2. Algoritmia

2.1. Introdrucción

Según la Real Academia Español la algoritmia es la ciencia del cálculo aritmético y algebraico y su objetivo es la búsqueda de operaciones que permitan hallar la solución de un problema. Por tanto, se denomina algoritmo a la solución encontrada para resolver una problemática dada.

La algoritmia nace en el siglo IX de la mano de Al-Juarismi, padre del álgebra y precursor del sistema de numeración usado en la actualidad. En este documento solo se abordará lo relativo a la algoritmia contemporánea, en concreto a la relacionada con la búsqueda de patrones.

La búsqueda de patrones, como su propio nombre indica, consiste en encontrar en un determinado texto un conjunto de N patrones, donde N va desde 1 hasta un número finito. No se tendrá en cuenta el caso de las expresiones regulares, en el que dicho conjunto puede alcanzar un tamaño infinito.

A lo largo de la historia, diversos autores han desarrollado algoritmos para la búsqueda de patrones. A día de hoy, no existe un único algoritmo para tal fin, puesto que cada uno cuenta con un rendimiento que dependerá de unas condiciones dadas. Estas condiciones pueden ser el número de patrones a buscar, la longitud de dichos patrones o las características del texto contra el que se busca.

2.2. Tipos de algoritmos

Los algoritmos de búsqueda de patrones pueden agruparse en algoritmos de búsqueda simple y múltiple. En el primer caso, cada ejecución del algoritmo buscará un único patrón en un texto, mientras que en el segundo caso, por cada ejecución se podrán buscar simultáneamente varios patrones. Llegados a este punto, se propone una agrupación de los algoritmos dependiendo de si llevan a cabo una búsqueda de patrones simple o múltiple. Separando en el segundo tipo los algoritmos basados en máquina de estados o en filtros Bloom (*hashing*).

2.2.1. Búsqueda simple

2.2.1.1. Naïve String Search Algorithm

El algoritmo más trivial de búsqueda uni-patrón, conocido como **Naïve String Search Algorithm**, realiza la búsqueda de un único patrón en un texto determinado de la manera más sencilla posible. Su funcionamiento consiste en recorrer el texto, carácter a carácter, y comparar cada uno con el primer carácter del patrón a buscar. En caso de coincidencia (o *match*) compara los siguientes

patrón: BCD
texto: ABADEBADD**BCD**FGHJ

```
          0123456789012345
          ABADEBADDBCDFGHJ posición
1         BCD·············  0
2          BCD············  1
3          BCD············  1
4           BCD···········  2
5            BCD·········  3
6             BCD········  4
7              BCD·······  5
8              BCD·······  5
9               BCD······  6
10               BCD·····  7
11                BCD····  8
12                 BCD···  9
13                 BCD···  9
14                 BCD···  (match)
15                  BCD··  0
16                   BCD·  1
17                    BCD  2
18                     BCD 3
```

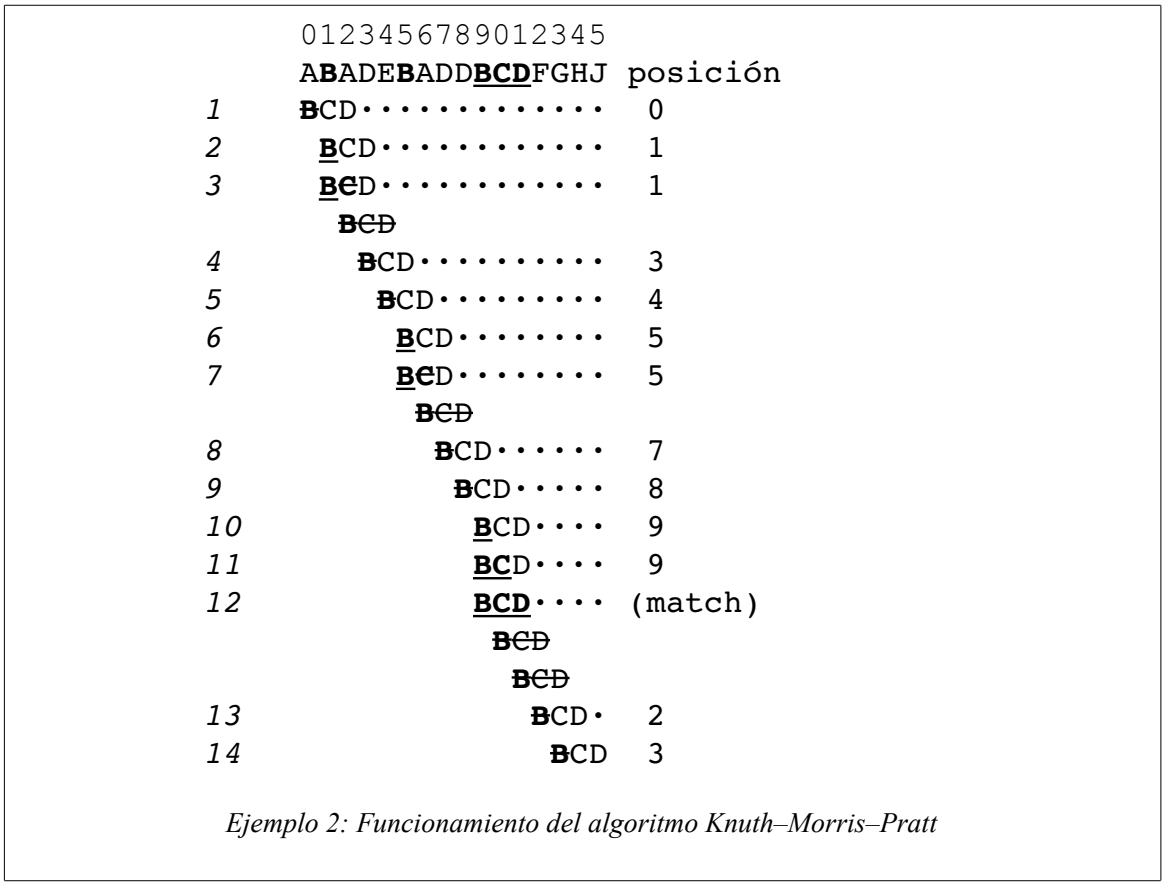
Ejemplo 1: Funcionamiento del algoritmo trivial de búsqueda simple de patrones (Naïve String Search Algorithm)

caracteres del texto y del patrón y así sucesivamente hasta conseguir un *match*. Si no se produce la coincidencia se continúa con el siguiente carácter del texto y se comienza de nuevo el proceso. En el ejemplo 1 se muestra con más detalle.

A pesar de su simplicidad, este algoritmo raramente es implementado debido a que ofrece un rendimiento muy pobre.

2.2.1.2. Knuth-Morris-Pratt

El algoritmo **Knuth-Morris-Pratt**, también conocido como **KMP**, fue creado en 1974 por Donald Knuth, James Hiram Morris y Vaughan Pratt. Sigue la línea del anterior algoritmo pero durante el proceso de búsqueda de patrones realiza una observación detallada, es decir no sólo compara un carácter con otro sino que memoriza lo que sucede previamente y cuando se produce un fallo en la comparación no desplaza solamente una posición el patrón respecto al texto, sino que avanza tantas



Ejemplo 2: Funcionamiento del algoritmo Knuth–Morris–Pratt

posiciones como sea necesario, siempre y cuando conozca de antemano que no se producirá coincidencia alguna en las posiciones anteriores. El proceso se muestra en el ejemplo 2, en el que aparecen tachadas las operaciones que se consiguen ahorrar respecto al algoritmo anterior.

Esta vez, después de la 3ª operación, dado que el carácter `C` del texto no corresponde a un posible inicio del patrón `BCD`, se avanza una posición en la búsqueda. Lo mismo ocurre después de la operación 7ª. Después de la operación 12ª, en la que se produce una coincidencia, se comprueba que los caracteres `C` y `D` tampoco forman el inicio, por tanto se omiten del mismo modo las dos siguientes operaciones.

2.2.1.3. Boyer-Moore

En 1977, Robert Stephen Boyer y J Strother Moore desarrollaron un algoritmo de búsqueda de patrones, hoy conocido como algoritmo **Boyer-Moore**. A diferencia del algoritmo KMP, no se inicia la inspección en el texto desde el primer carácter, sino que se comienza buscando por el último carácter del patrón, de esta forma, se ahorran muchas operaciones ya que si el último carácter no coincide se pueden llegar a avanzar tantas posiciones como dictamine la longitud del patrón. Se muestra su funcionamiento en el ejemplo 3.

En este ejemplo, en la 1ª operación se compara la letra `D` del patrón con la letra `A` del texto, que no coincide, pero no sólo eso, sino que se sabe que la letra `A` tampoco forma parte del patrón, por tanto se avanzan tantas posiciones como dicte la longitud del patrón, en este caso tres. En la siguiente

```
          0123456789012345
          ABADEBADDBCDFGHJ posición
1         BCÐ·············  0 (avance: 3-0=3)
2            BCÐ·········  3 (avance: 3-1=2)
3              BCD·······  5
4              BЄD·······  5
5                 BCÐ·····  8 (avance: 3-0=3)
6                  BCD····  9
7                  BCD····  9
8                  BCD···· (match)
9                     BCÐ·  0 (avance: 3-0=3)
```

Ejemplo 3: Funcionamiento del algoritmo Boyer-Moore

operación, la letra `B` comparada del texto sí se encuentra dentro del patrón, concretamente en la primera posición, por tanto se avanzan tantas posiciones como indique la longitud del patrón menos la posición en la que se encuentre la letra `B`, en este caso dos posiciones. Este mismo principio es el que se sigue de aquí en adelante hasta llegar al final del texto.

El recorrido hecho por los tres algoritmos anteriores muestra la evolución que han seguido los algoritmos de búsqueda uni-patrón hasta la actualidad, siendo el Boyer-Moore el que más rendimiento ofrece en esta categoría.

No obstante, cuando se trata de buscar varios patrones en un texto, existen otras alternativas que ofrecen mayor rendimiento. Estas alternativas son los algoritmos de búsqueda multi-patrón, que se explican en el siguiente punto.

2.2.2. Búsqueda múltiple

Dado un conjunto finito de patrones, denominado diccionario, un algoritmo de búsqueda múltiple debe ser capaz de buscar de manera simultánea todos los patrones del diccionario que se encuentren en un texto. Para ello, sólo se puede realizar una ejecución del algoritmo, es decir dicho texto sólo puede ser recorrido una vez, a diferencia de los algoritmos de búsqueda simple que deberán recorrer el texto tantas veces como patrones se quieran buscar en él.

Antes de seguir, conviene introducir algunos conceptos que ayudarán a entender mejor este bloque, como son la **máquina de estados finitos** o **FSM** (del inglés Finite State Machine) y los **filtros Bloom**. Posteriormente, se describirán algoritmos que usan estas técnicas.

2.2.2.1. Máquina de estados finitos: DFA y NFA

Una máquina de estados finitos, también conocida como **autómata finito** o **FA** (del inglés Finite Automaton), es un modelo matemático usado para solventar diversas situaciones, como puede ser el diseño de circuitos lógicos, la creación de protocolos de comunicación o la búsqueda de patrones. Esta última utilidad es la que resulta interesante en este documento y por tanto será en la que se

haga mayor hincapié. La relación del autómata finito con la búsqueda de patrones es que se usa dicho autómata para crear la estructura de detección que posteriormente utilizará la búsqueda multi-patrón.

Un autómata finito está formado por varios estados conectados entre sí mediante transiciones. Cada estado representa la situación actual del sistema y las transiciones entre estados vendrán determinadas por las acciones ejecutadas en el sistema.

En la figura 2.1 se muestra un ejemplo gráfico en el que se representan los estados por los que pasa una aplicación y sus transiciones. En este ejemplo, las entradas del sistema equivalen a las acciones que se ejecutan, como *startApp* y *pauseApp*, que provocarán la transición entre estados.

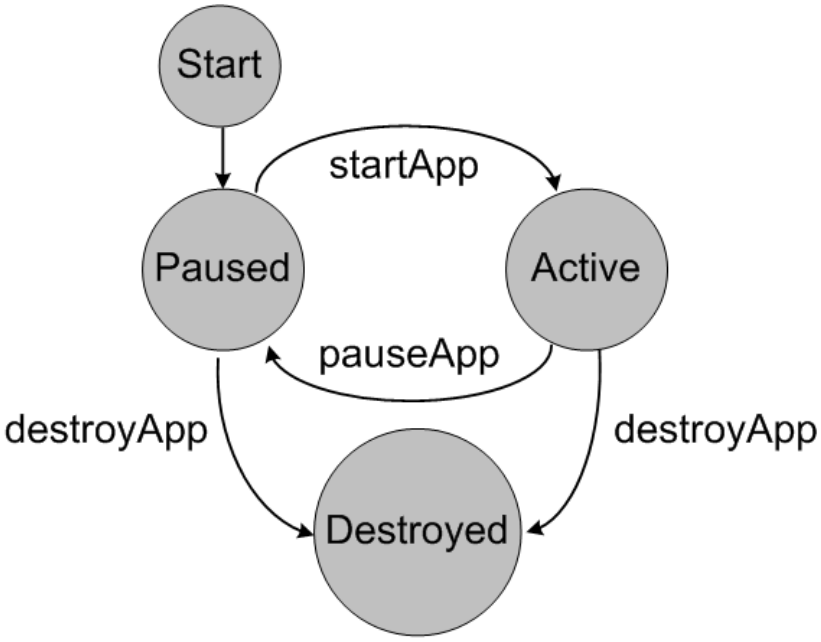


Figura 2.1: Ejemplo de máquina de estados

Existen dos tipos de autómatas finitos, el no determinista o **NFA** (Non-deterministic Finite Automaton) y el determinista o **DFA** (Deterministic Finite Automaton). La principal diferencia conceptual entre un NFA y un DFA es que, en el caso del DFA cada estado tiene exactamente una transición por cada posible valor de entrada, mientras que en un NFA una entrada puede conducir, en función del estado previo, a un sólo estado o a ninguno.

En la figura 2.2 se muestra una máquina de transición de estados en las formas no determinista y determinista. Se puede comprobar cómo el número de estados es el mismo, lo único que cambia son las transiciones entre estados. Mientras que en la máquina no determinista existen entradas sin una transición a otro estado intermedio, en la determinista cada entrada siempre conduce a otro estado. El ejemplo ha sido creado usando un diccionario con los patrones *halo*, *alto* y *ala*.

La ausencia de transiciones adicionales en los NFA provoca la necesidad de volver atrás en la búsqueda para que no se pierdan posibles positivos. A esta acción se la conoce como ***backtracking***. Por ejemplo, si en el texto *halago* queremos buscar los patrones *halo*, *alto* y *ala*, al leerlo desde el

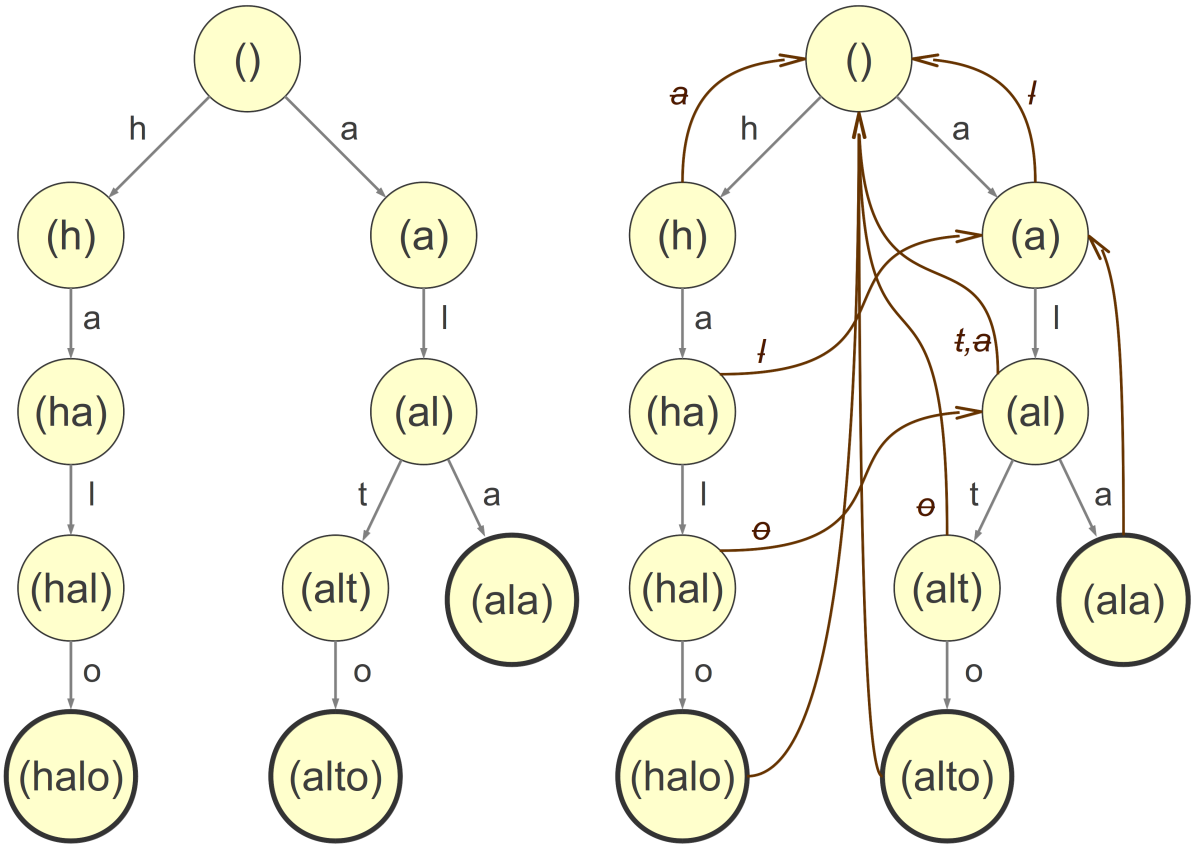


Figura 2.2: Autómata finito no determinista (izquierda) y determinista (derecha) usando los patrones 'halo', 'alto' y 'ala'. A L M I H A L A.

inicio recorrerá la máquina de estados por la rama izquierda, pasará del estado () al estado (hal) habiendo leído los caracteres *h*, *a* y *l*, y al leer el siguiente carácter volverá al inicio, es aquí cuando deberá volver a leer desde el sufijo '*alago*', repitiendo la lectura de los caracteres *a* y *l* ya leídos, para poder encontrar el patrón *ala*.

En este aspecto, los DFA son más eficientes, ya que del estado (hal) pasarían al estado (al) en el caso de que el siguiente carácter no fuera una *o*, por tanto no se debe recurrir al *backtracking* para que la búsqueda de los patrones no pierda eficacia.

Una desventaja de los DFA frente a los NFA es el incremento en el consumo de memoria, ya que necesitan almacenar más transiciones. Y no sólo se trata del consumo de memoria, sino que una transición se producirá de manera más rápida en un NFA, puesto que el número de transiciones posibles será menor que en el caso de un DFA. Aún así, los DFA suelen ser más utilizados cuando se prima el rendimiento frente al consumo de memoria.

2.2.2.2. Filtros Bloom

Los **filtros Bloom** datan de 1970 y reciben dicho nombre por su creador Burton Howard Bloom. Estos filtros sirven para comprobar si un elemento forma parte de un conjunto o no. Cuando el filtro encuentra un elemento dentro de un conjunto se dice que se ha producido un **positivo**, mientras que

en el caso contrario se trata de un **negativo**.

En los filtros Bloom es posible que se den **falsos positivos** pero es imposible que haya algún **falso negativo**. Es decir dado un elemento, si se produce un positivo puede darse el caso que tal elemento esté en el conjunto o no, pero en el caso de un negativo se podrá afirmar con certeza que el elemento no forma parte del conjunto y por lo tanto siempre se tratará de un **verdadero negativo**.

La seguridad que aportan los filtros Bloom en cuanto a la determinación de un negativo los convierte en un elemento muy útil a hora la de filtrar un conjunto de elementos. Antes de continuar con las utilidades de los filtros Bloom, será necesario introducir el concepto de **hash** para entender mejor el funcionamiento y los beneficios que puede aportar a la búsqueda de patrones.

Un hash es un resumen que se obtiene a partir de un dato determinado, bien sea de una palabra, un texto o un archivo. Cada dato proporciona un único resumen hash, pero diversos datos pueden producir un mismo hash, cuando se produce este último caso se dice que se ha dado una **colisión**. Dependiendo de la función hash que se utilice se obtendrán más, o menos, colisiones. Este hecho tiene mucho que ver con el rango de salida de la función. Cuanto mayor rango de salida ofrezca la función, la probabilidad de que se produzcan colisiones se reducirá, aunque estas funciones suelen tener un coste computacional mayor. No siempre conviene usar la función que menos colisiones devuelva, puede darse el caso de que compense usar una función con mayor número de colisiones si el coste computacional de dicha función es bastante menor. Se puede ver un ejemplo gráfico en la figura 2.3, en el se produce una colisión entre las palabras *alto* y *tono*.

Una vez explicado el concepto de hash, se podrá entender mejor el funcionamiento de un filtro Bloom. Recordando lo explicado previamente, un elemento que pase por un filtro Bloom, puede

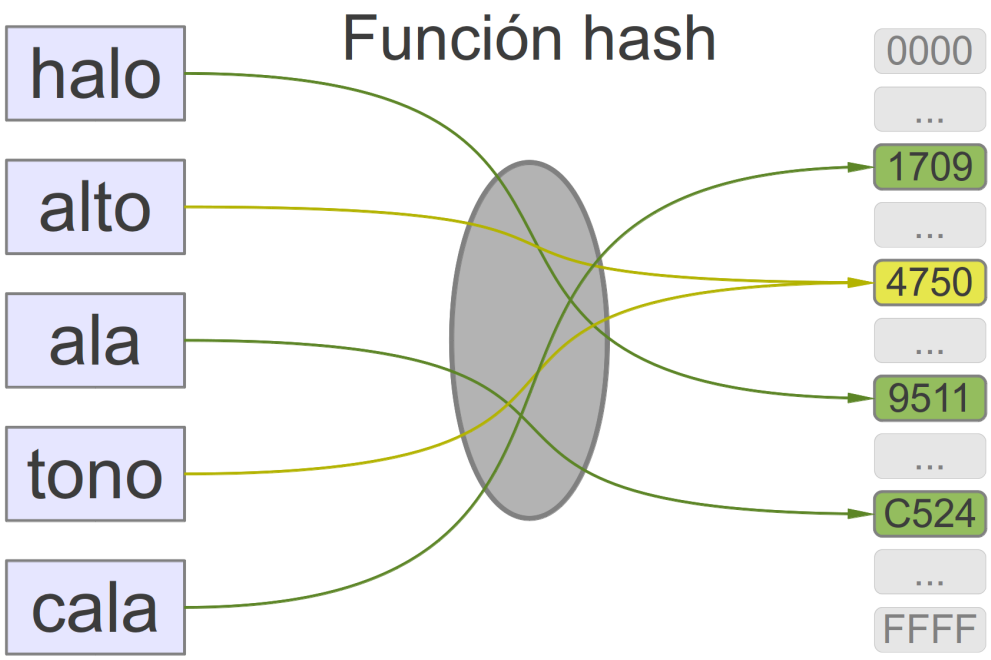


Figura 2.3: Ejemplo de aplicación de una función hash que devuelve, a partir del diccionario de la izquierda, un conjunto de resúmenes. Su rango de salida va desde 0x0000 a 0xFFFF. Se muestra una colisión entre 'alto' y 'tono'.

devolver un resultado positivo o negativo. Este funcionamiento se asimila bastante a una búsqueda de patrones corriente, en la que dado un texto o elemento y un diccionario de patrones, el texto o elemento será considerado como positivo si en él se encuentra al menos un patrón del diccionario, y negativo en el caso contrario. En el caso del uso de un filtro Bloom, en lugar de construir una máquina de estados para la búsqueda de patrones, la estructura de detección consistirá únicamente en una **matriz de almacenamiento binaria**, es decir una matriz que tomará sólo los valores 1 y 0.

En la creación de la estructura de búsqueda, también conocida como **fase *offline***, se crea la matriz binaria con todos sus valores a cero. A cada elemento del diccionario se le aplicará una función hash y el resultado será el indicador de la posición de la matriz que deberá cambiar su valor a un 1 lógico. Hay que tener en cuenta que diferentes elementos del diccionario pueden corresponder con el mismo hash, por consiguiente con la misma posición de la matriz. A esto se le denominaría una colisión en el filtro Bloom. La figura 2.4 ayuda a entender este proceso, donde se ha usado el mismo diccionario de la figura 2.3.

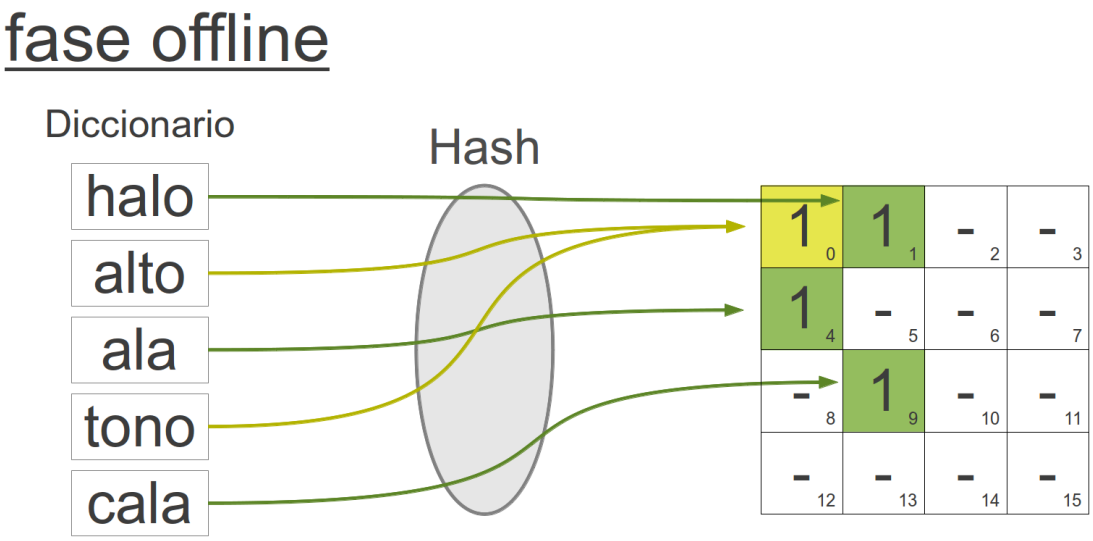


Figura 2.4: Fase offline, o creación de la estructura de búsqueda, de un filtro Bloom. En fondo amarillo se representa la colisión producida.

Una vez que se han evaluado todos los elementos del diccionario y se tiene la matriz binaria final, se procede la búsqueda de patrones, también conocida como **fase *online***. En esta fase, a cada elemento entrante, conocido como texto, se le aplica la misma función hash, y se comprueba si el resultado que devuelve corresponde a la posición de un 0 ó un 1 en la matriz. En el caso de ser un 0, se puede asegurar que tal elemento no pertenece al diccionario y por tanto será un verdadero negativo, mientras que si es un 1 cabe la posibilidad de que sí pertenezca al diccionario, por lo que sería un posible positivo. Para comprobar si verdaderamente se trata de un verdadero positivo, habría que realizar una comprobación posterior usando un algoritmo exacto. En el caso de la figura 2.5, el patrón *meta* no pasaría esta comprobación posterior, puesto que no forma parte del diccionario usado en la figura 2.4, y por tanto se tratará de un falso positivo.

Antes de seguir se planteará una consideración importante que no se debe pasar por alto a la hora de

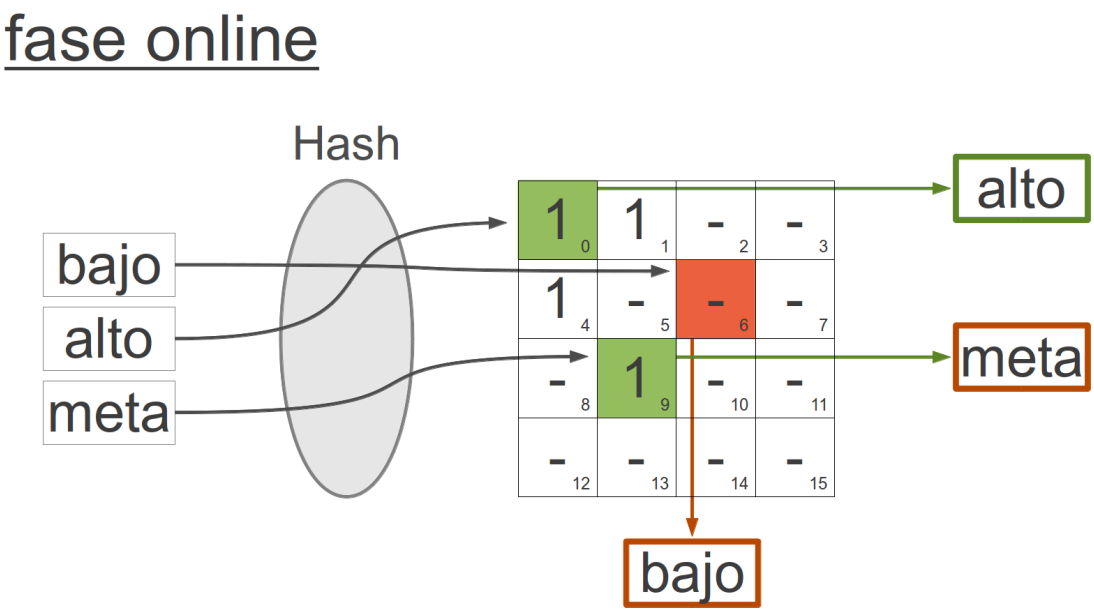


Figura 2.5: Fase online, o búsqueda de elementos, de un filtro Bloom. Los patrones 'alto' y 'meta' atraviesan el filtro (positivos), aunque 'meta' no forme parte del diccionario. El patrón 'bajo' es descartado (negativo).

trabajar con filtros Bloom. A la hora de realizar una búsqueda en un texto se deberá tener en cuenta el tipo de texto del que se trata y el diccionario con el que se está trabajando. Para la explicación se podría considerar como texto cualquier párrafo de este, o cualquier otro, documento.

Se propondrá un ejemplo que guíe al lector hacia la finalidad de la explicación, puesto que tratar de entender un razonamiento de este concepto seguramente sea menos efectivo que entender la problemática de una situación simulada. No obstante, después del ejemplo se desarrollará una explicación que se podrá comprender mejor.

En el ejemplo habrá dos situaciones parecidas aunque después se descubrirá que la diferencia existente tiene un efecto mayor del que se podría intuir a priori. En ambas situaciones se cuenta con el mismo texto, que será un fragmento de la obra *A un olmo seco* de Antonio Machado, la diferencia radicará en el conjunto de patrones que formará cada diccionario. A continuación se muestra el texto.

```
Mi corazón espera
también hacia la luz y hacia la vida,
otro milagro de la primavera.
```

Los diccionarios usados para cada situación se muestran en el siguiente cuadro. A primera vista se podrá apreciar que la diferencia fundamental es el tipo de patrones que usa cada uno. Mientras que en la primera situación se usan palabras extraídas de cualquier diccionario de la lengua española, en la segunda no es así, puesto que aparecen patrones formados por signos de puntuación, espacios y letras que aparentemente no tienen ningún sentido léxico.

Diccionario 1:	*Diccionario 2:*
espera	la luz
de	ida,
primavera	y ha
hacia	tro mi
luz	Mi c

En la primera situación, sabiendo que el diccionario contiene únicamente palabras extraídas de un diccionario de lengua, se podría realizar la búsqueda de los patrones en el texto mediante un tratamiento previo del mismo, obteniendo a partir del texto original fragmentos que formen textos más pequeños en el que cada uno de ellos sería una palabra. Para ello, el sistema que tratará el texto deberá reconocer una palabra como el conjunto de caracteres del alfabeto que no queda dividido por cualquier espacio o signo de puntuación. El resultado en este caso son los 16 textos que se muestran a continuación:

Mi	hacia	hacia	milagro
corazón	la	la	de
espera	luz	vida	la
también	y	otro	primavera

En una situación así, por tanto se podría actuar de una manera normal y predecible. En la fase *offline* se aplicarían la función hash a las palabras del diccionario para formar la matriz binaria. Mientras que en la fase *online* se trataría el texto original para obtener los 16 textos más pequeños, a cada uno se le aplicaría la función hash y se comprobaría el resultado de cada uno por separado para, a continuación, mostrar qué palabras se han detectado y cuáles no.

En la segunda situación el lector habrá podido comenzar a intuir que sería imposible operar de la misma manera. Efectivamente sería imposible puesto que no se podría realizar el tratamiento del texto, que se llevó a cabo en la primera situación, para obtener los 16 textos más pequeños. Hacer esto significaría no encontrar ningún patrón del diccionario, ya que ninguno de ellos corresponde a una palabra coherente, pero todos se encuentran dentro del texto. Evidentemente, aplicar la función hash al texto completo no sería concluyente en absoluto.

Si en este punto el lector se ha parado varios minutos para reflexionar acerca de una posible solución a esta problemática habrá notado que no se tratará de una solución trivial. Un camino a seguir para obtener una resolución al problema será pensar que se deberán leer trozos del texto realizando desplazamientos de carácter en carácter. Por ejemplo: '`Mi cor`', '`i cora`', '` coraz`', '`corazó`', y así sucesivamente hasta llegar al final del texto. En el ejemplo se han escogido todas las combinaciones posibles de 6 caracteres. A cada combinación se le aplicaría una función hash y su resultado se confrontaría con la matriz binaria para emitir un veredicto.

No obstante, el lector habrá observado que procediendo de esta forma únicamente se podrán encontrar los patrones de longitud 6, que en este caso serán '`la luz`' y '`tro mi`'. Para solucionarlo, se podría recorrer el texto tantas veces como distintas longitudes haya en el conjunto de patrones. En el caso del ejemplo hay patrones de longitud 4 y 6, por tanto se debería recorrer dos veces el texto, en la primera de ellas se formarían palabras de 4 caracteres y se les aplicaría la función hash correspondiente y en la segunda se procedería de la misma forma pero formando

palabras de 6 caracteres.

Pero, ¿qué ocurriría si se contara con un diccionario con multitud de patrones de longitudes muy diversas? ¿habría que recorrer tantas veces el texto como distintas longitudes hubiera? Esto sería muy ineficiente. Antes de seguir, no hay que olvidar que esta situación será muy frecuente puesto que el objetivo de los filtros Bloom, aunque aún no se ha explicado, es el de evaluar grandes cantidades de patrones rápidamente, es decir el rendimiento de un filtro Bloom deberá crecer a medida que aumenta el número de patrones del diccionario.

Para solucionar este problema se suele llegar a una solución de compromiso en la que se truncan los patrones a la menor longitud posible. Es decir, en nuestro ejemplo únicamente se rellenaría la matriz con los patrones de 4 caracteres y con los patrones resultantes de truncar a 4 caracteres los patrones de longitud 6. De esta forma únicamente se recorrería una vez el texto. Cuando la diferencia entre el patrón de mayor y menor longitud es muy grande, por ejemplo de 4 a 40, se suele elegir una longitud intermedia, como por ejemplo 10, y añadir al filtro únicamente los patrones que superen esta longitud. De esta forma los patrones de menor longitud se resolverían utilizando otro método de búsqueda de patrones, o construyendo un segundo filtro Bloom en el que se establecería a 4 la longitud mínima del patrón, y asumiendo el coste computacional de recorrer dos veces el texto.

En el caso de cualquier aplicación IDS/IPS las características del conjunto de patrones que forman el diccionario corresponderán a la segunda situación, por tanto se deberá proceder de una de las dos formas descritas anteriormente y que se resumen a continuación:

- Dado un conjunto de patrones con *n* longitudes distintas, en la fase *offline* se deberán formar *n* grupos de patrones con la misma longitud para construir *n* filtros Bloom distintos, mientras que en la fase *online* se calcularán *n* hashes por cada iteración en el texto y se comprobará la existencia de cada hash en su correspondiente matriz.
- Dado un conjunto de patrones con longitudes distintas, se establecerá una longitud mínima *k* a partir de la cual se truncarán los patrones para construir un único filtro Bloom formado por patrones de igual longitud. Si se tuvieran que descartar patrones cuyas longitudes fueran menores que *k*, se debería este conjunto de patrones cortos por otro método, o bien construyendo otro filtro Bloom de menor longitud de patrón.

Normalmente, en cualquier sistema que implemente el uso de filtros Bloom, se escogerá el segundo procedimiento por ser tener un coste computacional mucho menor.

No hay que olvidar que si se requiere una búsqueda exacta de patrones el uso de filtros Bloom no es suficiente puesto que únicamente devuelven posibles positivos, en lugar de verdaderos positivos. Aún así, una buena solución que ofrece un rendimiento adecuado es la combinación de filtros Bloom con algoritmos rápidos de búsqueda exacta uni-patrón que evalúen los posibles positivos, como por ejemplo Boyer-Moore.

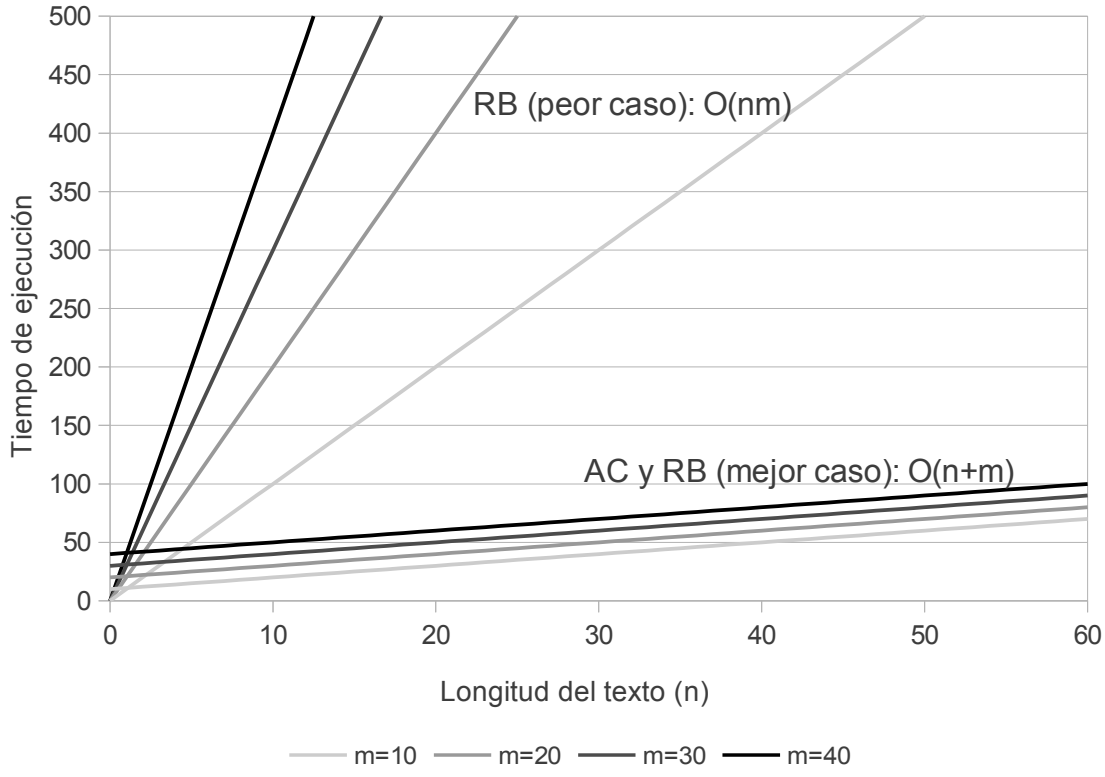
2.2.2.3. Algoritmos de búsqueda múltiple: Aho-Corasick y Rabin-Karp

Una vez introducidos los conceptos de máquina de estados y filtro Bloom, se pueden describir a continuación dos algoritmos que fueron pioneros en su época y que aún hoy en día son un referente de la búsqueda multi-patrón.

El primero de ellos es el **Aho-Corasick**, creado en por Alfred V. Aho y Margaret J. Corasick para los laboratorios Bell en 1975. Consiste en la búsqueda de un texto a través de una máquina de estados finitos, previamente creada analizando el conjunto de patrones a buscar. La máquina de estados es de tipo DFA, de esta forma se evita el *backtracking* presente en los NFA. Hasta la actualidad, es el algoritmo usado por defecto en Snort, y uno de los más eficientes, teniendo un rendimiento muy linear respecto al número de patrones usados. Para un texto de longitud n, y longitud de patrones constante y de valor m, el tiempo de ejecución es $O(n+m)$ en un espacio $O(m)$, siendo indiferente el número de patrones del diccionario. Hay que aclarar que, a pesar de haber tomado una longitud constante de patrones, el algoritmo Aho-Corasick no presenta esta restricción, por lo que se puede usar un diccionario que contenga patrones de distinta longitud.

El segundo algoritmo es el **Rabin-Karp**, creado por Michael Oser Rabin y Richard Manning Karp en 1987. Este algoritmo usa un filtro Bloom para encontrar cualquiera de los patrones de un diccionario. Hay que tener en cuenta lo explicado en el apartado de filtros Bloom, y es que para trabajar con funciones hash es necesario que todos los patrones tengan la misma longitud, o al menos una longitud mínima, descartando el resto del patrón desde una longitud en adelante. Esto último, unido al hecho de que existen falsos positivos, hace que el algoritmo Rabin-Karp deba ayudarse de una comprobación más exacta que se realizaría una vez que se obtuviera un positivo. En un espacio $O(p)$, siendo p el número de patrones del diccionario, el tiempo de ejecución de este algoritmo es $O(n+m)$ en el mejor de los casos y $O(nm)$ en el peor.

Como todo algoritmo de búsqueda multi-patrón que se precie, sus tiempos de ejecución no



Gráfica 2.1: Comparativa del tiempo de ejecución de los algoritmos Aho-Corasick (AC) y Rabin-Karp (RK). En este último se muestran el mejor y el peor de los casos.

dependen del número de patrones del diccionario *p*, sino únicamente de la longitud del texto *n* y de la longitud de los patrones *m*.

En la gráfica 2.1 se muestran los tiempos de ejecución de los algoritmos Aho-Corasick (AC) y Rabin-Karp (RK), en el caso de este último se distinguen el mejor y el peor de los casos.

Se podría llegar a pensar que no tendría sentido usar el algoritmo Rabin-Karp puesto que no sólo no lograría superar el rendimiento de Aho-Corasick, sino que además podría llegar a ser mucho peor. Esto es teóricamente cierto. No obstante, en la práctica estos algoritmos han de implementarse en un sistema y es donde llegan los problemas.

La creación de un DFA aumenta a medida que lo hace el número de patrones, mientras que en un filtro Bloom esto no tiene porque suceder, aunque si se tuviera que incrementar el tamaño de la matriz binaria para reducir el número de colisiones, la velocidad de crecimiento sería mucho menor que la de un DFA.

Esto quiere decir que al aumentar el número de patrones, el consumo de memoria crece más rápidamente al usar Aho-Corasick que al usar Rabin-Karp. En principio, este crecimiento no debería ser un factor limitante para el rendimiento, pero la realidad es que sí lo es, como se explicó en el primer capítulo.

Por tanto, el hecho de que el consumo de memoria del algoritmo Rabin-Karp permanezca casi constante, o crezca muy lentamente, al aumentar el número de patrones en el diccionario es el principal motivo por el que hoy en día sigue utilizándose en algunos sistemas. Lo que significa que en la práctica no es cierto que el número de patrones no influya en el rendimiento.