Proyecto Final de Carrera

Análisis de algoritmos de búsqueda de un solo patrón

Sergio Talens-Oliag

Noviembre 1997

Proyecto dirigido por
Francisco Casacuberta Nolla

Dept. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Contenidos

Capítulo 1. Introducción

El desarrollo de un programa informático consta de varias fases o pasos; comienza con la especificación del problema a resolver, continúa con el diseño, implementación y prueba de una o varias soluciones y termina con la evaluación de la aplicación obtenida, sin olvidar la documentación de todo el proceso.

En la etapa de diseño es fundamental identificar correctamente los datos y operaciones que vamos a manejar, pero, aunque los encontremos, esto no es suficiente para que el resultado sea satisfactorio; si deseamos un resultado óptimo deberemos evaluar cuales son los tipos de datos y algoritmos más adecuados para la implementación.

Para elegir esas estructuras y algoritmos debemos conocer las distintas alternativas y disponer de alguna herramienta que nos permita comprenderlas y compararlas. Este proyecto es un intento de proporcionar esa herramienta para el estudio de los algoritmos y estructuras implicados en el tratamiento de cadenas de símbolos alfabéticos, mediante la implementación de una biblioteca de clases C++ que incorpora mecanismos para estudiar los algoritmos desde distintos puntos de vista.

La implementación presentada aquí incluye fundamentalmente los tipos de datos y algoritmos relacionados con la búsqueda de una subcadena en otra, dejando para posibles ampliaciones la implementación de la búsqueda simultánea de varias subcadenas, la búsqueda aproximada o con errores y la búsqueda con expresiones regulares.

1.1. Origen del proyecto

En principio este proyecto iba a ser un estudio de los tipos de datos y algoritmos más adecuados para la gestión de un diccionario de sinónimos en castellano. El objetivo final era desarrollar un programa que permitiera generar y utilizar el diccionario en distintos sistemas.

Para un diccionario los datos serán registros con dos campos: una palabra y su definición. En nuestro caso una definición podría ser simplemente una lista de palabras, pero se iba a incluir un ejemplo o descripción con cada sinónimo que nos indicara en que contexto es adecuado utilizarlo. De cualquier forma, tanto las palabras como las definiciones serán cadenas de texto (aunque se representen codificadas).

Las operaciones a realizar dependerán del propósito de la aplicación: para consultar el diccionario sólo se precisará acceso rápido a las entradas utilizando las palabras como índices, mientras que para su edición necesitaremos mecanismos para la creación, modificación y almacenamiento de los registros.

En nuestro planteamiento la operación más importante es la de recuperación de las entradas, ya que es la que más va a emplear el usuario final. Por esta razón, el estudio debe comenzar por encontrar las estructuras y algoritmos más eficientes para la recuperación de los datos. Nos interesará una estructura que almacene las entradas de forma compacta y permita la recuperación de las definiciones a partir de las palabras.

En lugar de estudiar la gestión de bases de datos en general se comenzó por la bibliografía relacionada con el proceso de cadenas, ya que toda la información contenida en el diccionario es texto. Viéndolo así, se pensó que la edición podría hacerse escribiendo las entradas como texto estructurado, lo que reduciría nuestro programa de edición a una utilidad para transformar ese texto en la estructura más adecuada para el almacenamiento y recuperación de la información.

De cualquier forma, se abandonó el proyecto del diccionario y se replanteó el trabajo pensando en el diseño de una biblioteca para la gestión de información alfabética en general.

Este nuevo planteamiento contemplaba dos tipos de texto en función de la frecuencia con la que cambia: dinámico y estático [Bae92a]. La diferencia fundamental entre ambos es que las estructuras y algoritmos más eficientes para almacenar el texto o realizar operaciones de consulta necesitan preprocesarlo, algo que normalmente sólo vale la pena hacer cuando el texto cambia con poca frecuencia o el número de operaciones de consulta es muy elevado entre actualizaciones (como sucede hoy día con los buscadores de Internet).

Esta biblioteca se estructuraba en torno a las operaciones de edición (inserción, borrado, sustitución), búsqueda (de subcadenas y expresiones regulares), ordenación (teniendo en cuenta que el orden binario no es siempre el más adecuado), filtrado (recodificación, compresión y en general cualquier tipo de preproceso) y gestión de entrada/salida (mecanismos de *buffering* y almacenamiento de estructuras) de las cadenas de texto, definiendo los tipos de datos necesarios para cada una de ellas [Aho83]. En estos apartados distinguíamos entre texto dinámico y estático cuando era apropiado.

Como se ve, la idea era excesivamente ambiciosa, por lo que al final se decidió centrar el tema en el texto dinámico, comenzando por los algoritmos de búsqueda. Dada la gran cantidad de algoritmos existentes y lo difícil que era comprender el funcionamiento de algunos de ellos, sólo se implementaron los algoritmos de búsqueda de un solo patrón, diseñando herramientas para su análisis.

Este proyecto es el resultado de ese proceso.

1.2. Planteamiento general

Ya hemos dicho que la biblioteca proporciona un entorno para estudiar las estructuras y algoritmos adecuados para el *tratamiento de cadenas de símbolos alfabéticos*, pero, ¿a que nos referimos exactamente?. Comenzaremos por definir que entendemos por símbolo, cadena y alfabeto:

Símbolo
Elemento de un tipo de datos que tiene definida una relación de orden.

Alfabeto
Conjunto finito y no vacío de símbolos.

Cadena de símbolos
Secuencia finita de símbolos.

Con las cadenas realizaremos operaciones de edición (inserción, borrado y sustitución de símbolos o subcadenas), comparación (de igualdad o de orden) y búsqueda de unas en otras.

La idea es que los algoritmos y estructuras que se emplean en el tratamiento de cadenas no están restringidos a caracteres sino que, a efectos prácticos, son aplicables a cualquier tipo de datos que queramos tratar como secuencias.

Según la aplicación podríamos usar como símbolos las figuras geométricas elementales, las moléculas, las palabras reservadas de un lenguaje de programación o cualquier otro tipo de datos.

Las operaciones de edición (inserción, borrado y sustitución) son independientes del tipo de datos, ya que emplean únicamente las posiciones de los símbolos en la secuencia. En el caso de la búsqueda la única operación necesaria es la comparación, por lo que sólo necesitamos definir una relación de igualdad entre símbolos, de manera que dos cadenas serán iguales si todos sus símbolos lo son. Por último, para ordenar las cadenas necesitamos una relación de orden que nos diga si un símbolo es menor que otro, para poder compararlas símbolo a símbolo y determinar cual de es la menor cadena.

El alfabeto es el conjunto de símbolos que pueden aparecer en una cadena y en muchos casos no será necesario definirlo, ya que el propio tipo de datos define cuales son los valores aceptables. De todos modos, algunos de los algoritmos presentados dependen de él para construir tablas indexadas por símbolo y necesitan conocer el rango de valores posibles para una implementación eficiente[1].

Además podemos usar el alfabeto para redefinir el orden de los símbolos o crear clases de equivalencia, usando en las comparaciones el índice de los símbolos en el alfabeto (suponiendo que defina un orden interno) o asociando un entero a cada uno de ellos.

Lo que haremos en la biblioteca es definir estructuras para representar alfabetos y cadenas e implementar algoritmos que actúen sobre ellas. Las estructuras estarán parametrizadas por un tipo *símbolo* que debe definir los operadores *menor* e *igual* entre dos elementos del tipo.

Como la implementación se ha hecho en C++ y la biblioteca estándar de este lenguaje define un tipo `string` paramétrico, hemos definido los algoritmos en torno a él, usando la versión para caracteres de 8 bits. La ventaja de hacerlo así es que no necesitamos definir las operaciones de edición para las cadenas, ya que están incluidas en la clase `string`. También se ha definido un tipo alfabeto genérico (para cualquier tipo de datos), con una especialización para caracteres que tiene coste 1 para las operaciones de pertenencia y obtención del orden en el alfabeto. La versión genérica emplea el tipo conjunto de la STL [Ste95] para asociar índices a los símbolos, lo que la hace poco interesante, ya que los costes de las operaciones antes mencionadas es de orden logarítmico respecto al tamaño del alfabeto (aunque el usuario siempre puede definir una nueva versión más eficiente para su propio tipo de datos).

Para generar una biblioteca que emplease otro tipo de símbolos bastaría redefinir el tipo `astring` (equivalente a la cadena de caracteres estándar), reemplazando el tipo carácter por otro. Esto plantea el problema de no poder emplear la biblioteca usando dos tipos de símbolos en un mismo programa, ya que se usan los mismos nombres para los tipos; la solución sería modificar todas las clases que utilizan símbolos para que sean paramétricas o emplear espacios de nombres, definiendo cada versión de la biblioteca en un ámbito diferente.

Además, la biblioteca incluye herramientas para estudiar gráficamente la evolución de los algoritmos y medir sus costes temporales, tanto reales (tiempo de ejecución) como teóricos (cuenta de pasos). Las herramientas y su implementación serán comentadas en próximos capítulos.

1.3. Uso del C++

La biblioteca debía ser eficiente y transportable (no sólo entre compiladores, también entre sistemas operativos) y no deseaba usar un lenguaje interpretado (como PERL o JAVA) ni aprender uno nuevo, así que debía utilizar C o C++.

El lenguaje C era la apuesta más segura; es eficiente, transportable y ampliamente utilizado (además de existir buenos compiladores disponibles gratuitamente), pero carece de muchas de las características que buscaba (como las *templates* o la sobrecarga de operadores), y no es orientado a objetos (parte de mi diseño lo era, de hecho en forma de clases C++), por lo que elegí el C++.

Una vez decidido el nombre del lenguaje de programación me encontré con una decisión más difícil: ¿Qué C++ debía usar? Los compiladores actuales incorporan las caraterísticas del lenguaje definidas en [Str91], pero el C++ está aún evolucionando y el estándar[2], es aún una utopía para los usuarios del software gratuito de GNU.

Antes de codificar tenía que estudiar las características que necesitaba y las herramientas disponibles. En realidad, el problema no es sólo la disponibilidad de algunas características del lenguaje (como los *namespaces*), sino la falta de bibliotecas estándar completas, la necesidad de reinventar la rueda cuando, en un futuro cercano estarán disponibles herramientas estándar eficientes.

---

[1] Si empleamos vectores para las tablas es interesante que los símbolos puedan convertirse en enteros con coste de orden uno, ya que un método que tenga un orden mayor puede anular la eficacia del algoritmo que las emplee.

[2] En fase de revisión, con un borrador que tiene más de 700 páginas [C++96].

La solución adoptada es no emplear las nuevas características del lenguaje que no se han podido probar y utilizar las librerías disponibles que proporcionan características que estarán incluidas en el estándar (cómo la *Standard Template Library*, que se ha incorporado con pequeñas modificaciones) aun a coste de perder en eficiencia; cuando el estándar esté aceptado es muy probable que aparezcan implementaciones gratuitas y comerciales más eficientes y, se mejore la calidad de los compiladores.

1.4. Modelo de programación

Una de las características más peculiares del diseño de la biblioteca es la utilización del concepto de clase no sólo para encapsular los tipos de datos, sino también para representar los «tipos de algoritmos».

Ya hemos mencionado que la biblioteca implementa fundamentalmente algoritmos de búsqueda, pero, dado que todos los algoritmos de búsqueda implementados toman el mismo tipo y número de parámetros y retornan lo mismo, ¿por qué no definir una interfaz común para todos ellos?

La idea es definir una clase base abstracta (que es aquella que no puede ser instanciada, es decir, que no se pueden declarar objetos del tipo que define) que incluya todos los elementos comunes de los algoritmos, tanto la interfaz externa como las operaciones que debe definir cada uno. La separación entre interfaz y operaciones internas nos permite dividir el algoritmo en varias etapas o fases sin hacerlas visibles desde el exterior; siempre se llamará al algoritmo desde los métodos de la clase abstracta, ocultando los detalles de la implementación.

Ninguno de los métodos internos recibe parámetros; todas las variables que deban ser pasadas desde el exterior se declaran como atributos privados de la clase abstracta y las que deban intercambiar entre sí, como atributos de la clase que los implemente. De ese modo, antes de invocar al método o métodos internos podemos inicializar y validar las variables locales asociadas a los parámetros de entrada, con lo que simplificamos el tratamiento de errores y evitamos la repetición de operaciones.

Para el caso de los algoritmos de búsqueda definimos como interfaz un método que toma como parámetros una cadena a buscar y un texto. El retorno de la función dependerá de para qué queramos usarla; puede ser simplemente un valor booleano que indique si la cadena ha sido encontrada o no, un índice que nos de la posición de la primera ocurrencia de la cadena o un vector que nos de las posiciones de todas las ocurrencias. En los dos últimos casos, si el patrón no es encontrado, debemos indicarlo de alguna manera, por ejemplo retornando un índice fuera de rango en el primer caso y un vector vacío en el segundo.

Las operaciones internas serán el *preproceso del patrón* (que puede ser nulo, como en el caso de la búsqueda por el método de la fuerza bruta) y la *búsqueda del patrón preprocesado*. Como cada algoritmo emplea distintas estructuras de preproceso la clase que lo defina debe incluirlas como miembros privados de la clase, para que estén disponibles cuando sea necesario.

Los algoritmos trabajan con iteradores (punteros) al patrón y al texto, la clase base incluye atributos iterador que se inicializan con los punteros al principio y final del texto y el patrón y atributos enteros que contienen el tamaño de cada uno.

Como cada implementación define su propio método de preproceso y este sólo depende del patrón, se incluyen métodos para invocar al algoritmo asumiendo que ya lo hemos preprocesado, permitiéndonos reutilizar los valores obtenidos sin volver a ejecutar la función. Si se invoca el método de búsqueda que asume que el patrón ha sido preprocesado y no se ha hecho, el algoritmo termina devolviendo que el patrón no ha sido encontrado sin ejecutar la búsqueda.

Hemos comentado que existen distintas posibilidades para el retorno de las funciones según lo que queramos buscar y para que vayamos a utilizar el resultado. Para permitir la máxima flexibilidad se han definido dos métodos de acceso a la función de búsqueda, uno que retorna la posición en el texto de la primera ocurrencia del patrón (o la longitud del texto si el patrón no es encontrado) y otra que no retorna nada pero que toma un parámetro adicional, una *función de acierto* que se llama desde los algoritmos cada vez que se encuentra el patrón.

La función de acierto se emplea para indicar qué debemos hacer cuando encontramos el patrón y determinar si el algoritmo debe seguir o no después de hacerlo. Dado que es una clase externa se pueden derivar distintas versiones y almacenar en ella los resultados de la búsqueda. La biblioteca proporciona tres funciones de este tipo, una que encuentra todas las ocurrencias del patrón y las guarda en un vector y otras dos que encuentran la primera o todas las ocurrencias de una unidad con significado (*token*). Los métodos de búsqueda están preparados para emplear una función que modifique el texto; para definir algoritmos de búsqueda y reemplazo o de búsqueda y sustitución bastaría utilizar una función de acierto adecuada.

1.5. El entorno de trabajo

El proyecto ha sido desarrollado en su mayor parte en un sistema Debian GNU Linux ([Debian]) pero en realidad es bastante transportable, siempre que dispongamos de un compilador y unas bibliotecas mínimamente actualizadas[1].

De hecho, la biblioteca y el programa de ejemplo han sido compilados para Apple Macintosh utilizando el compilador de C++ de Metrowerks (concretamente la versión del CodeWarrior Academic número 10). Lo único que causó algún problema fue el programa de ejemplo, ya que emplea funciones de la biblioteca de C de GNU para UNIX ([GNU]), concretamente las funciones de proceso de parámetros de entrada (`getopt`). De todos modos, incluso estas se compilaron (como biblioteca de C alterando algunas macros, ya que haciéndolo en C++ daba errores), y enlazaron con el programa sin mayor problema.

En Linux se ha utilizado la versión 2.7.2.3 del compilador `gcc` y la versión 2.7.2.8 de la biblioteca de C++ (`libg++`). Para facilitar la compilación en distintas versiones de UNIX se han empleado los programas `autoconf`, `automake` y `libtool`, también de GNU.

El primero se emplea para detectar la disponibilidad de programas, bibliotecas, cabeceras, definiciones de tipos y funciones o determinar las características de los compiladores. En función de los resultados se proporcionan versiones alternativas del código o se eliminan parte de las capacidades del programa final. La biblioteca solo emplea la utilidad para detectar el compilador a utilizar, si realmente se desease portabilidad para múltiples sistemas seria necesario realizar pruebas adicionales. Aun así, cualquier sistema UNIX que utilice las últimas versiones del compilador y la biblioteca de C++ de GNU debería funcionar correctamente.

El segundo, `automake`, simplifica la generación de los archivos `Makefile` utilizados como entrada para el programa `make`, una herramienta disponible en prácticamente todos los sistemas UNIX que sirve para determinar que partes del código deben ser recompiladas después de una modificación y ejecutar los comandos necesarios para hacerlo y, en general, para generar ficheros a partir de otros mediante la ejecución de comandos, teniendo en cuenta las dependencias entre ficheros y las modificaciones de estos (sólo genera ficheros si los ficheros fuente son más nuevos que los generados).

Por último, el programa `libtool` ayuda en la generación de bibliotecas compartidas (de enlace dinámico) para distintos sistemas. Gracias a esta utilidad la biblioteca se genera en dos versiones: una con enlace dinámico y otra con enlace estático.

La primera versión de la memoria del proyecto fue escrita en formato SGML [SGML] utilizando las `sgml-tools` y el Linuxdoc DTD [sgmltools]. El paquete proporciona utilidades para convertir los archivos SGML a los formatos groff, LaTeX, HTML, GNU info, LyX y RTF. Para la edición del texto se empleó el editor Emacs en el modo PSGML.

La verdad es que las herramientas todavía necesitan evolucionar bastante, ya que están pensadas únicamente para generar documentos *HOWTO* de Linux y tienen algunos fallos, pero modificando

[1]Es suficiente con que incluyan las clases de la *Standard Template Library* ([Ste95]) y soporten el tipo `string` definido en el borrador del C++ estándar de abril de 1995.

ligeramente los ficheros de configuración se pueden obtener buenos resultados, con la ventaja de poder generar la documentación en varios formatos sin esfuerzo.

De cualquier modo, eran necesarias demasiadas modificaciones para obtener los resultados deseados, por lo que se realizó una conversión al formato del sistema `Lout` ([Kin96a], [Kin96b], [lout]), un sistema de composición de documentos similar al LaTeX [Lam86], pero más sencillo de utilizar. La memoria se completo en este sistema.

Capítulo 2. Algoritmos de búsqueda simple

Como la función principal de la biblioteca es estudiar los algoritmos de búsqueda de un sólo patrón, describiremos en este capítulo el problema de la búsqueda y la implementación de los algoritmos, dejando para el siguiente los detalles sobre las clases que integran la biblioteca.

2.1. Definición del problema

Podemos formular el problema de buscar una cadena dentro de otra como sigue:

> Dada una subcadena x, con $|x| = m$, y una cadena y, con $|y| = n$, donde $m > 0$, $n > 0$ y $m \geq n$, si x aparece como subcadena de y entonces determinar la posición en y de la primera ocurrencia de x, es decir, calcular el mínimo valor de i tal que $y_{i \ldots i+m-1} = x_{1 \ldots m}$.

El problema también se suele extender a buscar todas las apariciones de x en y.

Los algoritmos que vamos a estudiar son:

- Fuerza bruta
- Karp-Rabin [KR87]
- Knuth-Morris-Pratt [KMP77]
- Shift Or [Bae92b]
- Boyer-Moore [BM77]
- Boyer-Moore-Horspool [Hor80]
- Sunday Quick Search [Sun90]

Dividiremos el estudio de cada algoritmo en tres apartados: descripción del algoritmo, análisis de costes e implementación.

En la descripción de los algoritmos, denominaremos **patrón** a la subcadena buscada y **cadena** o **texto** a la cadena objeto de la búsqueda y emplearemos las letras m y n para referirnos a la longitud del patrón y la cadena respectivamente, asumiendo siempre que $n > m$.

En los ejemplos utilizaremos la salida del programa de prueba, donde el patrón aparece sobre el texto mostrando su posición relativa y se indica qué símbolos han sido comparados con éxito y cual es el símbolo actual colocando los caracteres «`*`» (comparados) y «`-`» (posición actual) bajo el patrón y el texto. Por ejemplo, si buscamos el patrón «`de`» en la cadena «`Buscando en un texto de prueba`» la salida:

```
      de
Buscando en un texto de prueba
      *-
```

indica que hemos encontrado la letra `d` y vamos a comparar la `o` del texto con la `e` del patrón.

Cada vez que se produce un acierto parcial (`-- PARTIAL MATCH --`) o un acierto total (`-- FULL MATCH --`) se indica mostrando un mensaje. Cuando el algoritmo utiliza estructuras de preproceso para saltar también se muestran sus valores intercalados entre los distintos pasos de la búsqueda, aunque únicamente cuando son utilizados. El significado de esos parámetros se comenta al describir los algoritmos.

El análisis de costes se divide en temporal y espacial. Para cada algoritmo comentaremos los costes temporales para los casos mejor, peor y promedio de la búsqueda de todas las ocurrencias del patrón en el texto, aunque sin dar una demostración formal, ya que esta se puede encontrar en la bibliografía relacionada con cada algoritmo. Del coste espacial nos limitaremos a indicar lo que ocupan las estructuras de preproceso de cada algoritmo.

En el apartado de implementación utilizamos una versión simplificada de los algoritmos escrita en C. Lo que se pretende con esto es mostrar claramente el funcionamiento del algoritmo sin incluir detalles que pueden dificultar la comprensión. De entrada, las versiones en C acceden al texto y al patrón a través de índices, mientras que la implementación real usa punteros. En la versión real los algoritmos retornan valores (tamaño del texto o posición del primer acierto) y cuando se encuentra un acierto se realizan verificaciones para saber si vamos a continuar o no. De cualquier modo, la versión completa en C++ se puede ver directamente en el código de la biblioteca.

La versión simplificada asume que tenemos definidas las variables `txt` y `pat` para acceder al texto y al patrón y los valores `txt_size` y `pat_size` para acceder a los tamaños de ambas estructuras (como realmente sucede en la implementación en C++). Además, se supone que el tamaño del texto es mayor o igual al del patrón y que ambos son distintos de 0 (la implementación no ejecuta el algoritmo si esto no se cumple).

Por último señalar que la implementación de los algoritmos no se corresponde con el planteamiento original de los autores, ya que los algoritmos que emplean tablas de preproceso se han modificado siguiendo un modelo similar al de la fuerza bruta, en el que el bucle principal compara el texto con el primer elemento del patrón (el último en el caso de los algoritmos de Boyer-Moore, Boyer-Moore-Horspool y Sunday QuickSearch) y actualiza el iterador del texto si no hay coincidencia (dependiendo del tipo de algoritmo se utilizan las tablas de preproceso, pero los valores que dependen del patrón son fijos y por tanto se almacenan en variables locales para evitar accesos a vector). Cuando encontramos un acierto parcial se entra en otro bucle que se desplaza en el patrón y usa la información del preproceso.

Con esta implementación, los valores que no se utilizan no son actualizados, evitando operaciones innecesarias sin introducir otros retardos. Esto se puede hacer porque todos los algoritmos tienen un comportamiento especial a partir del momento en el que es encontrado un acierto parcial, pero antes de haberlo hecho no necesitan utilizar ni actualizar muchas de las variables locales.

2.2. Fuerza bruta

2.2.1. Descripción

Este algoritmo es la forma más simple de aproximarse al problema de búsqueda de subcadenas. La idea es ir deslizando el patrón sobre el texto de izquierda a derecha, comparándolo con las subcadenas del mismo tamaño que empiezan en cada carácter del texto.

El funcionamiento es como sigue: vamos comparando el primer carácter del patrón con cada uno de los caracteres de la cadena, cuando se encuentra un acierto se compara el segundo carácter del patrón con el carácter del texto alineado con él (el que sigue al que causó el acierto), si coinciden seguimos con el tercero, cuarto, etc. hasta que se encuentra un fallo o se termina el patrón. Si alcanzamos el final del patrón hemos encontrado la subcadena, si nos detenemos antes volvemos a comparar el patrón con la subcadena que comienza en el carácter siguiente al primer acierto, es decir, deslizamos el patrón una posición a la derecha.

Veamos un ejemplo de la ejecución del algoritmo, generado con el programa de prueba de la biblioteca:

```
Text         : Este es un texto de prueba.
Text size    : 28
Pattern      : texto
Pattern size : 5

-*- Brute Force -*-

SEARCHING ...

texto
Este es un texto de prueba.
-

 texto
Este es un texto de prueba.
 -

  texto
Este es un texto de prueba.
  -

-- PARTIAL MATCH --

  texto
Este es un texto de prueba.
  *-

  texto
Este es un texto de prueba.
  **-

   texto
Este es un texto de prueba.
   -

    texto
Este es un texto de prueba.
    -

     texto
Este es un texto de prueba.
     -

      texto
Este es un texto de prueba.
      -

       texto
Este es un texto de prueba.
       -

        texto
Este es un texto de prueba.
        -

         texto
Este es un texto de prueba.
         -

          texto
Este es un texto de prueba.
          -
```

```
           texto
Este es un texto de prueba.
           _

-- PARTIAL MATCH --

           texto
Este es un texto de prueba.
           *_

           texto
Este es un texto de prueba.
           **_

           texto
Este es un texto de prueba.
           ***_

           texto
Este es un texto de prueba.
           ****_

           texto
Este es un texto de prueba.
           *****

-- FULL MATCH --

            texto
Este es un texto de prueba.
            _

             texto
Este es un texto de prueba.
             _

              texto
Este es un texto de prueba.
              _

-- PARTIAL MATCH --

              texto
Este es un texto de prueba.
              *_

               texto
Este es un texto de prueba.
               _

                texto
Este es un texto de prueba.
                _

                 texto
Este es un texto de prueba.
                 _

                  texto
Este es un texto de prueba.
                  _

                   texto
Este es un texto de prueba.
                   _
```

```
                    texto
Este es un texto de prueba.
                    -

                     texto
Este es un texto de prueba.
                     -

                      texto
Este es un texto de prueba.
                      -

-*- Brute Force -*-

Number of matches:           1
Match positions:            11
```

Este algoritmo necesita poder avanzar y retroceder en el texto, de manera que para textos almacenados en disco y no en memoria es necesario el empleo de *buffers*.

Por último indicar que, dadas las características de este algoritmo, el reinicio de la búsqueda consiste en continuar desde el carácter siguiente al que inició el acierto, como se puede observar en el ejemplo.

2.2.2. Costes

En el mejor caso, el primer elemento del patrón no está en el texto, de manera que ningún carácter coincide con él. Tenemos un coste temporal de orden Ω (n), el algoritmo es lineal.

En el peor caso el coste temporal de este algoritmo es de Ω (mn), que seria aquel en el que encontramos el patrón en todas las subcadenas del texto.

En promedio el coste temporal es menor que Ω (mn), ya que no precisamos comparar cada vez los m caracteres, sólo comparamos hasta que se detecta un fallo y las probabilidades de falsos comienzos son muy inferiores a 1 en general. Si buscamos en textos normales será de orden Ω (m+n) en la mayoría de los casos [Aho90].

El coste espacial es nulo, salvo que consideremos parte del algoritmo los *buffers* empleados para almacenar el patrón y la subcadena del texto con la que este está alineado, en cuyo caso será de Ω (m).

2.2.3. Implementación

Podemos expresar el algoritmo en C de la siguiente manera:

```
brute_force() {
  int ti, pi, tj;                            /* variables auxiliares */
  ti = 0;                                      /* índice en el texto */
  while (ti < txt_size - pat_size){
    if (txt[ti] = pat[0]) {                       /* Acierto parcial */
      pi = 0;                    /* índice acierto parcial en patrón */
      tj = ti;                    /* índice acierto parcial en texto */
      do {
        tj++; pi++;
        if (pi == pat_size){
        acierto_en (ti);
          break;                             /* continuamos en ti + 1 */
        }                   /* (ti se incrementa al salir del bucle) */
      } while (txt[tj] == pat[pi]);
    }                                         /* Fin Acierto Parcial */
    ti++;
  }                                                /* Fin bucle while */
```

```
}
```

Nótese que la condición de salida del bucle principal detiene la búsqueda si el texto que queda es menor que el tamaño del patrón, es decir, cuando ya no es posible encontrar un acierto, y que el empleo de una variable auxiliar nos evita el incremento y decremento del índice en el texto cuando se produce un acierto parcial; `ti` siempre nos dará la posición relativa del patrón respecto al texto.

Para mejorar el rendimiento se pueden definir variables para eliminar operaciones aritméticas (`txt_size - pat_size` es un valor constante durante la ejecución del algoritmo, pero el compilador no tiene porque detectarlo, si usamos una variable evitamos una resta en cada iteración) y accesos a vector (el valor del primer carácter del patrón es consultado en cada iteración del bucle principal, guardándolo en un registro eliminamos accesos a vector innecesarios).

La versión real aplica las modificaciones mencionadas y utiliza punteros para los tres índices definidos, ya que todos ellos son de acceso secuencial al texto y al patrón.

2.3. Karp-Rabin

2.3.1. Descripción

El algoritmo fue enunciado por Karp y Rabin en [KR87]. Se trata de un algoritmo probabilístico que adapta técnicas de dispersión (*hashing*) a la búsqueda de patrones. Se basa en tratar cada uno de los grupos de m caracteres del texto como un índice en una tabla de dispersión, de manera que si la función de dispersión de los m caracteres del texto coincide con la del patrón probablemente hemos encontrado un acierto (hay que comparar el texto con el patrón, ya que la función de dispersión elegida puede presentar colisiones).

La función de dispersión tiene la forma $d(k) = k \bmod q$ ($d(k)$ es igual al resto de la división k/q), con q un número primo grande que será el tamaño de la tabla de dispersión.

Para transformar cada subcadena de m caracteres en un entero lo que hacemos es representar los caracteres en una base B que en el planteamiento original coincide con el tamaño del alfabeto. Por ejemplo, el entero x_i correspondiente a la subcadena $txt_{i\ldots i+m-1}$ sería:

$$x_i = txt_i * B^{m-1} + txt_{i-1} * B^{m-2} + \ldots + txt_{i+m-1}$$

Para calcular el valor de la siguente subcadena ($txt_{i+1\ldots i+m}$) haríamos:

$$x_{i+1} = txt_{i+1} * B^{m-1} + txt_{i+2} * B^{m-2} + \ldots + txt_{i+m}$$

Pero para simplificar el cálculo podemos expresarlo en función de x_i:

$$x_{i+1} = x_i * B - txt_i * B^m + txt_{i+m}$$

Es decir, si la cadena es un número en base B, el nuevo valor será el resultado de multiplicar por la base el valor anterior eliminando el dígito de mayor peso (que ya no está en la cadena) y añadiendo como componente de menor peso el valor del nuevo símbolo.

Por ejemplo, si $B = 10$ y $m = 4$ (queremos obtener valores de 4 dígitos), dado el texto $txt = 256789$ calcularíamos el primer valor haciendo:

$$x_0 = 2 * 10^3 + 5 * 10^2 + 6 * 10 + 7 = 2567$$

Y obtendríamos x_1 haciendo:

$$x_1 = 2567 * 10 - 2 * 10^4 + 8 = 5678$$

El problema de este planteamiento es que, dependiendo de la longitud del patrón, el número entero equivalente a la cadena supera el rango de enteros representable por computador. Si la base (es decir, el tamaño del alfabeto) es pequeña, siempre que B^m sea un número dentro del rango de enteros representables, la representación de la cadena como un número en base B será una función de dispersión perfecta. Para enteros de 32 bits y alfabetos de tamaño 256 la función de dispersión es perfecta para patrones de tamaño menor o igual a cuatro.

Como la representación en base B causa problemas de rango lo que se hace es utilizar la función módulo (resto de la división). La ventaja fundamental es que la operación módulo es asociativa, por lo que podemos aplicarla después de cada operación de actualización.

Si empleamos como divisor un número primo tenemos la garantía de que el resto de la división es siempre el dividendo para todos los números menores que el divisor. Cómo todas las operaciones de actualización emplean sólo un símbolo del alfabeto, escogiendo el mayor número primo que multiplicado por el tamaño del alfabeto (en realidad el mayor índice de los símbolos más uno) nos de un valor representable en el computador tenemos un divisor que no causará nunca desbordamiento, es decir, no se saldrá del rango.

Así, la formula anterior para actualizar el valor de la cadena en base B se transforma en la función de dispersión siguiente:

$$d(txt_{i+1}) = (d(txt_i) - txt_i * B^m + txt_{i+m}) \mathrm{mod} Q$$

Que dividimos en dos operaciones para evitar desbordamientos:

$$d_1(txt_{i+1}) = (d(txt_i) + B * Q - txt_i * B^m) \mathrm{mod} Q$$
$$d(txt_{i+1}) = (d_1(txt_i) + txt_{i+m}) \mathrm{mod} Q$$

Donde el factor $B * Q$ del cálculo de $d_1(txt_{i+1})$ se emplea para evitar un valor negativo.

Al final, el algoritmo funciona de la siguiente manera:

En primer lugar se realiza el computo de la función de dispersión del patrón y los primeros m caracteres del texto. Mientras $d(pat)$ sea distinto de $d(txt)$ se calcula la clave siguiente haciendo uso de la clave actual, empleando la fórmula anterior. En caso de que las funciones de dispersión coincidan se realiza la comparación carácter a carácter para verificar que no ha sido una colisión.

Como se ve, este algoritmo reduce la búsqueda de caracteres a comparaciones de enteros, pero, aunque las operaciones matemáticas sean poco costosas, superan en mucho el coste de las comparaciones de símbolos del resto de algoritmos.

Para reducir algo el coste de las operaciones podemos utilizar una base que sea potencia de dos (la menor que sea mayor o igual que el tamaño del alfabeto), de manera que los productos de enteros pueden ser reemplazados por desplazamientos en registro, ya que $x * 2^p \equiv x << p$, donde $<<$ es un desplazamiento a la izquierda de tantos bits como indique el operando derecho.

La ejecución del ejemplo del apartado anterior para este algoritmo sería:

```
Text         : Este es un texto de prueba.
Text size    : 28
Pattern      : texto
Pattern size : 5

-*- Karp Rabin -*-

PREPROCESSING ...
```

```
texto
-
Bm   = 1
hpat = 116

texto
 -
Bm   = 256
hpat = 29797

texto
  -
Bm   = 65536
hpat = 7628152

texto
   -
Bm   = 30
hpat = 6653452

texto
    -
Bm   = 7680
hpat = 399444

SEARCHING ...

texto
Este es un texto de prueba.

htxt = 8161434
hpat = 399444

 texto
Este es un texto de prueba.

htxt = 7514109
hpat = 399444

  texto
Este es un texto de prueba.

htxt = 3017033
hpat = 399444

   texto
Este es un texto de prueba.

htxt = 7425248
hpat = 399444

    texto
Este es un texto de prueba.

htxt = 7793739
hpat = 399444

     texto
Este es un texto de prueba.

htxt = 2906344
hpat = 399444

      texto
Este es un texto de prueba.
```

```
htxt = 191471
hpat = 399444

       texto
Este es un texto de prueba.

htxt = 7466538
hpat = 399444

        texto
Este es un texto de prueba.

htxt = 3028809
hpat = 399444

         texto
Este es un texto de prueba.

htxt = 85319
hpat = 399444

          texto
Este es un texto de prueba.

htxt = 6899212
hpat = 399444

           texto
Este es un texto de prueba.

htxt = 399444
hpat = 399444

-- PARTIAL MATCH --

           texto
Este es un texto de prueba.
           _

           texto
Este es un texto de prueba.
           *_

           texto
Este es un texto de prueba.
           **_

           texto
Este es un texto de prueba.
           ***_

           texto
Este es un texto de prueba.
           ****_

           texto
Este es un texto de prueba.
           *****

-- FULL MATCH --

            texto
Este es un texto de prueba.

htxt = 21311
hpat = 399444
```

```
             texto
Este es un texto de prueba.

htxt = 8207868
hpat = 399444

              texto
Este es un texto de prueba.

htxt = 3017063
hpat = 399444

               texto
Este es un texto de prueba.

htxt = 7432928
hpat = 399444

                texto
Este es un texto de prueba.

htxt = 6876200
hpat = 399444

                 texto
Este es un texto de prueba.

htxt = 2896968
hpat = 399444

                  texto
Este es un texto de prueba.

htxt = 8145973
hpat = 399444

                   texto
Este es un texto de prueba.

htxt = 7750277
hpat = 399444

                    texto
Este es un texto de prueba.

htxt = 168653
hpat = 399444

                     texto
Este es un texto de prueba.

htxt = 7523351
hpat = 399444

                      texto
Este es un texto de prueba.

htxt = 7348996
hpat = 399444

-*- Karp Rabin -*-

Number of matches:           1
Match positions:            11
```

Si lo comparamos con el ejemplo anterior vemos que el único acierto parcial es el correspondiente al acierto completo. El valor `Bm` mostrado en el preproceso es el de la base (tamaño del alfabeto) elevada al tamaño del patrón. Este valor se calcula en el preproceso, ya que hasta no disponer de patrón no tenemos su tamaño. El cálculo se hace a la vez que el de la función de dispersión del patrón usando desplazamientos en registro, como veremos en la implementación.

Después de un acierto continuamos calculando la siguiente función de dispersión del texto, de modo similar al caso de la fuerza bruta.

La ventaja más comentada de este algoritmo es que es extensible a búsqueda en dos dimensiones.

2.3.2. Costes

El preproceso tiene coste temporal de Ω (m), aunque se suele ignorar ya que se puede hacer junto con el inicio de la búsqueda.

El coste temporal de la búsqueda es de Ω (m+n) en promedio. La m es por la comparación del patrón y el texto en un acierto, si no hay colisiones la n es el recorrido de todos los caracteres (el algoritmo es lineal). En el peor caso el coste es Ω (mn), si todas las secuencias de m caracteres tienen la misma función de dispersión realizaríamos una búsqueda por fuerza bruta (en la práctica esto supondría buscar un patrón con todos sus símbolos iguales en un texto que sólo contenga ese símbolo), aunque dadas las constantes empleadas este caso sólo se presentará cuando realmente todas las subcadenas del texto coincidan con el patrón ([Dav86]).

El coste espacial es nulo ya que no almacenamos la tabla de dispersión, salvo que consideremos el almacenamiento del patrón como parte del algoritmo, en cuyo caso será Ω (m).

2.3.3. Implementación

Presentaremos a continuación las versiones en C del preproceso y la función de búsqueda. En ambos casos se supone definida una base B que es potencia de dos, de modo que $B \equiv 2^{BS}$ y un número primo Q que cumple las condiciones mencionadas en la descripción del algoritmo.

El preproceso del algoritmo calcula los valores B^m y *hpat* (función de dispersión para el patrón):

```
preproceso_karp_rabin() {
  int pi;                                /* variables auxiliares */
  Bm = 1; hpat = pat[0];             /* inicialización de variables */
  for (pi = 1; pi < pat_size; pi++) {
    Bm = (Bm << BS) % Q;
    hpat  = ( (hpat << BS) + pat[pi]) ) % Q;
  }
};
```

El cálculo del primer valor de `hpat` se pone fuera del bucle para inicializar a la vez el valor de `Bm`.

La versión incluida en la biblioteca calcula los valores de `B` y `BS` en función del tamaño del alfabeto empleado y selecciona el valor del número primo `Q` de una tabla usando como índice el valor de `BS` obtenido antes.

Con el patrón preprocesado (es decir, suponiendo inicializados `hpat` y `Bm`) podemos ejecutar la función de búsqueda:

```
karp_rabin() {
  int ti, pi, tj;                           /* variables auxiliares */
  ti = 0;                                     /* índice en el texto */
  htxt = 0;                                     /* calculo de htext */
```

```
  while (ti < pat_size) {
    htxt = ( (htxt << BS) + txt[ti] ) % Q;
    ti++;
  }

  while (ti < txt_size - pat_size){
    if (htxt == hpat) {        /* Acierto parcial, comparar símbolos */
      pi = 0;                   /* índice acierto parcial en patrón */
      tj = ti;                   /* índice acierto parcial en texto */
      while (txt[tj] == pat[pi]) {
        tj++; pi++;
        if (pi == pat_size){
        acierto_en (ti);
          break;              /* continuamos en ti + 1, al salir del */
                              /* bucle incrementa su valor (en ti++) */
        }
      }
    }                                       /* Fin Acierto Parcial */
    htxt= ( htxt + (Q << BS) - txt[ti] * Bm ) % Q;
    htxt= ( (htxt << BS) + txt[ti+pat_size] ) % Q;
    ti++;
  }                                             /* Fin bucle while */
}
```

Para este algoritmo se pueden hacer los mismos comentarios que para el de fuerza bruta: el algoritmo se detiene cuando ya no se puede encontrar un acierto, `ti` siempre apunta a la posición relativa del patrón respecto al texto y `txt_size - pat_size` se puede almacenar en una variable local. La versión real aplica las modificaciones mencionadas y utiliza punteros para los índices.

2.4. Knuth-Morris-Pratt

2.4.1. Descripción

Este algoritmo es similar al de la fuerza bruta, pero en el caso de un acierto parcial utiliza el conocimiento de los caracteres previamente analizados para no retroceder en el texto. Además, como siempre avanza hacia adelante, no necesita mecanismos de *buffering* al realizar búsquedas en fichero. El algoritmo se describe en [KMP77].

Para evitar el retroceso, el algoritmo preprocesa el patrón para construir una tabla, denominada *tabla de siguientes*, que indica que posición del patrón debemos alinear con el texto en caso de fallo durante un acierto parcial.

A continuación veremos un ejemplo de la ejecución del algoritmo, que emplearemos más adelante como referencia para explicar como se realiza el preproceso y el uso de los valores calculados.

```
Text         : Este es un texto de prueba.
Text size    : 28
Pattern      : text
Pattern size : 4

-*- Knuth Morris Pratt -*-

PREPROCESSING ...

text
-
shift [0] = -1
resume  = -1
```

```
text
 -
shift [1] = 0
resume  = 0

text
 -
shift [1] = 0
resume  = -1

text
  -
shift [2] = 0
resume  = 0

text
  -
shift [2] = 0
resume  = -1

text
   -
shift [3] = -1
resume  = 0

text

shift [4] = 1
resume  = 1


PREPROCESSOR STRUCTURES

  0  1  2  3  : i
  t  e  x  t  : pat[i]
 -1  0  0 -1  : shift [i]

resume = 1


SEARCHING ...

text
Este es un texto de prueba.
-

 text
Este es un texto de prueba.
 -

  text
Este es un texto de prueba.
  -

-- PARTIAL MATCH --

  text
Este es un texto de prueba.
  *-

  text
Este es un texto de prueba.
  **-

shift [2] = 0
```

```
    text
Este es un texto de prueba.
    -

shift [0] = -1

     text
Este es un texto de prueba.
     -

      text
Este es un texto de prueba.
      -

       text
Este es un texto de prueba.
       -

        text
Este es un texto de prueba.
        -

         text
Este es un texto de prueba.
         -

          text
Este es un texto de prueba.
          -

           text
Este es un texto de prueba.
           -

-- PARTIAL MATCH --

           text
Este es un texto de prueba.
           *-

           text
Este es un texto de prueba.
           **-

           text
Este es un texto de prueba.
           ***-

           text
Este es un texto de prueba.
           ****

-- FULL MATCH --

resume  = 1

-- PARTIAL MATCH --

              text
Este es un texto de prueba.
              *-

shift [1] = 0

               text
Este es un texto de prueba.
               -
```

```
shift [0] = -1

                text
Este es un texto de prueba.
                -

                 text
Este es un texto de prueba.
                 -

                  text
Este es un texto de prueba.
                  -

                   text
Este es un texto de prueba.
                   -

                    text
Este es un texto de prueba.
                    -

                     text
Este es un texto de prueba.
                     -

                      text
Este es un texto de prueba.
                      -

                       text
Este es un texto de prueba.
                       -

-*- Knuth Morris Pratt -*-

Number of matches:           1
Match positions:            11
```

Como se ve en el ejemplo, cuando se produce un fallo en una posición del patrón cuyo siguiente es i (con $i \geq 0$) deberemos alinear el carácter que ocupa la posición i en el patrón con el carácter actual en el texto (desplazando el patrón, pero sin movernos en el texto), y continuar comparando a partir de esta posición. Cuando el siguiente es -1 deberemos desplazarnos a la siguiente posición del texto, alineando el primer caracter del patrón con ella y reiniciar la búsqueda.

En realidad, la tabla de siguientes se emplea para determinar el máximo prefijo del patrón que podemos encontrar inmediatamente antes de cada símbolo. Cuando se produce un fallo al comparar el contenido de una posición del patrón con el texto, la tabla de siguientes «recuerda» los símbolos que hemos visto anteriormente, de manera que si el fallo se produce despues de haber visto un prefijo, alineamos el patrón con él y continuamos comparando, sin retroceder en el texto.

El valor en la tabla de siguientes para el elemento que ocupa la posición j del patrón será el máximo índice $i < j$ que verifique $pat_i \neq pat_j$ para $i \geq 0$ y $pat_{0 \ldots i-1} \equiv pat_{j-i \ldots j-1}$ para $i > 0$. En caso de no existir un valor de i que verifique la condición le asignamos un valor que no sea un índice del vector (para simplificar usaremos $i = -1$). La tabla de siguientes se calcula para los valores de j entre 0 y m.

Para recuperarnos después de un acierto utilizamos el valor de la tabla de siguientes que correspondería a un símbolo que se encuentra en la posición que sigue al último elemento del patrón (el que estaría en la posición m), suponiendo que el símbolo adicional no coincide con ninguno de los visitados anteriormente. Este valor es el `resume` de los ejemplos.

Para el patrón `mimo`, la tabla de siguientes y el valor de recuperación serán:

```
i        :   0  1  2  3
pat[i]   :   m  i  m  o
shift[i] :  -1  0 -1  1
resume   = 0
```

El valor de $shift_0$ será siempre -1, ya que no existe un i que verifique $0 \leq i$ y $i < 0$, $shift_1$ será 0, ya que $pat_0 \neq pat_1$, $shift_2$ valdrá -1, ya que para $i \equiv 0$ no se cumple la primera condición ($pat_0 \equiv pat_2$), $shift_3$ será 1, ya que $pat_{0\ldots1-1} \equiv pat_{3-1\ldots3-1}$ y $pat_1 \neq pat_3$. Si existiera un símbolo adicional distinto de los anteriores su valor sería 0, ya que $pat_0 \neq pat_4$.

El funcionamiento de este algoritmo es similar al de *Aho-Corasik* [Aho75] para búsqueda de múltiples patrones, ya que ambos se basan en la misma idea. La diferencia está en que para el caso de un solo patrón este algoritmo mejora los costes espaciales y de tiempo de preproceso.

2.4.2. Costes

El coste temporal de este algoritmo es independiente del tamaño del alfabeto y es de orden Ω (n) en el mejor caso y de Ω (m+n) en el peor [KMP77].

El coste espacial será de Ω (m), ya que para almacenar la tabla de siguientes usamos un vector del mismo tamaño que el patrón.

2.4.3. Implementación

En el código en C del preproceso supondremos declarado un vector de enteros `shift` del mismo tamaño del patrón y una variable `resume` en la que almacenaremos el índice empleado para recuperarnos después de un acierto.

```c
preproceso_knuth_morris_pratt () {
  int pi;                                /* variables auxiliares */
  pi=0; resume = -1;                 /* inicialización de variables */
  shift[pi] = resume;
  while (pi < pat_size) {
    if (resume < 0 || pat[pi] == pat[resume]) {
      pi++; resume++;
      if ( (pat[pi] < pat_size) && (pat[i] == pat[resume]) ) {
        shift[pi] = shift[resume];
      } else {
        shift[pi] = resume;
      }
    } else {
      resume = shift[resume];
    }
  }
}
```

El preproceso comienza asignando a la primera posición el valor -1, y va avanzando en el patrón cada vez que `resume == -1` o el símbolo actual del patrón es igual al que ocupa la posición `resume` (cuando hemos alcanzado la posición actual o hemos encontrado un prefijo). Si `resume <= 0` actualizamos su valor asignándole el valor del siguiente de la posición a la que apunta.

Al avanzar en el patrón asignamos el valor de la tabla de siguientes en función del símbolo que encontremos, el siguiente será el mismo que el del simbolo que ocupa la posición `resume` si ambos símbolos son iguales (el símbolo actual forma parte de un prefijo, y por lo tanto ya hemos calculado su siguiente), o, en caso contrario, será el valor `resume`, que contendrá la posición alcanzada en el patrón.

En realidad, el preproceso es una ejecución del algoritmo (en su forma original), buscando el patrón sobre si mismo.

El algoritmo de búsqueda será:

```
knuth_morris_pratt (){
  int ti, pi;                                 /* variables auxiliares */
  ti = 0;                                       /* índice en el texto */
  while (ti < txt_size - pat_size){
    if (txt[ti] = pat[0]) {                        /* Acierto parcial */
      pi = 0;                  /* índice acierto parcial en patrón */
      do {
        ti++; pi++;
        if (pi == pat_size){
        acierto_en (ti - pat_size);
                                    /* Recuperación tras un acierto */
        if (resume < 0) {
        ti++;                           /* Continuamos en ti + 1 */
            break;
          } else if (pi == 0) {
            break;             /* Reinicio al principio del patrón */
          } else {
            pi = resume;         /* Seguimos en el acierto parcial */
          }
                               /* Fin Recuperación tras un acierto */
        }
                                     /* Paso de un acierto parcial */
        if (pat[pi] != txt[ti]) {
        if (shift[pi] < 0) {
        ti++;                           /* Avanzamos en el texto */
            break;
        } else {
        if (shift[pi] == 0)
              break;           /* Reinicio al principio del patrón */
            else
              pi = shift[pi];
          }
        }
                                  /* Fin Paso de un acierto parcial */
      } while (txt[ti] == pat[pi]);
    }                                         /* Fin Acierto Parcial */
    ti++;
  }                                               /* Fin bucle while */
}
```

Al igual que en otros algoritmos, la búsqueda se detiene cuando ya no se puede encontrar un acierto y la ejecución de la condición de salida se puede optimizar almacenando el valor de `txt_size - pat_size` en una variable local.

La recuperación tras un acierto es identica al paso de un acierto parcial cuando los símbolos actuales del patrón y el texto son distintos, pero resulta más sencillo y rápido repetir el código en lugar de usar condicionales (salvo que tengamos muchos aciertos, la recuperación se ejecuta pocas veces, pero usando condicionales haremos comprobaciones siempre que haya un acierto parcial).

La versión real aplica las modificaciones mencionadas y utiliza punteros para los dos índices..

2.5. Shift Or

2.5.1. Descripción

Este algoritmo se debe a Baeza-Yates y Gonnet ([Bae92b], [Bae91]). Se basa en la teoría de autómatas y en el uso de alfabetos finitos. Funciona recorriendo el texto de izquierda a derecha, carácter a carácter, sin retroceder jamás. El algoritmo representa el estado de la búsqueda con un número y usa desplazamientos en registro y una operación `O` lógica cada vez que se desplaza en el texto. Para simplificar la actualización del estado y eliminar las comparaciones entre símbolos se emplea una tabla del tamaño del alfabeto que asocia una máscara a cada símbolo.

El algoritmo emplea un método que en su planteamiento original es muy ineficiente, la idea es representar el estado de la búsqueda con un vector de valores binarios que nos indican, para cada posición i del patrón, si los últimos i caracteres del texto visitados son iguales a los primeros i caracteres del patrón. Es decir, si estamos mirando el carácter j del texto, el valor de v_i será verdadero si $pat_{0...i} \equiv txt_{j-i..j}$ o falso en caso contrario. Si el valor de la última posición es verdadero habremos encontrado un acierto (todos los símbolos del patrón serán iguales a los del texto).

Para actualizar los valores de este vector tras cada avance en el texto podemos ir desplazando la información de unas posiciones a otras, reduciendo las actualizaciones a la comparación del carácter actual en el texto. Si hacemos $v_{i+1} = v_i$ para $i < m - 1$, habremos comprobado si $pat_{0...i-1} \equiv txt_{j-(i-1)..j}$, y sólo nos faltará comprobar si $pat_i \equiv txt_{j+1}$ para cada $i < m - 1$. En realidad la última comprobación sólo se debe hacer para v_0 y los v_i que tienen valor verdadero tras el desplazamiento.

El algoritmo tal y como lo acabamos de plantear es muy costoso, ya que aunque sólo miremos un carácter del texto debemos compararlo con cada uno de los símbolos del patrón. Para evitar las comparaciones en cada paso aprovechamos el hecho de que el alfabeto es finito, construyendo una tabla de vectores binarios del tamaño del patrón para cada símbolo, que denominaremos *tabla de máscaras*. Para cada símbolo del alfabeto el vector máscara nos indicará en que posiciones del patrón lo podemos encontrar, asignando el valor verdadero al elemento o elementos correspondientes de la máscara.

Utilizando la tabla de máscaras eliminamos las comparaciones en la actualización después de cada paso. En primer lugar desplazamos los valores del vector ($v_{i+1} = v_i$ para $i < m - 1$) y luego añadimos el componente aportado por el nuevo símbolo, haciendo ($v_i = v_i) \wedge (mask_{txt_{j+1},i})$ ($\wedge$ es un `Y` lógico), es decir, si los primeros $i - 1$ elementos del patrón coincidían con los caracteres del texto correspondientes ($txt_{j-(i-1)..j}$) y txt_{j+1} está en la posición i del patrón, el valor de v_i será verdadero.

La ventaja de este planteamiento es que podemos emplear un número binario para representar los vectores de booleanos siempre que el patrón sea de menor tamaño que el número de bits en una palabra del computador, reduciendo las operaciones de actualización a un desplazamiento (*shift*) en registro y un `Y` lógico ($\wedge$) a nivel de bit.

Siguiendo el razonamiento anterior, el algoritmo debería denominarse *Shift-And* ([Bae92b]), pero como se ha implementado en C y los desplazamientos en registro de este lenguaje añaden ceros cuando nos movemos hacia la izquierda, se ha modificado el método de actualización anterior para que funcione utilizando operaciones `O` lógicas, representando con un 0 la presencia del símbolo y con un 1 la ausencia (cuando hacemos el `O` de dos valores sólo obtendremos un cero si ambos son 0, que es lo mismo que hacíamos antes con la operación `Y`).

Veamos un ejemplo:

```
Text         : Este es un texto de prueba.
Text size    : 28
Pattern      : texto
Pattern size : 5
```

```
-*- Shift Or -*-

PREPROCESSING ...

mask [ * ] = 11111111111111111111111111111111

texto
-
mask ['t'] = 01111111111111111111111111111111

texto
 -
mask ['e'] = 10111111111111111111111111111111

texto
  -
mask ['x'] = 11011111111111111111111111111111

texto
   -
mask ['t'] = 01101111111111111111111111111111

texto
    -
mask ['o'] = 11110111111111111111111111111111

PREPROCESSOR STRUCTURE

mask ['e'] :  10111111111111111111111111111111
mask ['o'] :  11110111111111111111111111111111
mask ['t'] :  01101111111111111111111111111111
mask ['x'] :  11011111111111111111111111111111
mask [ * ] :  11111111111111111111111111111111

SEARCHING ...

texto
Este es un texto de prueba.
-

 texto
Este es un texto de prueba.
 -

  texto
Este es un texto de prueba.
  -

-- PARTIAL MATCH --

  texto
Este es un texto de prueba.
  *-

state      = 01111111111111111111111111111111
mask ['e'] = 10111111111111111111111111111111
state << 1 = 00111111111111111111111111111111
nextstate  = 10111111111111111111111111111111

  texto
Este es un texto de prueba.
  **-

state      = 10111111111111111111111111111111
mask [' '] = 11111111111111111111111111111111
state << 1 = 01011111111111111111111111111111
nextstate  = 11111111111111111111111111111111
```

```
     texto
Este es un texto de prueba.
     _

      texto
Este es un texto de prueba.
      _

       texto
Este es un texto de prueba.
       _

        texto
Este es un texto de prueba.
        _

         texto
Este es un texto de prueba.
         _

          texto
Este es un texto de prueba.
          _

           texto
Este es un texto de prueba.
           _

-- PARTIAL MATCH --

           texto
Este es un texto de prueba.
           *_

state      = 01111111111111111111111111111111
mask ['e'] = 10111111111111111111111111111111
state << 1 = 00111111111111111111111111111111
nextstate  = 10111111111111111111111111111111

           texto
Este es un texto de prueba.
           **_

state      = 10111111111111111111111111111111
mask ['x'] = 11011111111111111111111111111111
state << 1 = 01011111111111111111111111111111
nextstate  = 11011111111111111111111111111111

           texto
Este es un texto de prueba.
           ***_

state      = 11011111111111111111111111111111
mask ['t'] = 01101111111111111111111111111111
state << 1 = 01101111111111111111111111111111
nextstate  = 01101111111111111111111111111111

           texto
Este es un texto de prueba.
           ****_

state      = 01101111111111111111111111111111
mask ['o'] = 11110111111111111111111111111111
state << 1 = 00110111111111111111111111111111
nextstate  = 11110111111111111111111111111111
```

```
           texto
Este es un texto de prueba.
           *****

-- FULL MATCH --

                texto
Este es un texto de prueba.
                -

                 texto
Este es un texto de prueba.
                 -

                  texto
Este es un texto de prueba.
                  -

                   texto
Este es un texto de prueba.
                   -

                    texto
Este es un texto de prueba.
                    -

                     texto
Este es un texto de prueba.
                     -

                      texto
Este es un texto de prueba.
                      -

-*- Shift Or -*-

Number of matches:          1
Match positions:           11
```

El preproceso asigna a cada símbolo del alfabeto una máscara con todos los valores a uno y despues va recorriendo el patrón símbolo a símbolo, asignando un cero al bit que ocupa la posición visitada en la máscara del símbolo correspondiente, sin modificar el valor del resto de bits.

La búsqueda se ejecuta mirando si la máscara de cada símbolo del texto tiene un cero en el primer bit. Si lo tiene, se produce un acierto parcial, en el primer momento el estado tendrá un cero en el primer bit y el resto serán unos, cuando pasamos al siguiente símbolo del texto desplazamos el cero (introduciendo un nuevo 0) y hacemos un `O` lógico con la máscara del símbolo. Si el valor del nuevo estado contiene algún cero continuamos desplazándonos y actualizando la máscara, si es todo unos salimos del acierto parcial. El acierto se produce cuando encontramos un cero en la posición correspondiente al último caracter del patón. Para recuperarnos del acierto eliminamos el ultimo cero del estado, si el valor es todo unos salimos del acierto parcial y continuamos y si no seguimos en el búcle, actualizando el nuevo estado.

La ventaja fundamental de este algoritmo es su fácil extensión a búsquedas con clases de caracteres, ya que basta con modificar el cálculo de la máscara para que ponga a 0 el bit correspondiente a la posición de la clase en el patrón para las máscaras de todos los símbolos de la clase. Con este cambio no hay que modificar el algoritmo de búsqueda para que se encuentre el acierto. De igual modo se soporta el uso de símbolos *don't care*, es decir, el uso de patrones para los que nos da igual que símbolo ocupe alguna de sus posiciones. La modificación es sencilla, podemos considerar que la clase de caracteres es todo el alfabeto (todas las máscaras contendrán un cero en la posición del símbolo que no nos importa).

En [Wu92] se presenta una versión de este algoritmo que permite la búsqueda aproximada o con errores. El algoritmo se utiliza en el programa `agrep`, una herramienta de búsqueda aproximada para Unix, que se ha incluido en el sistema de indexación y recuperación denominado `glimpse` ([Man93], [glimpse]).

2.5.2. Costes

El coste temporal del preproceso es de Ω (alpha_size + m), alpha_size por la inicialización del vector de máscaras a 1, y m para calcular la máscara de cada símbolo del patrón.

El coste temporal de la búsqueda es de Ω (n) y es independiente del tamaño del patrón y el alfabeto [Bae92b].

El coste espacial es de Ω (alpha_size) que es el espacio empleado para almacenar la tabla de máscaras.

2.5.3. Implementación

A continuación veremos el código en C del preproceso, supondremos definido un vector de enteros del tamaño del alfabeto (`mask`) y una máscara que nos indicará cual es el último estado de la búsqueda (`last_state`) y cual es el máximo número de estados (`max_states`).

```
preproceso_shift_or() {
  int i, st;                              /* variables auxiliares */
  for (i=0; i < alpha_size; i++)
    mask[i] = ~0;
  max_states = min(pat_size, sizeof(int) * 8);
  st = 1;
  for (i=0; i < max_states; i++) {
    mask[ pat[i] ] &= ~st;
    st <<= 1;
  }
  last_state = 1 << (max_states - 1);
}
```

La actualización de las máscaras se hace empleando una variable auxiliar que contiene un uno en la posición correspondiente al símbolo visitado, como inicializamos las máscaras poniendo todos los bits a uno, haciendo un Y lógico de la máscara con el valor de la variable auxiliar negada pondremos un cero en la posición correspondiente, manteniendo cualquier otro cero que hubiera en la máscara.

El valor de `max_states` será el tamaño del patrón o el número de bits de cada máscara si el patrón lo supera en tamaño.

El código en C de la búsqueda será:

```
shift_or() {
  int ti, pi, tj;                          /* variables auxiliares */
  ti = 0;                                    /* índice en el texto */
  while (ti < txt_size - pat_size){
    if ( (mask[ txt[ti] ] & 1) == 0) {          /* Acierto parcial */
      state = ~0;
      state = (state << 1) | (mask[txt[ti]]);
      while (state != ~0) {
        ti++;
        if ((state & last_state) == 0) {
          pi = max_states;       /* índice acierto parcial en patrón */
          tj = ti;                /* índice acierto parcial en texto */
            /* búsqueda lineal si el patrón es mayor que la máscara */
```

```
        if (pat_size > max_states) {
        while (txt[tj] == pat[pi]) {
        pi++; tj++;
        if (pi == pat_size) break;
        }
        }
          if (pi == pat_size){
            acierto_en (tj - pat_size);
        /* recuperación */
        state = state | last_state;       /* borramos el acierto */
        if (state == ~0)        /* si no hay aciertos pendientes */
        break;                  /* salimos del acierto parcial */
          }
        }
        state = (state << 1) | (mask[txt[ti]]);
      }
    }                                          /* Fin Acierto Parcial */
    ti++;
  }                                                /* Fin bucle while */
}
```

Como no sabemos cual es el tamaño del patrón antes de comenzar a buscar en la detección de los aciertos comprobamos que es menor que `max_states`, en caso de no serlo terminamos la búsqueda por fuerza bruta. Para los patrones de tamaño mayor al número de bits de la palabra del computador la recuperación retrocede en el texto, continuando a partir del último símbolo analizado por el método normal (usando el estado). El problema se podría evitar, pero la solución complicaría el algoritmo y en la práctica es raro encontrar patrones más largos de 32 caracteres, que es el tamaño de los enteros en la mayoría de computadores modernos.

En la versión real la condición de entrada en el acierto parcial compara el símbolo actual del texto con el primer caracter del patrón, ya que no precisa acceder al vector de máscaras (es más rápido). Esto implica que si queremos extender el algoritmo para que soporte clases de caracteres debemos modificar la implementación, ya que si ponemos la clase en la primera posición del patrón no se puede comparar un solo símbolo.

Como en otros algoritmos, la búsqueda se detiene cuando ya no se puede encontrar un acierto y accedemos al texto a través de punteros, aunque para la tabla de máscaras necesitamos usar el vector, ya que el acceso no es secuencial.

2.6. Boyer-Moore

2.6.1. Descripción

Este es el algoritmo más rápido para la búsqueda de un solo patrón en la teoría y la práctica. Se presentó por primera vez en [BM77].

Funciona recorriendo el texto de izquierda a derecha, pero comparando el patrón con el texto de derecha a izquierda. El aumento de velocidad se consigue saltando caracteres del texto que no pueden dar origen a un acierto. Para saltar empleamos dos tablas auxiliares que denominaremos *tabla de salto* y *tabla de desplazamiento*.

Supongamos que nos encontramos al principio de la búsqueda, con el patrón alineado con el inicio del texto. Según lo comentado antes, lo primero que haremos será comparar pat_{m-1} con txt_{m-1}. Si sus valores no coinciden y el caracter txt_{m-1} no aparece en el patrón, podremos desplazarnos sobre el texto m posiciones (alineando pat_0 con txt_m), ya que es seguro que no encontraremos ningun acierto que comience antes de txt_m (cualquier acierto que comenzara en una posición anterior a la m fallaría al comparar un caracter del patrón con txt_{m-1}). En este caso habremos dejado de comparar $m - 1$ símbolos del patrón y la siguiente comparación será entre pat_{m-1} y txt_{2m-1}.

Siguiendo con la técnica anterior llegamos al caso en el que el patrón está alineado con la posición $j - (m-1)$ del texto, y al comparar pat_{m-1} con txt_j los valores no coinciden pero el símbolo del texto aparece en otra posición del patrón. En este caso no podremos saltar m posiciones, pero si sabremos que, en caso de haber un acierto, el símbolo actual del texto deberá estar alineado con la posición del patrón en la que aparece. Si esa posición es $m - 1 - k$ alinearemos pat_{m-1-k} con txt_j y continuaremos comparando pat_{m-1} con txt_{j+k}, saltando k posiciones. Hay que señalar que si el símbolo aparece más de una vez en el patrón deberemos elegir el índice de la ocurrencia más a la derecha, ya que si no lo hacemos así podemos perder aciertos. Por ejemplo, si buscámos la palabra `texto` en la frase «`Un pretexto absurdo`» llegamos a la situación:

```
     texto
Un pretexto absurdo
         -
                       (desplazamiento para k=1)
      texto
Un pretexto absurdo
          -
                       (desplazamiento para k=4)
         texto
Un pretexto absurdo
             -
```

En este caso debemos desplazar el patrón una posición a la derecha, ($txt_j \equiv pat_{4-1} \equiv 't'$, luego $k = 1$), si escogemos $k = 4$ perderemos el acierto.

En el caso de que el símbolo actual en el texto coincidiera con el último carácter del patrón ($txt_j \equiv pat_{m-1}$), seguimos comparando los símbolos precedentes del texto y el patrón hasta encontrar un acierto completo o hasta que se produce un fallo. Si el fallo se produce en la posición $i - 1$ del patrón sabremos que $pat_{i..m-1} \equiv txt_{j-(m-1)+i..j}$, es decir, que los últimos $(m-1) - i$ caractes del patrón y el texto coinciden. En esta situación, si la ocurrencia más a la derecha de txt_{j-m+i} en el patrón es pat_g con $g < i - 1$ (antes del fallo) podemos desplazar el patrón g posiciones a la derecha para alinear pat_g con txt_{j-m+i}, continuando con la comparación de pat_{m-1} con $txt_{j-(m-1)+i+g}$. Si $g > i - 1$ (la ocurrencia está despues del fallo), no ganamos nada alineando, ya que eso implicaría retroceder en el texto a posiciones ya estudiadas, por lo que desplazamos el patrón sobre el texto una posición (lo alinearemos con $j + 1$) y continuaremos comparando pat_{m-1} con txt_{j+1}, ya que sabemos que en la posición actual no hay acierto, pero no podemos decir nada de la siguiente.

Hasta aquí hemos visto la denominada *heurística de salto*, para implementarla preprocesaremos el patrón asignando a cada símbolo del alfabeto el máximo salto que podemos dar cuando al encontrarlo se produce un fallo. Para todos los símbolos del alfabeto que no se encuentren en el patrón el salto será m, y para los símbolos que sí aparezcan en él el salto máximo será la distancia desde su última aparición en el patrón hasta el final del mismo.

La *heurística de desplazamiento* surge al estudiar el patrón de un modo similar al del algoritmo de Knuth, Morris y Pratt, al darnos cuenta de que cuando se produce un fallo trás un acierto parcial es posible avanzar más posiciones que con la tabla de saltos, teniendo en cuenta que despues del desplazamiento el patrón debe coincidir con los símbolos previamente comparados y que el símbolo alineado con la posición del texto que causó el fallo debe ser distinto al símbolo con el que se comparó antes.

Cuando $pat_{i..m-1} \equiv txt_{j-(m-1)+i..j}$ y $pat_{i-1} \neq txt_{j-m+i}$ si $pat_{i..m-1}$ aparece como la subcadena $pat_{i-g...(m-1)-g}$ en el patrón y $pat_{i-g} \neq pat_i$ con la $g < i$ de mayor valor (es decir, si hay más de una aparición de la subcadena, tomamos la que se encuentra más a la derecha), podemos avanzar más en el texto que con la heurística de saltos alineando $pat_{i-g\ldots(m-1)-g}$ con $txt_{j-(m-1)+i\ldots j}$ y continuar comparando pat_{m-1} con txt_{j+g}.

Para utilizar está heurística calculamos una tabla del tamaño del patrón que contiene para cada posición del mismo el desplazamiento g descrito antes más un desplazamiento adicional para apuntar a la última posición del patrón una vez desplazado. La definición formal de las entradas de la tabla *shift* será:

$$shift_i = \min(g + m - 1 - i : g \geq 1''\wedge''(g \geq i'\vee'pat_{i-g} \neq pat_i)''\wedge''((g \geq k'\vee'pat_{k-g} \equiv pat_k)''para''j < k < m))$$

En la ejecución del algoritmo usaremos ambas heurísticas, seleccionando en cada fallo el valor que nos permita saltar más en el texto. Así, en cada paso compararemos el último elemento del patrón con el símbolo del texto correspondiente, si no son iguales incrementaremos nuestra posición en el texto sumándole el valor $\max(skip_{txt_j}, shift_{m-1})$ y si son iguales continuaremos mirando los anteriores hasta encontrar el patrón completo o detenernos en un fallo, en cuyo caso haremos lo mismo que antes (sumaremos el valor $\max(skip_{txt_j}, shift_i)$ a la posición actual en el texto, siendo i la posición actual en el patrón).

Hay que señalar que la implementación propuesta asume que siempre usamos el mismo iterador para acceder a los elementos del texto y siempre apunta al símbolo a comparar, esta es la razón de que en el cálculo de *shift* se sume un desplazamiento adicional para alcanzar el final del patrón.

Veamos un ejemplo:

```
Text         : Vamos a beber un cocoloco bajo el baobab.
Text size    : 42
Pattern      : cocolo
Pattern size : 6

-*- Boyer Moore -*-

[ ... ]

PREPROCESSOR STRUCTURES

  0  1  2  3  4  5  : i
  c  o  c  o  l  o  : pat[i]
 11 10  9  8  3  1  : shift [i]

skip ['c'] :  3
skip ['l'] :  1
skip ['o'] :  0
skip [ * ] :  6

SEARCHING ...

cocolo
Vamos a beber un cocoloco bajo el baobab.
     -

max (shift [5], skip [' ']) = 6

      cocolo
Vamos a beber un cocoloco bajo el baobab.
           -

max (shift [5], skip ['e']) = 6

            cocolo
Vamos a beber un cocoloco bajo el baobab.
                 -
```

```
max (shift [5], skip ['c']) = 3

               cocolo
Vamos a beber un cocoloco bajo el baobab.
                    -

-- PARTIAL MATCH --

               cocolo
Vamos a beber un cocoloco bajo el baobab.
                   -*

max (shift [4], skip ['c']) = 3

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                      -

-- PARTIAL MATCH --

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                     -*

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                    -**

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                   -***

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                  -****

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                 -*****

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                 ******

-- FULL MATCH --

max (shift [0], skip ['c']) = 11

                       cocolo
Vamos a beber un cocoloco bajo el baobab.
                            -

max (shift [5], skip ['j']) = 6

                             cocolo
Vamos a beber un cocoloco bajo el baobab.
                                  -

max (shift [5], skip ['b']) = 6

                                   cocolo
Vamos a beber un cocoloco bajo el baobab.
                                        -

max (shift [5], skip ['.']) = 6

-*- Boyer Moore -*-
```

```
Number of matches:          1
Match positions:           17
```

En este ejemplo el algoritmo ejecuta 16 comparaciones para buscar un patrón de longitud 6 en un texto de 42 caracteres. Para este mismo caso el algoritmo de la fuerza bruta hubiera necesitado 48 comparaciones y el de Knuth-Morris-Pratt 38. Como se ve, este algoritmo ejecuta muchas menos comparaciones que los otros.

La recuperación tras un acierto es sencilla, basta con incrementar el valor del iterador del texto como si el primer carácter del patrón no hubiera sido acertado, por la definición de las tablas sabemos que el acierto no volverá a ser encontrado (al menos nos desplazará al final del patrón más uno).

Este algoritmo ha sido muy estudiado y existen multitud de variantes: unas utilizan una versión distinta de la tabla de desplazamiento otras la eliminan por completo e incluso alteran el orden de comparación de los símbolos del patrón con el texto. En los proximos apartados veremos dos variantes de este algoritmo, para ver la descripción de algunos más se puede consultar [Ste92].

Utilizando técnicas similares a las de este algoritmo se ha desarrollado una versión para búsqueda de múltiples patrones ([Com79]).

En [Wat92] y [Wat95] se presenta una taxonomía de algoritmos de búsqueda de uno o multiples patrones ordenada por los detalles que introduce cada versión y que por tanto incluye las variantes del Boyer-Moore y Commenz-Walter.

2.6.2. Costes

El coste temporal del preproceso es de Ω (alpha_size + m) para la tabla de saltos (alpha_size para inicializar a m y m para el cálculo del salto para cada símbolo del patrón) y de Ω (m) para el cálculo de la tabla de desplazamientos. El coste total del preproceso será de Ω (alpha_size + m).

El coste temporal de la búsqueda en el peor caso es de Ω (m+n) para la búsqueda de una sola ocurrencia (Ω (m+rn) si buscamos todas las ocurrencias y el patrón aparece r veces en el texto) y en el mejor caso (si ningún símbolo del patrón aparece en el texto), el coste será de Ω (n/m).

El coste espacial es de Ω (alpha_size+m) que es el tamaño de las tablas de salto y siguientes respectivamente.

2.6.3. Implementación

Presentamos a continuación el código en C del preproceso, supondremos definidas las tablas `skip` y `shift` del tamaño del alfabeto y el patrón respectivamente.

Para el cálculo de `shift` emplearemos una tabla auxiliar `f`, que se define como $f_{m-1} = m + 1$ y $f_{j-1} = \min(i : j < i < m \wedge pat_{i..m-1} = pat_{j..j+m-1-i})$ para $0 < j < m$.

```
preproceso_boyer_moore () {
  int i, j, k;                       /* variables auxiliares */
  int f[pat_size];
                                            /* cálculo de skip */
  for (i=0; i < alpha_size; i++)
    skip[i] = pat_size;
  for (i=1; i < pat_size; i++)
    skip [ pat[i-1] ] = pat_size - i;
                                           /* cálculo de shift */
  for (i=1; i <= pat_size; i++)      /* valores máximos de shift */
    shift[i-1] += pat_size - i;
                                                          /*  */
```

```
  k = pat_size + 1;
  for (j=pat_size; j > 0; j--) {
    f[j-1] = k;
    while (k <= pat_size && pat[j-1] != pat[k-1]) {
      shift[k-1] = min ( shift[k-1], pat_size - j);
      k = f[k-1];
    }
    k--;
  }
                                                          /* */
  for (i=1; i<=k; i++) {
    shift[i-1] = min( shift[i-1], pat_size + k - i );
  }
                /* Corrección de Rytter planteada por Mehlhorn */
  j = f[k-1];
  while(k < pat_size){
    while (k <= j) {
      shift[k-1] = min( shift[k -1], pat_size + j - k);
      k++;
    }
    j = f[j-1];
  }
}
```

El código en C de la búsqueda será:

```
boyer_moore() {
  int ti, pi;                               /* variables auxiliares */
  ti = pat_size - 1;                          /* índice en el texto */
  while (ti < txt_size){
    if (txt[ti] = pat[pat_size-1]) {             /* Acierto parcial */
      pi = pat_size - 1;         /* índice acierto parcial en patrón */
      do {
        if (pi == 0){
        acierto_en (ti);
          break;                    /* continuamos en ti + max(...) */
                           /* (ti se incrementa al salir del bucle) */
        }
        ti--; pi--;
      } while (txt[ti] == pat[pi]);
      ti += max(skip[txt[ti]], shift[pi]);
    } else {                                /* Fin Acierto Parcial */
      ti += max(skip[txt[ti]], shift[pat_size-1]);
    }
  }                                              /* Fin bucle while */
}
```

Al igual que para otros algoritmos presentados antes, la versión real emplea optimizaciones triviales como el almacenamiento del valor del último símbolo del patrón y su valor asociado en la tabla `shift` en registros, además de usar punteros en lugar de índices para acceder al texto y al patrón.

2.7. Boyer Moore Horspool

2.7.1. Descripción

Este algoritmo es una versión simplificada del de Boyer y Moore debida a Horspool [Hor80] que elimina el uso de la *tabla de desplazamiento*, ya que esta tabla sólo mejora la velocidad cuando buscamos patrones muy repetitivos, que en la práctica no suelen aparecer.

Esta versión reduce el coste espacial (sólo necesitamos una tabla del tamaño del alfabeto), el problema es que ahora el peor caso tiene un coste temporal de Ω (mn), aunque en situaciones normales tendrá un comportamiento similar al del algoritmo original.

Además, se modifica el cálculo de la tabla de salto no asignando valor al símbolo pat_{m-1} (que en el caso del algoritmo de Boyer-Moore era siempre 0, ya que cuando encontrabamos un fallo en ese símbolo seguíamos saltando gracias a la tabla de desplazamiento), de manera que $skip_{pat_{m-1}}$ valdrá m si pat_{m-1} es la única ocurrencia del símbolo en el patrón o menos si hay alguna ocurrencia en una posición anterior.

Por último, si se produce un acierto parcial y no se encuentra el patrón, usando el desplazamiento asignado al símbolo del texto que provoca el fallo podemos encontrarnos con que el nuevo alineamiento deje alguno de los símbolos ya comparados alineado con su ocurrencia más a la derecha en el patrón, lo que implicaría un retroceso en el patrón que debe ser detectado, ya que puede causar recursión y detener la búsqueda. Para evitarlo siempre saltamos a partir del símbolo del texto que inició el acierto parcial, usando su valor en la tabla de salto. Por ejemplo, si buscamos el patrón `cero` en la cadena «`es un letrero enorme`» nos encontraremos con la siguiente secuencia:

```
Text         : Es un letrero enorme.
Text size    : 22
Pattern      : cero
Pattern size : 4

-*- Boyer Moore Horspool -*-

PREPROCESSING ...

skip [ * ] = 4

cero
-
skip ['c'] = 3

cero
 -
skip ['e'] = 2

cero
  -
skip ['r'] = 1

PREPROCESSOR STRUCTURES

skip ['c'] :  3
skip ['e'] :  2
skip ['r'] :  1
skip [ * ] :  4

SEARCHING ...

cero
Es un letrero enorme.
   -

skip ['u'] = 4

    cero
Es un letrero enorme.
       -
```

```
skip ['e'] = 2

      cero
Es un letrero enorme.
         -

skip ['r'] = 1

       cero
Es un letrero enorme.
          -

skip ['e'] = 2

         cero
Es un letrero enorme.
            -

-- PARTIAL MATCH --

         cero
Es un letrero enorme.
           -*

         cero
Es un letrero enorme.
          -**

         cero
Es un letrero enorme.
         -***
```

Llegados a este punto, se produce un fallo en la primera `r` de `letrero`, si usamos el desplazamiento dado por `skip['r']` llegamos a un búcle infinito:

```
skip ['r'] = 1

       cero
Es un letrero enorme.
          -

skip ['e'] = 2

         cero
Es un letrero enorme.
            -

-- PARTIAL MATCH --

         cero
Es un letrero enorme.
           -*

         cero
Es un letrero enorme.
          -**

         cero
Es un letrero enorme.
         -***

skip ['r'] = 1
```

Pero si saltamos usando el valor de `skip['o']` la búsqueda continua sin problemas:

```
skip ['o'] = 4

             cero
Es un letrero enorme.
                -

-- PARTIAL MATCH --

             cero
Es un letrero enorme.
               -*

skip ['o'] = 4

                 cero
Es un letrero enorme.
                    -

skip ['.'] = 4

-*- Boyer Moore Horspool -*-

Number of matches:           0
```

La recuperación de la búsqueda es similar a la del algoritmo de Boyer-Moore, lo que hacemos es suponer que ha habido un fallo y continuar.

Si realizaramos la misma búsqueda que en el ejemplo del Boyer-Moore veríamos que al eliminar la tabla de desplazamiento este algoritmo tiene un comportamiento peor que el original en algunos casos, ya que despues del acierto, al no recordar los símbolos previamente analizados, saltaría menos de lo posible:

```
Text         : Vamos a beber un cocoloco bajo el baobab.
Text size    : 42
Pattern      : cocolo
Pattern size : 6

-*- Boyer Moore Horspool -*-

[ ... ]

PREPROCESSOR STRUCTURES

skip ['c'] :  3
skip ['l'] :  1
skip ['o'] :  2
skip [ * ] :  6

SEARCHING ...

[ ... ]

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                 ******

-- FULL MATCH --

skip ['o'] = 2
```

```
                   cocolo
Vamos a beber un cocoloco bajo el baobab.
                        -

-- PARTIAL MATCH --

                   cocolo
Vamos a beber un cocoloco bajo el baobab.
                       -*

skip ['o'] = 2

                     cocolo
Vamos a beber un cocoloco bajo el baobab.
                          -

[ ... ]

-*- Boyer Moore Horspool -*-

Number of matches:            1
Match positions:             17
```

Al saltar a partir del valor de `skip['o']` sólo saltamos 2 posiciones a la derecha, mientras que usando la tabla de desplazamiento conseguíamos un salto de 6 posiciones hacia la derecha (11 desde el primer símbolo del patrón).

2.7.2. Costes

El coste temporal del preproceso es de Ω (alpha_size + m) para la tabla de saltos (alpha_size para inicializar a m y m para el cálculo del salto para cada símbolo del patrón).

El coste temporal de la búsqueda en el peor caso es de Ω (mn) y en el mejor caso (si ningún símbolo del patrón aparece en el texto) de Ω (n/m). El coste en promedio es similar al del algoritmo de Boyer-Moore para textos normales, aunque ahora el peor caso sea de orden Ω (mn).

El coste espacial es de Ω (alpha_size) que es el tamaño de la tabla de salto.

2.7.3. Implementación

El código en C del preproceso es sencillo, sólo asume que tenemos declarado el vector `skip` del tamaño del alfabeto:

```
preproceso_boyer_moore_hoorspool () {
  int i;                                /* variables auxiliares */
  for (i=0; i < alpha_size; i++)
    skip[i] = pat_size;
  for (i=1; i < pat_size; i++)
    skip[ pat[i-1] ] = pat_size - i;
}
```

El código en C de la búsqueda es muy similar al del algoritmo de Boyer-Moore, salvo que aquí no empleamos la función máximo y utilizamos un índice auxiliar en los aciertos parciales.

```
boyer_moore_hoorspool() {
  int ti, pi, tj;                            /* variables auxiliares */
  ti = pat_size - 1;                           /* índice en el texto */
  while (ti < txt_size){
```

```
    if (txt[ti] = pat[pat_size-1]) {              /* Acierto parcial */
      pi = pat_size - 1;         /* índice acierto parcial en patrón */
      tj = ti;                    /* índice acierto parcial en texto */
      do {
        if (pi == 0){
        acierto_en (tj);
          break;               /* continuamos en ti + skip[txt[ti]] */
                           /* (ti se incrementa al salir del bucle) */
        }
        tj--; pi--;
      } while (txt[tj] == pat[pi]);
    }                                         /* Fin Acierto Parcial */
    ti += skip[txt[ti]];
  }                                                 /* Fin bucle while */
}
```

La versión real del algoritmo utiliza punteros para acceder al patrón y al texto y almacena en registro el valor del último símbolo del patrón.

2.8. Sunday Quick Search

2.8.1. Descripción

Este algoritmo es otra versión simplificada del de Boyer y Moore que elimina el uso de la *tabla de desplazamiento* (igual que en el Boyer-Moore-Horspool) y emplea una técnica diferente para saltar en el texto. Esta versión se debe a Sunday [Sun90].

La idea básica es que cuando se produce un fallo y pat_{m-1} está alineado con el símbolo txt_i, en lugar de saltar a partir del valor de esté último podemos usar el del siguiente símbolo del texto (txt_{i+1}), ya que, para el mínimo desplazamiento (una posición a la derecha), txt_{i+1} forma parte de la nueva subcadena a examinar. De este modo, los desplazamientos de la tabla *skip* son iguales a los de la tabla del algoritmo original más uno en promedio.

Además, el algoritmo hace uso del hecho de que las comparaciones entre el patrón y la subcadena del texto alineada con él pueden hacerse en cualquier orden, lo que nos permite elegir a partir de qué símbolo queremos comenzar las comparaciones. Esta versión compara comenzando a partir del primer símbolo del patrón, continuando hacia la derecha. Las modificaciones del algoritmo son sencillas, en principio inicializamos los valores de la tabla de salto a partir de $m+1$ para incorporar el uso del símbolo siguiente para indexar los saltos y además si estamos en la posición i del texto utilizamos el símbolo $i+m$ actualizar el índice.

Sunday propone dos versiones más, una que ordena de mayor a menor los símbolos en función de la longitud del salto que proporcionan (*Maximal Shift*) y otra que utiliza una estadística de la frecuencia de aparición de los símbolos en el texto para comparar primero los de menor frecuencia (*Optimal Mismatch*).

Veamos un ejemplo:

```
Text         : Vamos a beber un cocoloco bajo el baobab.
Text size    : 42
Pattern      : cocolo
Pattern size : 6

-*- Sunday Quick Search -*-


PREPROCESSING ...
```

```
skip [ * ] = 7

cocolo
-
skip ['c'] = 6

cocolo
 -
skip ['o'] = 5

cocolo
  -
skip ['c'] = 4

cocolo
   -
skip ['o'] = 3

cocolo
    -
skip ['l'] = 2

cocolo
     -
skip ['o'] = 1


PREPROCESSOR STRUCTURES

skip ['c'] :  4
skip ['l'] :  2
skip ['o'] :  1
skip [ * ] :  7


SEARCHING ...

cocolo
Vamos a beber un cocoloco bajo el baobab.
-

skip ['a'] = 7

       cocolo
Vamos a beber un cocoloco bajo el baobab.
       -

skip [' '] = 7

              cocolo
Vamos a beber un cocoloco bajo el baobab.
              -

skip ['o'] = 1

               cocolo
Vamos a beber un cocoloco bajo el baobab.
               -

skip ['l'] = 2

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                 -

-- PARTIAL MATCH --
```

```
                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                 *_

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                 **_

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                 ***_

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                 ****_

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                 *****_

                 cocolo
Vamos a beber un cocoloco bajo el baobab.
                 ******

-- FULL MATCH --

skip ['c'] = 4

                     cocolo
Vamos a beber un cocoloco bajo el baobab.
                     _

skip ['a'] = 7

                            cocolo
Vamos a beber un cocoloco bajo el baobab.
                            _

skip ['b'] = 7

                                   cocolo
Vamos a beber un cocoloco bajo el baobab.
                                   _

skip ['
'] = 7

-*- Sunday Quick Search -*-

Number of matches:           1
Match positions:            17
```

La recuperación de la búsqueda es similar a la del algoritmo de Boyer-Moore, lo que hacemos es suponer que ha habido un fallo y continuar.

2.8.2. Costes

El coste temporal del preproceso es de Ω (alpha_size + m) para la tabla de saltos (alpha_size para inicializar a m y m para el cálculo del salto para cada símbolo del patrón).

El coste temporal de la búsqueda en el peor caso es de Ω (mn) y en el mejor caso (si ningún símbolo del patrón aparece en el texto) de Ω (n/m). En promedio es similar al Boyer-Moore y al Boyer-Moore-Horspool, aunque su rendimiento es algo mejor con patrones pequeños.

El coste espacial es de Ω (alpha_size) que es el tamaño de la tabla de salto.

2.8.3. Implementación

El código en C del preproceso es sencillo, sólo asume que tenemos declarado el vector `skip` del tamaño del alfabeto:

```
preproceso_sunday_quick_search () {
  int i;                                /* variables auxiliares */
  for (i=0; i < alpha_size; i++)
    skip[i] = pat_size + 1;
  for (i=0; i < pat_size; i++)
    skip[ pat[i] ] = pat_size - i;
}
```

El código en C de la búsqueda será muy parecido al del algoritmo Boyer-Moore-Horspool, salvo que en este caso incrementamos los iteradores en un acierto parcial y el incremento de `ti` usa el caracter que está justo despues del patrón en el texto:

```
sunday_quick_search() {
  int ti, pi, tj;                            /* variables auxiliares */
  ti = 0;
  while (ti < txt_size){
    if (txt[ti] = pat[0]) {                       /* Acierto parcial */
      pi = 0;                   /* índice acierto parcial en patrón */
      tj = ti;                   /* índice acierto parcial en texto */
      do {
        tj++; pi++;
        if (pi == pat_size){
        acierto_en (ti);
          break;     /* continuamos en ti + skip[txt[ti + pat_size]] */
                            /* (ti se incrementa al salir del bucle) */
        }

      } while (txt[tj] == pat[pi]);
    }                                        /* Fin Acierto Parcial */
    ti += skip[txt[ti + pat_size]];
  }                                              /* Fin bucle while */
}
```

La versión real del algoritmo utiliza punteros para acceder al patrón y al texto y almacena en registro el valor del primer símbolo del patrón.

Capítulo 3. Descripción del código

En este capítulo comentaremos como se implementan y que función tienen las clases que integran la biblioteca. Comenzaremos con una visión rápida que nos dará una idea de cual es la estructura de la biblioteca y las relaciones entre las clases, para pasar despues a una descripción más detallada de las clases básicas de la biblioteca, las clases auxiliares y de ayuda al análisis y implementación de los algoritmos de búsqueda de un solo patrón.

El último apartado del capítulo lo dedicaremos a comentar el programa de análisis de los algoritmos, centrandonos más en su funcionalidad que en la codificación.

3.1. Estructura de la biblioteca

En esta sección describiremos brevemente todas las clases y estructuras de la biblioteca, dejando para puntos posteriores los detalles de la implementación.

3.1.1. Clases básicas

Símbolo
: Valor de un tipo elemental que tiene definidos los operadores de comparación. La biblioteca no incluye ningún tipo especial para representarlos, un símbolo se considera alfabético cuando se accede a él usando un alfabeto.

Alfabeto
: Se define como un conjunto de símbolos a los que se asocia un índice o número de orden dentro del mismo. Todos los símbolos que no pertenecen al alfabeto tienen un índice igual al tamaño del mismo.

Cadena de símbolos
: Es una secuencia finita de símbolos. La representación de las cadenas no utiliza el alfabeto, por lo que podemos tener cadenas que contengan símbolos alfabéticos y no alfabéticos.

Algoritmo
: Clase para dar una base común a distintos tipos de algoritmos.

3.1.2. Clase auxiliares y soporte para el análisis

Función de acierto (`mfun`)
: Se usa desde los algoritmos de búsqueda simple para indicar qué debemos hacer con un acierto.

Unidad (`token`)
: Clase que se emplea para definir una unidad con algún significado dentro de una cadena: palabra, frase, etc. Se incorporan dos versiones, una que define los elementos por los símbolos válidos dentro de la unidad y otra por los que pueden aparecer antes y después de ella.

Cronómetro (`stopwatch`)
: Clase empleada para medir lapsos de tiempo.

Dispositivo de análisis (`adev`)
: Clase de análisis gráfico. Se define para los algoritmos de búsqueda simple e incorpora métodos para imprimir las distintas estructuras de preproceso y la evolución de la búsqueda. Los métodos deben definirse para cada formato y sistema de salida deseado (el formato puede ser texto simple, texto html, texto con atributos gráficos, etc., mientras que los sistemas de salida pueden ser los *streams* de C++ o cualquier sistema de ventanas, desde el del MacOS, Windows o XWindows hasta el proporcionado por la biblioteca *curses* para terminales).

Contador de pasos (`stepacct`)
: Se emplea para llevar la contabilidad de pasos de un algoritmo a partir de una serie de puntos de ruptura definidos en él.

3.1.3. Algoritmos de búsqueda

Algoritmo de búsqueda de un solo patrón (`skmalgo`)
: Clase base para derivar algoritmos de búsqueda simple. Incorpora métodos para ajustar los parámetros de análisis e invocar los algoritmos de múltiples formas.

Búsqueda por Fuerza bruta (`bfskm`)
: Implementación del algoritmo de fuerza bruta para búsqueda de un solo patrón.

Algoritmo de Karp-Rabin (`krskm`)
: Implementación del algoritmo de Karp y Rabin para búsqueda de un solo patrón. Se basa en el uso de técnicas de dispersión, representa el patrón y cada subcadena del texto del mismo tamaño que el patrón como un número, reduciendo la comparación de símbolos a la comparación de enteros.

Algoritmo de Knuth-Morris-Pratt (`kmpskm`)
: Implementación del algoritmo de Knuth, Morris y Pratt para búsqueda de un solo patrón. Busca hacia adelante sin retroceder en el texto.

Algoritmo Shift-Or (`soskm`)
: Implementación del algoritmo Shift-Or para búsqueda de un solo patrón. Busca hacia adelante sin retroceder en el texto, emplea operaciones a nivel de bit para representar el estado de la búsqueda.

Algoritmo de Boyer-Moore (`bmskm`)
: Implementación del algoritmo de Boyer y Moore para búsqueda de un solo patrón. Desplaza el patrón de izquierda a derecha en el texto comparando los caracteres de derecha a izquierda. Gracias al empleo de dos tablas de preproceso desplaza el patrón sobre el texto saltando en función de los caracteres visitados

Algoritmo de Boyer-Moore-Horspool (`bmhskm`)
: Implementación del algoritmo de Boyer, Moore y Horspool para búsqueda de un solo patrón. Versión simplificada del algoritmo de Boyer y Moore que elimina una de las tablas de preproceso, simplificando el algoritmo a coste de un peor funcionamiento en el peor caso.

Algoritmo Quick Search de Sunday (`sqsskm`)
: Implementación del algoritmo Quick Search de Sunday para búsqueda de un solo patrón. Versión simplificada del algoritmo de Boyer y Moore que elimina una de las tablas de preproceso y emplea un sistema diferente para construir y emplear la otra (usa la técnica de Boyer-Moore pero compara el patrón con el texto de izquierda a derecha).

3.2. Clases básicas

3.2.1. Símbolo

Como ya hemos mencionado en la introducción, no existe un tipo símbolo definido por la biblioteca, ya que la idea es que pueda ser usada con distintos tipos de datos; un *símbolo* sería un valor de un tipo cualquiera. Además, no necesitamos darle ningún atributo o propiedad, eso se hace al definir un alfabeto y emplearlo junto a los símbolos, por lo que no vale la pena encapsular el tipo paramétrico en una clase.

3.2.2. Alfabeto

El alfabeto es un conjunto ordenado de símbolos. La clase se define como `template`, proporcionando una especialización para el tipo `char`. Se definen métodos para comprobar si un símbolo pertenece o no al alfabeto, conocer su tamaño del alfabeto o si está vacío y obtener el índice de un símbolo dentro del conjunto o el símbolo correspondiente a un índice.

El alfabeto es estático, una vez definido no se puede modificar. Se construye pasándole un vector de símbolos del que se extraen los posibles valores eliminando duplicados.

Existe un constructor por defecto que se emplea para inicializar el alfabeto. La versión genérica genera un alfabeto vacío que no sirve más que para poder declarar variables sin inicializar (como por ejemplo dentro de una clase). La versión especializada para caracteres inicializa las tablas como alfabeto identidad en lugar de como alfabeto vacío.

Los métodos definidos son:

- `index(symbol)`. Retorna el índice del símbolo, si no pertenece al alfabeto retorna el tamaño del mismo.
- `value(index)`. Retorna el valor asociado al índice. El valor de un índice mayor que el tamaño del alfabeto no está definido.
- `size()`. Retorna el tamaño del alfabeto.
- `empty()`. Retorna verdadero si el tamaño del alfabeto es igual a cero.

Para representar el conjunto se emplean dos estructuras: un vector de símbolos y otro de enteros (indexado por símbolos e implementado usando el tipo `map` de la STL). El primer vector se utiliza para acceder a los símbolos correspondientes a cada índice en el alfabeto (con coste de orden 1) y el segundo para obtener el índice de cada símbolo (con coste de orden logarítmico respecto al tamaño del alfabeto).

La especialización para caracteres usa un vector en lugar de un mapa para asociar símbolos e índices (permitiendo acceder a estos últimos con un coste de orden 1). Dependiendo del compilador el acceso a los índices causa problemas: si el tipo usado para representar caracteres es con signo no podemos usarlos para acceder al vector (los caracteres por encima del 127 se convierten en enteros negativos, al menos en el `gcc`). La versión implementada lo soluciona convirtiendo los símbolos a caracteres sin signo antes de usarlos como índices del vector.

3.2.3. Cadena de símbolos

Al igual que sucede con los símbolos, no se define un nuevo tipo, en este caso por utilizar la clase `string` de la biblioteca estándar. En realidad se usan muy pocas operaciones del tipo, sobre todo se emplean los métodos de acceso a los iteradores inicio y fin de la cadena y el que nos da su tamaño.

3.2.4. Texto

Se define como una cadena de símbolos. Si la biblioteca fuese adecuada para implementar programas de edición de texto se podría considerar el texto como un vector de cadenas. De todos modos, tal y como está ahora, se podrían emplear los mismos algoritmos llamándolos para cada cadena del vector.

La biblioteca asume que el texto está en memoria, si se desarrolla algún programa que necesite buscar en el texto desde un fichero y no se puede mantener completo en memoria basta con hacer una composición similar al caso del vector de cadenas, dividiendo el texto en bloques. Después de acceder al primer bloque cargaríamos los últimos caracteres del primero (tantos como el tamaño del patrón menos uno) seguidos del segundo bloque, de este modo no perderíamos un acierto que estuviera en la frontera entre dos bloques. De hecho, con este método no repetimos operaciones, ya que todos los algoritmos terminan cuando ya no pueden encontrar un acierto; cuando quedan menos caracteres que el tamaño del patrón dejamos de buscar.

3.2.5. Algoritmos

Clase de la que se derivan los distintos tipos de algoritmos. Aunque podría incluir más funciones, sólo obliga a que las clases derivadas definan métodos para darle un identificador a cada algoritmo e indicar su nombre y su tipo. El identificador es útil para la selección de los algoritmos (se puede usar para comparar con la entrada del usuario en un menú), mientras que el tipo y el nombre son útiles para la salida de los algoritmos o para saber que algoritmo estamos usando cuando accedemos a través de un puntero a una clase base (podemos identificar el tipo de datos en tiempo de ejecución)

La clase podría incluir algunos de los métodos y atributos que ahora se incluyen en la clase que representa los algoritmos de búsqueda de un solo patrón, como el soporte para la contabilidad de pasos o el análisis gráfico (aunque este último debería ser modificado).

3.3. Clases auxiliares y de soporte para el análisis

3.3.1. Función de aciertos

Se emplea desde los algoritmos de búsqueda para procesar un acierto e indicar si se debe o no continuar buscando. La función de acierto definida no procesa el patrón y no detiene la búsqueda después de los aciertos (se buscan todas las ocurrencias del patrón, aunque no se utilizan para nada las posiciones de los aciertos ni se almacenan en ningún sitio), en realidad sólo debe ser empleada para definir clases derivadas que realicen algún tipo de tarea.

Define los métodos `start` y `found`, además de una función de salida para streams.

El método `start` toma como único parámetro un puntero al algoritmo que ha invocado la función. Se hace así y no pasando el texto y el patrón para que cualquier función derivada pueda acceder a información sobre el algoritmo. Durante la ejecución de la función tenemos la garantía de que el algoritmo (y sus métodos) están accesibles, pero después de la ejecución no es necesariamente cierto. En teoría la función de acierto no debe utilizar el algoritmo después de haber recogido los resultados.

El segundo método, `found`, es el empleado para recibir los aciertos y determinar si se debe o no seguir buscando. Toma como parámetro la posición del patrón en el texto y retorna un booleano, falso indica continuar buscando y verdadero detener la búsqueda.

Se derivan versiones de la clase para buscar todas las ocurrencias del patrón y para buscar una unidad con sentido (la primera o todas sus ocurrencias). Estas clases incluyen estructuras para almacenar las posiciones en las que ha habido un acierto.

En una versión preliminar de esta clase la función `found` podía editar el texto, pero esto provocaba el reinicio total de los algoritmos, complicando el código innecesariamente, ya que pasándole a la función `find_in(txt, txt_off)` el segundo parámetro (desplazamiento en el texto) reiniciamos la búsqueda en el punto que queramos sin repetir el preproceso.

De hecho, en la antigua versión, si se quería seguir buscando se debía indicar con cuidado desde dónde continuar la búsqueda, y en muchos casos se plantean problemas. En función de las operaciones realizadas sobre el texto se debía reiniciar en distintos puntos:

- Si el patrón es eliminado continuamos desde la posición en la que se encontró el acierto. Por ejemplo, si buscamos el patrón `bab` en el texto `bababoom` encontramos `[*bab*]aboom` y después de la eliminación continuaríamos con `[a]boom`. En este caso no habríamos eliminado el segundo `bab`, ya que la primera eliminación hace desaparecer la `b`. Este comportamiento puede ser el correcto o no, dependiendo del tipo de aplicación.

- Si el patrón es reemplazado continuaremos desde la antigua posición del acierto sumándole el tamaño del nuevo patrón (lo que deja la continuación de la búsqueda después de este). Por ejemplo, si queremos reemplazar el patrón `bab` por `ex` en el texto `bababoom` encontramos `[*bab*]aboom` y después del reemplazo continuamos con `ex[a]boom`. Aquí se plantearían dos problemas, uno similar al mencionado antes (el segundo `bab` desaparece antes de poder reemplazarlo) y otro en el caso de que el nuevo patrón fuera igual al antiguo (se podrían perder reemplazos). El segundo caso se daría en el ejemplo: después de reemplazar `bab` por sí mismo nos encontraríamos en la situación `bab[a]boom`, perdiendo el acierto `ba[*bab*]oom`. Aunque esto no es importante para el reemplazo, hemos perdido un acierto. La alternativa sería continuar buscando después del primer símbolo del nuevo patrón, pero esto es mucho peor, ya que podría causar reemplazos infinitamente recursivos.

- Si se añade un nuevo patrón como prefijo continuaremos desde la antigua posición de acierto sumándole el tamaño del nuevo patrón. Ejemplo: siguiendo con el patrón, texto y nuevo patrón anteriores, encontramos `[*bab*]aboom` y después de la inserción continuamos con `exb[a]baboom` y si seguimos llegamos a `exba[*bab*]oom`, que despues de insertar nos deja con `exbaexb[a]boom`. Este es el único caso que no causa problemas.

- Si se añade un nuevo patrón como sufijo continuamos a partir de la antigua posición de acierto más el tamaño del patrón buscado mas el tamaño del nuevo patrón. Ejemplo: encontramos `[*bab*]aboom` y después de la inserción continuamos con `babex[a]boom`. Como se puede observar, esta inserción también pierde aciertos (de nuevo se pierde el segundo `bab`) y la única solución es insertar los sufijos despues de haber encontrado todos los aciertos.

Por último indicaremos que junto a las funciones de acierto incluidas en la biblioteca se definen funciones de adaptación que toman como parámetros el algoritmo de búsqueda, el patrón y el texto y retornan la función de acierto. Su utilidad está en que indican claramente que hace la función de acierto, pero no son realmente necesarias.

3.3.2. Unidad

Clase que sirve para definir una unidad con algún significado dentro de una cadena: palabra, frase, etc. Se definen tres clases, una clase base abstracta (`atoken`) y dos clases derivadas (`actoken` y `adtoken`).

La clase base define operaciones para encontrar el principio o el final de una unidad a partir de un punto (o ambos) y para verificar si un rango del texto es una unidad. Las operaciones se definen en función de métodos que indican si un símbolo es un delimitador, separando entre delimitadores anteriores y posteriores. La detección de delimitadores se implementa en las clases derivadas.

La clase `actoken` define las unidades por su contenido, es decir, un rango es una unidad si todos sus símbolos pertenecen al conjunto de símbolos válidos. Ese conjunto se define al declarar variables (en el constructor), y no puede ser modificado.

La clase `adtoken` utiliza delimitadores para detectar las unidades, empleando dos conjuntos de símbolos, los que podemos encontrar antes y despues del elemento. Los conjuntos se definen al construir un objeto de la clase y no se pueden modificar.

La determinación de si un rango es o no una unidad se hace comprobando únicamente que los símbolos contenidos en él no son delimitadores, si deseamos verificar que está bien delimitado deberemos invocar el método `is_predelim()` para el símbolo que precede al rango y el método `is_postdelim()` para el que está inmediatamente despues de él.

El empleo de dos métodos de definición de unidades permite mayor flexibilidad a la hora de utilizarlos, ya que ambos métodos no son simétricos, aunque las diferencias son sutiles. Por ejemplo, la versión que emplea delimitadores anteriores y posteriores podría obtener, a partir de un símbolo, una unidad que contuviera símbolos delimitadores finales que no pertenecen al conjunto de delimitadores iniciales y viceversa, ya que al buscar el principio de una unidad sólo miramos que los símbolos no sean delimitadores iniciales y al buscar el final que no sean delimitadores finales. Si los conjuntos de inicio y final son iguales ambas clases son simétricas (siempre que el conjunto de símbolos válidos sea el de todos los símbolos que no son delimitadores).

3.3.3. Análisis temporal

Para el análisis temporal se define una clase cronómetro denominada **stopwatch**. Sólo se definen las operaciones de inicio y paro del cronómetro y un método que nos da la resolución como el número de valores que se pueden representar en un segundo. Se define como clase para poder implementar distintas versiones según el sistema operativo sin tener que modificar la interfaz y declarar variables de tipo cronómetro, lo que permite lanzar una temporización aunque ya haya otra en marcha.

La clase incluida en la biblioteca define versiones especiales del cronómetro para los sistemas operativos **MacOS** y **DOS** (usando el compilador `djgpp`, de libre distribución)[1]. En ambos casos se consiguen resoluciones del orden de microsegundos. La resolución de la versión genérica depende del sistema operativo, el hardware empleado y la implementación de la biblioteca de C. En teoría, en los sistemas POSIX (como el Linux), la resolución es de microsegundos, pero en la práctica nos encontramos con resoluciones mucho menores (en la versión del Linux para arquitecturas Intel la resolución es de centésimas de segundo)

El cronómetro podía haberse incluido en la interfaz de otra clase (por ejemplo para medir los tiempos de ejecución de los algoritmos), pero es más razonable dejar que sean los programas los que la utilicen directamente, ya que eso nos da la máxima flexibilidad. Por ejemplo, podemos medir el tiempo de búsqueda con o sin preproceso sin ninguna dificultad, mientras que la integración en la clase supondría definir métodos explícitos para ambos casos.

3.3.4. Pasos del algoritmo

Para ayudar al análisis se introduce el concepto de *paso de un algoritmo* que definiremos como el «conjunto de instrucciones que debemos ejecutar para modificar el estado de un algoritmo, entendiendo como estado el conjunto de valores significativos de sus variables internas en un momento dado». Distinguiremos distintos tipos de pasos en función de los valores que hayan sido modificados.

Por ejemplo, en el caso de los algoritmos de búsqueda, vamos recorriendo el texto de manera que, en un momento dado, nos encontramos en una posición cualquiera de éste. Según el algoritmo del que se trate, realizamos alguna operación relacionada con el símbolo actual (lo comparamos con otro del patrón, usamos su valor para acceder a una tabla o modificamos una variable relacionada con él) y pasamos a visitar otro símbolo (que vendrá dado por los resultados obtenidos en la operación). Con la definición

[1]También se definió una versión especial para **Windows**, pero como no se ha probado ha sido eliminada.

anterior, cada vez que nos desplacemos en el texto habremos ejecutado un *paso* del algoritmo. El ejemplo anterior sugería que lo único que se modifica en cada paso es la posición en el texto (y quizás alguna variable relacionada con él), si además modificamos nuestra posición en el patrón o alguna condición nos indica que el estado actual tiene una propiedad especial (como por ejemplo que hemos encontrado un acierto) también habremos ejecutado un paso, pero de un *tipo* distinto.

Esta definición de paso es útil para la representación gráfica de los algoritmos y para relacionar su ejecución real con el coste temporal teórico. Para utilizar los pasos definimos puntos de ruptura dentro del código, que no son más que llamadas a una función que toma como parámetro el tipo de paso que hemos completado. La idea de los puntos de ruptura es similar a la empleada en los depuradores de programas (*debuggers*) disponibles en la mayoría de entornos de programación. La diferencia está en que los depuradores nos permiten definir los puntos mientras se ejecuta nuestro programa y nuestro modelo los define dentro del código, en tiempo de compilación.

En cuanto a la relación entre la ejecución real y el coste temporal teórico tenemos que señalar que, si contamos el número de veces que se atraviesa un punto de ruptura durante una ejecución del algoritmo, estamos calculando el número de operaciones de modo similar a lo que se hace para obtener el coste asintótico en función del tamaño de los valores de entrada. De hecho, los resultados nos pueden ayudar a verificar si los cálculos asintóticos se corresponden con los resultados reales, aunque siempre pensando que trabajamos con una entrada real y por tanto no consideramos el cálculo en el caso promedio, sólo el de la entrada actual. Para ver el comportamiento en el caso promedio deberemos diseñar un conjunto de casos de prueba y obtener estadísticas a partir de los resultados obtenidos empíricamente. Para el peor y el mejor caso bastará ejecutar los algoritmos con la entrada adecuada.

Además, los distintos puntos de ruptura suelen corresponder con los distintos bloques estudiados para obtener los costes teóricos. Por ejemplo, en los algoritmos de búsqueda suele haber un coste directamente relacionado con el recorrido del texto y otro con las operaciones realizadas en caso de un acierto parcial. En general el cálculo asintótico del primer coste se corresponde con el número de veces que pasamos por el punto de ruptura que nos indica una actualización de nuestra posición en el texto y el segundo con el número de pasos que ejecutamos comparando distintas posiciones del patrón con el texto manteniendo la posición relativa del primero con el segundo, aunque esto no siempre sea así.

Por otro lado, el calculo del número de pasos nos sirve para valorar la relación entre coste asintótico y coste real, ya que no todos los pasos tienen el mismo coste en ciclos de procesador (en realidad al hablar de coste de orden n estamos obviando una constante que nos indique el coste de las operaciones en cada paso). Por ejemplo, en el caso de los algoritmos de búsqueda, los costes en número de pasos del algoritmo de fuerza bruta suelen ser superiores a los del algoritmo de Karp-Rabin (aunque son del mismo orden), pero en la práctica el primero es mucho más rápido que el segundo, ya que sus pasos sólo incluyen comparaciones de caracteres e incrementos de variables, mientras que el segundo realiza en cada paso varias operaciones aritméticas que necesitan ciclos extra de CPU.

Para contabilizar los pasos que se ejecutan en cada algoritmo se define una clase (`stepacct`) que almacena un vector de enteros que representan el número de pasos a través de cada punto de ruptura. El número y nombre de los puntos de ruptura se pasan como parámetros en el constructor.

Se definen métodos para acceder, borrar e incrementar los valores de cada punto de ruptura individual y para poner a cero todos los contadores e imprimirlos (indicando sus nombres).

Se emplea desde `skm_algo` para llevar la cuenta de pasos. En realidad la función que cumple esta clase podría haberse integrado en `skm_algo`, pero implementándola de manera independiente podemos reutilizarla si definimos otro tipo de algoritmos.

Es responsabilidad de la implementación de cada algoritmo definir que llamadas al dispositivo de análisis se deben realizar al detenerse en un punto de ruptura concreto, asi como la inserción de esos mismos puntos en las posiciones adecuadas dentro del código del algoritmo.

3.3.5. Análisis Gráfico

Para el análisis gráfico se define una clase denominada `adev` que incluye métodos para imprimir la salida de los pasos del algoritmo. En realidad la clase es abstracta y sólo sirve para definir las distintas operaciones de impresión, debemos derivar clases que los implementen dependiendo del formato (texto simple, texto html, etc.) y el sistema de salida que vayamos a emplear (streams, ventanas, etc.).

La biblioteca incluye un dispositivo de análisis que trabaja con la entrada salida estándar (`cin/cout`), que es el empleado por el programa de prueba, y otro que utiliza la biblioteca `curses` para terminales Unix (y que solo se compila si nuestro sistema dispone de las cabeceras y bibliotecas necesarias).

En la clase se define la salida del patrón y el texto marcando las posiciones coincidentes y el símbolo actual en cada paso y la salida de los tipos de datos empleados en el preproceso del modo más genérico posible. Se pueden imprimir valores simples enteros, desplazamientos en el texto o el patrón y cadenas de caracteres (que es en lo que se deben de convertir los tipos de datos no disponibles, como sucede con las máscaras de bits). En cuanto a las tablas de preproceso se definen métodos para imprimir vectores de enteros, desplazamientos y cadenas asociados al alfabeto o al patrón. La pega es que si tenemo un tipo nuevo con formato de vector debemos transformar todos sus elementos a cadenas para poder imprimirlos.

Para poder mostrar el estado de la búsqueda se define la clase auxiliar `adev_mpatt`, que almacena la posición actual en el patrón y los símbolos acertados. Además, define métodos para actualizar la posición actual y marcar rangos de símbolos acertados. La actualización de la posición en el patrón anula las informaciones sobre aciertos anteriores, permitiendo la reutilización de objetos del tipo sin tener que borrar los aciertos anteriores explícitamente.

Por último se introducen dos herramientas de interacción con el usuario, los retardos y las funciones de obtención de caracteres. Lo que se hace es incluir en la clase de análisis vectores asociados a los puntos de ruptura para saber si debemos detenernos y solicitar la entrada de un carácter al usuario despues de mostrar un paso.

Para los retardos nos limitamos a definir un vector de enteros del tamaño del número de puntos de ruptura del algoritmo. Este vector será empleado para detener la ejecución del algoritmo tantos milisegundos como haya almacenados en la posición correspondiente al punto de ruptura actual.

La función de obtener un caracter se define como elemento independiente, lo que nos permite definir clases derivadas que procesen de alguna manera la entrada del usuario. Su operación elemental toma como entrada un caracter y retorna un booleano que índica si debemos detener la búsqueda o no. El caracter de entrada debe serle proporcionado por el dispositivo de análisis.

Estas funciones se llaman desde el método `algoritm_step` de la clase `skm_algo`. Cada versión del algoritmo utilizará las funciones adecuadas dependiendo de las estructuras de preproceso que emplee y el punto de ruptura desde el que haya sido invocado. La entrada/salida de cada paso se invoca automáticamente desde `skm_algo` cada vez que se informa de un paso del algoritmo.

3.4. Algoritmos de búsqueda simple

Ya mencionamos en la introducción del capítulo anterior que todos los algoritmos de búsqueda se derivan de la clase `skm_algo`. Esta clase es fundamental ya que, salvo el acceso al nombre y al identificador de cada algoritmo, todos los métodos de búsqueda implementados son invocados desde ella.

En esta sección comentaremos los distintos atributos y métodos de la clase en función de su propósito y nivel de acceso; dentro de los métodos públicos hablaremos de los de acceso a la lectura de variables, los de ajuste de parámetros y los de búsqueda, mientras que en los privados separaremos entre las utilidades internas y los métodos abstractos que debe definir cada algoritmo de búsqueda. Para terminar comentaremos algunos detalles sobre la codificación y compilación que son aplicables a todos los algoritmos implementados.

En los apendices se puede encontrar el código completo de la clase.

3.4.1. Construcción y destrucción de objetos

El constructor toma como parámetro un alfabeto, si no se pasa ninguno utiliza el alfabeto por defecto (para caracteres la identidad y para otro tipo de símbolos un alfabeto vacío). Además, todos los parámetros de análisis toman valores nulos, si se desea utilizarlos se deben activar después de construir el objeto.

El algoritmo no crea ninguna variable dinámica, de manera que el destructor no hace nada.

La razón de incluir un constructor por defecto es que cada clase derivada define una variable con el mismo nombre que el algoritmo que implementa, de manera que un usuario puede emplear el objeto como si se tratase de una función.

3.4.2. Métodos públicos de sólo lectura

La clase define una serie de métodos para leer valores almacenados en ella, ya que nuestra biblioteca se emplea para analizar los algoritmos y resulta interesante poder acceder a información sobre ellos desde el exterior.

La clase `adm_algo` definía únicamente métodos abstractos que retornan cadenas de caracteres indicando el tipo, nombre e identificador de cada algoritmo. Dado que la clase `skm_algo` define un tipo de algoritmos, se define el método que nos da el tipo. Las clases derivadas deben implementar los que nos indican el identificador y el nombre.

Aunque los datos de entrada para la ejecución de las búsquedas los pasamos desde nuestro programa, para algunas de las clases de análisis resulta interesante poder acceder a ellos sin recibirlos como parámetro. Una solución sería declararlas como amigas pero, si añadimos una nueva clase de análisis, deberemos modificar también esta para indicarle su existencia. Para evitar la declaración de clases amigas se definen métodos públicos para leer los parámetros de entrada, lo que permite conocer que alfabeto, patrón, texto y desplazamiento inicial estamos utilizando o hemos utilizado en la última ejecución.

Como ya hemos comentado, existe una clase que nos permite saber cuantos pasos ha ejecutado un algoritmo. La clase `skm_algo` incluye un objeto contador de esa clase que sólo se utiliza si el usuario lo activa desde el exterior.

Para obtener información sobre el contador se incluye un método de lectura que nos devuelve una copia del objeto y otro que nos dice si está o no activado.

Por último, como no conocemos que tipos y variables se van a emplear para el preproceso y no hay forma de forzar a que los algoritmos definan una estructura que podamos emplear desde la base, se incluye un método para imprimir los datos del preproceso. Gracias a esta función podemos imprimir tanto los datos de entrada (accesibles directamente) como los generados en el preproceso sin tener que utilizar el dispositivo de análisis durante la búsqueda.

3.4.3. Métodos públicos de ajuste de parámetros

Se definen tan solo dos, uno que selecciona que alfabeto vamos a usar (para símbolos de un tipo distinto a `char` el alfabeto por defecto está vacío, si usamos el constructor por defecto para los algoritmos debemos tener una manera de reemplazarlo por uno útil) y otro que activa o desactiva el uso del contador.

3.4.4. Métodos públicos de ejecución de la búsqueda

Los métodos de búsqueda se implementan utilizando dos operaciones internas que son definidas en las clases derivadas: el preproceso y la búsqueda del patrón preprocesado.

Desde el exterior se definen métodos para acceder a ambas operaciones por separado (`prep` y `find_in`) y conjuntamente (`find`).

El preproceso toma como entrada el patrón y no retorna nada. El resultado de su ejecución queda almacenado en la clase. El método que busca el patrón preprocesado toma como parámetros el texto y, opcionalmente, un desplazamiento inicial desde el que comenzar a buscar y retorna la posición de la primera ocurrencia del patrón o el tamaño del texto si no es encontrado.

El método conjunto toma como parámetros el patrón, el texto y el desplazamiento y retorna la posición del acierto. En realidad lo que hace es llamar a las dos funciones anteriores en orden, preprocesa el patrón y lo busca en el texto.

Para poder emplear objetos de la clase como funciones se define el operador «paréntesis» o «llamada a función» (`operator()`) con los mismos parámetros y retorno que el método de búsqueda conjunto. En realidad la definición del operador es simplemente una llamada a la función compuesta.

El método de preproceso ajusta los atributos relacionados con el patrón, reinicializa la cuenta de pasos y llama al método interno de preproceso adecuado (utilizando la versión simple o la que incorpora soporte para el análisis).

Para utilizar la función de aciertos se definen versiones de los métodos de búsqueda que toman como primer parámetro un puntero a un objeto del tipo `mfun` y no retornan nada (los resultados se almacenan en el objeto). El resto de parámetros son iguales a los de las versiones simples. El puntero a la función de acierto es un atributo de la clase, pero una vez ejecutado el algoritmo no se debe usar, así que los métodos lo asignan antes de comenzar y lo anulan al terminar. De cualquier forma, las modificaciones del objeto están disponibles para el que llama a la función, ya que se modifica directamente el valor pasado a través del puntero.

3.4.5. Utilidades internas

Se incluyen varias utilidades para simplificar la actualización y utilización de los atributos desde los métodos internos. De hecho, gracias a algunos de ellos se podrían modificar las estructuras internas sin tener que tocar las clases derivadas. Se definen utilidades para ajustar variables internas relacionadas con el texto y el patrón y de acceso al alfabeto, la función de aciertos y el contador.

El texto y el patrón se emplean a través de variables internas, así que se definen métodos para inicializar o actualizar sus valores (tamaño y punteros de inicio y final). Estas funciones se emplean al inicializar una búsqueda.

El primer método de acceso se emplea para obtener el índice de un símbolo en el alfabeto. Aunque en la formulación actual el alfabeto siempre existe, si decidiéramos acceder a él desde un puntero o no utilizarlo no sería necesario alterar ninguna clase derivada. Por ejemplo, para los caracteres la identidad no necesita el alfabeto, se utiliza por no complicar las cosas, pero si no nos interesa utilizar alfabetos en ningún caso podemos modificar la función `aindex` para que retorne su entrada. En caso de no utilizar alfabeto es necesario que, al menos, se le de un valor a la variable interna que contiene su tamaño, ya que se utiliza como tamaño de algunas tablas de preproceso.

La función de aciertos puede estar definida o no (de hecho depende de como se llame a la función que ejecuta la búsqueda). Para aislar los algoritmos de este problema se define la función `report_match` que detiene el algoritmo si no hay función de acierto o la ejecuta y actualiza valores si la hay. Si por alguna razón nos interesase eliminar las funciones de acierto, bastaría con redefinir está función, las clases derivadas no tendrían que ser alteradas.

Por último se definen métodos para indicar que se ha ejecutado uno o varios pasos del algoritmo. La función definida, `report_step()`, actualiza el contador y
o ejecuta la función asociada al dispositivo de análisis. Además, si estamos utilizando un dispositivo de

análisis, se encarga de llamar a la función que recibe entrada del usuario despues de ejecutar la función del algoritmo que le indica al dispositivo los datos que debe mostrar. Con el retorno de esta función se actualiza el valor de la variable interna `stop` que los algoritmos utilizan para saber si deben continuar buscando despues de ejecutar cada paso del algoritmo.

3.4.6. Métodos abstractos

En primer lugar, por la herencia de la clase `adm_algo`, cada implementación debe definir como métodos públicos las funciones que nos dan el nombre y el identificador del algoritmo.

El resto de métodos abstractos son privados y se emplean para preprocesar el patrón, buscar el patrón preprocesado o indicarle al dispositivo de análisis que sucede en nuestro algoritmo.

Ninguno de los métodos de búsqueda recibe parámetros, lo que llega del exterior es verificado y copiado en atributos internos, por lo que está disponible nada más comenzar la ejecución del algoritmo. En cuanto a las estructuras de preproceso, se definen como atributos en la clase derivada y se inicializan al ejecutar el preproceso. Los métodos de la clase base se encargan de garantizar que el preproceso se ejecuta antes de llamar a la función de búsqueda.

El único método que toma parámetros es el que informa del paso por un punto de ruptura. Recibe dos valores, uno que le indica qué punto de ruptura hemos atravesado y otro que le da el número de veces que lo hemos hecho. El segundo valor es igual a uno por defecto, pero se ha incluido porque algunas operaciones que ejecutan varios pasos (como la iniciación de los valores de un vector) no las realiza nuestro código, se ejecutan a través de llamadas a funciones o métodos de biblioteca que no podemos modificar.

Los métodos que debe definir cada algoritmo serán:

- `preprocess()`, preproceso del patrón, accede al patrón a través de las variables internas y calcula los valores de las estructuras de preproceso. No utiliza el texto, si hay operaciones de preproceso que utilizan el texto se hacen dentro del método de búsqueda. Si el algoritmo no tiene preproceso se define vacía.
- `run_preprocessor()`, es la misma función de preproceso con puntos de ruptura.
- `match()`, función de acierto, busca el patrón en el texto. Asume que todos los valores que dependen de la entrada y el preproceso están inicializados. Cuando encuentra un acierto llama al método report_match y en función de su retorno reinicia la búsqueda o termina.
- `run_matcher()`, igual que la función anterior incluyendo puntos de ruptura.
- `algorithm_step(break_point, int)`, llama a las funciones del dispositivo de análisis adecuadas según el punto de ruptura recibido. Se llama desde `report_step()` cuando el usuario a solicitado el análisis y ha proporcionado un dispositivo. Las operaciones de cuenta de pasos se incluyen en el método `report_step` y por tanto no deben ser definidas en las clases derivadas.

3.4.7. Notas sobre la codificación de los algoritmos

Los métodos de preproceso y búsqueda (`preprocessor()` y `matcher()`) se definen en ficheros con la extensión `.cpp` que también se emplean para compilar las versiones de análisis (`run_preprocessor()` y `run_matcher()`) utilizando macros. Lo que se hace es incluir dos veces el fichero `.cpp` en el archivo que implementa los métodos de la clase, definiendo en cada caso las macros adecuadas. De ese modo ambas versiones son siempre iguales y disponemos de una función *eficiente* sin tener que utilizar condicionales en tiempo de ejecución, aunque el código de la clase es más grande.

Todos los métodos de búsqueda definen la recuperación tras un acierto. Dependiendo del tipo de algoritmo se reinicia totalmente o se hace uso de los resultados anteriores. Los sistemas de recuperación se comentan en la descripción de cada algoritmo.

Como ya se ha dicho, los métodos de búsqueda emplean punteros para el acceso al texto y al patrón, evitando el uso de índices siempre que es posible, lo que nos ahorra una suma para cada acceso a los elementos de un vector.

En todas las implementaciones se aplican optimizaciones triviales, evitando depender de la inteligencia del compilador. Fundamentalmente consisten en el uso de variables locales (a ser posible almacenadas en registro) para el acceso a elementos de vectores que son utilizados varias veces, como por ejemplo el primer o último carácter del patrón.

Hay que señalar que, para los métodos de preproceso y búsqueda que no incorporan análisis (`preprocessor()` y `matcher()`), se redeclaran algunos de los atributos de la clase como variables locales en registro (como por ejemplo el iterador en el texto, que se usa en todos los algoritmos), ya que no se va acceder a ellas durante la ejecución. Esta optimización anula en parte el funcionamiento como clase, pero se hace así porque no se puede declarar un atributo para que se almacene en registro y el compilador no puede optimizar como nosotros (no sabe que el valor del atributo no va a ser usado desde otro método y por tanto no lo guarda en registro). Para algoritmos como el de la fuerza bruta el aumento de velocidad es considerable.

Respecto a la compilación hay que indicar que muchas funciones y métodos de se declaran como `inline`, por lo que es imprescindible utilizar las opciones adecuadas para que el compilador no generen llamadas a función, lo que empeora sensiblemente el rendimiento (para el `gcc` esto implica utilizar la opción `-O2` o `-O3`).

3.5. El programa de análisis

El programa de ejemplo se ha escrito para probar los algorítmos y utilizar las distintas opciones de la biblioteca. Está pensado para su uso en terminales, siguiendo el modelo de la mayoría de los comandos del sistema UNIX, el ajuste de opciones y la selección del patrón y el texto se hacen en la línea de comandos.

El formato para invocarlo es `skmtest [OPCIONES] [PATRÓN] TEXTO`, donde `PATRÓN` es la cadena a buscar (si contiene espacios en blanco se debe pasar entre comillas), `TEXTO` corresponde a un fichero que contiene el texto en el que se va a buscar el patrón y las `OPCIONES` pueden ser:

`-a, --algorithm=[algoritmo]`
: Permite seleccionar el algoritmo a utilizar, si no se da el nombre del algoritmo obtenemos una lista de valores posibles. Para utilizar más de un algoritmo se pueden incluir varias opciones `-a` en la línea de comandos. Si no se incluye esta opción se ejecutan todos los algoritmos.

`-A, --alphabet=[cadena]`
: Hace que los algoritmos utilicen como alfabeto el conjunto de caracteres incluidos en la cadena pasada como parámetro. Si se pasa este parámetro el patrón debe contener sólo símbolos alfabéticos.

`-b, --breakpoints=[cadena]`
: Indica en que puntos de ruptura debe detenerse el programa para preguntarle al usuaro si debemos continuar. Esta opción sólo es efectiva cuando el algoritmo se invoca con las opciones -i y -s activadas.

`-c, --compact`
: Muestra una salida más compacta para la medida de tiempos y el recuento de pasos. Los datos de cada algoritmo estan en una línea que contiene el identificador del algoritmo, el tiempo de ejecución y las cuentas de pasos separadas por comas. Los campos se separan usando ':'

`-d, --delays=[cadena]`
Indica en que puntos de ruptura debe esperar un programa un determinado lapso de tiempo (por defecto 1 segundo) cuando se ejecuta el programa con las opciones -i y -s activadas.

`-D, --delay-time=[ms]`
Indica el tiempo (en milisegundos) que debe durar la pausa en los puntos de ruptura marcados con -d. La opción sólo es efectiva si se ejecuta el programa con las oprciones -i y -s activadas.

`-f, --find-first`
Le indica al programa que sólo búsque la primera ocurrencia del patrón en el texto (por defecto las busca todas).

`-h, --help`
Imprime un mensaje con las opciones

`-i, --interactive`
Aplica retardos a todos los pasos de los algoritmos y se detiene para recibir entrada del usuario cuando nos encontramos en un acierto parcial. Sólo es útil si se emplea junto con -s

`-n, --no-prep`
Preprocesa el patrón antes de la búsqueda. Esta opción es útil para medir tiempos, ya que si buscamos más de una vez el patrón (opción `-T`) el preproceso sólo se ejecuta una vez y no se calcula su tiempo de ejecución.

`-p, --patterns[=ARCHIVO]`
Lee los patrones de un archivo, cada uno es una línea. Si se usa esta opción no se debe pasar un patrón en la línea de comandos, sólo el texto.

`-r, --report`
Imprime los pasos ejecutados en cada algoritmo

`-s, --show`
Muestra la ejecución de los algoritmos.

`-t, --time`
Calcula el tiempo que tarda en ejecutarse cada algoritmo

`-T, --text-size[=TAM]`
Ejecuta cada algoritmo varias veces para simular que el texto tiene un tamaño de TAM bytes. Esta opción sólo se emplea junto a -t

`-v, --version`
Imprime la versión del programa de prueba

Respecto al código del programa hay poco que decir, está escrito siguiendo un modelo imperativo de programación, definiendo unas pocas funciones auxiliares y el bloque principal del programa. Lo primero que hace es crear un vector con los algoritmos de búsqueda implementados e inicializar las variables auxiliares con los valores por defecto. Despues lee los parámetros de entrada usando la función `getopt_long` de GNU. Según las opciones leidas se asignan valores a variables o se generan mensajes de error. Cuando se han leido los parámetros y no se han producido errores se lee el patrón (excepto si se ha leido la opción `-p`, en cuyo caso tendremos los patrones almacenados en una cola) y el texto (almacenando todo el contenido del fichero como una sola cadena). Despues ajusta los parámetros que dependen del texto o el patrón y entra en el bloque principal, que ejecuta los algoritmos seleccionados para cada patrón. El código del programa se puede ver en los apendices.

Capítulo 4. Análisis y resultados experimentales

En este capítulo describiremos el análisis de los algoritmos implementados, comentando el diseño de los casos de prueba (elección de los textos y patrones de entrada, condiciones del análisis, etc.), los resultados obtenidos y las conclusiones que se derivan de ellos.

4.1. Diseño de los casos de prueba

Realizaremos dos tipos de análisis experimentales, uno para medir la eficiencia temporal de los algoritmos (Mb procesados por segundo) y otro para relacionar el coste asintótico con la ejecución (cuenta de pasos).

En [Dav86], [Bae89] y [Hum91] se pueden encontrar análisis experimentales de la eficiencia de los algoritmos de búsqueda de un solo patrón y en [Wat94] se analiza además la eficiencia de los algoritmos de búsqueda de múltiples patrones.

4.1.1. Textos de prueba

Para las pruebas se seleccionaron tres textos de características diferentes:

1. Ulysses de *James Joyce* (en inglés)[1]. Texto literario. Para las pruebas se empleó el texto resultante de concatenar por orden todos los capítulos (1.585.338 bytes).

2. Constitución española de 1978 (en castellano)[2]. Texto técnico. Para las pruebas se usó el texto tal cual se obtuvo del navegador (113.343 bytes).

3. Secuencia de ADN HC21-000020 del Cromosoma 21[3]. Cadena de ADN. Para las pruebas se transformó el texto para simplificar las búsquedas. Primero se agruparon los bloques de cada línea (cada una se divide en 6 bloques de 10 bases) eliminando los espacios en blanco y posteriormente se eliminaron los saltos de linea, con lo que se obtubo un texto de una sola línea con toda la secuencia de bases (1.296.826 bytes).

La elección de los textos se debe a su tamaño (dos relativamente grandes y otro más pequeño), el tipo de texto (literario, legal y cadena de ADN), al uso de alfabetos distintos (ASCII de 7 bits, ISO-8859-1 y bases 'A', 'C', 'G' y 'T' de ADN) y a su disponibilidad (todos se pueden obtener a traves de Internet).

Algunos de los estudios de la eficiencia de los algoritmos de búsqueda emplean textos sintéticos generados aleatoriamente o usando las propiedades estadísticas del tipo de texto ([Dav86], [Bae89]), pero nosotros hemos preferido hacerlo con textos reales que son los que se utilizan en la mayoría de aplicaciones.

4.1.2. Subcadenas de entrada

Para los dos primeros textos se han empleado dos conjuntos de subcadenas, uno tomando palabras del

[1]Obtenido de `ftp://blaze.trentu.ca/pub/jjoyce/ulysses/ascii_texts/ulys*.txt`.

[2]Obtenida en el URL `http://ccdis.dis.ulpgc.es/~secrdis/constitucion.txt`

[3]URL `http://www-eri.uchsc.edu/chr21/dna/HC21-000020.html`

propio texto y otro seleccionándolas del diccionario del sistema (`/usr/dict/english` para Ulysses y `/usr/dict/spanish` para la Constitución). En el caso de la secuencia de DNA se eligieron fragmentos de distintos tamaños de la misma secuencia.

Para los dos primeros textos las subcadenas se eligen seleccionando palabras agrupadas por tamaño [1]. Para la ejecución se seleccionan aleatoriamente 20 subcadenas *distintas* de cada grupo. En los casos en los que no hay 20 subcadenas diferentes del mismo tamaño se usan todas las disponibles.

Para la secuencia de ADN se eligieron aleatoriamente cinco subcadenas de 10, 25, 50, 75, 100, 250, 500 y 750 elementos, tomadas directamente del texto a buscar.

4.1.3. Entorno de las pruebas

Las pruebas se ejecutaron en un PC con un procesador Pentium a 150 Mhz y 40 Mb de RAM bajo el sistema operativo MS-DOS. Aunque el entorno de desarrollo del proyecto ha sido un sistema Debian GNU Linux, se opto por el MS-DOS para las pruebas por que para este sistema disponemos de funciones de cronómetro de mayor resolución y además, al tratarse de un sistema monoproceso, no hay otros procesos que puedan alterar la medida de los tiempos, algo que no podemos asegurar en los sistemas Unix.

Dado que la ejecución de los algoritmos es muy rápida, se simuló que cada texto tenía un tamaño de aproximadamente 20 Mb ejecutando cada búsqueda sobre el mismo texto tantas veces como sea necesario.

4.1.4. Parámetros del programa de prueba

Todas las pruebas se realizaron invocando el programa de prueba con las opciones: `-a ALGO -c -r -t -T20M -p PATT_FILE TEXTO`, donde `ALGO` es el algoritmo a estudiar (`bf`, `kr`, `kmp`, `so`, `bm`, `bmh` o `sqs`), `PATT_FILE` el fichero con las palabras a buscar y `TEXTO` el texto sobre el que se realiza la búsqueda. Para la secuencia de DNA se indicó además cual es el alfabeto utilzado con la opción `-A ACGT`, reduciendo el tamaño de las tablas de preproceso y los tiempos relacionados para los algoritmos que emplean el alfabeto.

4.2. Resultados de la ejecución

En este apartado presentamos tablas, para cada fichero de entrada, que representan la eficiencia temporal, la cuenta de pasos del preproceso y la cuenta de pasos de la búsqueda de cada uno de los algoritmos.

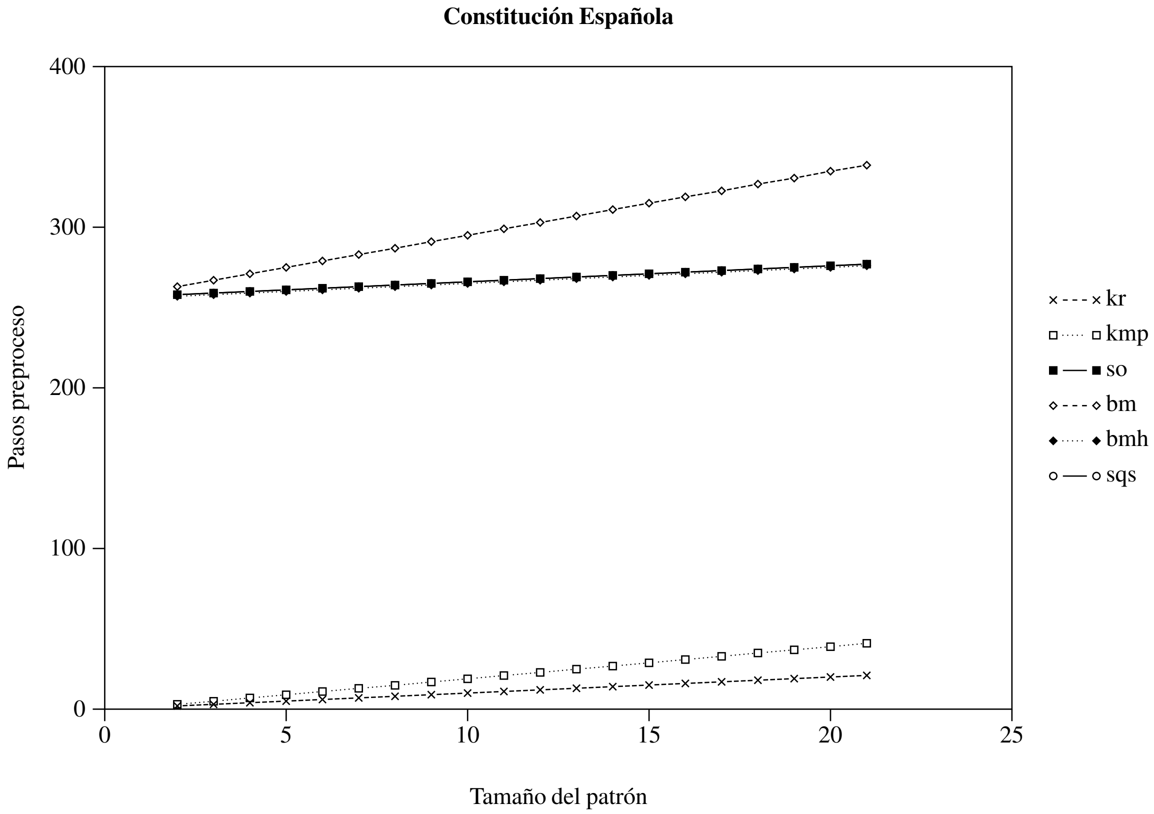
Las tablas de cuenta de pasos relacionan los costes asintóticos con los caracteres del texto y el patrón visitados realmente. En el preproceso usaremos los *pasos* del preprocesador, ya que todos ellos implican desplazamientos relacionados con el patrón y/o el alfabeto, mientras que para la búsqueda se usará la suma de los pasos del buscador y los invertidos en aciertos parciales.

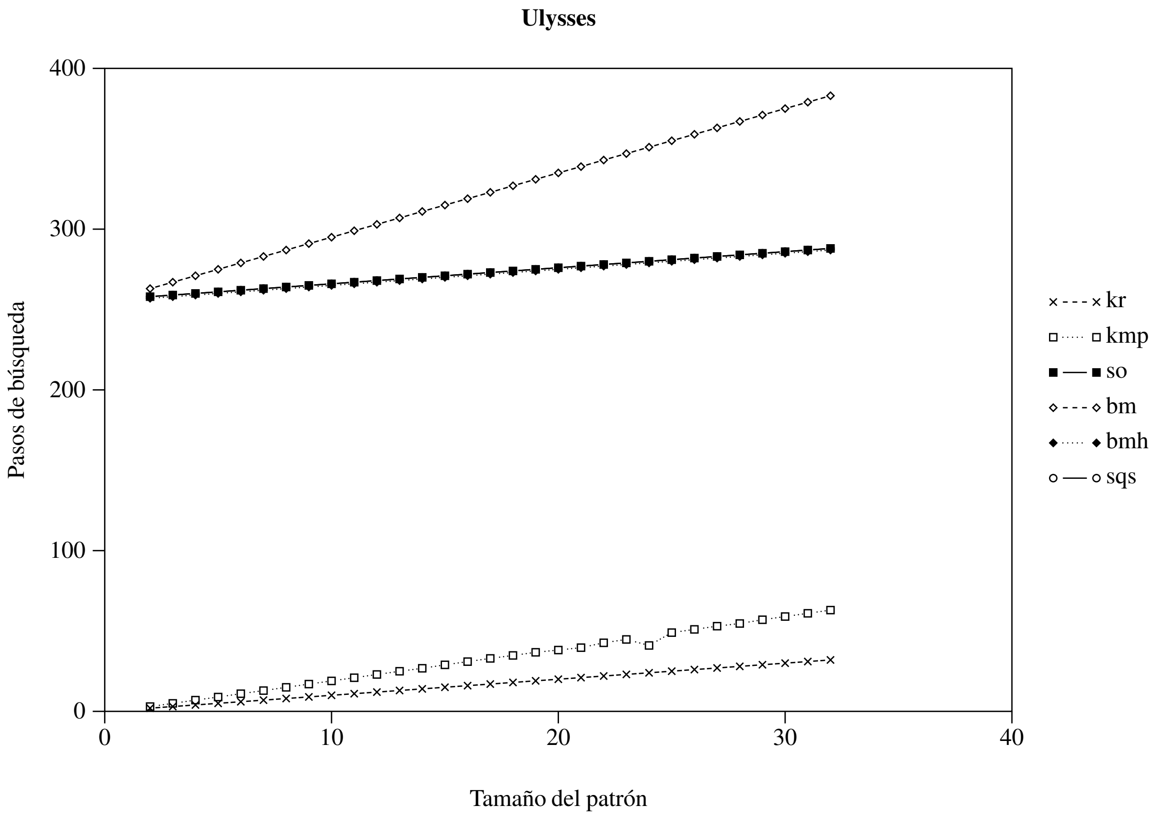
La eficiencia temporal se mide en Mb procesados por segundo, lo que permite comparar las tablas de los distintos textos directamente. Nótese que, con esta representación, los algoritmos más rápidos serán aquellos cuyos valores sean más altos, y no más pequeños como cuando comparamos usando directamente los tiempos de ejecución.

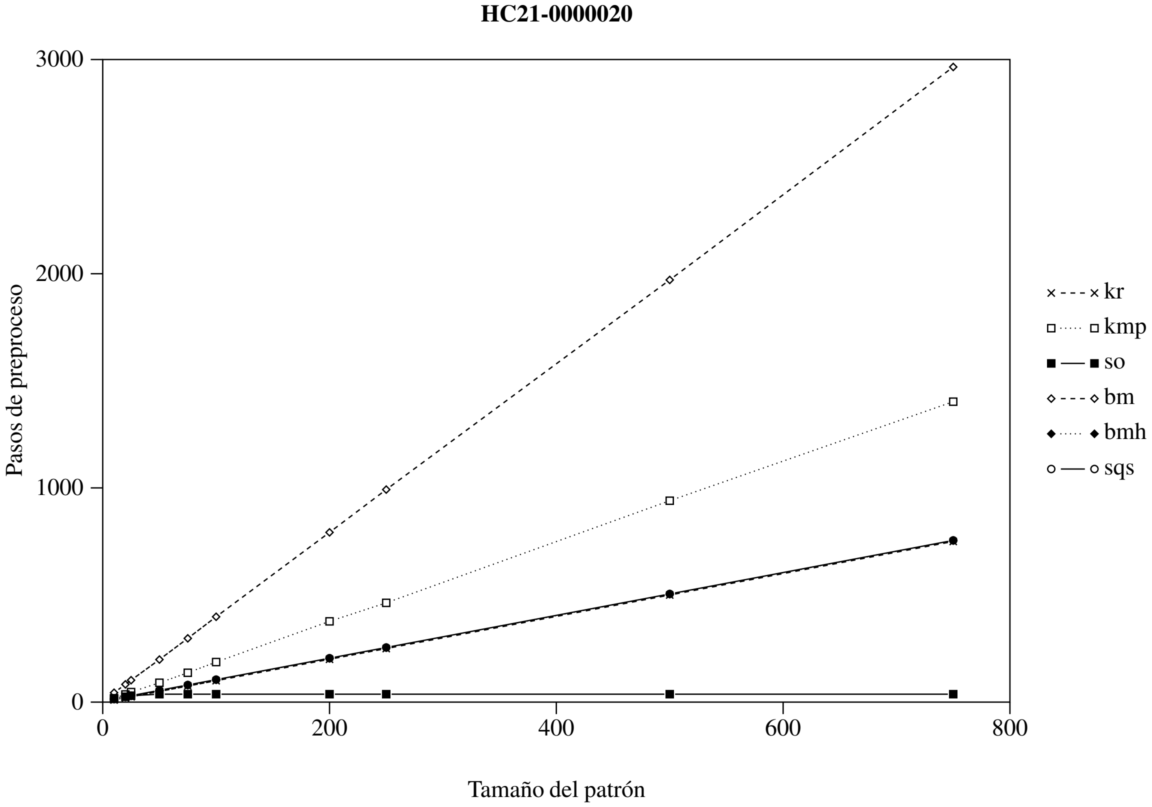
En la medida de tiempos se incluye el preproceso, aunque su influencia es mínima, ya que trabajamos con textos grándes ($n > 100$Kb) y patrones pequeños ($m < 32$). En el caso de la secuencia de ADN los patrones son más largos (hasta 750 bytes), pero tambien lo es el texto (más de 1Mb).

[1]Las palabras se definen como subcadenas que no contienen espacios (entendiendo como tales los que retornan verdadero al ejecutar la función `isspace()` de C++), lo que ocasiona que encontremos subcadenas que contienen símbolos de puntuación y números al seleccionar palabras de los textos (en los diccionarios también podría pasar, pero no hay ninguna puntuación).

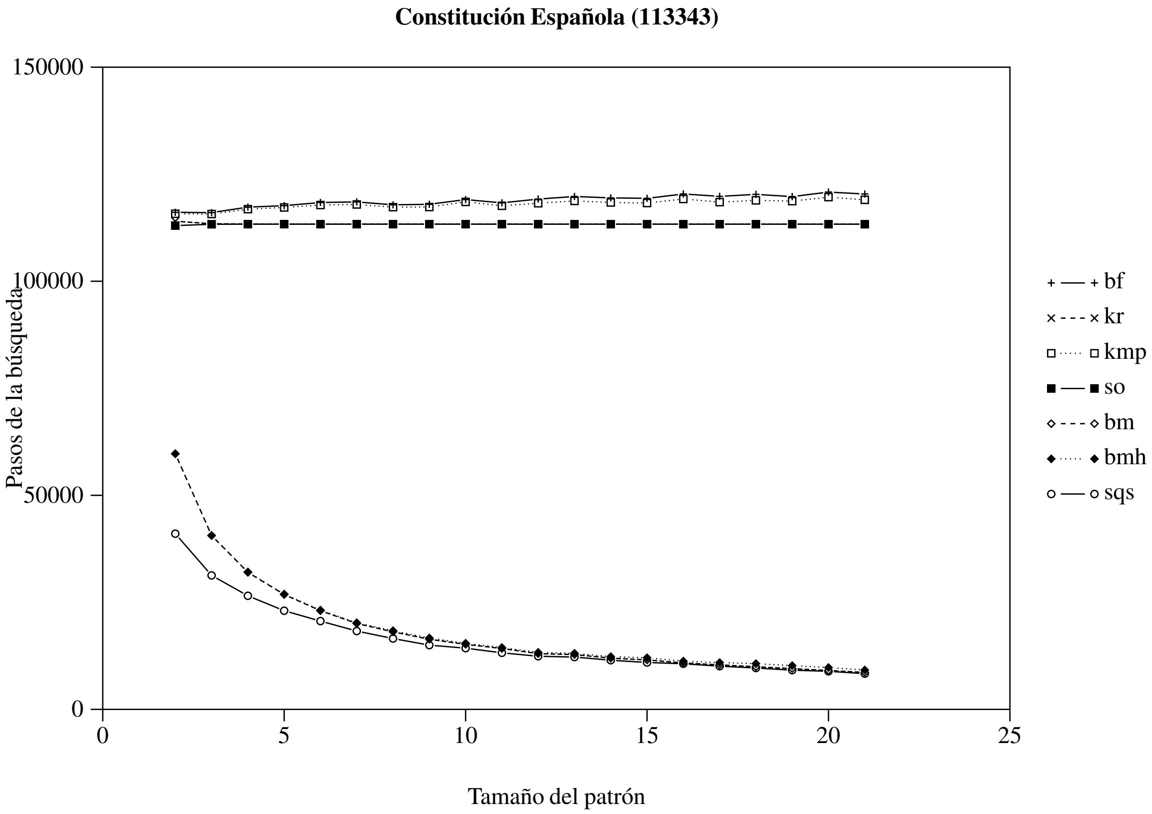
4.2.1. Cuenta de pasos (preproceso)

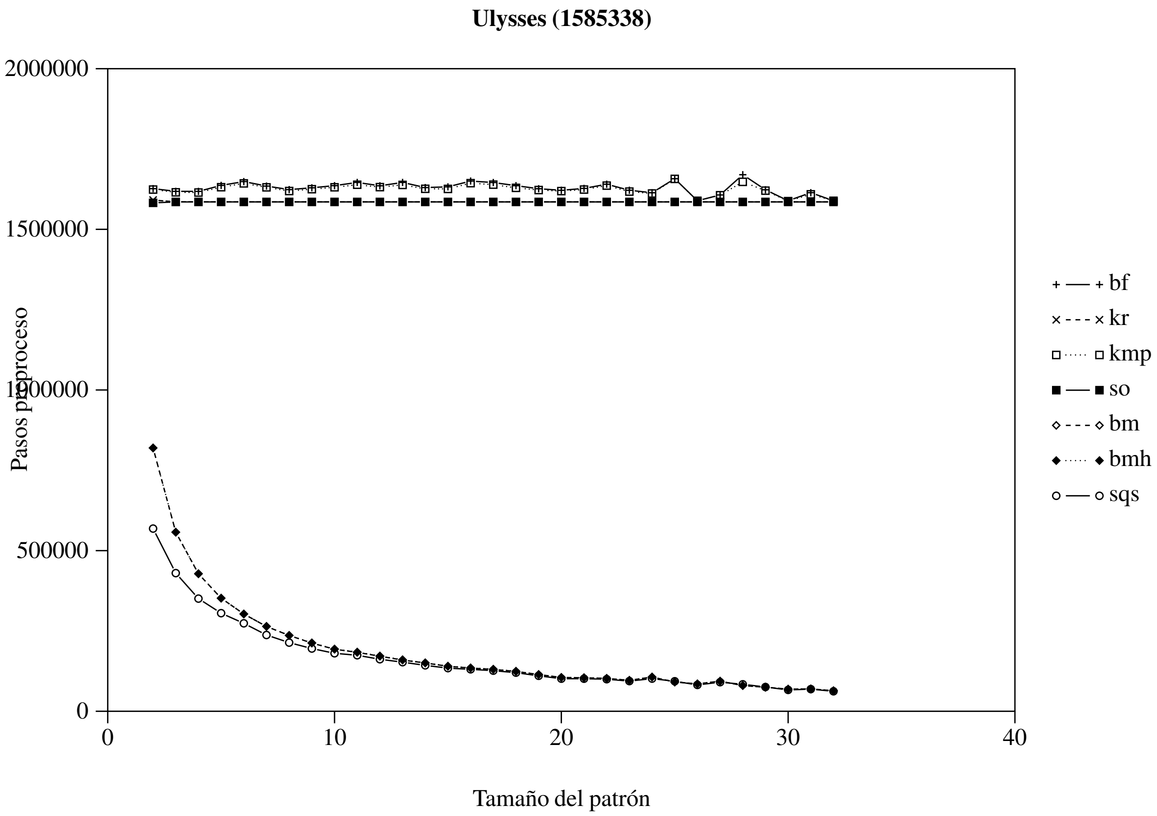


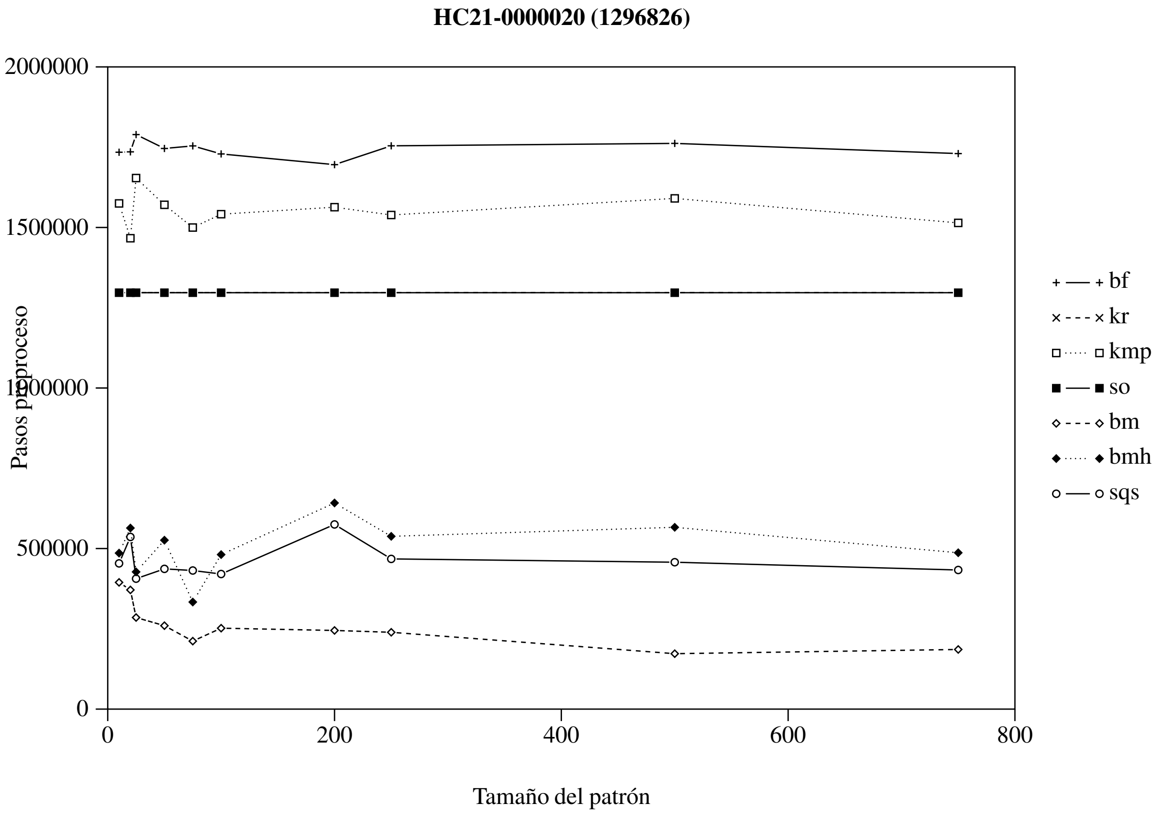




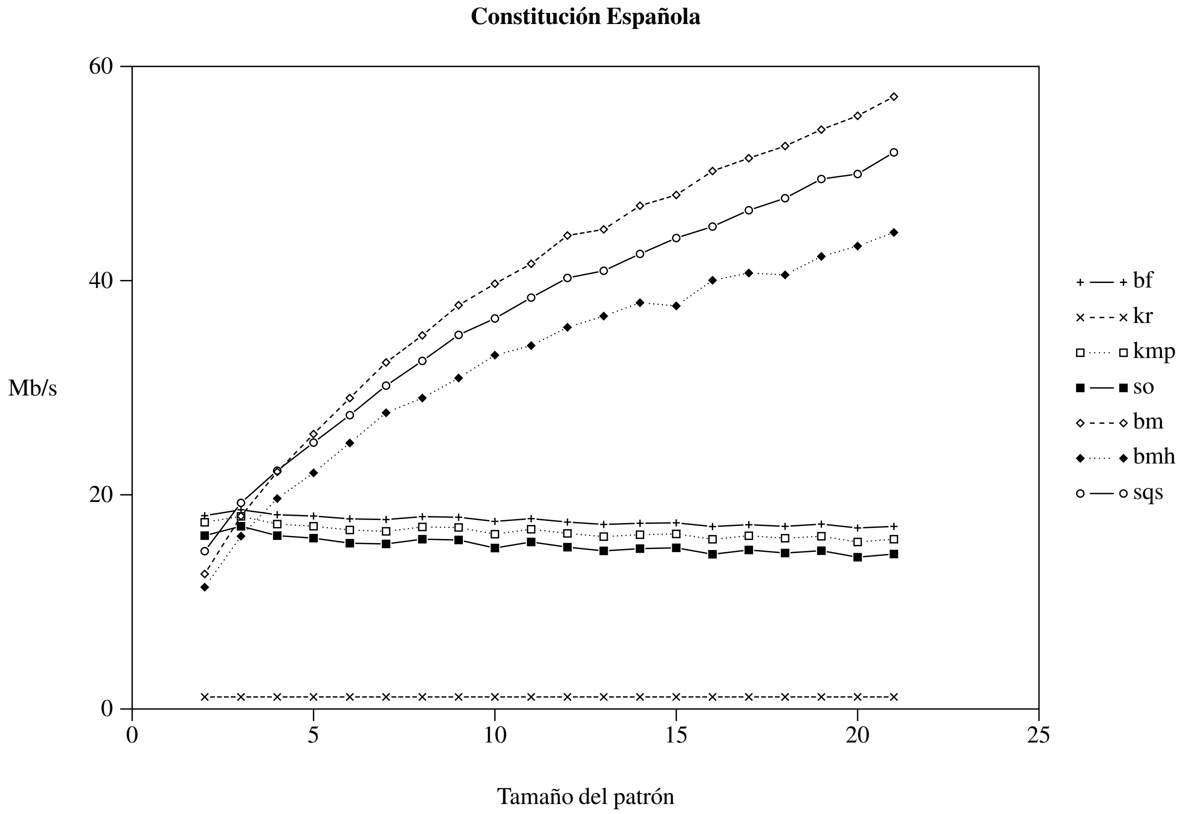
4.2.2. Cuenta de pasos (búsqueda)

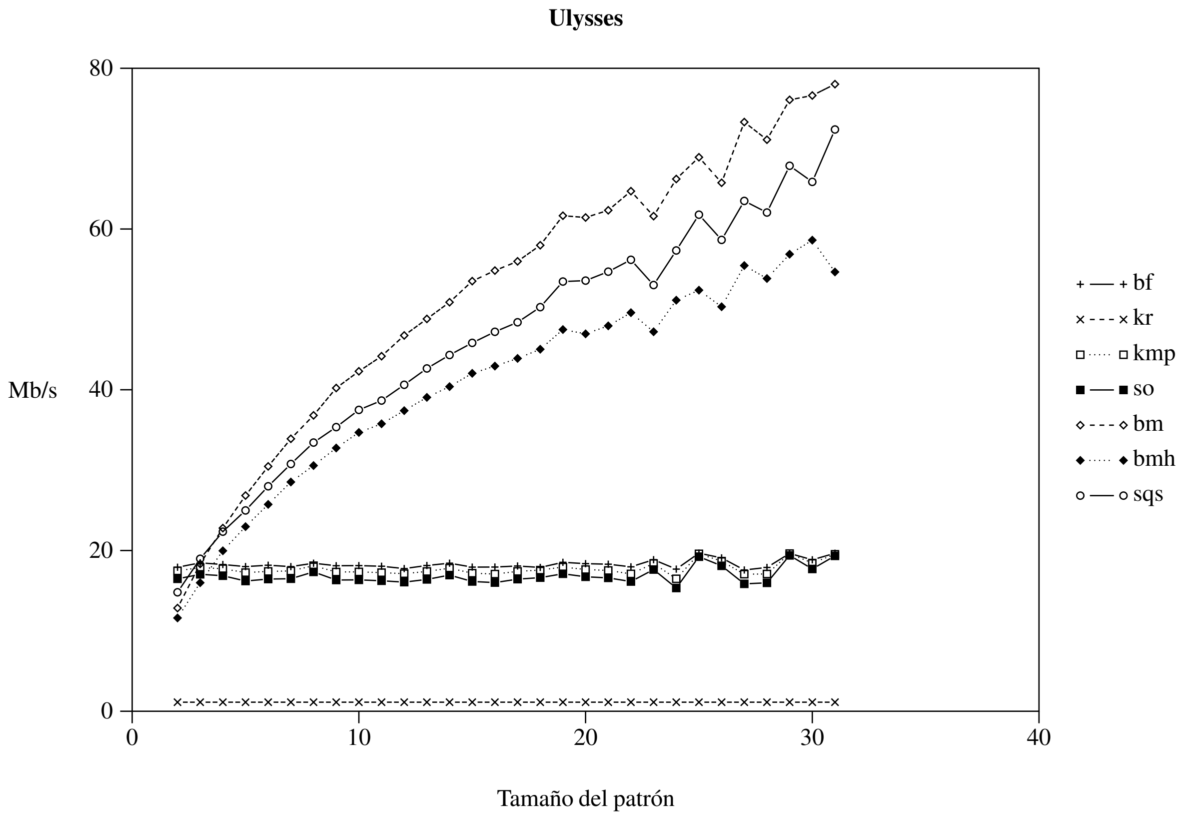


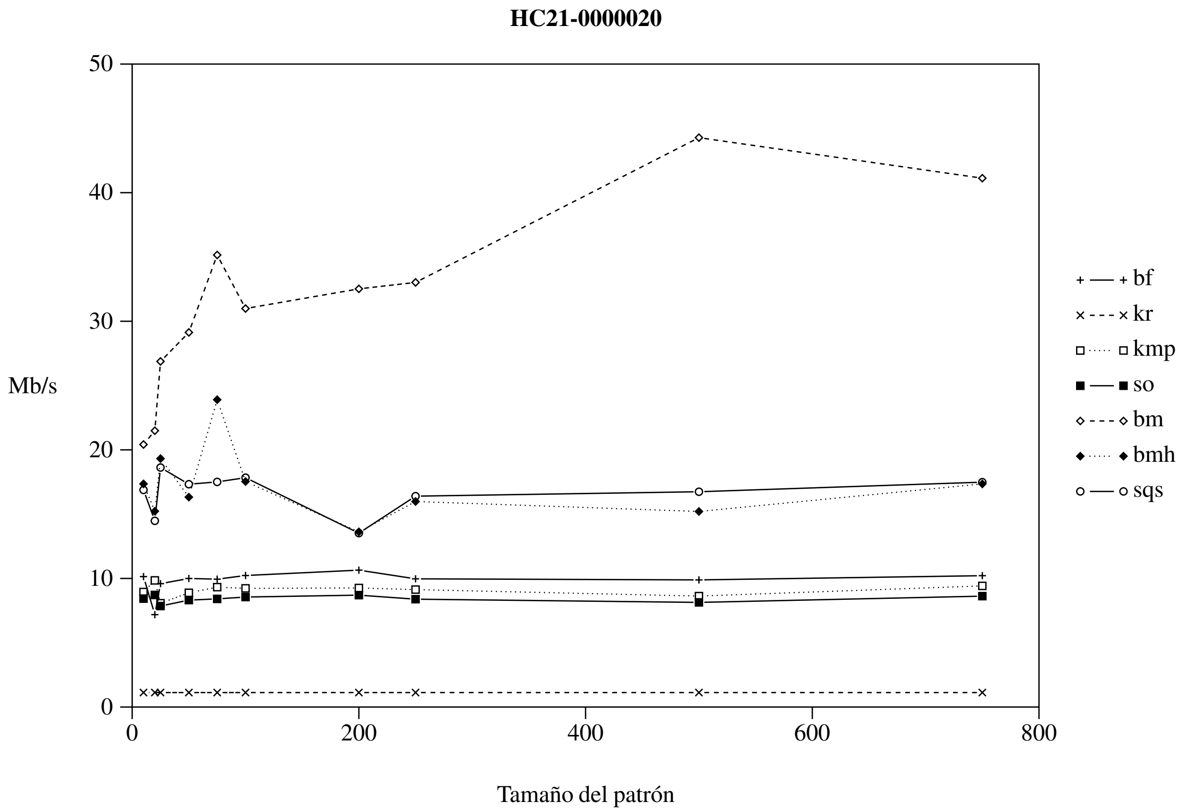

Ulysses (1585338)
2000000
1500000
1000000
500000
0
0
10
20
30
40
Pasos preproceso
Tamaño del patrón
bf
kr
kmp
so
bm
bmh
sqs


HC21-0000020 (1296826)
2000000
1500000
1000000
500000
0
0
200
400
600
800
Pasos preproceso
Tamaño del patrón
bf
kr
kmp
so
bm
bmh
sqs

4.2.3. Eficiencia temporal







4.3. Conclusiones

Comentaremos a continuación los resultados de los análisis presentados antes y las posiblidades gráficas de la biblioteca implementada.

4.3.1. Análisis asintótico

En primer lugar señalaremos que los resultados para los dos primeros textos son similares pero en el caso de la secuencia de ADN la cosa cambia; el reducido tamaño del alfabeto reduce también los pasos del preprocesador y su tiempo de ejecución.

Separando por algoritmos podemos decir que los basados en la teoría de automatas siguen su modelo establecido, el número de pasos de preproceso de los algoritmos es prácticamente igual al tamaño del patrón. En el caso de los de la familia booyer-moore e incluso el algoritmo shift-or el coste es el tamaño del patrón más el tamaño del alfabeto empleado.

La ejecución de la búsqueda nos da resultados que tambien se corresponden con los costes teóricos de cada algoritmo.

4.3.2. Análisis temporal

Respecto al análisis temporal hay varias cosas interesantes que decir. En primer lugar señalar que el peor algoritmo en todos los casos es el de Karp Rabin, el hecho de emplear opreraciones módulo y división para calcular las funciones de dispersión hacen de él una solución muy lenta en condicones normales, donde una instrucción aritmética tiene un coste muy superior al de una comparación.

Los algoritmos de fuerza bruta, knuth-morris-pratt y shift_or tienen un comportamiento temporal similar al teórico, aunque por las optimizaciones aplicadas su implementación, el más rápido es el de fuerza bruta, lo que no quiere decir que en textos repetitivos cualquiera de los otros dos no pueda ser muy superior.

Los algoritmos de la familia Boyer-Moore tambien se comportan como era de esperar, incluso si miramos la búsqueda sobre la cadena de ADN veremos que la versión completa (Boyer-Moore) es más rápida que las otras dos, ya que hace uso del alfabeto para mejorar la heurística de salto mientras que los otros dos son modificaciones del algoritmio que se basan en eliminar esa tabla, que sólo es útil para tabajar con textos altamente repetitivos.

4.3.3. Análisis gráfico

Además de las posiblidades como herramienta de análisis temporal podemos usar los programas de prueba para ver la evolución de cualquiera de los algoritmos de búsqueda, tomando el patrón y texto que deseemos. Esta capacidad de *trazar* la evolución del algoritmo puede sey muy útil para explicar el funcionamiento de los algoritmos y generar ejemplos apropiados para acompañar una explicación sobre ellos.

Referencias

[Aho75]
Aho, A. V. y Corasick, M. J.. Efficient String Matching: An Aid to Bibliographic Search. *Comunications of the ACM* **18** (6), 333–340 (Junio 1975).

[Aho83]
Aho, A. V.; Hopcroft, J. E. y Ullman, J. D.. *Data Structures and Algorithms.* Addison-Wesley, Reading, Massachusetts, 1983.

[Aho90]
Alfred V. Aho. Algorithms for Finding Patterns in Strings. En J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, páginas 255–300. Elsevier Science Publishers, New York, 1990.

[BM77]
Boyer, R. S. y Moore, J. S.. A Fast String Searching Algorithm. *Comunications of the ACM* **20** (10), 62–72 (Octubre 1977).

[Bae89]
Ricardo A. Baeza-Yates. Efficient Text Searching. Research Report CS-89-17 (1989), Departament of Computer Science, University of Waterloo, Ontario.

[Bae91]
Baeza-Yates, R. A. y Gonnet, G. H.. *Handbook of Algorithms and Data Structures: in Pascal and C*. Addison-Wesley, Reading, Massachusetts. Segunda Edición, 1991.

[Bae92a]
Ricardo A. Baeza-Yates. Text Retrieval: Theory and Practice, Twelfth IFIP World Computer Congress, Madrid, España, Septiembre 1992.

[Bae92b]
Baeza-Yates, R. A. y Gonnet, G. H.. A New Aproach to Text Searching. *Comunications of the ACM* **35** (10), 74–82 (Octubre 1992).

[C++96]
Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++, Diciembre 1996. URL http://www.maths.warwick.ac.uk/c++/pub/.

[Com79]
Commentz-Walter, B.. A string matching algorithm fast on the average. En H. A. Maurer, editor, *Proc. 6th International Coll. on Automata, Languajes and Programing*, páginas 118–132, Springer, Berlin, 1979.

[Dav86]
Davies, G. y Bowsher, S.. Algorithms for Pattern Matching. *Software – Practice and Experience* **16** (6), 575–601 (Junio 1986).

[Debian]
Debian/GNU Linux. URL http://www.debian.org/. Páginas Web relacionadas con la distribución Debian del sistema operativo Linux

[GNU]
GNU WWW. URL http://www.gnu.org/. Páginas Web relacionadas con el proyecto GNU (*GNU's Not Unix*) de la FSF (*Free Software Foundation*)

[Hor80]
Horspool, R. N.. Practical Fast Searching in String. *Software – Practice and Experience* **10** (6), 501–506 (1980).

[Hum91]
Hume, A. y Sunday, D. M.. Fast String Searching. *Software – Practice and Experience* **21** (11), 1221–1248 (Noviembre 1991).

[KMP77]
Knuth, D. E.; Morris, J. H. y Pratt, V. R.. Fast Pattern Matching in Strings. *SIAM Journal on Computing* **6** (2), 323–350 (Junio 1977).

[KR87]
Karp, R. M. y Rabin, M. O.. Efficient Randomized Pattern Matching Algorithms. *IBM J. Res. Develop.* **31** (2), 249–260 (1987).

[Kin96a]
Jeffrey H. Kingston. *An Expert's Guide to the Lout Document Formatting System (Version 3.10).* Basser Departament of Computer Science, University of Sydney, Octubre 1996.

[Kin96b]
Jeffrey H. Kingston. *A User's Guide to the Lout Document Formatting System (Version 3.10).* Basser Departament of Computer Science, University of Sydney, Noviembre 1996.

[Lam86]
Leslie Lamport. *LaTeX User's Guide and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1986.

[Man93]
Manber, U. y Wu, S.. GLIMPSE: A Tool to Search Through Entire File Systems. TR 93-34 (Octubre 1993), Department of Computer Science, University of Arizona, Tucson, Arizona. URL ftp://ftp.cs.arizona.edu/glimpse/glimpse.ps.Z.

[SGML]
SGML Web Page. URL http://www.sil.org/sgml/. Páginas Web relacionadas con el SGML (*Standard Generalized Markup Languaje*). Incluye punteros a información y programas relacionados con el SGML

[Ste92]
Graham A. Stephen. String Search. ?? TR-92-gas-01 (Octubre 1992), School of Electronic Engineering Science, University College of North Wales.

[Ste95]
Stepanov, A. A. y Lee, M.. The Standard Template Library, Hewlett-Packard Laboratories, Palo Alto, California, Febrero 1995. URL http://www.cs.rpi.edu/~musser/.

[Str91]
Bjarne Stroustroup. *The C++ Programing Languaje.* Addison-Wesley, Reading, Massachusetts. Segunda Edición, 1991.

[Sun90]
Daniel M. Sunday. A Very Fast Substring Search Algorithm. *Comunications of the ACM* **33** (8), 132–142 (Agosto 1990).

[Wat92]
Watson, B. W. y Zwaan, G.. A taxonomy of keyword pattern matching algorithms. Computing Science Notes 92/27 (Diciembre 1992), Eindhoven University of Technology, Eindhoven, Holanda.

[Wat94]
Bruce W. Watson. The performance of single-keyword and multiple-keyword pattern matching algorithms. Computing Science Notes 94/19 (Mayo 1994), Eindhoven University of Technology, Eindhoven, Holanda.

[Wat95]
Watson, B. W. y Zwaan, G.. A taxonomy of sublinear multiple keyword pattern matching algorithms. Computing Science Notes 95/ (Abril 1995), Eindhoven University of Technology, Eindhoven, Holanda.

[Wu92]
Wu, S. y Manber, U.. Fast Text Searching Allowing Errors. *Comunications of the ACM* **35** (10), 83–91 (Octubre 1992).

[glimpse]
GLIMPSE: GLobal IMPlicit SEarch. URL http://glimpse.cs.arizona.edu/. Sistema de indexación y recuperación que permite realizar búsquedas muy eficientes en multiples archivos

[lout]
The Basser Lout Document Formating System. URL ftp://ftp.cs.su.oz.au/jeff/lout/. Sistema de composición de documentos similar al LaTeX

[sgmltools]
SGML-Tools. URL http://web.inter.NL.net/users/C.deGroot/sgmltools/. Sistema basado en el SGML que permite generar versiones en distintos formatos a partir de un solo documento SGML escrito empleando el *linuxdoc-dtd* (usado para escribir los *HOWTO* de Linux)