utilizar un único bucle while en el que con la ayuda otra vez de la función find_first_of miro los delimitadores y si el primero que encuentro no se trata

de un @ entonces la función devuelve false, reduciendo una complejidad de $O(n+m)$ a una complejidad $O(k)$ en la que k es menor que n y m (tamaño del string izq y derecha).

También he mejorado la eficiencia de TokenizarNumero() y Filtrar() (La función que se encarga de pasar mayúsculas a minúsculas y quitar acentos) empleando la misma idea.

En vez de ir uno a uno en TokenizarNumero primero miro si hay una letra en el string tokenizado usando la función find_first_not_of ("0123456789,.\$%", 0) lo cual me permite luego crear un bucle while que voy "saltando" entre delimitadores (,.\$%) reduciendo mucho el número de iteraciones. (Antes miraba carácter a carácter con muchos if)

Lo mismo con Filtrar() pero usando dos strings, uno con todas las mayúsculas y otro con todos los acentos y Ñ que reconoce el programa, realizando un bucle que no acaba hasta que no encuentra mayúsculas o acentos en el string pasado por parametro.

La mayor mejora del programa se encuentra en la función auxiliar OpenFile() la cual llamo desde GestionarFicheros() (Función auxiliar que uso para evitar código repetido a la hora de llamar a todas las funciones relacionadas con ficheros)

La función OpenFile() lo que hace es leer TODO el fichero que se le pasa por parámetro y lo vuelca en una variable char * la cual devuelvo como string después de liberar memoria, pasándole a la función Tokenizar(string,list<string>) el fichero entero como parámetro.

Además, también he cambiado el bucle que servía para escribir en los ficheros de salida por uno mejor.

```
if(NomFichSal != "")
    fSalida.open(NomFichEntr.c_str());
else
    fSalida.open((NomFichEntr + ".tk").c_str());

for(const auto& v : tokens)
    fSalida << v << '\n' ;

fSalida.close();
return true;
```

Estos dos cambios provocaron el mayor cambio con respecto a la complejidad temporal, a la hora de pasar el CORPUS que hay en prácticas pasó de 11-12 segundos de ejecución a 3,8-4,2 segundos aproximadamente.

Análisis de complejidad (teórica y práctica) del tokenizador

Complejidad temporal

El **peor de los casos** viene dado cuando las booleanas `pasarAminuscSinAcentos` y `casosEspeciales` son **true**.

Obteniendo las siguientes complejidades:

(5) `Filtrar(str);` → $O(\text{str})$ //str= nº de mayúsculas y acentos del string pasado por parámetro

(26) `while(string::npos != pos || string::npos != lastPos)` → $O(n)$ // n= nº iteraciones

El peor caso del switch es cuando un `str[pos] == '.'`, es el case en el que se llama a más funciones auxiliares, lo cual resulta en la siguiente complejidad y suponemos que es trata de un acrónimo, dando false en el resto de las llamadas para llegar a la última llamada.

(31) `switch(str.at(pos))` → $O(a+p)$

Justificación suponiendo que la cadena es: `s= "9.9.9.9.a:9"`; //Caso ejemplo, podría repetirse `9.9...a:9`

(34) `TokenizarNumero()` → $O(a)$

a= nº de delimitadores especiales a eliminar (ej.: para email habría que eliminar `.-_@` a= 4)

No se tiene en cuenta el coste lineal de `find_first_not_of()` es lineal

(36) `TokenizarAcronimo()` $O(p)$

p= nº de puntos en el acrónimo

Por tanto, nuestro Tokenizador obtiene al final una complejidad de $O(\text{str} + n^{a+p})$

En el mejor de los casos sería cuando la cadena introducida no contiene ningún delimitador, además de no tener la necesidad de pasar la cadena a minúsculas y sin acentos y con lo que solo realizaría una iteración resultando en una complejidad: $\Omega(1)$

Complejidad espacial

Respecto a la complejidad espacial, el peor de los casos vendría dado por la misma situación que hemos mencionado anteriormente en la complejidad temporal.

La función **Filtrar()** tendría una complejidad de $O(\text{mayus} + \text{acentos}) = O(a)$

mayus → Un string que nunca modifica su tamaño que contiene todo el alfabeto excepto Ñ

acentos → Un string que nunca modifica su tamaño que contiene todos los acentos y Ñ que se contempla en el enunciado de la práctica

Antes del switch se declaran dos string especiales y delimitadores a los cuales le asignaremos que tienen un tamaño **b**.

En el switch se llamarán a las funciones TokenizarNumero() y TokenizarAcronimo()

TokenizarNumero() tiene una complejidad de $O(\text{esp0} + \text{delimitadores} + \text{delim} + \text{num}) = O(n)$

esp0 → Tamaño del string que nunca se modifica que contiene “.”

delimitadores → Tamaño del string donde se almacena los delimitadores que ignoran los delimitadores especiales.

delim → Tamaño del string auxiliar que se usa en EliminaEspeciales

num → Tamaño del string donde se almacena el supuesto número

TokenizarAcronimo() tiene una complejidad de $O(\text{delimitadores} + \text{acron} + \text{punto}) = O(m)$

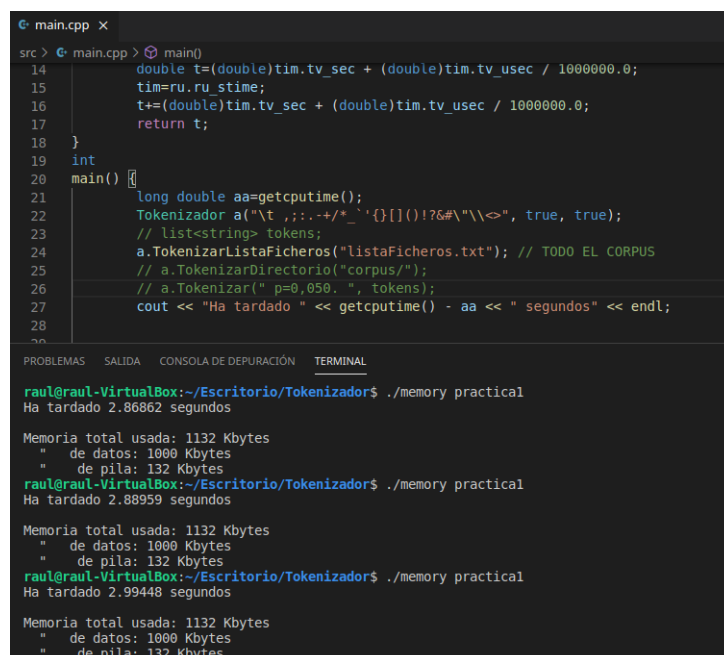
delimitadores → Tamaño del string donde se almacena los delimitadores que ignoran los delimitadores especiales.

acron → Tamaño del string donde se almacena el supuesto acrónimo

punto → `string::size_type` que se usa en la función auxiliar Acronimo()

Por tanto, al final el Tokenizador tiene una complejidad de **$O(a+b+n+m)$**

En el mejor de los casos, cuando todas las respectivas booleanas son false y el string a tokenizar no tenga delimitadores la complejidad será de **$\Omega(\text{lastPos} + \text{pos})$** (Dos `string::size_type`)



```
main.cpp x
src > main.cpp > main()
14 double t=(double)tim.tv_sec + (double)tim.tv_usec / 1000000.0;
15 tim=ru.ru_time;
16 t+=(double)tim.tv_sec + (double)tim.tv_usec / 1000000.0;
17 return t;
18 }
19 int
20 main() {
21     long double aa=getcputime();
22     Tokenizador a("\t ,;.:+/*_ '{}[]!@#%^\\";
23     // list<string> tokens;
24     a.TokenizarListaFicheros("listaFicheros.txt"); // TODO EL CORPUS
25     // a.TokenizarDirectorio("corpus/");
26     // a.Tokenizar("p=0,050. ", tokens);
27     cout << "Ha tardado " << getcputime() - aa << " segundos" << endl;
28 }
```

```
raul@raul-VirtualBox:~/Escritorio/Tokenizador$ ./memory practical
Ha tardado 2.86862 segundos

Memoria total usada: 1132 Kbytes
" de datos: 1000 Kbytes
" de pila: 132 Kbytes
raul@raul-VirtualBox:~/Escritorio/Tokenizador$ ./memory practical
Ha tardado 2.88959 segundos

Memoria total usada: 1132 Kbytes
" de datos: 1000 Kbytes
" de pila: 132 Kbytes
raul@raul-VirtualBox:~/Escritorio/Tokenizador$ ./memory practical
Ha tardado 2.99448 segundos

Memoria total usada: 1132 Kbytes
" de datos: 1000 Kbytes
" de pila: 132 Kbytes
```


Nuevo Tokenizador

```
1. void Tokenizador::Tokenizar (const string& s, list<string>& tokens) const {
2.     tokens.clear();
3.     string str= s;
4.     if(pasarAminuscSinAcentos)
5.         Filtrar(str);
6.     if(!casosEspeciales) {
7.         string::size_type lastPos = str.find_first_not_of(delimiters,0);
8.         string::size_type pos = str.find_first_of(delimiters,lastPos);//El primer delimitador
9.         while(string::npos != pos || string::npos != lastPos) {
10.             tokens.push_back(str.substr(lastPos, pos - lastPos));
11.             lastPos = str.find_first_not_of(delimiters, pos);
12.             pos = str.find_first_of(delimiters, lastPos);
13.         }
14.     }
15.     else {
16.         string especiales= "_.?&-=#@,";
17.         string delimitadores= this->delimiters
18.         if(delimiters.find(' ') == string::npos)//No está ' '
19.             delimitadores += ' ';
20.         if(delimiters.find('\n') == string::npos)//No está '\n'
21.             delimitadores += '\n';
22.
23.         string::size_type lastPos = str.find_first_not_of(delimitadores,0);
24.         string::size_type pos = str.find_first_of(delimitadores,lastPos);
25.         while(string::npos != pos || string::npos != lastPos)
26.         {
27.             bool especial= true
28.             bool numero= true;
29.             if(pos < str.size()) {
30.                 if(especiales.find(str.at(pos)) != string::npos &&
31.                    delimitadores[pos] != ' ' && delimitadores[pos+1] != '\n')
32.                     switch(str.at(pos)) {
33.                         case '.':
34.                             if(!TokenizarNumero(str, delimitadores,pos, lastPos, tokens)){
35.                                 if(!TokenizarAcronimo(str, delimitadores,pos, lastPos, tokens))
36.                                     especial= false;
37.                             }
38.                             else
39.                                 numero= false;
40.                             break;
41.                         case ',':
42.                             if(!TokenizarNumero(str, delimitadores, pos, lastPos, tokens))
43.                                 especial= false;
44.                             else
45.                                 numero= false;
46.                             break;
47.                         case '-':
48.                             if(!TokenizarMultipalabra(str, delimitadores, pos, lastPos, tokens))
49.                                 especial= false;
50.                             break;
51.                         case '@'://Si el primer delimitador NO es @ entonces nunca puede ser Email
52.                             if(!TokenizarEmail(str, delimitadores, pos, lastPos, tokens))
53.                                 especial= false;
54.                             break;
55.                         default:
56.                             if(!TokenizarURL(str, delimitadores,pos, lastPos, tokens))
57.                                 especial= false;
58.                             break;
59.                     }
60.             }
        }
    }
```

```

61.         else
62.             especial= false;//Token no es caso especial
63.         }
64.         else
65.             especial= false;
66.         if(numero && !especial)
67.             if(str[lastPos - 1] == '.' || str[lastPos - 1] == ',')
68.                 if(TokenizarNumero(str, delimitadores,pos, lastPos, tokens))
69.                     especial= true;
70.         if(!especial)
71.         {
72.             tokens.push_back(str.substr(lastPos, pos - lastPos));//Tokenización normal
73.             lastPos = str.find_first_not_of(delimitadores, pos);
74.             pos = str.find_first_of(delimitadores, lastPos);
75.         }
76.     }
77. }
78. }
79.

```

Se ha modificado la disposición de saltos de línea y {} para que el código quedara mejor en el pdf

Complejidad de funciones:

find_first_of: length() - pos

substr: length() of the returned object.

find_first_not_of(): length() - pos