Tokenizador

Introducción

En el procesamiento de texto, la segmentación de strings en palabras es una tarea fundamental y esencial para una amplia gama de aplicaciones. El tokenizador es un componente clave en el procesamiento del lenguaje natural y en el análisis de datos de texto. Sin embargo, la creación de un tokenizador eficiente y preciso puede ser un desafío. Por lo tanto, en este trabajo se propone la implementación de una clase Tokenizador en C++ para segmentar strings en palabras, con la opción de detectar una serie de casos especiales de palabras.

La hipótesis de partida es que, al implementar un algoritmo basado en reglas específicas, se puede lograr un tokenizador eficiente y preciso. Además, al agregar la capacidad de detectar casos especiales de palabras, como palabras compuestas y números, el tokenizador será capaz de manejar una amplia variedad de textos y aumentar su precisión en la segmentación de strings. Este trabajo se centrará en el diseño y la implementación de la clase Tokenizador, y se evaluará su rendimiento mediante pruebas y comparaciones con otros tokenizadores.

Partimos de la siguiente versión de un tokenizador, vista en clase:

```
void
Tokenizador::Tokenizar (const string& str, list<string>& tokens) {
   string::size_type lastPos = str.find_first_not_of(delimiters,0);
   string::size_type pos = str.find_first_of(delimiters,lastPos);

   while(string::npos != pos || string::npos != lastPos)
   {
       tokens.push_back(str.substr(lastPos, pos - lastPos));
       lastPos = str.find_first_not_of(delimiters, pos);
       pos = str.find_first_of(delimiters, lastPos);
   }
}
```

Esta función recorre una cadena detectando secuencias de caracteres delimitadas por caracteres que se hayan definido como delimitadores, las extrae y las añade a la lista de tokens.

Además, en el enunciado de la práctica se proponen varias funciones para tokenizar ficheros y realizar accesos a directorios.

En cuanto a los casos especiales se detectarán URL, números decimales, e-mail, acrónimos y palabras compuestas (separadas por guiones). Además se van a poder quitar acentos y mayúsculas.

Solución implementada

Estructuras de datos

Para guardar los delimitadores se utiliza un *bitset* de 256 bits. Los 256 bits se corresponden a los 256 valores de la tabla ASCII, de tal forma que si un carácter se ha introducido como delimitador, el bit de la posición correspondiente al valor del carácter ASCII estará a 1, si no a 0. La consulta es de complejidad constante, pero además mejor que en una tabla hash, ya que

se evita calcular el valor hash. La complejidad espacial es muy baja ya que solamente ocupará 256 bits.

Los *bitset* se utilizan tanto para guardar los delimitadores de palabras generales, como los delimitadores especiales para los casos que tienen más de un delimitador especial.

```
bitset<256> delimitersBitset; // Delimitadores de t?rminos en forma de bitset
bitset<256> URLdelimiters;
bitset<256> decimaldelimiters;
bitset<256> emaildelimiters;
```

Algoritmo

La función para tokenizar un string simplemente llama a una función que implementa una forma de tokenizar simple para cuando los casos especiales no están activados y a otra para cuando sí lo están. De la misma forma pasará el string antes por la función que lo pasa todo a minúsculas y quita los acentos si la variable de control está activada.

```
void Tokenizador::Tokenizar(const string &str, list<string> &tokens)
{
    // Limpiar la lista de tokens
    tokens.clear();
    if (casosEspeciales)
        if (pasarAminuscSinAcentos)
            TokenizarEspeciales(pasar_a_minusculas_sin_acentos(str), tokens);
        else
            TokenizarEspeciales(str, tokens);
    else
        if (pasarAminuscSinAcentos)
            TokenizarSimple(pasar_a_minusculas_sin_acentos(str), tokens);
        else
            TokenizarSimple(str, tokens);
}
```

La función de *TokenizarSimple* implementa una versión mejorada del tokenizador visto en clase. Esta función resetea el bitset, ya que este bitset se puede modificar en *TokenizarEspeciales*. El algoritmo recorre el string carácter a carácter y en cuanto se encuentre el primer carácter no delimitador se guarda en la variable token. Se van guardando los caracteres siguientes hasta encontrar un delimitador. Cuando se encuentra el delimitador, se guarda el token que se ha generado en la lista de tokens y se vacía la variable *token*.

```
void Tokenizador::TokenizarSimple(const string &str, list<string> &tokens)
{
    delimitersBitset.reset();
    for (unsigned char c : delimiters) // loop through each character in s
        delimitersBitset[c] = 1; // set the bit for c to 1
    std::string token; // temporary string to store token
    for (unsigned char c : str) { // loop through each character in str
        if (delimitersBitset[c]) { // if c is a delimiter
            if (!token.empty()) { // if token is not empty
                tokens.push_back(token); // add token to vector
                token.clear(); // clear token
            }
        } else { // if c is not a delimiter
            token += c; // append c to token
        }
    }
    if (!token.empty()) { // if token is not empty after loop
        tokens.push_back(token); // add last token to vector
    }
}
```

La función de *TokenizarEspeciales* activa los bits correspondientes al espacio y al salto de línea ya que estos siempre se consideran delimitadores en los casos especiales.

```
void Tokenizador::TokenizarEspeciales(const string &str, list<string> &tokens) {

    // A?adir el espacio y el salto de l?nea al set de delimitadores
    delimitersBitset[' '] = 1;
    delimitersBitset['\n'] = 1;

    ESTADO estado = INICIO;
    unsigned char c;
    size_t start;
    bool todo_digitos = true;
```

```
 // Booleano para saber si se ha confirmado un caso especial
 bool confirmado = false;

 // Booleano que controla si se encuentra "." o "," al principio de un token
 bool delim_at_start = false;
```

El algoritmo recorre cada carácter del string y comprueba si es delimitador. Si lo es pero el estado es *INICIO* (empieza en este estado) comprueba si se ha encontrado un delimitador de decimales, en cuyo caso se entra en el caso especial de los decimales que sigue comprobando y si es un decimal lo guardará, si no se seguirá recorriendo. Al encontrar el primer carácter no delimitador se marca dicho carácter como inicio de un token y el estado cambia a *TOKEN*. En este estado cada carácter recorrido que sea delimitador se irá comprobando si pertenece a uno de los casos especiales, en el orden proporcionado en el enunciado. Si es posible que pertenezca a cierto caso especial, entrará en la función del caso donde se realizarán comprobaciones adicionales y si es un caso especial se guarda el token y se sigue a partir del

final de ese token. Si no se comprueba el siguiente caso especial. Si no pertenece a ningún caso especial, se guarda como token normal.

```
// Recorrer la cadena a tokenizar de izquierda a derecha hasta encontrar un delimitador o un espacio
for (size_t i = 0; i < str.size(); i++) {
    c = str[i];
    // Si estamos dentro de un token comprobamos si es un delimitador especial
    if (delimitersBitset[c]) {
        if (estado == TOKEN) {
            if (URLdelimiters[c])
                confirmado = TokenizarURL(URLdelimiters, str, tokens, start, i);
            if (!confirmado && decimaldelimiters[c])
                confirmado = TokenizarDecimal(decimaldelimiters, str, tokens, start, i, delim_at_start);
            if (!confirmado && c == '@')
                confirmado = TokenizarEmail(emaildelimiters, str, tokens, start, i);
            if (!confirmado && c == '.')
                confirmado = TokenizarAcronimo(str, tokens, start, i);
            if (!confirmado && c == '-')
                confirmado = TokenizarGuion(str, tokens, start, i);
            if (!confirmado){
                tokens.push_back(str.substr(start, i-start));
                estado = INICIO;
                confirmado = false;
            }
            if (confirmado) {
                estado = INICIO;
                confirmado = false;
            }
        }
        // Para los casos en que el "." o "," aparece al principio del token
        else if (decimaldelimiters[c]) {
            delim_at_start = true;
            confirmado = TokenizarDecimal(decimaldelimiters, str, tokens, start, i, delim_at_start);
            if (confirmado) {
                estado = INICIO;
                confirmado = false;
            }
            delim_at_start = false;
        }
    }
    else {
        // Si estamos al inicio nos saltamos todos los delimitadores hasta encontrar un car?cter
        // Al encontrar el primer car?cter es donde empieza el token
        if (estado == INICIO) {
            start = i;
            estado = TOKEN;
        }
    }
}
```

Justificación de la solución

Como primera mejora de la versión inicial del tokenizador se han implementado contenedores asociativos para guardar los caracteres delimitadores. Para esto se ha usado el *unordered_set*.

```
delimiters = delimitadoresPalabra;
delimitersSet = unordered_set<char>(delimiters.begin(), delimiters.end());
```

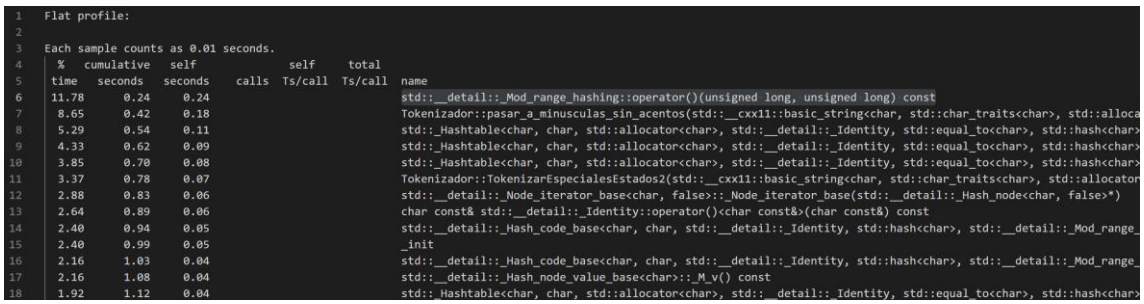
Como se ha visto en teoría, la complejidad temporal del tokenizador inicial es de $O(n * d)$ siendo *n* la cantidad de caracteres en el string a tokenizar y *d* la cantidad de caracteres en el string de delimitadores. Al usar un contenedor asociativo cuya búsqueda tiene una complejidad promedio constante, se puede sustituir el *find_first_of()* y el *find_first_not_of()* por una versión más eficiente.

```cpp
size_t Tokenizador::efficient_find_first_of(const string &str, const size_t &pos) const {
    for (size_t i = pos; i < str.size(); i++) {
        if (this->delimitersSet.count(str[i])>0) {
            return i;
        }
    }
    return string::npos;
}

size_t Tokenizador::efficient_find_first_not_of(const string &str, const size_t &pos) const {
    for (size_t i = pos; i < str.size(); i++) {
        if (this->delimitersSet.count(str[i])==0) {
            return i;
        }
    }
    return string::npos;
}
```

Con esta mejora la complejidad temporal promedio sería de $O(n)$.

Por otro lado, se ha añadido una complejidad espacial adicional. Además, se podría mejorar todavía más la complejidad temporal ya que el cálculo de los valores hash de los *unordered_set* consumen la mayor parte del tiempo de computación:



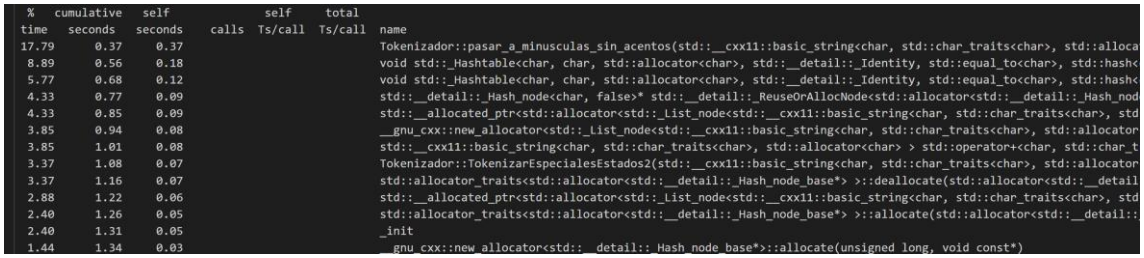
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 11.78      0.24     0.24                             std::__detail::_Mod_range_hashing::operator()(unsigned long, unsigned long) const
  8.65      0.42     0.18                             Tokenizador::pasar_a_minusculas_sin_acentos(std::__cxx11::basic_string<char, std::char_traits<char>, std::alloca
  5.29      0.54     0.11                             std::_Hashtable<char, char, std::allocator<char>, std::__detail::_Identity, std::equal_to<char>, std::hash<char>
  4.33      0.62     0.09                             std::_Hashtable<char, char, std::allocator<char>, std::__detail::_Identity, std::equal_to<char>, std::hash<char>
  3.85      0.70     0.08                             std::_Hashtable<char, char, std::allocator<char>, std::__detail::_Identity, std::equal_to<char>, std::hash<char>
  3.37      0.78     0.07                             Tokenizador::TokenizarEspecialesEstados2(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
  2.88      0.83     0.06                             std::__detail::_Node_iterator_base<char, false>::_Node_iterator_base(std::__detail::_Hash_node<char, false>*)
  2.64      0.89     0.06                             char const& std::__detail::_Identity::operator()<char const&>(char const&) const
  2.40      0.94     0.05                             std::__detail::_Hash_code_base<char, char, std::__detail::_Identity, std::hash<char>, std::__detail::_Mod_range_
  2.40      0.99     0.05                             _init
  2.16      1.03     0.04                             std::__detail::_Hash_code_base<char, char, std::__detail::_Identity, std::hash<char>, std::__detail::_Mod_range_
  2.16      1.08     0.04                             std::__detail::_Hash_node_value_base<char>::_M_v() const
  1.92      1.12     0.04                             std::_Hashtable<char, char, std::allocator<char>, std::__detail::_Identity, std::equal_to<char>, std::hash<char>
```

Para la función de pasar a minúsculas y quitar acentos, se planteó una comprobación basada en el valor decimal del carácter:

```cpp
string Tokenizador::pasar_a_minusculas_sin_acentos(const string &str) const {
    string result = "";
    for (unsigned char c : str) {
        int code = int(c);
        if (code >= 192 && code <= 198) {
            result += 'a';
        } else if (code == 199) {
            result += 'c';
        } else if (code >= 200 && code <= 203) {
            result += 'e';
        } else if (code >= 204 && code <= 207) {
            result += 'i';
        } else if (code == 208) {
            result += 'd';
        } else if (code == 209) {
            result += 'ñ';
        } else if (code >= 210 && code <= 214) {
            result += 'o';
```

Sin embargo esta solución implica muchas comprobaciones para cada carácter lo que aumenta el tiempo de ejecución considerablemente:



% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
17.79	0.37	0.37				Tokenizador::pasar_a_minusculas_sin_acentos(std::__cxx11::basic_string<char, std::char_traits<char>, std::alloca
8.89	0.56	0.18				void std::_Hashtable<char, char, std::allocator<char>, std::__detail::_Identity, std::equal_to<char>, std::hash<
5.77	0.68	0.12				void std::_Hashtable<char, char, std::allocator<char>, std::__detail::_Identity, std::equal_to<char>, std::hash<
4.33	0.77	0.09				std::__detail::_Hash_node<char, false>* std::__detail::_ReuseOrAllocNode<std::allocator<std::__detail::_Hash_nod
4.33	0.85	0.09				std::__allocated_ptr<std::allocator<std::_List_node<std::__cxx11::basic_string<char, std::char_traits<char>, std
3.85	0.94	0.08				__gnu_cxx::new_allocator<std::_List_node<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
3.85	1.01	0.08				std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > std::operator+<char, std::char_t
3.37	1.08	0.07				Tokenizador::TokenizarEspecialesEstados2(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
3.37	1.16	0.07				std::allocator_traits<std::allocator<std::__detail::_Hash_node_base*> >::deallocate(std::allocator<std::__detail
2.88	1.22	0.06				std::__allocated_ptr<std::allocator<std::_List_node<std::__cxx11::basic_string<char, std::char_traits<char>, std
2.40	1.26	0.05				std::allocator_traits<std::allocator<std::__detail::_Hash_node_base*> >::allocate(std::allocator<std::__detail::
2.40	1.31	0.05				_init
1.44	1.34	0.03				__gnu_cxx::new_allocator<std::__detail::_Hash_node_base*>::allocate(unsigned long, void const*)

Análisis de las mejoras

Al cambiar la estructura de datos de un *unordered_set* a un *bitset* junto con las operaciones correspondientes se ha bajado de un tiempo promedio de 2.6s a 1.0s. Esto es una mejora de 2,6x sobre el original.

Al implementar la función *pasar_a_minusculas_sin_acentos* usando las fórmulas explicadas en lugar de una serie de *if*'s el tiempo promedio de ejecución ha bajado de 1.0s a 0.84s, lo que implica una aceleración de 1,19x sobre el original.

Otras mejoras implementadas incluyen refactorizar el código para evitar repetir declaraciones y llamadas de forma innecesaria, con lo que se consiguió hasta 0.77s de promedio.

Análisis de la complejidad

Teórica

Práctica