UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

LOS AUTÓMATAS FINITOS EN EL APAREAMIENTO DE CADENAS

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADA EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

MARÍA FERNANDA SÁNCHEZ PUIG

DIRECTORA DE TESIS: . ELISA VISO GUROVICH

FACULTAD DE CIENCIAS
U.N.A.M.

2005

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

Los autómatas finitos en el apareamiento de cadenas

realizado por María Fernanda Sánchez Puig

con número de cuenta 09954294-6 , quien cubrió los créditos de la carrera de:
Licenciatura en Ciencias de la Computación

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director Propietario	Dra. Elisa Viso Gurovich
Propietario	M. en C. José de Jesús Galaviz Casas
Propietario	Lic. en C. C. Francisco Lorenzo Solsona Cruz
Suplente	Lic. en C. C. Iván Hernández Serrano
Suplente	Mat. Mónica Leñero Padierna

Consejo Departamental de
Matemáticas

Dr. Francisco Hernández Quiroz

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMÁTICAS

A mis padres: *Fernando Sánchez Bravo*
Victoria Puig Salitres

A mi hermana: *Nuria Sánchez Puig*

Agradecimientos

El apoyo y la ayuda de mucha gente hicieron posible este trabajo.

Agradezco a todos mis profesores por compartir su tiempo y un poco de sus vidas.

Primero, agradezco a la Dra. Elisa Viso Gurovich por ser la excelente persona y maestra que es y por todas sus enseñanzas y ayuda en la realización de este trabajo.

Agradezco a José de Jesús Galavíz Casas por su amistad, tiempo y paciencia. Porque siempre tiene un buen chiste que contar y alguno que otro consejo que dar.

Quisiera darle especialmente las gracias a Mónica Leñero Padierna, quien me abrió su corazón, por siempre tener tiempo para una charla y muchas otras cosas.

Las gracias especiales van a toda la gente que comparte su amistad conmigo y me dan apoyo moral y académico siempre. Por todos los buenos momentos dentro y fuera de la escuela, agradezco sinceramente a Francisco Solsona Cruz, Iván Hernández Serrano, Arturo Vázquez Corona, Karla Ramírez Pulido y Manuel Sugawara Muro.

A mis amigos de siempre y de toda la vida, por brindarme su amistad y cariño incondicional.

Finalmente, agradezco a mis padres y hermana por su amor y su apoyo a lo largo de mi vida entera.

Todo mi corazón va a Mauricio Aldazosa Mariaca, por su amor, amistad, paciencia y comprensión incondicional brindada durante todos estos años.

Índice general

Índice de figuras

Índice de algoritmos

Capítulo 1

Introducción

Un texto es la manera más simple y natural de representar la información en diferentes áreas.

El procesamiento de texto empieza con la criptografía, bajo la idea de codificar mensajes escritos protegiéndolos contra una posible intercepción, y después las guerras impulsaron mucho este campo; en particular la segunda guerra mundial. El diseño de algoritmos que procesan texto se remonta aproximadamente treinta años atrás. En la última década se han producido muchos resultados nuevos al respecto. Dicho progreso se debe en parte a las investigaciones del genoma humano donde los algoritmos sobre texto son muy utilizados.

Este trabajo se ocupa de uno de los problemas básicos de procesamiento de texto: el problema de apareamiento de cadenas.

El problema de apareamiento de cadenas común . Consiste en encontrar todas las presencias de un patrón $P = p_1 p_2 ... p_m$ en un texto largo $T = t_1 t_2 ... t_n$, ambas sucesiones de símbolos tomados de un conjunto finito Σ. Dichos símbolos pueden ser caracteres en cualquier idioma: pares de bases en una secuencia de ADN, líneas de código fuente de algún programa, ángulos entre aristas en polígonos, máquinas o partes de máquinas de una producción, notas musicales o tiempos en una pieza musical, imágenes que se desean comparar, por mencionar algunas.

El apareamiento de cadenas es fundamental para el procesamiento de textos, ya que vivimos en un mundo propenso a errores tipográficos. Cualquier programa para corregir palabras debe poder identificar el apareamiento más cercano de cualquier cadena no encontrada en el diccionario.

En algunos casos el texto buscado no es exacto, por ejemplo, podemos no recordar la escritura exacta de una palabra que estamos buscando o la palabra puede estar mal escrita dentro del texto; incluso el texto puede corresponder a una secuencia de números con ciertas propiedades del cual no tenemos el patrón exacto, también puede ser una secuencia de moléculas de ADN en la que estamos buscando patrones aproximados.

Este trabajo se enfoca en el problema de apareamiento de cadenas con errores, también

llamado *apareamiento aproximado de cadenas*. La idea principal es realizar el apareamiento de un patrón en un texto donde uno o ambos han sufrido algún tipo de corrupción.

El problema, en su forma más general, es encontrar un texto donde aparezca un patrón dado, permitiendo un número limitado de errores durante el apareo.

El apareamiento aproximado de cadenas se define como la búsqueda de todas las ocurrencias de un patrón $P = p_1 p_2 ... p_m$ en un texto $T = t_1 t_2 ... t_n$ permitiendo a lo más k errores dentro del texto.

El número de errores permitidos en una subcadena es determinado por una distancia que se define como el mínimo número de operaciones necesarias para convertir el patrón P en la subcadena encontrada.

Entre las distancias utilizadas más frecuentemente en la medición del número de errores permitidos en una cadena de texto se encuentran la distancia de *Hamming* que permite la operación de reemplazo, la distancia de *Levenshtein* que permite las operaciones de reemplazo, inserción y eliminación y la distancia *generalizada de Levenshtein* que permite las operaciones de inserción, eliminación, reemplazo y transposición.

Cuando las únicas operaciones permitidas sobre las cadenas son las operaciones de inserción, reemplazo y eliminación de caracteres simples, como es el caso de la distancia de Levenshtein, dicho modelo de error se llama *distancia de edición*.

La distancia de edición ha recibido mucha atención porque resulta ser un medida de distancia muy poderosa para muchas aplicaciones.

Existen otros modelos de distancia diferentes; algunos de ellos son: *subcadena común más larga* (Longest Common Substring, LCS) entre dos cadenas que permite las operaciones de inserción y eliminación; *distancia de bloque* que permite reordenación y permutación entre las cadenas; distancias que permiten saltos entre las cadenas; distancias que permiten invertir las cadenas; *distancia episodio* que sólo permite la operación de inserción pero no es una distancia simétrica, etc.

Los algoritmos que resuelven el problema de apareamiento de patrones generalmente constan de dos etapas: una fase de preprocesamiento y un fase de búsqueda. En la fase de preprocesamiento se utiliza una pequeña cantidad de tiempo para analizar la estructura de la cadena de texto y evitar algunas comparaciones durante la ejecución del algoritmo. Después de la fase de preprocesamiento se realiza la fase de búsqueda, donde la información encontrada en la fase de preprocesamiento es utilizada para reducir el trabajo que se realiza durante la búsqueda. Cuando un texto es dinámico (como en aplicaciones de edición) el preprocesamiento es aplicado al patrón. Cuando el texto es estático (por ejemplo un diccionario) el preprocesamiento es aplicado al texto de manera que se construye un índice que después puede soportar diferentes tipos de búsquedas.

Se puede utilizar el preprocesamiento como un criterio para la clasificación general en búsquedas de texto. Hay cuatro categorías en esta clasificación:

1. Ni el patrón ni el texto son preprocesados. Los algoritmos elementales pertenecen a esta categoría.

2. El patrón es preprocesado. Los autómatas para el apareamiento de patrones pertenece a esta categoría.

3. El texto es preprocesado. Los autómatas de factor y métodos de indexado pertenecen a esta categoría.

4. Tanto el texto como el patrón son preprocesados. Los métodos de firma, autómatas de factor y autómatas para el apareamiento de patrones pertenecen a esta categoría.

Los autómatas finitos resultan ser herramientas útiles para entender y resolver problemas de apareamiento de cadenas. Algunos algoritmos que resuelven este tipo de problemas pueden ser considerados como simuladores de autómatas finitos no determinísticos (AFN), de modo que pueden servir como modelos para esos algoritmos.

Existen muchas áreas en las Ciencias de la Computación que utilizan autómatas finitos para resolver problemas.

Los autómatas finitos son dispositivos poderosos que se utilizan para el cálculo sobre sucesiones de caracteres en algunos campos del análisis léxico y biología computacional donde el apareamiento aproximado de cadenas está a la cabeza de muchos algoritmos que tratan con secuencias genéticas.

En este trabajo se presentan soluciones en las que la fase de búsqueda está basada en autómatas. Se presentan autómatas finitos no determinísticos para el apareamiento aproximado de cadenas utilizando la distancia de Hamming, la distancia de Levenshtein y la distancia generalizada de Levenshtein. Así como, problemas de apareamiento en el cual el texto no ha sido preprocesado y no se consideran algoritmos que pueden saltar alguna parte del texto.

1.1. Áreas de aplicación en el apareamiento de cadenas

Las primeras referencias que se tienen del problema de apareamiento de cadenas datan de los años 60's y 70's, donde el problema apareció en un gran número de áreas. La motivación principal para este tipo de investigación surge de la biología computacional, procesamiento de señales y recuperación de texto.

1.1.1. Biología computacional

El ADN y las secuencias de proteínas pueden ser vistas como largas cadenas de texto sobre alfabetos específicos. La búsqueda de secuencias específicas sobre estos textos aparecen como operaciones fundamentales para problemas tales como el ensamblaje de cadenas ADN de piezas obtenidas experimentalmente, búsqueda de rasgos en cadenas de ADN, o determinar la diferencia entre dos secuencias genéticas.

Estos problemas fueron modelados como búsqueda de patrones dados en un texto. Sin embargo, las búsquedas exactas tuvieron un uso limitado en esta área debido a que los patrones difícilmente se aparean de forma exacta en el texto; los resultados experimentales

presentan errores de diferentes tipos y aún las cadenas correctas pueden tener pequeñas diferencias, algunas de ellas significando mutaciones y alteraciones evolutivas.

Establecer las diferencias entre dos secuencias resulta importante para poder reconstruir el árbol de evolución.

Todos estos problemas requieren un concepto de *similitud*. Esto motiva a realizar búsquedas que permiten errores.

La biología computacional ha evolucionado mucho, con un empuje especial en los últimos años, gracias al proyecto del genoma humano de decodificar el ADN humano completo y sus potenciales aplicaciones.

Existen otros problemas importantes como el apareamiento de estructuras o búsqueda de patrones desconocidos.

1.1.2. Procesamiento de señales

Una de las más grandes áreas del procesamiento de señales es el reconocimiento de lenguaje, donde el problema general es determinar, dado una señal de audio, un mensaje textual que es transmitido. Aún problemas aparentemente simples como diferenciar una palabra de un pequeño número de alternativas, son complejos ya que algunas partes de la señal pueden estar comprimidas o perderse durante la transmisión. Un apareamiento exacto es prácticamente imposible.

Otro problema es la corrección de errores. La transmisión física de señales es propensa a errores. Para asegurar una correcta transmisión sobre un canal físico es necesario ser capaz de recuperar el mensaje correcto, después que un posible error es introducido durante la transmisión. En algunos casos no se sabe lo que se está buscando, sino sólo se desea un texto correcto y cercano al mensaje recibido.

El procesamiento de señales es un área muy activa hoy en día. La rápida evolución de bases de datos multimedia, demanda la habilidad de búsquedas por contenido en imágenes, audio y video. Se espera en los próximos años una mejor precisión en la comunicación no escrita humano-computadora, lo cual involucra reconocimiento de lenguaje. También presenta especial interés en redes inalámbricas ya que el aire es un medio de baja calidad de transmisión.

1.1.3. Recuperación de texto

El problema de corrección de errores de texto es quizá la aplicación más vieja para el apareamiento aproximado de cadenas.

Desde los años 60's el apareamiento aproximado de cadenas es una de las herramientas más populares y demandantes para este problema.

Sin embargo, el apareamiento de cadenas clásico no es suficiente ya que los textos cada vez son más grandes y menos homogéneos (por ejemplo, diferentes idiomas) y más propensos a errores. Muchos textos son muy grandes y su crecimiento es muy rápido de manera que es casi imposible controlar su calidad. Los textos digitalizados mediante reconocimiento de caracteres ópticos contienen un gran porcentaje de errores; lo mismo ocurre con los errores de escritura.

En nuestros días, todos los producto de recuperación de texto permiten textos o patrones que contienen errores. Otras aplicaciones para procesamiento de texto son correctores ortográficos, interfaces de lenguaje natural, aprendizaje de lenguas, por nombrar algunas.

1.1.4. Otras áreas

El número de aplicaciones para el apareamiento aproximado de cadenas crece cada día.

Se han encontrado soluciones a los problemas más diversos basados en apareamiento aproximado de cadenas como son reconocimiento de escritura, detección de virus e intrusos, compresión de imágenes, reconocimiento de patrones, reconocimiento de caracteres ópticos, comparación de archivos, por nombrar algunos.

Capítulo 2

Fundamentos

En este capítulo presentaremos algunos conceptos básicos necesarios, a partir de los cuales se desarrolla el resto de este trabajo.

2.1. Alfabetos y cadenas

Definición 2.1.1. Un *alfabeto* Σ es un conjunto finito, no vacío de símbolos.
Un alfabeto puede tener o no un orden establecido. El ordenamiento es total sobre el conjunto completo. Los alfabetos sin un orden establecido se llaman *alfabetos generales.*

Ejemplo 2.1.2. $\Sigma = \{0, 1\}$ es el alfabeto binario.
$\Sigma = \{a, b, ..., z\}$ es el conjuntos que contiene las letras minúsculas del alfabeto para el español.

Definición 2.1.3. Una *cadena* es una sucesión finita de símbolos dentro de un alfabeto. La cadena vacía ϵ es la sucesión vacía de símbolos y la cadena con cero presencias de símbolos.

Ejemplo 2.1.4. La cadena $c = 01101$ es una cadena formada con el alfabeto binario $\Sigma = \{0, 1\}$.

Definición 2.1.5. La *longitud de una cadena* es el número de posiciones de símbolos que contiene la cadena. Se denota $|w|$ a la longitud de la cadena w.

Ejemplo 2.1.6. $|011| = 3$ y $|\epsilon| = 0$

Definición 2.1.7. La operación de *concatenación de cadenas* se define sobre un conjunto de cadenas de forma que si x y y son cadenas sobre un alfabeto Σ, entonces la concatenación de estas cadenas, denotada como xy, es la cadena que resulta de copiar la cadena x seguida de copiar la cadena y.
Es decir, si x es la cadena formada con i símbolos, $x = a_1a_2...a_i$ y y es la cadena formada con j símbolos $y = b_1b_2...b_j$, entonces xy es la cadena de longitud $i + j$: $xy =$

$a_1a_2...a_ib_1b_2...b_j$.
La operación de concatenación es asociativa, es decir $(xy)\,z = x\,(yz)$ pero no es conmutativa, es decir $xy \neq yx$.
La cadena vacía ϵ es el elemento neutro dentro de la concatenación, es decir $x\epsilon = \epsilon x = x$.
Para una cadena x sobre un alfabeto Σ que contiene repeticiones se utiliza la notación de exponentes, donde x^k representa a x concatenada consigo misma k veces. Por ejemplo, $a^0 = \epsilon, a^1 = a, a^2 = aa, ..., \forall a \in \Sigma$.

Ejemplo 2.1.8. Sea $x = 01101$ y $y = 110$.
Entonces, $xy = 01101110$ y $yx = 11001101$. Además, $x\epsilon = 01101$

Definición 2.1.9. Si Σ es un alfabeto entonces el conjunto de todas las cadenas dentro del alfabeto Σ, que tienen una determinada longitud se puede expresar utilizando la notación exponencial. Se define Σ^k al conjunto de cadenas de longitud k que pertenecen al alfabeto Σ.

Se denota como Σ^* al conjunto de todas la cadenas sobre un alfabeto Σ, incluyendo a la cadena vacía ϵ. Dicho conjunto siempre es infinito y numerable. El conjunto de cadenas no vacías sobre un alfabeto Σ se denota como Σ^+. Entonces, $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

Ejemplo 2.1.10. Sea $\Sigma = \{0, 1\}$
Entonces $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$
$\Sigma^0 = \{\epsilon\}$ sin importar de qué alfabeto Σ se trate porque ϵ es la única cadena con longitud 0.

Definición 2.1.11. El *complemento de un alfabeto* Σ para un conjunto de símbolos B, $B \subseteq \Sigma$, se denota como $\overline{B} = \Sigma - B$. Usaremos $\overline{a}$ para denotar $\Sigma - \{a\}$.

Ejemplo 2.1.12. Sea $\Sigma = \{0, 1, 2, 3\}$, $B = \{1, 3\}$ y $a = 2$
Entonces, $\overline{B} = \Sigma - B = \{0, 2\}$ y $\Sigma - \{a\} = \{0, 1, 3\}$.

Definición 2.1.13. Para cualquier conjunto A, el conjunto de todos los subconjuntos de A es llamado el *conjunto potencia* de A y se denota $\mathcal{P}(A)$.

2.2. Operaciones de edición

Definición 2.2.1. La operación de edición de *reemplazo* es una operación que convierte un cadena wav en la cadena wbv, donde $w, v \in \Sigma^*, a, b \in \Sigma, a \neq b$. Es decir, un símbolo es reemplazado por otro, manteniendo su posición original dentro de la cadena.

Ejemplo 2.2.2. Sea $w = 12$, $v = 34$, $a = 5$ y $b = 6$. Entonces si se reemplaza el símbolo a por el símbolo b dentro de la cadena $wav = 12534$ se obtiene $wbv = 12634$.

Definición 2.2.3. La operación de edición de *inserción* es una operación que convierte una cadena wv en la cadena wbv, donde $w, v \in \Sigma^*, b \in \Sigma$. Es decir, un símbolo es agregado dentro de la cadena sin alterar el resto de la cadena.

Ejemplo 2.2.4. Sea $w = 12$, $v = 34$ y $b = 6$. Entonces si se inserta el símbolo b dentro de la cadena $wv = 1234$ se obtiene $wbv = 12634$.

Definición 2.2.5. La operación de edición de *eliminación* es una operación que convierte un cadena wav en la cadena wv, donde $w, v \in \Sigma^*, a \in \Sigma$. Es decir, un símbolo es quitado de la cadena sin alterar el resto de la cadena.

Ejemplo 2.2.6. Sea $w = 12$, $v = 34$ y $a = 5$. Entonces si se elimina el símbolo s dentro de la cadena $wav = 12534$ se obtiene $wv = 1234$.

Definición 2.2.7. La operación de edición de *transposición* es una operación que convierte un cadena $wabv$ en la cadena $wbav$, donde $w, v \in \Sigma^*, a, b \in \Sigma, a \neq b$. Es decir, dos símbolos son intercambiados entre sí.

Ejemplo 2.2.8. Sea $w = 12$, $v = 34$, $a = 5$ y $b = 6$. Entonces si se transpone el símbolo a con el símbolo b en la cadena $wabv = 125634$ se obtiene $wbav = 126534$.

Definición 2.2.9. La distancia de *Hamming* $D_H(v, w)$ entre dos cadenas $v, w \in \Sigma^*$, con $|v| = |w|$, es el mínimo número de operaciones de edición de reemplazo que son necesarias para convertir a v en w.

Definición 2.2.10. La distancia de *Levenshtein* $D_L(v, w)$ entre dos cadenas $v, w \in \Sigma^*$, con $|v| = |w|$, es el mínimo número de operaciones de edición de reemplazo, inserción y eliminación que son necesarias para convertir a v en w.

Definición 2.2.11. La distancia *generalizada de Levenshtein* $D_G(v, w)$ entre dos cadenas $v, w \in \Sigma^*$, con $|v| = |w|$, es el mínimo número de operaciones de edición de reemplazo, inserción, eliminación y transposición que son necesarias para convertir a v en w. Cada símbolo de v puede participar en a lo más una operación de edición de transposición.

2.3. Apareamiento de cadenas

Definición 2.3.1. El *apareamiento aproximado de cadenas* se define como la búsqueda de todas las ocurrencias del patrón $P = p_1 p_2 ... p_m$ en el texto $T = t_1 t_2 ... t_n$ permitiendo a lo más k errores en el texto. El número de errores permitidos en una subcadena es determinada por las distancias de Hamming, Levenshtein y Levenshtein generalizado.

El apareamiento aproximado de cadenas utilizando la distancia de Hamming se llama también *apareamiento aproximado de cadenas con k errores*, y el apareamiento aproximado de cadenas utilizando la distancia de Levenshtein se llama también *apareamiento aproximado de cadenas con k diferencias.*

Definición 2.3.2. El *apareamiento exacto de cadenas* es un apareamiento aproximado de cadenas donde k, el máximo número de errores, es 0.

2.4. Autómatas finitos

Enunciaremos brevemente algunas nociones básicas de los autómatas finitos y se extenderá el tema en el capítulo 4.

Definición 2.4.1. Un *autómata finito no determinístico* (AFN) N es un quíntuplo $N = (Q, \Sigma, \delta, S, F)$ donde:

Q es un conjunto finito de estados.

Σ es un conjunto finito de símbolos de un alfabeto.

δ es la función de transición; $\delta : (Q \times \Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$.

S es un subconjunto de estados iniciales; $S \subseteq Q$.

F es un conjunto finito de estados finales; $F \subseteq Q$.

Definición 2.4.2. Sea $N = (Q, \Sigma, \delta, S, F)$ un AFN y $P \subseteq Q$. La $\epsilon - cerradura(P) = \{q' \mid q' \in \delta(q, \epsilon), q \in P\} \cup P$.

Definición 2.4.3. La *configuración* de un AFN $N = (Q, \Sigma, \delta, q_0, F)$ es un par $(q, v) \in Q \times \Sigma^*$.

La *configuración inicial* del AFN es un par (q_0, v) y una *configuración final* del AFN es un par (q_f, ε). Se dice que el AFN se encuentra en una configuración que acepta si $q_f \in F$.

Definición 2.4.4. La *transición* de un AFN $N = (Q, \Sigma, \delta, q_0, F)$ es una relación $\vdash_N \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ definida como $(q_1, aw) \vdash_N (q_2, w)$ donde $q_2 \in \delta(q_1, a), a \in \Sigma \cup \{\varepsilon\}, w \in \Sigma^*, q_1, q_2 \in Q$.

El símbolo $\vdash_N^*$ denota la cerradura de transitividad y reflexión de la relación $\vdash_N$.

Definición 2.4.5. El *lenguaje aceptado por un AFN* $N = (Q, \Sigma, \delta, q_0, F)$ es el conjunto $L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q_f, \epsilon), q_f \in F\}$.

Ejemplo 2.4.6. El lenguaje de todas las cadenas que contienen un número arbitrario de 0's y que terminan con un número arbitrario de 1's es: $\{\epsilon, 01, 0011, 000111, ...\}$.
Σ^* es un lenguaje para cualquier alfabeto Σ.
El lenguaje vacío, $\emptyset$, es un lenguaje sobre cualquier alfabeto.

Definición 2.4.7. Un *estado inaccesible* de un *AFN* $N = (Q, \Sigma, \delta, q_0, F)$ es un estado q tal que no existe ninguna cadena $w \in \Sigma^*$ tal que $(q_0, w) \vdash_M^* (q, \epsilon)$.

Definición 2.4.8. Un *estado activo* de un *AFN* $N = (Q, \Sigma, \delta, q_0, F)$ después de leer la cadena de entrada $w \in \Sigma^*$ es cada estado q tal que $(q_0, w) \vdash_M^* (q, \epsilon)$.

Definición 2.4.9. La *profundidad* de un estado q en un AFN $N = (Q, \Sigma, \delta, q_0, F)$, $q \in Q$, es el mínimo número de movimientos que son necesarios para llegar desde un estado inicial q_0 al estado q sin utilizar transiciones-ε y sin ciclos. En las figuras de los autómatas finitos presentadas en los siguientes capítulos, los estados con la misma profundidad se encuentran alineados verticalmente.

Definición 2.4.10. El *nivel* de un estado q en un AFN $N = (Q, \Sigma, \delta, q_0, F), q \in Q$, es el mínimo número de errores asociados con todos los estados finales alcanzables desde q. En las figuras de los autómatas finitos presentadas en los siguientes capítulos, los estados con el mismo nivel se encuentran alineados horizontalmente.

En el siguiente capítulo se revisará la clasificación de los problemas de apareamiento.

Capítulo 3

Clasificación de problemas de apareamiento

En este capítulo revisaremos cómo se clasifican los problemas de apareamiento de patrones, de acuerdo a diferentes características.

Llamaremos *dimensión*, en el contexto de apareamiento de patrones, al grado de libertad que se tiene dentro del espacio de todos los posibles problemas de apareamiento de patrones. Por ejemplo, los problemas de apareamiento de patrones de una dimensión son los problemas secuenciales.

De esta forma, los problemas para el apareamiento de patrones de una dimensión para un alfabeto de tamaño finito pueden ser clasificados de acuerdo a muchos criterios.

Se utilizan principalmente seis criterios de clasificación dentro de un espacio de seis dimensiones en el cual cada punto corresponde a un problema particular de apareamiento de patrones.

Todas las dimensiones posibles incluyendo los valores posibles dentro de cada dimensión son:

1. Naturaleza del patrón:
   - cadena
   - sucesión
2. Integridad del patrón:
   - patrón completo
   - subpatrón

3. Número de patrones:

   - un patrón
   - número finito de patrones mayor a uno
   - número infinito de patrones

4. Forma de apareamiento:

   - apareamiento exacto [1]
   - apareamiento aproximado con distancia de Hamming [1]
   - apareamiento aproximado con distancia de Levenshtein [1]
   - apareamiento aproximado con distancia generalizada de Levenshtein [1]
   - apareamiento aproximado con distancia Δ [2]
   - apareamiento aproximado con distancia Γ [2]
   - apareamiento aproximado con distancia (Δ, Γ) [2]

5. Importancia de símbolos dentro del patrón:

   - importancia de todos los símbolos
   - algunos símbolos sin importancia

6. Sucesión de patrones:

   - una sucesión
   - un número finito de sucesiones

Esta clasificación se puede visualizar en la figura 3.1.

[1] Este tipo de apareamiento se definirá con detalle más adelante.
[2] Tipo de distancias definidas para alfabetos ordenados

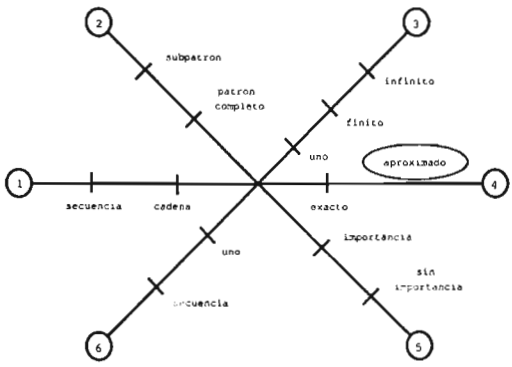


Figura 3.1: Clasificación de problemas de apareamiento de patrones

De manera que el número posible de problemas de apareamiento de patrones se obtiene $N = 2 * 2 * 3 * 7 * 2 * 2 = 336$.

Si se toman en cuenta todos los posibles ejemplares del problema, el número de problemas para el apareamiento de patrones crece rápidamente.

3.1. Dos formas de apareamiento de patrones

Existen diferentes maneras en las cuales se puede realizar el apareamiento de patrones:

- Apareamiento de patrones hacia adelante.
- Apareamiento de patrones hacia atrás.

El principio básico del apareamiento de patrones hacia adelante se muestra en la figura 3.2.

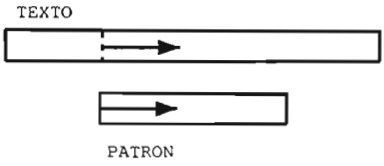


Figura 3.2: Apareamiento de patrones hacia adelante

El texto y el patrón se aparean en dirección "hacia adelante", lo que significa que la comparación de símbolos se realiza de izquierda a derecha. Todos los algoritmos para

el apareamiento de patrones hacia adelante deben comparar cada símbolo del texto al menos una vez contra el patrón. Entonces, el menor tiempo de complejidad es igual a la longitud del texto.

El principio básico del apareamiento de patrones hacia atrás se muestra en la figura 3.3.

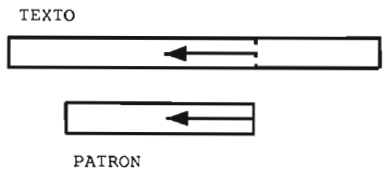


Figura 3.3: Apareamiento de patrones hacia atrás

Esto significa que la comparación de símbolos se realiza de derecha a izquierda.
Hay dos principios para el apareamiento de patrones hacia atrás:

- Buscar repeticiones de un sufijo del patrón.
- Buscar un prefijo del patrón.

Los algoritmos para el apareamiento de patrones hacia atrás permiten saltar algunas partes del texto de manera que el número de comparaciones puede ser menor que la longitud del texto.

La mayoría de los algoritmos hacia adelante basan su funcionamiento en el modelo de autómatas finitos, mientras que los algoritmos hacia atrás trabajan con sufijos. Nos concentraremos en los primeros, para lo cual en el siguiente capítulo se revisará la relación que existe entre los autómatas finitos y los conjuntos de cadenas.

Capítulo 4

Autómatas finitos y expresiones regulares

Un autómata finito es un modelo matemático de un sistema, con entradas y salidas discretas. El sistema puede estar en cualquiera de un número finito de configuraciones o "estados". El estado del sistema contiene información acerca de las entradas anteriores y que son necesarias para determinar el comportamiento futuro del sistema; de este modo, los estados conforman la memoria del sistema.

En las ciencias de la computación se pueden encontrar muchos ejemplos de sistemas de estados finitos; programas tales como editores de texto y analizadores léxicos dentro de la mayoría de los compiladores están usualmente diseñados con sistemas de estados finitos. Por ejemplo, un analizador léxico busca los símbolos de un programa para localizar las cadenas de caracteres correspondientes a identificadores, constantes numéricas o palabras reservadas. En dicho proceso el analizador necesita recordar tan solo una cantidad finita de información, tal como el número de caracteres que ha leído de una palabra reservada desde que inició el proceso.

Se puede ver al cerebro humano como una red de sistemas de estados finitos. El número de neuronas es finito, de modo que se puede pensar que el estado de cada neurona puede ser descrito por un pequeño número de bits y cada neurona puede estar en un número finito de estados. Así, podemos utilizar la teoría de autómatas finitos para tratar de entender al cerebro humano.

Sin embargo, la razón más importante para el estudio de sistemas de estados finitos es la propia naturaleza del concepto, su simplicidad, la riqueza en su estructura y el potencial de sus aplicaciones.

A continuación se presentará la noción formal de un autómata finito; aunque no se desarrollarán todas las demostraciones de las aseveraciones que se realicen, el lector interesado puede consultar [9], [10], [16] y [19] para más detalle de dichas demostraciones.

4.1. Definición básica

Un autómata finito (AF) consiste de un conjunto finito de estados y un conjunto de transiciones de un estado a otro, dado un símbolo de entrada tomado de un alfabeto finito Σ. Por cada símbolo de entrada hay definida exactamente una transición ya sea hacia otro estado o hacia sí mismo [1]. Un estado, usualmente denotado q_0, es el estado inicial en el cual empieza el autómata. Existen otros estados designados como finales o estados que aceptan. El AF acepta una cadena x si la secuencia de transiciones correspondientes a los símbolos de x lleva al autómata, desde el estado inicial hasta un estado que acepta; de otra forma el autómata rechaza la cadena.

Formalmente, un autómata finito M es un quíntuplo $M = (Q, \Sigma, \delta, q_0, F)$ donde:

Q es un conjunto finito de estados.
Σ es un conjunto finito de símbolos de un alfabeto.
δ es la función de transición; $\delta : Q \times \Sigma \rightarrow Q$.
q_0 es el estado inicial; $q_0 \in Q$.
F es un conjunto finito de estados finales; $F \subseteq Q$.

Se puede pensar en el funcionamiento del autómata como una sucesión de movimientos (transiciones) cada uno de ellos en un instante dado de tiempo t, donde t_0 representa el momento en que el autómata no ha empezado a trabajar aún; el autómata comienza en un estado inicial. En cada instante t, M recibe un símbolo $s \in \Sigma$. La capacidad del autómata para retener información respecto a símbolos anteriores reside en el conjunto de estados, de modo que lo que recuerda el autómata es aquello que lo obligó a entrar en un cierto estado. Cuando se alimenta un símbolo al autómata se produce en él un cambio de estado, que depende exclusivamente del estado en el que se encuentra y del símbolo que recibió como entrada.

Para poder describir formalmente el comportamiento de un AF sobre una cadena debemos extender la función de transición, para aplicarla a un estado y una cadena en lugar de a un estado y un símbolo.

Formalmente extendemos la función de transición de la siguiente manera:

1. $\hat{\delta}(q, \epsilon) = q$.
2. $\forall w \in \Sigma^*, a \in \Sigma, \hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$.

La regla 1 indica que el autómata no puede ejecutar una transición sin leer ningún símbolo.

[1] Trabajaremos primero con el modelo de autómata determinístico (AFD).

La regla 2 indica inductivamente cómo encontrar un estado después de leer una cadena de entrada no vacía.

De esta forma es posible formalizar la noción de un lenguaje aceptado por un autómata finito.

El lenguaje aceptado por M, designado como $L(M)$, se define:

$$L(M) = \left\{ x \in \Sigma^* \mid \hat{\delta}(q_0, x) = q_i, \textit{ con } q_i \in F \right\}$$

En adelante se omitirá el gorrito de la δ, ya que no aporta información adicional.

Al conjunto de lenguajes aceptados por los AF se les denomina *lenguajes regulares.*

Una gráfica dirigida, llamada *diagrama de transición*, se asocia con el AF de modo que los vértices de la gráfica corresponden a los estados del AF. Si hay una transición desde el estado q hacia el estado p dada la entrada a, entonces existe un arco etiquetado con a desde el estado q hacia el estado p en el diagrama de transición.

Para construir los diagramas de transición se siguen las reglas:

1. Se dibuja un nodo de la digráfica por cada estado en Q y se etiqueta el nodo con el nombre del estado.
2. Al estado inicial se le marca de alguna forma que lo distinga de los demás, como puede ser una flecha o la palabra "inicio".
3. A los estados finales se les marca de alguna forma especial, como puede ser una doble delimitación del nodo.
4. Se traza un arco dirigido del estado q_i al estado q_j y se etiqueta con el símbolo a si y sólo si $\delta(q_i, a) = q_j$.

Ejemplo 4.1.1. Se presenta el diagrama de estados del autómata finito M.

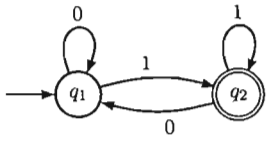


Figura 4.1: Diagrama de estados para el autómata finito M

La descripción formal del autómata es $M = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\})$.

La función de transición δ es:

$$\delta(q_1, 0) = q_1$$
$$\delta(q_2, 0) = q_1$$
$$\delta(q_1, 1) = q_2$$
$$\delta(q_2, 1) = q_2$$

4.2. Autómatas finitos no deterministicos

Consideremos una modificación al modelo de los autómatas finitos de forma que permitan cero, una o más transiciones de un estado a otro bajo el mismo símbolo de entrada. Este modelo se conoce como *autómata finito no determinístico (AFN).*

Una sucesión de entrada $a_1\, a_2 \ldots a_n$ es aceptada por un autómata finito no determinístico si existe una sucesión de transiciones correspondiente a la sucesión de entrada que vaya desde el estado inicial a algún estado final.

De este modo los autómatas finitos son un caso particular de los autómatas finitos no deterministicos, en los que por cada estado existe una única transición por cada símbolo.

Formalmente, un autómata finito no determinístico N es un quíntuplo $N = (Q, \Sigma, \delta, S, F)$ donde:

Q es un conjunto finito de estados.
Σ es un conjunto finito de símbolos de un alfabeto.
δ es la función de transición; $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$.
S es un subconjunto de estados iniciales; $S \subseteq Q$.
F es un conjunto finito de estados finales; F $\subseteq$ Q.

La intención es que $\delta(q, a)$ sea un subconjunto de estados $\{p_i\}$, tal que hay una transición de q a cada uno de los estados p_i al leer el símbolo a.

Podemos extender formalmente la función de transición de la siguiente manera:

1. $\hat{\delta}(q, \epsilon) = q$.
2. $\forall w \in \Sigma^*, a \in \Sigma, \hat{\delta}(q, wa) = \left\{p \mid \text{para algún estado } r \in \hat{\delta}(q, w), p \in \delta(r, a)\right\}$.

La regla 1 indica que el autómata no puede ejecutar una transición sin leer ningún símbolo.

La regla 2 indica que empezando en el estado q y después de leer la cadena wa, la única forma de terminar en el estado p es que r sea uno de los posibles estados a los que se transfiere el autómata desde q después de leer w y p sea uno de los posibles estados a los que se transfiera N desde r al leer a.

También se puede extender el dominio de δ a subconjuntos de estados $\delta : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q)$ de la siguiente manera:

$$\delta(P, w) = \bigcup_{p \in P} \delta(p, w).$$

De esta forma es posible formalizar la noción de un lenguaje aceptado por un AFN. El lenguaje aceptado por N, designado como $L(N)$, se define:

$$L(N) = \left\{ x \in \Sigma^* \mid \hat{\delta}(q_0, x) \text{ contiene algún estado en } F \right\}$$

Ejemplo 4.2.1. Se presenta un autómata finito no determinístico que solo acepta cadenas formadas por 0's y 1's que terminan en 01.

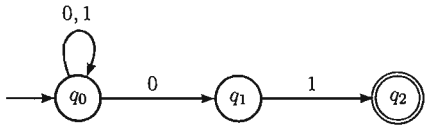


Figura 4.2: AFN que acepta todas las cadenas que terminan en 01

El autómata contiene dos arcos etiquetados con 0 que salen del estado q_0. De modo que cuando el autómata se encuentre en el estado q_0 y lea una cadena 0, se transferirá a los estados q_0 y q_1 simultáneamente.

4.2.1. Equivalencia entre AFD y AFN

Se puede demostrar que la clase de los lenguajes aceptados por los AFD es la misma que la clase de lenguajes aceptados por los AFN, ver [19]. Así, la clase de lenguajes aceptados por los AFN incluyen a los conjuntos regulares y de igual forma, por cada AFN se puede construir un AFD equivalente que acepte el mismo lenguaje.

La manera en que un AFD simula un AFN es permitiendo que los estados del AFD correspondan a subconjuntos de estados de Q en el AFN.

El AFD que se construye mantiene el rastro de todos los estados en los que el AFN puede estar después de leer la misma entrada que el AFD ha leído.

4.3. Autómatas finitos con transiciones-ϵ

Es posible extender el modelo de autómata finito no determinístico para incluir transiciones sobre la entrada vacía ϵ.

Formalmente, un autómata finito no determinístico con transiciones-ϵ es un quíntuplo $N = (Q, \Sigma, \delta, S, F)$ donde:

Q es un conjunto finito de estados.
Σ es un conjunto finito de símbolos de un alfabeto.
δ es la función de transición; $\delta : (Q \times \Sigma \cup \{\epsilon\} \rightarrow \mathcal{P}(Q)$.
S es un subconjunto de estados iniciales; $S \subseteq Q$.
F es un conjunto finito de estados finales; F $\subseteq$ Q.

La intención es que $\delta(q, a)$ sea un subconjunto de estados p, tal que hay una transición de q a p al leer a, donde a es un símbolo en Σ^*.

Resulta importante calcular el conjunto de estados que se pueden alcanzar desde un estado dado q usando únicamente transiciones-ϵ; esto es equivalente a calcular los vértices que son alcanzables desde un vértice dado en una gráfica dirigida utilizando para ello únicamente arco etiquetados con ϵ.

Utilizamos el término $\epsilon - cerradura(q)$ para denotar al conjunto de todos los vértices p tales que existe un camino de q a p donde todos los arcos tienen como etiqueta a ϵ.

Entonces, la $\epsilon - cerradura(P)$, donde P es un conjunto de estados, consiste de:

$$\bigcup_{q \in P} \epsilon - cerradura(q)$$

Existe una equivalencia entre los autómatas finitos con transiciones-ϵ y sin transiciones-ϵ, de modo que si L es el lenguaje aceptado por un AFN con transiciones-ϵ, entonces L es aceptado por un AFN sin transiciones-ϵ.

Ejemplo 4.3.1. Se presenta un autómata finito no determinístico con transiciones-ϵ que sólo acepta cadenas formadas por 0's y 1's que terminan en 01.

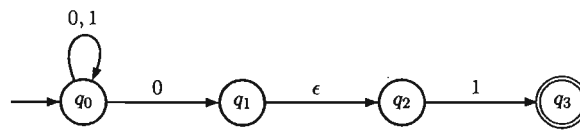


Figura 4.3: AFN con transiciones-ϵ que acepta todas las cadenas que terminan en 01

El autómata contiene un arco etiquetado con ϵ que sale del estado q_1 al estado q_2. De modo que cuando el autómata se encuentre en el estado q_1 se transfiere automáticamente al estado q_2 sin tener que leer un símbolo de la cadena de entrada.

4.4. Expresiones regulares

Los lenguajes aceptados por autómatas finitos pueden ser descritos por expresiones simples llamadas *expresiones regulares*. Las expresiones regulares tienen una gran importancia en aplicaciones que involucran búsquedas en texto, pues se puede estar buscando una cadena que satisface ciertos patrones. Las expresiones regulares brindan métodos para describir dichos patrones. Programas como AWK, GREP, PERL y editores de texto proveen mecanismos para la descripción de patrones utilizando expresiones regulares.

Las expresiones regulares sobre Σ y los conjuntos que denotan se definen recursivamente como sigue:

Sea Σ un alfabeto.

1. $\emptyset$ es una expresión regular y denota el conjunto vacío.
2. ϵ es una expresión regular y denota el conjunto $\{\epsilon\}$.
3. Por cada $a \in \Sigma$, $\boldsymbol{a}$ es una expresión regular que denota el conjunto $\{a\}$.
4. Si r y s son expresiones regulares que denotan a los conjuntos de cadenas R y S respectivamente, entonces $(r+s)$, (rs) y (r^*) son expresiones regulares que denotan los conjuntos $R \cup S$, RS, R^* respectivamente.
5. Éstas y sólo éstas son expresiones regulares.

Dos expresiones regulares son equivalentes si y sólo si describen al mismo conjunto de cadenas. Esto se puede demostrar por inducción sobre el número de operadores de la expresión regular. Derivadas de estas definiciones se pueden establecer ciertas equivalencias entre expresiones regulares y algunas propiedades como son el hecho de que el reverso de una expresión regular es una expresión regular.

Ejemplo 4.4.1. Un ejemplo de una expresión regular es

$$(0 \cup 1)^*$$

El valor de esta expresión es el lenguaje que consiste de todas las posibles cadenas de 0's y 1's.

4.4.1. Equivalencia entre autómatas finitos y expresiones regulares

Los lenguajes aceptados por autómatas finitos son los lenguajes denotados por expresiones regulares. Esta equivalencia es la que permite llamar a los lenguajes aceptados por autómatas finitos, lenguajes regulares.

4.4.2. Construcción de autómatas finitos a partir de expresiones regulares

Para cada expresión regular se puede construir un AFN M con transiciones-ϵ equivalente. El AFN que se construye tiene un solo estado inicial al que no llegan transiciones y un solo estado final del cual no salen transiciones.

A continuación se mostrará cómo convertir una expresión regular en un AFN que reconoce un cierto lenguaje.

Sea r una expresión regular; entonces los AFN que se presentan a continuación aceptan $L(r)$.

Cuando la expresión regular contiene cero operadores, entonces la expresión regular debe ser $\epsilon, \emptyset$, o a para algún símbolo $a \in \Sigma$. La construcción de los autómatas que aceptan a estas expresiones regulares se pueden ver en la figura 4.4.

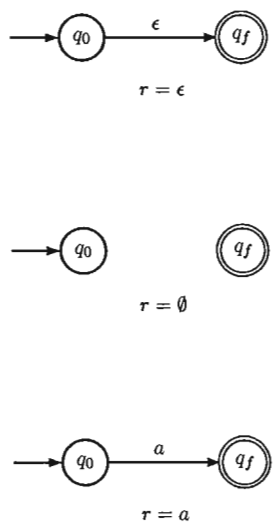


Figura 4.4: Autómatas para expresiones regulares sin operadores

Si la expresión regular tiene uno o más operadores entonces hay tres casos, dependiendo de la forma de r:

Caso 1. $r = r_1 r_2$.
Sean $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, q_{f_1})$ y $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, q_{f_2})$ dos autómatas que aceptan a

r_1 y r_2 respectivamente, cada uno de ellos con un único estado inicial y un único estado final del que no salen transiciones.

Se construye un AFN $M' = (Q', \Sigma', \delta', q_1, q_{f_2'})$ con transiciones-ϵ de la siguiente manera:

- El estado inicial del nuevo autómata es el estado inicial de M_1.
- El estado final de M' es el estado final de M_2.
- $Q' = Q_1 \cup Q_2$.
  El conjunto de estados del nuevo autómata es la unión de los estados de los autómatas a concatenar.
- $\Sigma' = \Sigma_1 \cup \Sigma_2$.
  El alfabeto de entrada es la unión de ambos alfabeto.
- $\delta' = \delta_1 \cup \delta_2 \cup \{\delta(q_{f_1}, \epsilon) \supseteq q_2\}$

Se sigue cumpliendo la condición de tener un único estado inicial (q_1) y un único estado final (q_{f_2}). Además, al estado inicial no llega ninguna transición y del estado final no sale ninguna transición.

Si $\epsilon \in r_1 r_2$ significa que estaba en ambas expresiones regulares. Entonces, en cada uno de los autómatas hay un camino etiquetado con-ϵ, que lleva del estado inicial al estado final correspondiente. Dado que se agregó una transición-ϵ desde q_{f_1} hacia q_2, es posible juntar los dos caminos en un solo camino, dentro del nuevo autómata, que vaya desde el estado inicial q_1 hasta el estado final q_{f_2}de modo que ϵ se encuentra en la concatenación.

La construcción de los autómatas que aceptan la concatenación de expresiones regulares se puede ver en la figura 4.5.

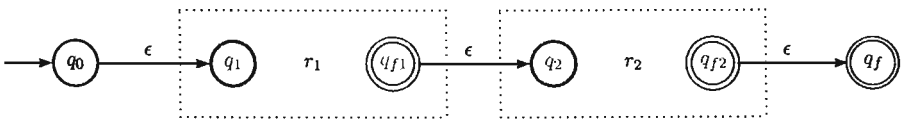


Figura 4.5: Concatenación de expresiones regulares

Caso 2. $r = r_1 + r_2$.
Sean $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, q_{f_1})$ y $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, q_{f_2})$ dos autómatas que aceptan a r_1 y r_2 respectivamente, cada uno de ellos con un único estado inicial y un único estado final del que no salen transiciones.

Se construye un AFN $M' = (Q', \Sigma', \delta', q_0, q_{f'})$ con transiciones-ϵ de la siguiente manera:

- $Q' = Q_1 \cup Q_2 \cup \{q_0\} \cup \{q_{f'}\}$.
  Se agrega un nuevo estado inicial y un nuevo estado final.

- $\Sigma = \Sigma_1 \cup \Sigma_2$.
  El alfabeto de entrada es la unión de ambos alfabeto.

- $\delta' = \delta_1 \cup \delta_2 \cup \{\delta(q_0, \epsilon) = \{q_1, q_2\}$
  $\delta(q_{f_1}, \epsilon) = q_{f'}$
  $\delta(q_{f_2}, \epsilon) = q_{f'}\}$

Se agregan transiciones-ϵ desde el estado inicial de M' hacia los estados iniciales de M_1y M_2, desde los estados finales de M_1y M_2 hacia el nuevo estado final de M'; esto garantiza que el nuevo autómata tiene un único estado inicial y un único estado final del cual no salen transiciones.

La construcción de los autómatas que aceptan la unión de expresiones regulares se puede ver en la figura 4.6.

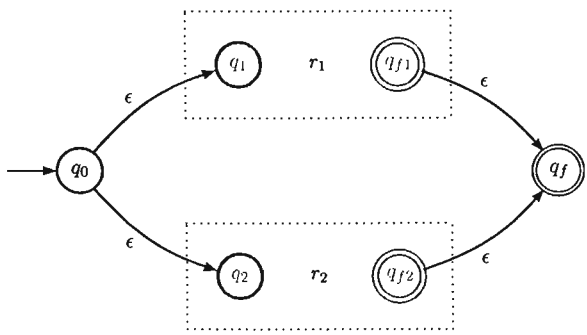


Figura 4.6: Unión de expresiones regulares

Caso 3. $r = r^*$.
Sea $M = (Q_1, \Sigma_1, \delta_1, q_1, q_{f_1})$. Es necesario agregar un nuevo estado inicial q_0 con una transición-ϵ hacia el estado inicial, q_1, original. De igual forma, es necesario agregar un nuevo estado final, q_f, con una transición-ϵ del estado final original, q_{f_1}, al nuevo estado final. También se agrega una transición-ϵ del estado final original al estado inicial original.

Para aceptar ϵ, se agrega una transición-ϵ desde el nuevo estado inicial hacia el nuevo estado final.

La construcción de los autómatas que aceptan la cerradura de Kleene de expresiones regulares se puede ver en la figura 4.7.

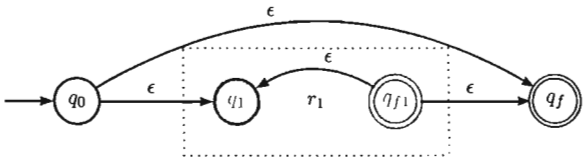


Figura 4.7: Cerradura de Kleene de expresiones regulares

4.4.3. Aplicaciones de autómatas finitos

Editores de texto

Algunos editores de texto permiten la sustitución de una cadena por otra cualquiera que se aparee con una expresión regular dada. Un ejemplo de ello son los editores de texto de los sistemas UNIX, como *gedit* o *emacs*, que sustituye un blanco único por la primera cadena de dos o más blancos encontrados en una línea dada. Se podría convertir una expresión regular r a un AFD que acepte cualquier r; sin embargo, la conversión de una expresión regular a un AFD toma más tiempo que el que tomaría examinar una línea corta usando al AFD; además el AFD podría tener un número de estados que son una función exponencial de la longitud de la expresión regular.

Lo que realmente sucede en éstos editores es que la expresión regular r es convertida a un AFN con transiciones-ϵ y luego el AFN es simulado directamente. Sin embargo, una vez que una columna ha sido construida enumerando todos los estado que el AFN puede alcanzar con un prefijo particular de la entrada, la columna anterior ya no es necesaria y se descarta para ahorrar espacio.

Analizadores léxicos

Una de las aplicaciones de los autómatas finitos y las expresiones regulares es especificar el analizador léxico de un compilador. Dicho componente revisa un programa fuente y reconoce todas las subcadenas de caracteres consecutivos que pertenecen a una misma categoría lógica llamados *símbolos terminales.*

El programa *lex (Lexical Analyzer Generator)* de UNIX acepta como entrada una lista de expresiones regulares, seguidas por una sección de código que indica lo que debe hacer el analizador léxico cuando encuentre un símbolo terminal. Utiliza el proceso de conversión entre expresiones regulares y autómatas finitos para generar un función eficiente que separe código fuente en símbolos terminales.

El conjunto de expresiones se convierte en un AFN con transiciones-ϵ que es capaz de decir qué símbolos terminales se van a reconocer.

4.5. Propiedades de los lenguajes regulares

Existen muchas preguntas que uno puede hacerse respecto a los lenguajes regulares. Quisiéramos saber cuándo los lenguajes regulares denotados por diferentes expresiones son iguales, o encontrar el autómata finito con el mínimo número de estados que denota el mismo lenguaje que un AFD dado.

A continuación se presentan algunas de las propiedades más importantes de los AF y que resuelven dichas preguntas.

4.5.1. Propiedades de cerradura

Existen muchas operaciones con lenguajes que preservan conjuntos regulares, en el sentido de que operaciones aplicadas a conjuntos regulares dan como resultado conjuntos regulares, y que se derivan de la definición de expresiones regulares y su equivalencia con aquéllos. Entre ellas se tienen la operación de unión, concatenación y cerradura de Kleene de los lenguajes regulares.

4.5.2. Operaciones booleanas

Propiedad La clase de los lenguajes regulares es cerrada bajo complemento.

Propiedad Si L es un lenguaje regular y $L \subseteq \Sigma^*$, entonces $\Sigma^* - L$ es un lenguaje regular.

Propiedad Los lenguajes regulares son cerrados bajo intersección.

4.5.3. Sustituciones e isomorfismos

Propiedad La clase de lenguajes regulares tiene la propiedad de cerradura bajo sustitución. Un tipo de sustitución interesante es el isomorfismo.

Sean $M = (Q, \Sigma, \delta, q_0, F)$ y $M' = (Q', \Sigma', \delta', q_{0'}, F')$ dos autómatas finitos.
Entonces, M y M' son *isomorfos* si y sólo si existe un mapeo $h : Q \rightarrow Q'$ tal que:

- $h(q_0) = q_{0'}$
- $\forall q \in Q, \forall a \in \Sigma, \delta'(h(q), a) = h(\delta(q, a))$
- $\forall q \in Q, h(q) \in F'$si y solo si$q \in F$

De modo que la clase de lenguajes regulares es cerrada bajo isomorfismos.

4.5.4. Cociente de un lenguaje

Se define el *cociente* entre dos lenguajes L_1 y L_2 (L_1/L_2) como:

$$\{x | \exists y \in L_2 \, tal \, que \, xy \in L_1\}$$

Propiedad La clase de conjuntos regulares es cerrada bajo cociente con conjuntos arbitrarios.

4.5.5. Algoritmos de decisión para lenguajes regulares

Definiremos las condiciones bajo las cuales un lenguaje aceptado por un autómata finito es vacío, finito o infinito y la equivalencia entre conjuntos regulares.

Vacuidad, finitud e infinitud

El conjunto de cadenas aceptadas por un autómata finito M con n estados es:

1. No vacío si y sólo si el autómata acepta a una cadena de longitud menor que n.
2. Infinito si y sólo si el autómata acepta alguna cadena de longitud m, donde $n \leq m < 2n$.

De modo que podemos asegurar que existe un algoritmo para determinar si un autómata finito acepta cero, un número finito o un número infinito de cadenas.

Un AFD acepta a un conjunto vacío si al tomar su diagrama de transiciones y eliminar todos los estados que no se pueden alcanzar sobre cualquier entrada desde el estado inicial permanecen uno o más estados finales, entonces el lenguaje aceptado por el autómata es no vacío. De modo que es posible eliminar los estados que no son finales y de los cuales no se puede alcanzar un estado final sin cambiar el lenguaje aceptado por el autómata. El AFD acepta un lenguaje infinito si y sólo si el diagrama de transiciones resultante tiene un ciclo.

Equivalencia

Existe un algoritmo para determinar si dos autómatas finitos son equivalentes, es decir, si aceptan el mismo lenguaje. Para esto, es necesario detectar y eliminar estados redundantes en el diagrama de transición del autómata sin alterar su comportamiento básico y obtener un autómata mínimo y único que sea equivalente al original.

La eliminación de estados redundantes es importante en un autómata pues el costo de implementación es proporcional al número de estados.

Un estado es redundante si existe otro estado en el autómata que muestra el mismo comportamiento frente a toda cadena de entrada. Esto es, si partiendo de cada uno de

los estados de los autómatas y después de leer una cadena w, $\forall w \in \Sigma^*$, se llega a un estado final o no final en ambos casos. Para poder decidir esto es necesario seguir el comportamiento del autómata. Para ello definimos una función de salida F:

$$F(q, \epsilon) = \begin{cases} 0 & q \notin F \\ 1 & q \in F \end{cases}$$

$$F(q, aw) = F(q, \epsilon) \cdot F(p, w) \text{ donde } p = \delta(q, a)$$

Dos estados p y q de un autómata finito M son distinguibles si y sólo si existe una cadena de entrada, de longitud finita, tal que cuando se le aplica a M, produce diferentes respuestas.

Formalmente:
Sea $M = (Q, \Sigma, \delta, q_0, F)$ tal que q_i y q_j están en Q.
Entonces q_i y q_j son distinguibles si y sólo si $\exists w \in \Sigma^*$ tal que $F(q_i, w) \neq F(q_j, w)$; decimos además que w distingue a q_i de q_j.

De manera similar, q_i y q_j son k-distinguibles si:

1. existe una cadena w de longitud k que distingue a q_i y q_j
2. $\forall x \in \Sigma^*$, con $|x| = m$, $m < k$, x no distingue a q_i de q_j.

Entonces, dos estados son equivalentes si son k-distinguibles para toda k. De este concepto definimos la equivalencia de la siguiente forma:
Dos autómatas finitos $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ y $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$ son equivalentes si y sólo si:

1. $\Sigma_1 = \Sigma_2$.
2. q_1 y q_2 son estados equivalentes.

Las relaciones de equivalencia inducen en su dominio una partición en clases ajenas llamadas "*clases de equivalencia*" de modo que uno de los estados está en una y solo una de esas clases.

4.5.6. Minimización de autómatas finitos

Para minimizar un autómata finito lo que se busca es una partición de clases ajenas de manera tal que queden en una misma clase los estados equivalentes entre sí. Dada esta partición se puede tomar a un representante de cada una de las clases y con dichos representantes construir el autómata mínimo.

Decimos que dos estados q y p son equivalentes si se cumple:

1. Condición de compatibilidad: los estados p y q deberán ser al mismo tiempo estados que aceptan o estados que rechazan, es decir, p y q deben ser 0-equivalentes.

2. Condición de propagación: para todo símbolo de entrada, p y q deben transferirse a estados equivalentes.

Entonces p y q son estados equivalentes si y sólo si no son distinguibles.
Existe un único AFD con el mínimo número de estados para cada conjunto regular, aunque pudiera haber otros AFD mínimos pero isomorfos a éste y en el que se pueden renombrar los estados.

Algoritmo de minimización

1. Dar una primera partición de los estados del autómata finito en dos clases de equivalencia: la clase de los estados que aceptan y la clase de los estados que no aceptan.

2. Dada una partición P_i, a partir de ella se obtiene P_{i+1} de la siguiente manera: p y q están en un mismo bloque en P_{i+1} si y sólo si

   a) p y q están en un mismo bloque en P_i.
   b) $\forall a \in \Sigma$, $\delta(p,a)$, $\delta(q,a)$ están en un mismo bloque en P_i.

3. El proceso termina cuando $P_i = P_{i+1}$.

Una vez definida la partición final del autómata, se construye un nuevo autómata de la siguiente manera:

1. Se asigna a cada bloque de la partición un nuevo estado en el nuevo autómata.

2. Se construye la tabla de transiciones con los símbolos para las columnas y los nuevos estados para los renglones.

3. El nuevo estado inicial es el que corresponde al bloque que contenga al estado inicial del autómata original.

4. Son estados finales los correspondientes a todos aquellos bloques que consistan de estados finales del autómata original.

5. La función de transición del nuevo autómata está definida como sigue: $\delta'(B_i, a) = B_j$ en el nuevo autómata si y sólo si

   a) $q \in B_i$ y $p \in B_j$
   b) $\delta(q,a) = p$

La partición obtenida al final del algoritmo es única y la demostración se puede verificar en [9].

Además de estados redundantes, en el sentido de que su función es llevada a cabo por algún otro estado, el autómata puede tener estados inalcanzables. El procedimiento para detectar estados inalcanzables consiste en elaborar una lista de estados alcanzables desde el estado inicial como sigue:

1. Construir la lista con el estado inicial.
2. Agregar a la lista todos los estados que pueden ser alcanzados desde el estado inicial con cadenas de longitud 1.
3. Para cada nuevo estado en la lista, agregar los estados alcanzables desde aquél bajo cadenas de un símbolo y que no se encuentran ya en la lista.

El procedimiento termina cuando no se puede agregar a la lista ningún nuevo estado.

Un autómata finito es *reducido* si no contiene estados equivalentes. Un autómata finito es *conectado* si todos sus estados son alcanzables desde el estado inicial. Un autómata finito es *fuertemente conectado* si todo estado en el autómata es alcanzable desde cualquier otro estado.

En el siguiente capítulo se describirán las distancias utilizadas más frecuentemente para la medición del número de errores permitidos en una cadena.

Capítulo 5

Distancias para la detección de errores en texto

Frecuentemente se desea medir la diferencia o distancia entre dos cadenas. Existen muchas maneras de formalizar la noción de distancia entre cadenas, la más común consiste en transformar una cadena en otra mediante una serie de operaciones sobre caracteres individuales.

El problema de detección de errores en una cadena de entrada x que debió ser y consiste en transformar x en y llevando a cabo una serie de operaciones de edición sobre x. Dichas operaciones de edición pueden ser inserción, eliminación, reemplazo y transposición.

El número de errores permitidos en una subcadena está determinado por una distancia $d(x, y)$ que se define como el mínimo número de operaciones de edición necesarias para convertir al patrón x en la subcadena y.

Las medidas de distancia satisfacen los axiomas matemáticos de métrica, de manera que desde el punto de vista formal cumplen con las siguientes propiedades:

Sean x, y, z elementos cualesquiera dentro de un campo.

- Positiva definida.

$$d(x, y) \geq 0$$

$$d(x, y) = 0 \text{ si y sólo si } x = y.$$

- Simétrica.

$$d(x, y) = d(y, x).$$

- Desigualdad del triángulo.

$$d(x,z) \leq d(x,y) + d(y,z).$$

En cuanto a manejo de texto, podemos definir la distancia entre dos cadenas de diversas maneras.

Se consideran cuatro operaciones de edición sobre x[1]:

- Eliminación: $d(a,\epsilon)$ consiste en borrar un símbolo a de la cadena.
- Inserción: $d(\epsilon,a)$ consiste en agregar un nuevo símbolo a en la cadena.
- Reemplazo: $d(a,b), a \neq b$ consiste en reemplazar un símbolo a de la cadena por el símbolo b.
- Transposición: $d(ab,ba), a \neq b$ consiste en intercambiar dos símbolos adyacentes a,b de la cadena. Esta operación representa la situación en la cual dos caracteres se encuentran en orden inverso.

De este modo es posible definir las funciones de distancia utilizadas más frecuentemente para la edición de texto.

5.1. Distancia de Hamming

La distancia de Hamming se define como el número de posiciones en dos cadenas de igual longitud para las cuales los elementos correspondientes son diferentes. Es decir, mide el número de sustituciones necesarias para convertir una cadena en otra.

Resulta conveniente pensar en la distancia de Hamming entre dos cadenas como el resultado de aplicar la operación de disyunción exclusiva entre las dos cadenas, la operación bit a bit conocida como OR exclusivo o XOR.

Ejemplo 5.1.1. La distancia de Hamming entre las cadenas "*TRABAJO*" y "*PRIMATE*" es 5 dado que ambas cadenas difieren en las posiciones $\{0,2,3,5,6\}$.

```
T R A B A J O
P R I M A T E
- - - - - - -
1 0 1 1 0 1 1
```

[1]Las coincidencias entre las cadenas no son tomadas en cuenta.

Esta distancia recibe el nombre de su autor Richard Hamming. Se utiliza en telecomunicaciones para contar el número de bits intercambiados en una palabra binaria. Esta distancia es útil en algunos casos, pero en general no es suficientemente flexible debido a que las cadenas deben tener la misma longitud y generalmente no existe una correspondencia fija entre la posición de sus caracteres; un desplazamiento o la inserción errónea de caracteres en alguna de las cadenas suele exagerar los valores en la distancia de Hamming.

5.2. Distancia de Levenshtein

La distancia de Levenshtein entre dos cadenas está dada por el mínimo número de operaciones necesarias para transformar una cadena en otra, donde las operaciones permitidas son inserción, reemplazo y eliminación.

Ejemplo 5.2.1. Denotemos con I a la operación de inserción, E denota la operación de eliminación, R denota la operación de reemplazo y C denota una coincidencia; entonces la distancia de Levenshtein entre las cadenas "*TRABAJO*" y "*PASAJERO* es 5, dado que son necesarios cinco cambios para transformar una cadena en la otra y no hay forma de hacerlo en menos ediciones.

```
T R A B A J    O
  P A S A J E R O
- - - - - - - - -
I R C R C C E E C
```

A la transformación de una cadena en otra se llama *transcripción* de las dos cadenas.

El problema de edición de cadenas consiste en calcular la distancia entre dos cadenas mediante una transcripción óptima.

En general, dadas dos cadenas de entrada S_1 y S_2 y dada una transcripción para S_1 y S_2, la transformación se realiza aplicando sucesivamente la operación especificada en la transcripción al siguiente carácter de la cadena. La transcripción se aplica de izquierda a derecha.

La definición de distancia implica que todas las operaciones se realizan sobre una cadena solamente; aunque algunas veces también puede ser pensada como el mínimo número de operaciones realizadas sobre ambas cadenas, para transformarlas en una tercera cadena común. Este punto de vista es equivalente, dado que una operación de inserción en una cadena puede ser vista como una operación de eliminación en la otra cadena y viceversa.

La distancia de Levenshtein es considerada como una generalización de la distancia de Hamming y recibe su nombre de su autor Vladimir Levenshtein, quien introdujera el término en 1966 con códigos correctores de errores.

La distancia de Levenshtein se utiliza en revisión de archivos, corrección de palabras, reconocimiento de lenguaje, análisis de ADN, detección de intrusos en el ámbito de seguridad de sistemas, entre otras aplicaciones.

5.3. Distancia generalizada de Levenshtein

En la distancia generalizada de Levenshtein se utiliza, además de las operaciones de inserción, reemplazo y eliminación, la operación de transposición en la cual dos símbolos adyacentes del patrón P, $p_i p_{i+1}$, $0 < i < m$, donde m es la longitud del patrón, son acomodados en la cadena en orden inverso, es decir, $p_{i+1} p_i$.

La distancia generalizada de Levenshtein se utiliza en biología para encontrar secuencias similares de ácidos nucleicos en ADN o aminoácidos en proteínas; en correctores de ortografía cuando se encuentra una palabra desconocida y para estimar la proximidad en la pronunciación de un dialecto.

Ejemplo 5.3.1. Denotemos con I a la operación de inserción, E denota la operación de eliminación, R denota la operación de reemplazo, C denota una coincidencia y R denota la operación de transposición; entonces la distancia generalizada de Levenshtein entre las cadenas "*TRABAJO*" y "*CARACOL*" es 5, dado que son necesarios cinco cambios para transformar una cadena en la otra y no hay forma de hacerlo en menos ediciones.

```
T R A B A J O
C A R   A C O L
- - - - - - - -
R T C I C R C E
```

Ahora entraremos al tema central de este trabajo: el apareamiento de cadenas con autómatas finitos.

Capítulo 6

Autómatas para apareamiento de patrones

El área de los algoritmos basados en autómatas es muy conocida. Resulta interesante porque proporciona en el peor caso el mejor tiempo algorítmico ($O(n)$ donde n es la longitud del texto), lo cual resulta ser la cota mínima del problema.

Sin embargo, el espacio y tiempo dependen exponencialmente de la longitud del patrón y el número de errores permitidos, lo que limita, en la práctica, la aplicación de éstos algoritmos.

Una forma alternativa y práctica de considerar el problema es modelar la búsqueda con un autómata finito no determinístico (AFN).

6.1. Autómatas y apareamiento de patrones sobre cadenas

Todos los problemas de apareamiento de patrones de una dimensión son problemas secuenciales, por lo que es posible resolverlos utilizando un autómata finito no determinístico. Hay tres formas en las que los autómatas pueden ser utilizados:

1. Como modelos de algoritmos para resolver diferentes problemas.
2. Para simular el autómata finito no determinístico de manera determinística.
3. Para construir un autómata finito determinístico equivalente.

El uso de autómatas finitos para modelar algoritmos de apareamiento de patrones proporciona un método formal en esta área de las ciencias de la computación, lo que permite describir el problema utilizando una visión unificada. Entre las consecuencias de la formalización de los autómatas finitos están: la construcción de algoritmos para el

problema de apareamiento de patrones con un tiempo lineal de complejidad y con un espacio de complejidad que depende del problema particular.

Los AFN buscan la primera presencia del patrón P en un texto T y pueden continuar la búsqueda de todas la ocurrencias del patrón en el texto; de tal modo que los autómatas pueden determinar:

- La primera presencia del patrón en el texto.
- El número de presencias del patrón en el texto.
- Todas las presencias del patrón en el texto.

También se pueden encontrar todas las presencias traslapadas del patrón.

Por ejemplo para el texto $T = abbb$ y el patrón $P = bb$ se daría como respuesta que el patrón ha sido encontrado dos veces ($a\underline{bb}b, ab\underline{bb}$). También es posible encontrar las presencias del patrón que no están traslapadas; esto se logra haciendo que el autómata se reinicie una vez que ha alcanzado un estado final y sólo el estado inicial permanece activo; de esta forma el autómata "olvida" toda la información sobre los prefijos del patrón encontrados y no puede encontrar ninguna presencia del patrón traslapada con la presencia actual.

El autómata finito no determinístico acepta una cadena de entrada si y sólo si existe un camino desde el estado inicial hasta un estado final que acepte dicha cadena de entrada. En algunos estados del AFN existe más de una posibilidad de transición; en tales casos el autómata recorre todos los diferentes caminos posibles al mismo tiempo; sin embargo, en la práctica no se da la simultaneidad.
Si se construye el AFN para resolver el problema de apareamiento de cadenas no es posible utilizar el autómata directamente dado el no determinismo del AFN. Una manera de utilizar el AFN es transformándolo a su equivalente autómata finito determinístico (AFD), pero dicho AFD tiene $O(2^m)$ estados, donde m es el número de estados del AFN. Otra forma de utilizar el AFN es simulando su comportamiento no determinístico.
Se utilizarán dos versiones de AFN para el apareamiento aproximado de cadenas, una donde las operaciones de edición de inserción y sustitución están etiquetadas con el complemento de un símbolo y otra en el que están etiquetadas con el alfabeto completo. La segunda versión simplifica la forma de simulación mencionada, mientras que el comportamiento del AFN prácticamente no cambia.

6.2. Apareamiento perfecto de cadenas

En el apareamiento perfecto de cadenas se buscan todas las ocurrencias del patrón $P = p_1 p_2 \dots p_m$ en el texto $T = t_1 t_2 \dots t_n$. De modo que lo que se desea es construir un AFN M que acepte al lenguaje $L(M) = \{wPy \mid w, y \in \Sigma^*\}$.

En el autómata cada estado q_i, representa que los primeros i símbolos del patrón han sido encontrados en el texto. Para saltar los símbolos frente a una ocurrencia del patrón se agrega un ciclo en el estado inicial por cada símbolo del alfabeto Σ. La construcción del autómata se presenta en el algoritmo 1.

Algoritmo 1 Construcción del AFN para el apareamiento perfecto de cadenas

Entrada: Patrón $P = p_1 p_2 \dots p_m$.

Salida: AFN M que acepta el lenguaje $L(M) = \{wPy \mid w, y \in \Sigma^*\}$.

Método: AFN $M = (\{q_0, q_1, \dots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$, donde la función de mapeo δ se construye de la siguiente manera:

$\delta(q_0, a) \leftarrow \{q_0\}, \forall a \in \Sigma \setminus \{p_1\}$ /* ciclo del estado inicial */
$\delta(q_0, p_1) \leftarrow \{q_0, q_1\}$
for $i \leftarrow 1, 2, \dots, m-1$ **do**
  $\delta(q_i, p_{i+1}) \leftarrow \{q_{i+1}\}$ /* transiciones hacia adelante */
end for
$\delta(q_m, a) \leftarrow \emptyset, \forall a \in \Sigma$ /* no hay transiciones definidas para el estado final */

Un ejemplo de dicho autómata para $m = 4$ se muestra en la figura 6.1.

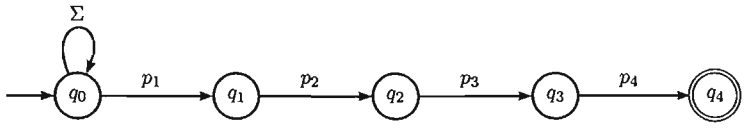


Figura 6.1: AFN para el apareamiento perfecto de cadenas

Lema 6.2.1. *El autómata mínimo AFN para el apareamiento perfecto de cadenas tiene* $(m+1)$ *estados, donde* m *es la longitud del patrón buscado* $P = p_1 p_2 \dots p_m$.

Demostración. Ver el algoritmo 1. □

El autómata finito determinístico para el apareamiento perfecto de cadenas puede ser construido utilizando la propiedad de los autómatas finitos de equivalencia entre AFN y AFD o bien puede ser construido directamente en tiempo $O(m|\Sigma|)$ como se muestra en el algoritmo 2. En este algoritmo, el estado r en el i-ésimo paso es el estado que es alcanzado desde el estado inicial y leyendo la cadena w (es decir, $(q_0, w) \vdash^* (r, \epsilon)$) que es el prefijo más largo de $p_1 p_2 \dots p_{i-1}$ y al mismo tiempo es un sufijo de $p_1 p_2 \dots p_i$.

Ejemplo 6.2.2. Sea $P = aba$. El AFD para el apareamiento perfecto de una ocurrencia de P en un texto dado se muestra en la figura 6.2

Algoritmo 2 Construcción del AFD para el apareamiento perfecto de cadenas

Entrada: Patrón $P = p_1 p_2 \ldots p_m$.

Salida: AFD M que acepta el lenguaje $L(M) = \{wPy \mid w, y \in \Sigma^*\}$.

Método: AFD $M = (\{q_0, q_1, \ldots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$, donde la función de mapeo δ se construye de la siguiente manera:

```
for a ∈ Σ do
    δ(q0, a) ← q0                          /* ciclo del estado inicial */
end for
for i ← 1, 2, ..., m do
    r ← δ(q_{i-1}, p_i)
    δ(q_{i-1}, p_i) ← {q_i}                /* transiciones hacia adelante */
    for a ∈ Σ do
        δ(q_i, a) ← δ(r, a)
    end for
end for
```

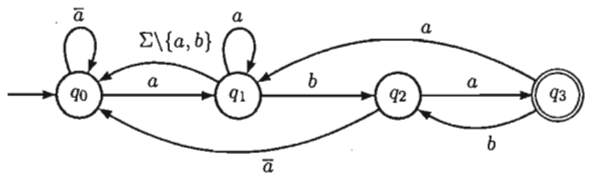


Figura 6.2: Ejemplo de un AFD para $P = aba$

6.3. Apareamiento aproximado de cadenas utilizando la distancia de Hamming

En el apareamiento de cadenas utilizando la distancia de Hamming para el patrón $P = p_1 p_2 ... p_m$ se debe considerar el máximo número k de errores permitidos. De esta manera se utilizan $k+1$ copias del autómata construido para el mapeo exacto de cadenas $(M_0, M_1, ..., M_k)$ donde un nivel es utilizado para $k = 0$, es decir, cuando no se permiten errores (nivel 0); un nivel para cuando se permite 1 error (nivel 1); ...; un nivel para cuando se permiten k errores (nivel k).

El $(k+1)$ - autómata está conectado por transiciones que representan la operación de *reemplazo*. Cada transición está etiquetada por el símbolo $\overline{p}_{j+1}$ (el error del símbolo p_{j+1} en el patrón P) y direcciona desde el estado p_j del autómata M_i al estado q_{j+1} del autómata M_{i+1}, $0 \leq i < k$,$0 \leq j < m$. La profundidad del autómata se incrementa tanto

como el mínimo número de errores. El estado inicial del $k+1$ - autómata es el estado inicial del autómata M_0.

El autómata puede ser construido conectando $k+1$ copias del autómata descrito o directamente usando el algoritmo 3.

Algoritmo 3 Construcción del AFN para el apareamiento aproximado de cadenas utilizando la distancia de Hamming (versión $\overline{p}$)

Entrada: Patrón $P = p_1 p_2 \ldots p_m$, k máximo número de errores permitidos, $k < m$.
Salida: AFN M que acepta el lenguaje $L(M) = \{wu \mid w, u \in \Sigma^*, D_H(P,u) \leq k\}$.
Método: AFN $M = (Q, \Sigma, \delta, q_0, F)$, donde Q, δ, F se construyen de la siguiente manera:

$Q \leftarrow \{q_0, q_1, \ldots, q_{|Q|-1}\}, \mid Q \mid = (k+1)\left(m+1-\frac{k}{2}\right)$
$l \leftarrow 0$ /* número de nivel */
$r \leftarrow 1$ /* profundidad del estado */
for $i \leftarrow 0, 1, \ldots, \mid Q \mid -1$ **do**
  if $r > m$ **then**
    $\delta(q_i, a) \leftarrow \emptyset, \forall a \in \Sigma$
    $F \leftarrow F \bigcup \{q_i\}$ /* estado final */
    $l \leftarrow l+1; r \leftarrow l+1$ /* ir al primer estado del siguiente nivel */
  else
    $\delta(q_i, p_r) \leftarrow \{q_{i+1}\}$
    if $l < k$ **then**
      $s \leftarrow i+m+1-l$ /* s = número del vecino menor derecho */
      $\delta(q_i, a) \leftarrow \{q_s\}, \forall a \in \Sigma \setminus \{p_r\}$ /* transición de reemplazo */
    end if
    $r \leftarrow r+1$ /* siguiente profundidad del nivel l */
  end if
end for
$\delta(q_0, a) \leftarrow \delta(q_0, q) \bigcup \{q_0\}, \forall a \in \Sigma$ /* ciclo del estado inicial */

Un ejemplo de dicho autómata para $m = 4$ y $k = 3$ se muestra en la figura 6.3.

Existe otra posibilidad para representar la transición de *reemplazo*. Se puede etiquetar la transición de *reemplazo* mediante Σ en lugar de $\overline{p_i}$. Dicho autómata puede ser construido conectando $k+1$ copias del autómata descrito o directamente usando el algoritmo 4, donde la transición de *reemplazo* está modificada.

Un ejemplo de dicho autómata para $m = 4$ y $k = 3$ se muestra en la figura 6.4.

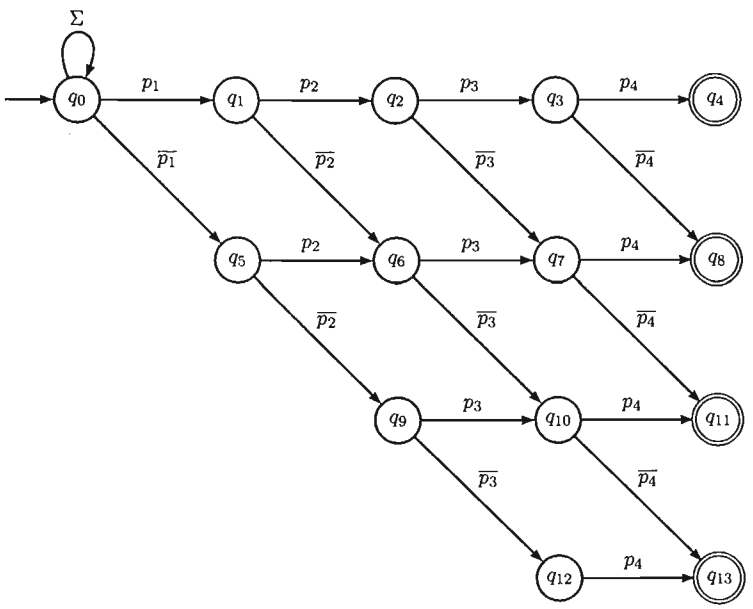


Figura 6.3: AFN para el apareamiento aproximado de cadenas utilizando la distancia de Hamming - versión $\bar{p}$ $(m = 4, k = 3)$

Algoritmo 4 Construcción del AFN para el apareamiento aproximado de cadenas utilizando la distancia de Hamming (versión Σ)

Entrada: Patrón $P = p_1 p_2 \ldots p_m$, k máximo número de errores permitidos, $k < m$.
Salida: AFN M que acepta el lenguaje $L(M) = \{wu \mid w, u \in \Sigma^*, D_H(P, u) \leq k\}$.
Método: AFN $M = (Q, \Sigma, \delta, q_0, F)$, donde Q, δ, F se construyen de la siguiente manera:

$Q \leftarrow \{q_0, q_1, \ldots, q_{|Q|-1}\}, \mid Q \mid = (k+1)\left(m + 1 - \frac{k}{2}\right)$
$l \leftarrow 0$ /* número de nivel */
$r \leftarrow 1$ /* profundidad del estado */
for $i \leftarrow 0, 1, \ldots, \mid Q \mid -1$ **do**
  if $r > m$ **then**
    $\delta(q_i, a) \leftarrow \emptyset, \forall a \in \Sigma$
    $F \leftarrow F \bigcup \{q_i\}$ /* estado final */
    $l \leftarrow l + 1; r \leftarrow l + 1$ /* ir al primer estado del siguiente nivel */
  else
    if $l < k$ **then**
      $s \leftarrow i + m + 1 - l$ /* s =número del vecino menor derecho */
      $\delta(q_i, p_r) \leftarrow \{q_{i+1}, q_s\}$ /* transición de apareo */
      $\delta(q_i, a) \leftarrow \{q_s\}, \forall a \in \Sigma \setminus \{p_r\}$ /* transición de reemplazo */
    else
      $\delta(q_i, p_r) \leftarrow \{q_{i+1}\}$ /*transición de apareo en el k – *ésimo* nivel */
    end if
    $r \leftarrow r + 1$ /* siguiente profundidad del nivel l */
  end if
end for
$\delta(q_0, a) \leftarrow \delta(q_0, q) \bigcup \{q_0\}, \forall a \in \Sigma$ /* ciclo del estado inicial */

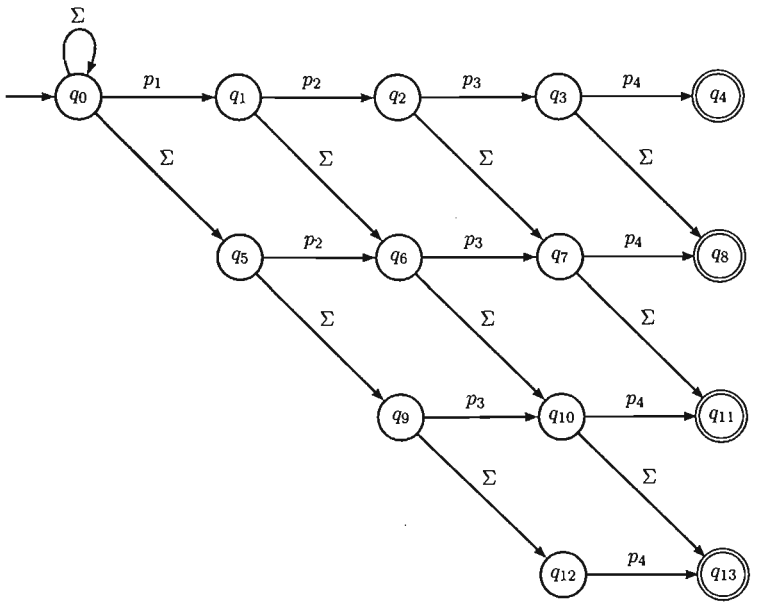


Figura 6.4: AFN para el apareamiento aproximado de cadenas utilizando la distancia de Hamming - versión Σ ($m = 4, k = 3$)

Lema 6.3.1. *El AFN para el apareamiento aproximado de cadenas utilizando la distancia de Hamming, construido mediante el algoritmo 3 y el algoritmo 4 para un patrón de longitud m, y permitiendo un máximo número de errores k, tiene* $(k+1)\left(m+1-\frac{k}{2}\right)$ *estados.*

Demostración. Se construyen $k+1$ copias del AFN para el apareamiento perfecto de cadenas (construido por el algoritmo 1) y en cada $i-ésima$ copia, $0 \leq i \leq k$, se eliminan los primeros i estados.

Entonces, el número de estados es

$$\begin{aligned}(m+1)+(m+1-1)+(m+1-2)+\ldots &= \sum_{i=0}^{k}(m+1-i) \\ &= (k+1)(m+1-\frac{k}{2})\end{aligned}$$

□

Teorema 6.3.2. *El AFN construido por el algoritmo 3 (versión $\overline{p}$) y el AFN construido por el algoritmo 4 (versión Σ) son equivalentes respecto al apareamiento aproximado, utilizando la distancia de Hamming.*

Demostración. Ambos autómatas difieren solo en la representación de las transiciones de *reemplazo*.

En el AFN construido mediante el algoritmo 4 (versión Σ), todos los estados bajo el estado activo superior en la misma profundidad están también activos, pero en el AFN construido mediante el algoritmo 3 (versión $\overline{p}$), solo el estado activo superior en cada profundidad está activo. Esto es debido a que en cada estado (excepto los estados finales) hay una sola transición por cada símbolo del alfabeto Σ a la siguiente profundidad.

En el apareamiento aproximado de cadenas utilizando la distancia de Hamming nos interesa el mínimo número de errores, es decir el estado final activo, el cual resulta ser el mismo en ambos AFN. Entonces los AFN son equivalentes con respecto al apareamiento aproximado de cadenas utilizando la distancia de Hamming. □

6.4. Apareamiento aproximado de cadenas utilizando la distancia de Levenshtein

En el apareamiento aproximado de cadenas utilizando la distancia de Levenshtein se usan $k+1$ copias del AFN para el apareamiento perfecto de cadenas $(M_0, M_1, \ldots, M_k)$ y se conectan mediante transiciones que representen la operación de *reemplazo*. Debido a que en la distancia de Levenshtein tiene como operaciones permitidas, además de la operación de *reemplazo*, las operaciones de *inserción* y de *eliminación*, es necesario agregar nuevas

transiciones para dichas operaciones en el AFN para la distancia de Hamming. El estado inicial del AFN resultante es el estado inicial del autómata M_0.

La transición que representa la operación de *inserción* está etiquetada por el símbolo $\overline{p}_{j+1}$ y va desde el estado q_j del autómata M_i al estado q_j del autómata M_{i+1}, $0 \leq i < k$, $0 < j < m$; el mínimo número de errores se incrementa, pero la profundidad en el autómata sigue siendo la misma.

La transición que representa la operación de *eliminación* está etiquetada por el símbolo $\overline{p}_{j+1}$ y va desde el estado q_j del autómata M_i al estado q_j del autómata M_{i+1}, $0 \leq i < k$, $0 \leq j < m$; la profundidad del autómata se incrementa tanto como el mínimo número de errores, pero ningún símbolo es leído desde la entrada.

Un ejemplo de dicho autómata para $m = 4$ y $k = 3$ se muestra en la figura 6.5.

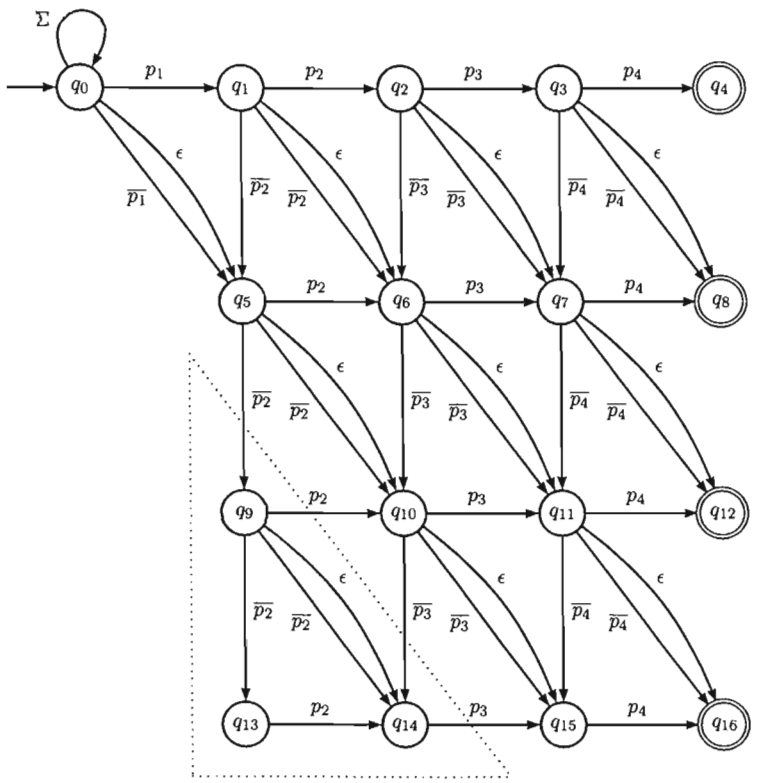


Figura 6.5: AFN para el apareamiento aproximado de cadenas utilizando la distancia de Levenshtein ($m = 4, k = 3$)

El autómata construido contiene estados redundantes que pueden ser removidos (línea punteada de la figura 6.5).

Teorema 6.4.1. *En el AFN para apareamiento aproximado de cadenas utilizando la distancia de Levenshtein, los estados bajo la diagonal principal desde el estado inicial (es decir, los estados de profundidad i en el autómata M_j para el apareamiento exacto de cadenas tales que $1 < j \leq k+1$, $1 \leq i < j$) pueden ser eliminados sin alterar el comportamiento del AFN.*

Demostración. Dado el ciclo del estado inicial q_0 del AFN, el estado inicial siempre está activo. Como los estados localizados en la diagonal 0, la cual se dirige desde el estado inicial q_0, están conectados con el estado inicial mediante transiciones-ε, también están activos todo el tiempo (es decir, los estados de la $\varepsilon - cerradura\,\{q_0\}$ siempre están activos).

Los estados localizados bajo la diagonal 0 están separados de cualquiera de los estados finales por estados de la diagonal 0. Cualquier camino encabezado por cualquier estado bajo la diagonal 0 debe cruzar los estados de la diagonal 0, pero esos estados siempre están activos. Los estados bajo la diagonal 0 no influyen sobre ningún estado de la diagonal 0 y por tanto no influyen sobre ningún estado final. Entonces dichos estados son redundantes y pueden ser eliminados. □

Es decir, cada estado redundante de nivel j y profundidad i, $1 < j \leq k$, $1 \leq i < j$, representa un número j de errores mayores que la profundidad i. Esto hace que el autómata no sea mínimo, pues es un número más grande que si se utilizara i veces la operación de *eliminación*.

Para eliminar dichos estados directamente, en la construcción del AFN se ponen transiciones de *inserción* en los estados q_j (de profundidad j) en el autómata M_i, $0 \leq i < k$, $i < j < m$.

El AFN sin estados redundantes puede ser construido mediante una modificación en la construcción del AFN para la distancia de Hamming.

Un ejemplo de dicho autómata para $m = 4$ y $k = 3$ se muestra en la figura 6.6.

Es posible construir un AFN equivalente sin transiciones-ε como se muestra en el algoritmo 5.

Cuando todas las transiciones para el estado q de nivel l, $0 \leq l < k$, y profundidad r, $l \leq r < m$, se construyen (excepto las transiciones de *eliminación*), todas las transiciones del estado q' de nivel $l+1$ y profundidad $r+1$ se agregan a estas transiciones del estado q, porque hay una transición-ε del estado q al estado q'.

Si se reemplaza la línea para la transición de *eliminación* por $\delta(q_i, \varepsilon) = \{q_s\}$ en el algoritmo 5, se obtiene el algoritmo que construye el AFN con transiciones-ε descrito antes.

Se puede construir un AFN para el apareamiento aproximado de cadenas utilizando la distancia de Levenshtein en el que las transiciones que representan *inserción* y *reemplazo*

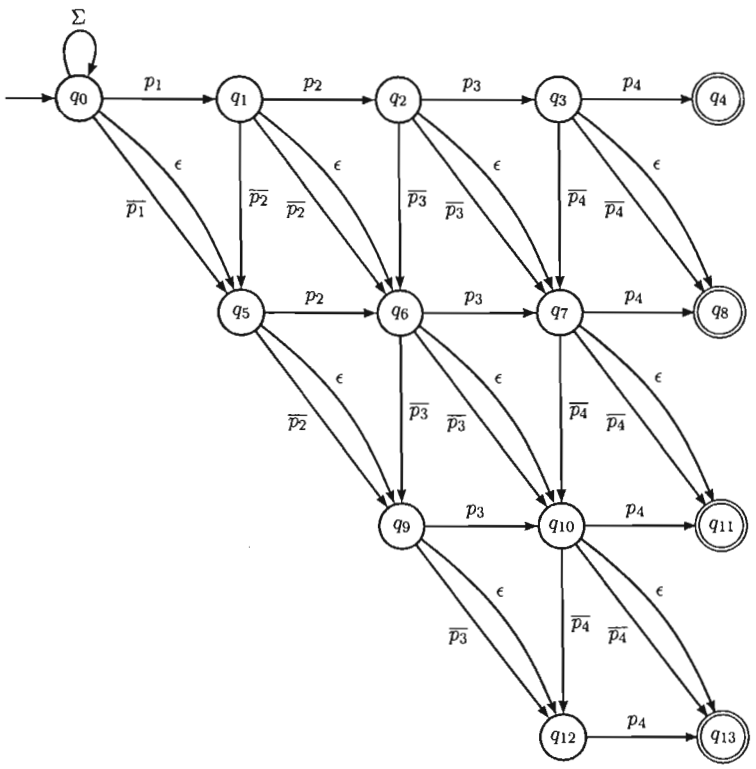


Figura 6.6: AFN para el apareamiento aproximado de cadenas utilizando la distancia de Levenshtein - versión $\bar{p}$ $(m = 4, k = 3)$

Algoritmo 5 Construcción del AFN para el apareamiento aproximado de cadenas utilizando la distancia de Levenshtein (versión $\overline{p}$)

Entrada: Patrón $P = p_1 p_2 \ldots p_m$, k máximo número de errores permitidos, $k < m$.
Salida: AFN M que acepta el lenguaje $L(M) = \{wu \mid w, u \in \Sigma^*, D_L(P, u) \leq k\}$.
Método: AFN $M = (Q, \Sigma, \delta, q_0, F)$, donde Q, δ, F se construyen de la siguiente manera:

$Q \leftarrow \{q_0, q_1, \ldots, q_{|Q|-1}\}, \mid Q \mid = (k+1)\left(m + 1 - \frac{k}{2}\right)$
$l \leftarrow k$ /* número de nivel */
$r \leftarrow m$ /* profundidad del estado */
for $i \leftarrow \mid Q \mid -1, \mid Q \mid -2, \ldots, 0$ **do**
  if $r = m$ **then**
    $\delta(q_i, a) \leftarrow \emptyset, \forall a \in \Sigma$
    $F \leftarrow F \bigcup \{q_i\}$ /* estado final */
  else
    $\delta(q_i, p_{r+1}) \leftarrow \{q_{i+1}\}$ /* transición de apareo */
    if $l < k$ **then**
      $s \leftarrow i + m + 1 - l$ /* s =número del vecino menor derecho */
      $\delta(q_i, a) \leftarrow \{q_s\}, \forall a \in \Sigma \setminus \{p_{r+1}\}$ /* transición de reemplazo */
    end if
    if $r > l$ **then**
      $\delta(q_i, a) \leftarrow \delta(q_i, a) \bigcup \{q_{s-1}\}, \forall a \in \Sigma \setminus \{p_{r+1}\}$ /* transición de inserción */
    end if
    $\delta(q_i, a) \leftarrow \delta(q_i, a) \bigcup \delta(q_s, a), \forall a \in \Sigma$ /* transición de eliminación */
  else
    $\delta(q_i, a) \leftarrow \emptyset, \forall a \in \Sigma \setminus \{p_{r+1}\}$
  end if
  $r \leftarrow r - 1$ /* profundidad anterior del nivel l */
  /*si r esta fuera de los estados del nivel l */
  if $r < l$ **then**
    $r \leftarrow m$ /* ir al estado final del nivel anterior */
    $l \leftarrow l - 1$
  end if
end for
$\delta(q_0, a) \leftarrow \delta(q_0, q) \bigcup \{q_0\}, \forall a \in \Sigma$ /* ciclo del estado inicial */

se etiquetan con Σ. Este autómata puede ser construido conectando $k+1$ copias del AFN o mediante el algoritmo 6.

Un ejemplo de dicho autómata para $m = 4$ y $k = 3$ se muestra en la figura 6.7.

Lema 6.4.2. *El AFN para el apareamiento aproximado de cadenas utilizando la distancia de Levenshtein, construido por el algoritmo 5 o el algoritmo 6, para un patrón de longitud m y máximo número de errores k tiene $(k+1)\left(m+1-\frac{k}{2}\right)$ estados.*

Demostración. El AFN construido mediante el algoritmo 5 y el algoritmo 6 tiene el mismo número de estados que el AFN para la distancia de Hamming construido por el algoritmo 3 y el algoritmo 4. □

El siguiente ejemplo muestra la diferencia entre los autómatas finitos versión-$\overline{p}$ y versión-Σ.

Ejemplo 6.4.3. Sea $P = abc$, $T = dabcddd$, y $k = 2$.

Se construyen los autómatas $M_{\overline{p}}$ y M_{Σ} para la versión-$\overline{p}$ y la versión-Σ de los autómatas finitos para el apareamiento aproximado de cadenas utilizando la distancia de Levenshtein.

Dichos autómatas se muestran en la figura 6.8.

Al ejecutar los autómatas con la cadena de entrada T, se obtienen las siguientes configuraciones:

$$
\begin{aligned}
M_{\overline{p}}: \quad (\{q_0, q_4, q_7\}, dabcddd) &\vdash (\{q_0, q_4, q_7\}, abcddd) \\
&\vdash (\{q_0, q_1, q_4, q_5, q_7, q_8\}, bcddd) \\
&\vdash (\{q_0, q_2, q_4, q_6, q_7, q_8\}, cddd) \\
&\vdash (\{q_0, q_3, q_4, q_7, q_8\}, ddd) \\
&\vdash (\{q_0, q_4, q_7\}, dd) \\
&\vdash (\{q_0, q_4, q_7\}, d) \\
&\vdash (\{q_0, q_4, q_7\}, \epsilon)
\end{aligned}
$$

$$
\begin{aligned}
M_{\Sigma}: \quad (\{q_0, q_4, q_7\}, dabcddd) &\vdash (\{q_0, q_4, q_7\}, abcddd) \\
&\vdash (\{q_0, q_1, q_4, q_5, q_7, q_8\}, bcddd) \\
&\vdash (\{q_0, q_2, q_4, q_5, q_6, q_7, q_8\}, cddd) \\
&\vdash (\{q_0, q_3, q_4, q_6, q_7, q_8\}, ddd) \\
&\vdash (\{q_0, q_4, q_7, q_8\}, dd) \\
&\vdash (\{q_0, q_4, q_7\}, d) \\
&\vdash (\{q_0, q_4, q_7\}, \epsilon)
\end{aligned}
$$

Algoritmo 6 Construcción del AFN para el apareamiento aproximado de cadenas utilizando la distancia de Levenshtein (versión Σ)

Entrada: Patrón $P = p_1 p_2 \ldots p_m$, k máximo número de errores permitidos, $k < m$.
Salida: AFN M que acepta el lenguaje $L(M) = \{wu \mid w, u \in \Sigma^*, D_L(P, u) \leq k\}$.
Método: AFN $M = (Q, \Sigma, \delta, q_0, F)$,donde Q, δ, F se construyen de la siguiente manera:

$Q \leftarrow \{q_0, q_1, \ldots, q_{|Q|-1}\}$, $\mid Q \mid = (k+1)\left(m + 1 - \frac{k}{2}\right)$
$l \leftarrow k$ /* número de nivel */
$r \leftarrow m$ /* profundidad del estado */
for $i \leftarrow \mid Q \mid -1. \mid Q \mid -2, \ldots, 0$ **do**
  if $r = m$ **then**
    $\delta(q_i, a) \leftarrow \emptyset, \forall a \in \Sigma$
    $F \leftarrow F \bigcup \{q_i\}$ /*estado final */
  else
    if $l < k$ **then**
      $s \leftarrow i + m + 1 - l$ /* s =número del vecino menor derecho */
      $\delta(q_i, p_{r+1}) \leftarrow \{q_{i+1}, q_s\}$ /* transición de apareo */
      $\delta(q_i, a) \leftarrow \{q_s\}, \forall a \in \Sigma \setminus \{p_{r+1}\}$ /* transición de reemplazo */
      if $r > l$ **then**
        $\delta(q_i, a) \leftarrow \delta(q_i, a) \bigcup \{q_{s-1}\}, \forall a \in \Sigma$ /* transición de inserción */
      end if
      $\delta(q_i, a) \leftarrow \delta(q_i, a) \bigcup \delta(q_s, a), \forall a \in \Sigma$ /* transición de eliminación */
    else
      $\delta(q_i, p_{r+1}) \leftarrow \{q_{i+1}\}$ /* transición de apareo en el $k - ésimo$ nivel */
      $\delta(q_i, a) \leftarrow \emptyset, \forall a \in \Sigma \setminus \{p_{r+1}\}$
    end if
  end if
  $r \leftarrow r - 1$ /* profundidad anterior del nivel l */
  /*si r esta fuera de los estados del nivel l */
  if $r < l$ **then**
    $r \leftarrow m$ /* ir al estado final del nivel anterior */
    $l \leftarrow l - 1$
  end if
end for
$\delta(q_0, a) \leftarrow \delta(q_0, q) \bigcup \{q_0\}, \forall a \in \Sigma$ /* ciclo del estado inicial */

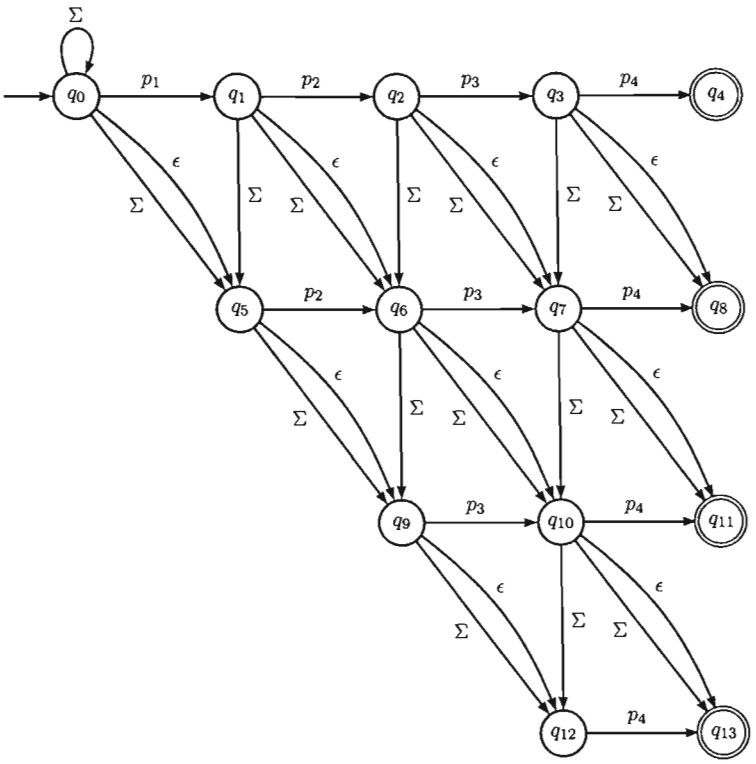


Figura 6.7: AFN para el apareamiento aproximado de cadenas utilizando la distancia de Levenshtein - versión Σ ($m = 4, k = 3$)

La siguiente tabla muestra los resultados (número de errores de las cadenas encontradas) reportados por los autómatas:

T:	d	a	b	c	d	d	d
$M_{\overline{p}}$	-	2	1,2	0,1,2	2	-	-
M_{Σ}	-	2	1,2	0,1,2	-	-	-

M_{Σ} reporta una presencia con 2 errores en la posición 5 ocasionada por la inserción de un símbolo c en P (en la posición 4) y substituyendo el símbolo c en P por el símbolo d (en la posición 5).

Esta ocurrencia no es encontrada en $M_{\overline{p}}$ debido a que fue causada por una ocurrencia con errores.

Comparando $M_{\overline{p}}$ y M_{Σ} se puede ver que $M_{\overline{p}}$ tiene normalmente menos estados activos que M_{Σ} (a lo mas tantos como M_{Σ}).

6.5. Apareamiento aproximado de cadenas utilizando la distancia generalizada de Levenshtein

En el apareamiento aproximado de cadenas utilizando la distancia generalizada de Levenshtein se usan $k+1$ copias del AFN para el apareamiento perfecto de cadenas $(M_0, M_1, , \ldots, M_k)$ y se conectan mediante transiciones que representen las operaciones de *reemplazo, inserción y eliminación.* Como en la distancia generalizada de Levenshtein hay, además de las operaciones de *reemplazo, inserción y eliminación* la operación de *transposición*, es necesario insertar nuevas transiciones para dicha operación en el AFN para la distancia de Levenshtein.

La transición que representa la operación de *transposición* va desde el estado q_j, $0 < j < m-1$, del autómata M_i, $0 \leq i < k$, al nuevo estado q' y es etiquetado con el símbolo p_{j+2} del autómata M_{i+1}. De modo que dos símbolo adyacentes q_{j+1} y q_{j+2} son aceptados en orden inverso; la profundidad del autómata se incrementa en 2 y el mínimo número de errores se incrementa en 1.

La operación de *transposición* puede ser reemplazada por dos operaciones de *reemplazo* adyacentes, pero esto incrementa el número de errores en 2. Entonces, si lo que se desea es evaluar la operación de *transposición* sólo en 1 error como en las operaciones de *substitución, inserción* y *eliminación,* es necesario usar la distancia generalizada de Levenshtein.

Un ejemplo de dicho autómata para $m = 4$ y $k = 3$ se muestra en la figura 6.9.

Es posible construir un AFN equivalente sin transiciones-ε como se muestra en el algoritmo 7.

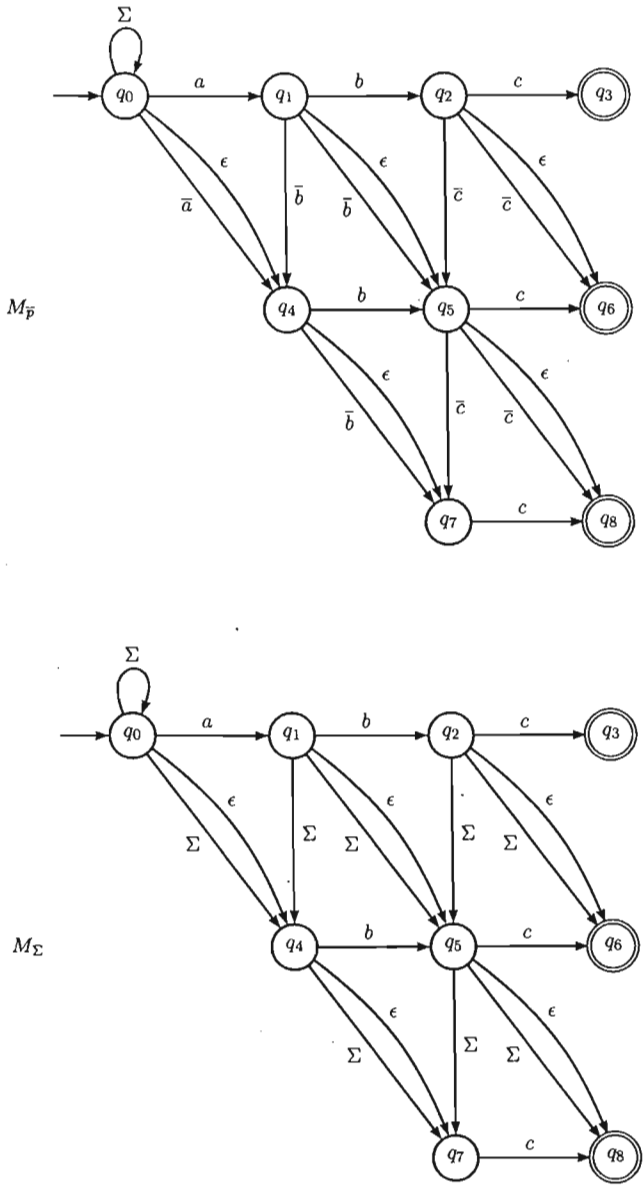


Figura 6.8: Autómatas $M_{\overline{p}}$ y M_Σ

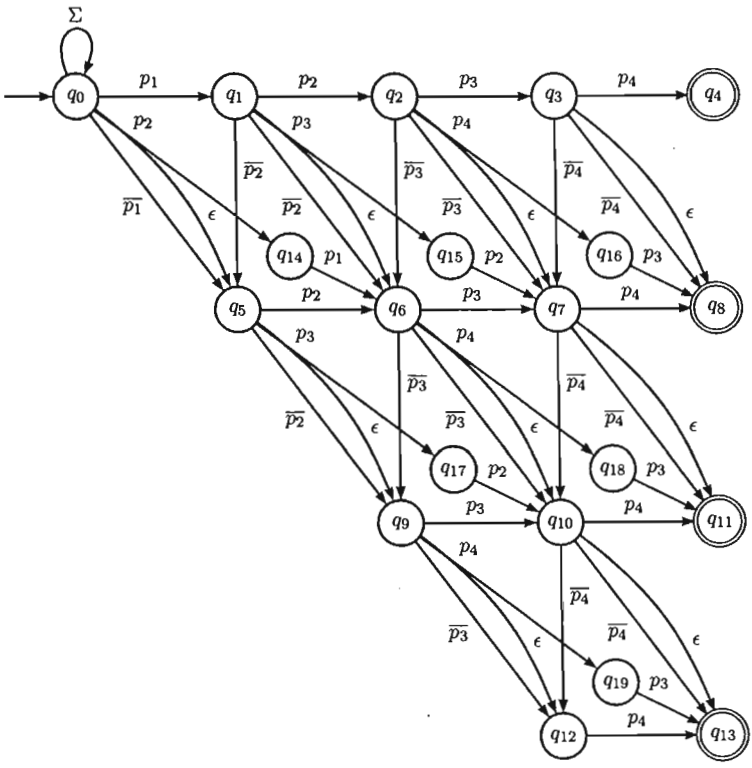


Figura 6.9: AFN para el apareamiento aproximado de cadenas utilizando la distancia generalizada de Levenshtein - versión $\bar{p}$ $(m = 4, k = 3)$

Algoritmo 7 Construcción del AFN para el apareamiento aproximado de cadenas utilizando la distancia generalizada de Levenshtein (versión - $\overline{p}$)

Entrada: Patrón $P = p_1 p_2 \ldots p_m$, k máximo número de errores permitidos, $k < m$.
Salida: AFN M que acepta el lenguaje $L(M) = \{wu \mid w, u \in \Sigma^*, D_G(P, u) \leq k\}$.
Método: AFN $M = (Q, \Sigma, \delta, q_0, F)$, donde Q, δ, F se construyen de la siguiente manera:

$Q \leftarrow \{q_0, q_1, \ldots, q_{|Q|-1}\}, \mid Q \mid = (m+1) + k(2m-k)$
$l \leftarrow k$ /* número de nivel */
$r \leftarrow m$ /* profundidad del estado */
$j \leftarrow \mid Q \mid -1$ /* índice del último estado de transposición */
for $i \leftarrow (k+1)\left(m+1-\frac{k}{2}\right)-1, (k+1)\left(m+1-\frac{k}{2}\right)-2, \ldots, 0$ **do**
  if $r = m$ **then**
    $\delta(q_i, a) \leftarrow \emptyset, \forall a \in \Sigma$
    $F \leftarrow F \bigcup \{q_i\}$ /* estado final */
  else
    $\delta(q_i, p_{r+1}) \leftarrow \{q_{i+1}\}$ /* transición de apareo */
    if $l < k$ **then**
      $s \leftarrow i + m + 1 - l$ /* s =número del vecino menor derecho */
      $\delta(q_i, a) \leftarrow \{q_s\}, \forall a \in \Sigma \setminus \{p_{r+1}\}$ /* transición de reemplazo */
      if $r > l$ **then**
        $\delta(q_i, a) \leftarrow \delta(q_i, a) \bigcup \{q_{s-1}\}, \forall a \in \Sigma \setminus \{p_{r+1}\}$ /* transición de inserción */
      end if
      if $r < m-1$ **then**
        $\delta(q_i, p_{r+2}) \leftarrow \delta(q_i, p_{r+2}) \bigcup \{q_j\}$ /* transición de transposición */
        $\delta(q_j, p_{r+1}) \leftarrow \delta(q_{s+1})$
        $j \leftarrow j - 1$
      end if
      $\delta(q_i, a) \leftarrow \delta(q_i, a) \bigcup \delta(q_s, a), \forall a \in \Sigma$ /* transición de eliminación */
    else
      $\delta(q_i, a) \leftarrow \emptyset, \forall a \in \Sigma \setminus \{p_{r+1}\}$
    end if
  end if
  $r \leftarrow r - 1$ /* profundidad anterior del nivel l */
  /*si r está fuera de los estados del nivel l */
  if $r < l$ **then**
    $r \leftarrow m$ /* ir al estado final del nivel anterior */
    $l \leftarrow l - 1$
  end if
end for
$\delta(q_0, a) \leftarrow \delta(q_0, q) \bigcup \{q_0\}, \forall a \in \Sigma$ /* ciclo del estado inicial */

Si se reemplaza la línea para la transición de *eliminación* por $\delta(q_i, \varepsilon) = \{q_s\}$ en el algoritmo 7, se obtiene el algoritmo que construye el AFN con transiciones-ε para el apareamiento aproximado de cadenas utilizando la distancia generalizada de Levenshtein.

Se puede construir un AFN para el apareamiento aproximado de cadenas utilizando la distancia generalizada de Levenshtein en el que las transiciones que representan *inserción* y *reemplazo* se etiquetan con Σ. Este autómata puede ser construido conectando $k+1$ copias del AFN o mediante el algoritmo 8.

Un ejemplo de dicho autómata para $m = 4$ y $k = 3$ se muestra en la figura 6.10.

Lema 6.5.1. *El AFN para el apareamiento aproximado de cadenas utilizando la distancia generalizada de Levenshtein construido por el algoritmo 7 o el algoritmo 8, para un patrón de longitud m y máximo número de errores k, tiene $(m+1) + k(2m-k)$ estados.*

Demostración. Si al autómata para el apareamiento aproximado de cadenas utilizando la distancia generalizada de Levenshtein, se agregan los estados que son necesarios para las transiciones de *transposición*, se obtiene

$$(m+1) + (m-1+m+1-1) + (m-2+m+1-2) + \ldots =$$

$$(m+1) + \sum_{i=1}^{k} (2m+1-2i) = (m+1) + k(2m-k)$$

estados. □

Algoritmo 8 Construcción del AFN para el apareamiento aproximado de cadenas utilizando la distancia generalizada de Levenshtein (versión - Σ)

Entrada: Patrón $P = p_1 p_2 \ldots p_m$, k máximo número de errores permitidos, $k < m$.
Salida: AFN M que acepta el lenguaje $L(M) = \{wu \mid w, u \in \Sigma^*, D_G(P, u) \leq k\}$.
Método: AFN $M = (Q, \Sigma, \delta, q_0, F)$, donde Q, δ, F se construyen de la siguiente manera:

$Q \leftarrow \{q_0, q_1, \ldots, q_{|Q|-1}\}, \mid Q \mid = (m+1) + k(2m-k)$
$l \leftarrow k$ /* número de nivel */
$r \leftarrow m$ /* profundidad del estado */
$j \leftarrow \mid Q \mid -1$ /* índice del último estado de transposición */
for $i \leftarrow (k+1)\left(m+1-\frac{k}{2}\right)-1, (k+1)\left(m+1-\frac{k}{2}\right)-2, \ldots, 0$ **do**
  if $r = m$ **then**
    $\delta(q_i, a) \leftarrow \emptyset, \forall a \in \Sigma$
    $F \leftarrow F \bigcup \{q_i\}$ /* estado final */
  else
    if $l < k$ **then**
      $s \leftarrow i + m + 1 - l$ /* s =número del vecino menor derecho */
      $\delta(q_i, p_{r+1}) \leftarrow \{q_{i+1}, q_s\}$ /* transición de apareo */
      $\delta(q_i, a) \leftarrow \{q_s\}, \forall a \in \Sigma \setminus \{p_{r+1}\}$ /* transición de reemplazo */
      if $r > l$ **then**
        $\delta(q_i, a) \leftarrow \delta(q_i, a) \bigcup \{q_{s-1}\}, \forall a \in \Sigma$ /* transición de inserción */
      end if
      if $r < m - 1$ **then**
        $\delta(q_i, p_{r+2}) \leftarrow \delta(q_i, p_{r+2}) \bigcup \{q_j\}$ /* transición de transposición */
        $\delta(q_j, p_{r+1}) \leftarrow \delta(q_{s+1})$
        $j \leftarrow j - 1$
      end if
      $\delta(q_i, a) \leftarrow \delta(q_i, a) \bigcup \delta(q_s, a), \forall a \in \Sigma$ /* transición de eliminación */
    else
      $\delta(q_i, p_{r+1}) \leftarrow \{q_{i+1}\}$ /* transición de apareo en el $k - ésimo$ nivel*/
      $\delta(q_i, a) \leftarrow \emptyset, \forall a \in \Sigma \setminus \{p_{r+1}\}$
    end if
  end if
  $r \leftarrow r - 1$ /* profundidad anterior del nivel l */
  /*si r esta fuera de los estados del nivel l */
  if $r < l$ **then**
    $r \leftarrow m$ /* ir al estado final del nivel anterior */
    $l \leftarrow l - 1$
  end if
end for
$\delta(q_0, a) \leftarrow \delta(q_0, q) \bigcup \{q_0\}, \forall a \in \Sigma$ /* ciclo del estado inicial */

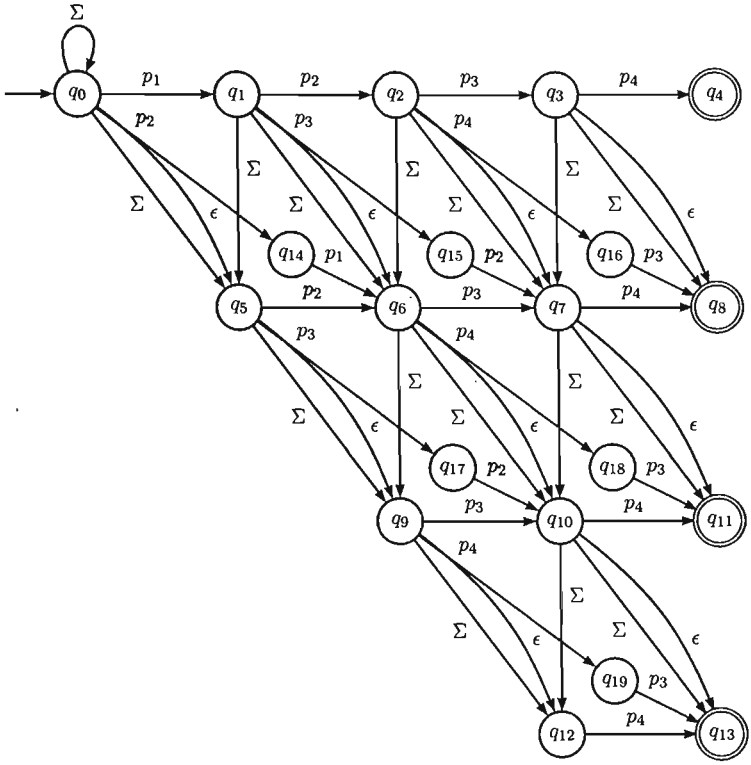


Figura 6.10: AFN para el apareamiento aproximado de cadenas utilizando la distancia generalizada de Levenshtein - versión Σ ($m = 4, k = 3$)

Capítulo 7

Simulación del AFN para el apareamiento de cadenas

El autómata finito determinístico para el apareamiento de cadenas utiliza en algunos casos, una gran complejidad de espacio. Esto resulta especialmente cierto en los autómatas para el apareamiento aproximado de cadenas, lo que nos lleva a la construcción de algoritmos que simulen los autómatas finitos no deterministicos para el apareamiento perfecto de cadenas. Estos algoritmos tienen una complejidad de espacio aceptable, aunque su complejidad de tiempo no es, en algunos casos, lineal.

Uno de los métodos utilizados en la simulación del autómata finito no determinístico para el apareamiento perfecto de cadenas usa una *función de fallo.*

El grupo de métodos que utilizan la función de fallo están basados en el siguiente principio:

- En el autómata no determinístico para el apareamiento perfecto de cadenas los ciclos del estado inicial hacia sí mismo, pueden se eliminados para obtener un autómata finito determinístico. Si lo anterior ocurre, y el autómata finito resultante es determinístico, entonces puede ser utilizado directamente; sin embargo, es necesario agregar algunas transiciones llamadas *Transiciones de retroceso o fallo.* En estas transiciones, ningún símbolo es leído y son utilizadas en los casos en los que las transiciones hacia adelante no puede ser utilizadas. [1]

El algoritmo de Knuth-Morris-Pratt pertenece a esta categoría.

7.1. Algoritmo de Knuth-Morris-Pratt

El algoritmo de Knuth-Morris-Pratt es el algoritmo más conocido, de tiempo lineal, para resolver el problema de apareamiento exacto de cadenas.

[1] Las transiciones de retroceso puede ser vistas como transiciones-ϵ.

Este algoritmo fue diseñado por Knuth y Pratt y de manera independiente por Morris en 1976; está basado en el teorema de Cook, el cual enuncia que hay un método para resolver el problema de apareamiento de cadenas en tiempo $O(n+m)$ en el peor de los casos, donde n es la longitud del texto y m es la longitud del patrón buscado dentro del texto.

El algoritmo de Knuth-Morris-Pratt simula el comportamiento de un autómata finito en el que los caracteres del patrón y del texto son comparados en orden de izquierda a derecha. Si un error ocurre, el algoritmo busca el sufijo más largo dentro del error, el cual también resulta ser un prefijo del patrón y entonces determina qué tan lejos se puede encontrar el patrón, recorrido a la derecha.

Además, evita el cálculo de la función de transición δ del autómata finito. El tiempo que tarda durante la búsqueda es lineal en el tamaño del texto y utiliza una función auxiliar $\pi[1...m]$ para el preprocesamiento del patrón, calculada también en tiempo lineal en el tamaño del patrón. El arreglo π permite calcular la función de transición δ del autómata finito de manera eficiente conforme se va necesitando. Es decir, para cualquier estado $q = 0, 1...m$ y cualquier carácter $a \in \Sigma$, el valor de $\pi[q]$ contiene la información que es independiente de a y necesaria para calcular $\delta(q, a)$. Como el arreglo π tiene m entradas, mientras que la función δ tiene $m|\Sigma|$ entradas, se elimina el factor $|\Sigma|$ en el tiempo del preprocesamiento al calcular π en lugar de δ.

7.1.1. Función de prefijo para un patrón

La función de prefijo π para un patrón guarda la información acerca de cómo el patrón se aparea contra desplazamientos de sí mismo. Esta información puede ser utilizada para evitar comparaciones innecesarias en el algoritmo de fuerza bruta o para evitar el cálculo de la función δ del autómata finito.

Dado un alineamiento de P con T, supongamos que el algoritmo de Fuerza Bruta aparea los primeros i caracteres de P dentro de T y ocurre un error en la siguiente comparación.

El algoritmo de fuerza bruta desplazaría a P un lugar y volvería a comparar otra vez desde el extremo izquierdo del patrón. Sin embargo, se puede hacer un desplazamiento más largo.

Por ejemplo, si $P = ababaca$ y en la alineación actual de P con T, el error ocurre en la posición $q = 6$ de P, entonces P puede ser desplazado algunos lugares sin saltar ninguna presencia de P en T y sólo es necesario conocer el lugar del error dentro de P. La información de qué q caracteres han sido apareados permite determinar que algunos saltos resultarán inútiles o bien saltarán una posible presencia del patrón.

La figura 7.1 muestra un desplazamiento para el patrón P. El desplazamiento $s + 1$ resulta inadecuado, ya que el primer carácter del patrón (a) sería alineado con un carácter del texto que ya se sabe se aparea con el segundo carácter del patrón (b). El desplazamiento $s' = s + 2$ alinea los primeros tres caracteres del patrón con tres caracteres del texto. La información para deducir los desplazamientos válidos puede obtenerse comparando el

patrón contra sí mismo. En el ejemplo, el prefijo más grande de P, que también es un sufijo propio de P_5, es P_3. Esta información es calculada y guardada en el arreglo π, asignando $\pi[5] = 3$.

Como q caracteres han sido apareados en el desplazamiento s, el siguiente desplazamiento válido está en la posición $s' = s + (q - \pi[q])$.

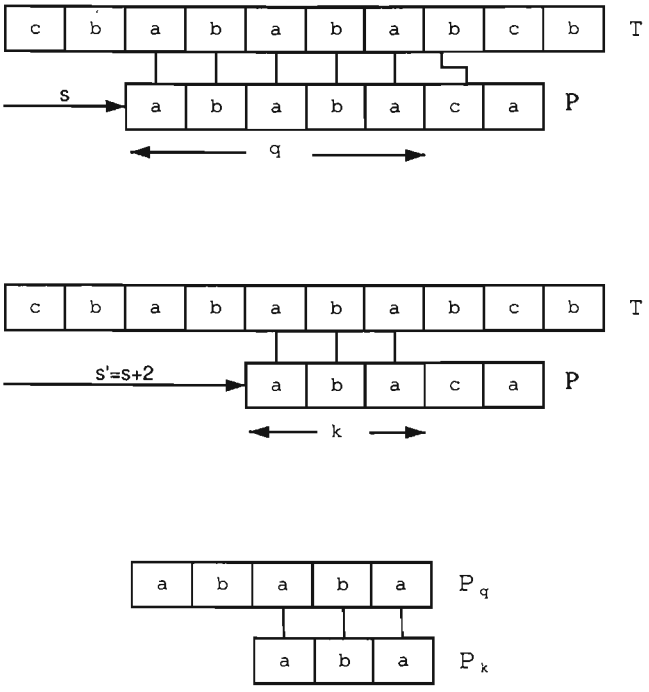


Figura 7.1: La función de prefijo π

En general, dados los caracteres $P[1...q]$ del patrón que se aparean con los caracteres $T[s+1...s+q]$ del texto, se desea saber cuál es el desplazamiento $s' > s$ tal que:

$$P[i...k] = T[s'+1...s'+k]$$

donde $s' + k = s + q$.

El desplazamiento s' es el primer desplazamiento válido mayor que s, dado el conocimiento de $T[s+1...s+q]$. En el mejor caso se tiene que $s' = s + q$ y los desplazamientos $s+1, s+2, ..., s+q-1$ son descartados.

En cualquier caso, en el desplazamiento s' no es necesario comparar los primeros k caracteres de P con los correspondientes caracteres de T. La información necesaria puede ser obtenida comparando el patrón contra sí mismo, como ya se mencionó. Como $T[s'+1...s'+k]$ es parte de la porción del texto conocida, entonces es un sufijo de la cadena P_q, es decir, se busca la máxima $k < q$ tal que P_k es un sufijo de P_q ($P_k \sqsupset P_q$). Entonces, $s' = s + (q-k)$ es el siguiente desplazamiento válido.

Formalizando:

Sean x y w cualesquiera dos cadenas. Entonces, la cadena w es *prefijo* de la cadena x, denotada como $w \sqsubset x$, si $x = wy$ para alguna cadena $y \in \Sigma^*$. Además, si $y \neq \epsilon$ y $w \neq \epsilon$ entonces w es *prefijo propio* de x.

Si $w \sqsubset x$, entonces $|w| \leq |x|$.

De igual forma, la cadena w es *sufijo* de la cadena x, denotada como $w \sqsupset x$, si $x = yw$ para alguna cadena $y \in \Sigma^*$. Además, si $y \neq \epsilon$ y $w \neq \epsilon$ entonces w es *sufijo propio* de x.

Si $w \sqsupset x$, entonces $|w| \leq |x|$.

Dado un patrón $P[1...m]$ la **función de prefijo** del patrón P es la $\pi : 1, 2, ..., m \rightarrow 0, 1, ..., m-1$ tal que

$$\pi[q] = max\{k : k < q; P_k \sqsupset P_q\}.$$

Es decir, $\pi[q]$ es la longitud del prefijo más largo de P que también es un sufijo propio de P_q.

Ejemplo 7.1.1. Si $P = abcaeabcabd$, entonces

- $\pi[2] = \pi[3] = 0$

  dado que $P_2 = [ab]$ y $P_3 = [abc]$ y no hay una cadena que sea prefijo y sufijo propio de P.

- $\pi[4] = 1$

  dado que $P_4 = [abca]$; entonces la cadena a de longitud 1 es prefijo y sufijo propio de P.

- $\pi[8] = 3$

  dado que $P_8 = [abcaeabc]$; entonces la cadena abc de longitud 3 es prefijo y sufijo propio de P.

- $\pi[10] = 2$

  dado que $P_{10} = [abcaeabcab]$; entonces la cadena ab de longitud 2 es prefijo y sufijo propio de P.

El apareamiento de Knuth-Morris-Pratt se presenta en el algoritmo 9.

7.1.2. Análisis de tiempo de ejecución

El tiempo de ejecución para el cálculo de la función de prefijo es $O(m)$.

Dentro del algoritmo para calcular la función de prefijo (algoritmo 10), la variable k tiene un valor inicial de 0 y se decrementa dentro del ciclo interno del algoritmo puesto que $\pi[k] < k$. Como $\pi[k] \geq 0$, $\forall k$, entonces k nunca toma valores negativos.

La variable k también se incrementa a lo más en una unidad durante cada ejecución del ciclo externo del algoritmo. Como $k < q$ al entrar al ciclo principal, y como q se incrementa en cada iteración del ciclo, entonces se mantiene que $k < q$.

El número de iteraciones del ciclo externo es $O(m)$; además el máximo valor que alcanza la variable k es por lo menos tan grande como su valor inicial. De manera que el tiempo de ejecución total para el cálculo de la función de prefijo es $O(m)$.

Se puede hacer el mismo análisis para el cálculo del tiempo de ejecución del algoritmo de Knuth-Morris-Pratt, por lo que el tiempo total de ejecución del algoritmo de Knuth-Morris-Pratt es $O(n)$.

Comparando el algoritmo de Knuth-Morris-Pratt con el autómata finito para el apareamiento exacto de cadenas se puede observar que utilizando la función π en lugar de la función δ se reduce el tiempo de preprocesamiento del patrón de $O(m|\Sigma|)$ a $O(m)$, mientras que se mantiene el tiempo de búsqueda en $O(n)$.

7.1.3. Correctez del algoritmo de Knuth-Morris-Pratt

El algoritmo de Knuth-Morris-Pratt puede ser visto como una reimplementación del autómata finito para el apareamiento exacto de cadenas. En lugar de utilizar los valores guardados de la función δ, éstos se calculan conforme es necesario desde la función π.

Como el algoritmo de Knuth-Morris-Pratt simula el comportamiento del autómata finito, la correctez del algoritmo de Knuth-Morris-Pratt se sigue de la correctez del autómata finito.

La función π del algoritmo de Knuth-Morris-Pratt puede ser vista como una función de fallo dentro del autómata finito, de forma que es posible utilizarla para reconstruir al autómata finito que simula.

El algoritmo 11 reconstruye el autómata finito para el apareamiento perfecto de cadenas a partir de la función π del algoritmo de Knuth-Morris-Pratt.

Algoritmo 9 Algoritmo de Knuth-Morris-Pratt

Entrada: Patrón $P = p_1p_2 \ldots p_m$. Texto $T = t_1t_2 \ldots t_n$

Salida: Apareamiento del patrón P con el texto T.

```
n ← length[T]
m ← length[P]
π ← FUNCION − PREFIJO(P)
q ← 0                                        /* Número de caracteres apareados*/
for i ← 1, 2, ..., n do

  while q > 0  and  P[q + 1] ≠ T[i] do
    q ← π[q]                                 /* El siguiente carácter no se aparea */
  end while
  if P[q + 1] = T[i] then
    q ← q + 1                                /* El siguiente carácter se aparea */
  end if
  if q = m then
    "El patrón aparece en "i − m
    q ← π[q]                                 /* Busca el siguiente apareamiento */
  end if
end for
```

Algoritmo 10 FUNCION-PREFIJO

Entrada: Patrón $P = p_1p_2 \ldots p_m$.

Salida: Función π.

```
m ← length[P]
π[1] ← 0
k ← 0
for q ← 2, 3 ..., m do

  while k > 0  and  P[k + 1] ≠ P[q] do
    k ← π[k]
  end while
  if P[k + 1] = P[q] then
    k ← k + 1
  end if
  π[q] ← k
end for
```

Algoritmo 11 Construcción del AFN para el apareamiento perfecto de cadenas a partir de la función π del algoritmo de Knuth-Morris-Pratt

Entrada: Función $\pi : \{1, 2, ..., m\} \rightarrow \{0, 1, ..., m-1\}$. Patrón $P = p_1 p_2 \dots p_m$.
Salida: AFD M que acepta el lenguaje $L(M) = \{wPy \mid w, y \in \Sigma^*\}$
Método: AFD $M = (\{q_0, q_1, \dots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$, donde la función de mapeo δ se construye de la siguiente manera:

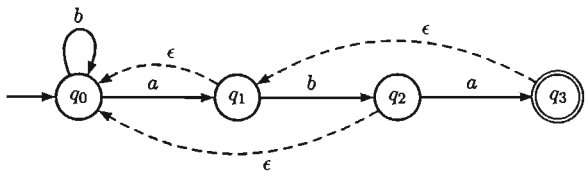
$\delta(q_0, a) \leftarrow \{q_0\}, \forall a \in \Sigma \setminus \{p_1\}$ /* ciclo del estado inicial */
$\delta(q_0, p_1) \leftarrow \{q_0, q_1\}$
for $i \leftarrow 1, 2, \dots, m-1$ **do**
  $\delta(q_i, p_{i+1}) \leftarrow \{q_{i+1}\}$ /* transiciones hacia adelante */
end for
for $i \leftarrow 1, 2, \dots, m-1$ **do**
  $\delta(q_i, \epsilon) \leftarrow \{q_{\pi[i]}\}$ /* transiciones de fallo */
end for

Ejemplo 7.1.2. Sea $P = aba$.

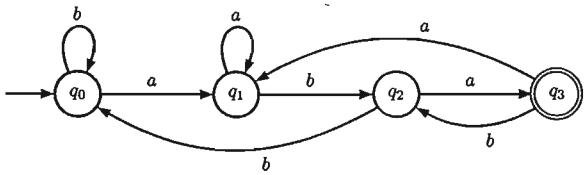
La construcción del autómata Knuth-Morris-Pratt se muestra en la figura 7.2. Las transiciones de fallo generadas por la función π están dibujadas con líneas cortadas.

Utilizando las propiedades de los autómatas finitos es posible convertir el AFN con transiciones-ϵ construido en un AFD equivalente.

Es posible verificar que el AFD obtenido es precisamente el AFD presentado en 6.2.



Autómata Knuth-Morris-Pratt para $P = aba$



Autómata Knuth-Morris-Pratt para $P = aba$ convertido a un AFD

Figura 7.2: Autómata Knuth-Morris-Pratt para $P = aba$

Capítulo 8

Implementación

AUPAM, (*AUtomata PAttern Matching*) es una herramienta dentro del área de Teoría de la Computación, que provee una aplicación gráfica para el apareamiento de cadenas con autómatas finitos.

En este capítulo se explica a grandes rasgos cómo se desarrolló AUPAM, desde su diseño, hasta su publicación en *sourceforge.net* y también se explica cómo utilizarlo.

8.1. Descripción

Las características que cubre AUPAM como sistema para el apareamiento exacto y aproximado de cadenas utilizando autómatas finitos son:

- Crear los autómatas para el apareamiento exacto de cadenas, a partir de un patrón dado.
- Crear los autómatas para el apareamiento aproximado de cadenas, a partir de un patrón dado.
- Crear los autómatas con transiciones que contengan todo el alfabeto o el complemento de un símbolo.
- Permitir que los autómatas contengan o no transiciones-ϵ.
- Permitir un cierto número de errores en el apareamiento aproximado de cadenas y poder variar dicho número.
- Permitir variar la distancia con la que se mide el número de errores, en este caso las distancias contempladas son la distancia de Hamming, Levenshtein y Levenshtein generalizada.
- Mostrar el autómata generado en cada caso.

- Mostrar las posiciones en el texto donde se haya encontrado una presencia del patrón.

Entre las características no funcionales se tienen:

- Ser un sistema con independencia de plataforma.
- Tener un buen desempeño en la creación y visualización de los autómatas, siempre y cuando el tamaño del patrón sea pequeño.
- Tener un buen desempeño en la búsqueda del patrón en el texto (aproximadamente un máximo de 10 caracteres).

8.2. Diseño y componentes del sistema

Para el diseño de AUPAM primero se definió el funcionamiento general del sistema y después se desarrollaron las partes específicas.

La figura 8.1 muestra una visión general de la herramienta y su interfaz gráfica.

De la interfaz gráfica se pueden extraer los siguientes componentes:

- Una barra de menú con las operaciones [*File*] y [*Extras*].

  En el menú [*File*] se permite:

  - Cargar cualquier texto desde un archivo en formato de texto plano. [1]
  - Cargar un patrón con formato XML o crear directamente un patrón de acuerdo al alfabeto del archivo de texto.

  La figura 8.2 muestra el menú [*File*].

  El menú [*Extras*] permite:

  - Ver la definición formal del autómata generado a partir del patrón.
  - Obtener la descripción del autómata generado.

  La figura 8.3 muestra el menú [*Extras*].

- Un componente [*Pattern*] que contiene al patrón buscado dentro del texto.

[1] Se utiliza la codificación de caracteres que tenga instalado por omisión el sistema operativo sobre el que se ejecute el programa.

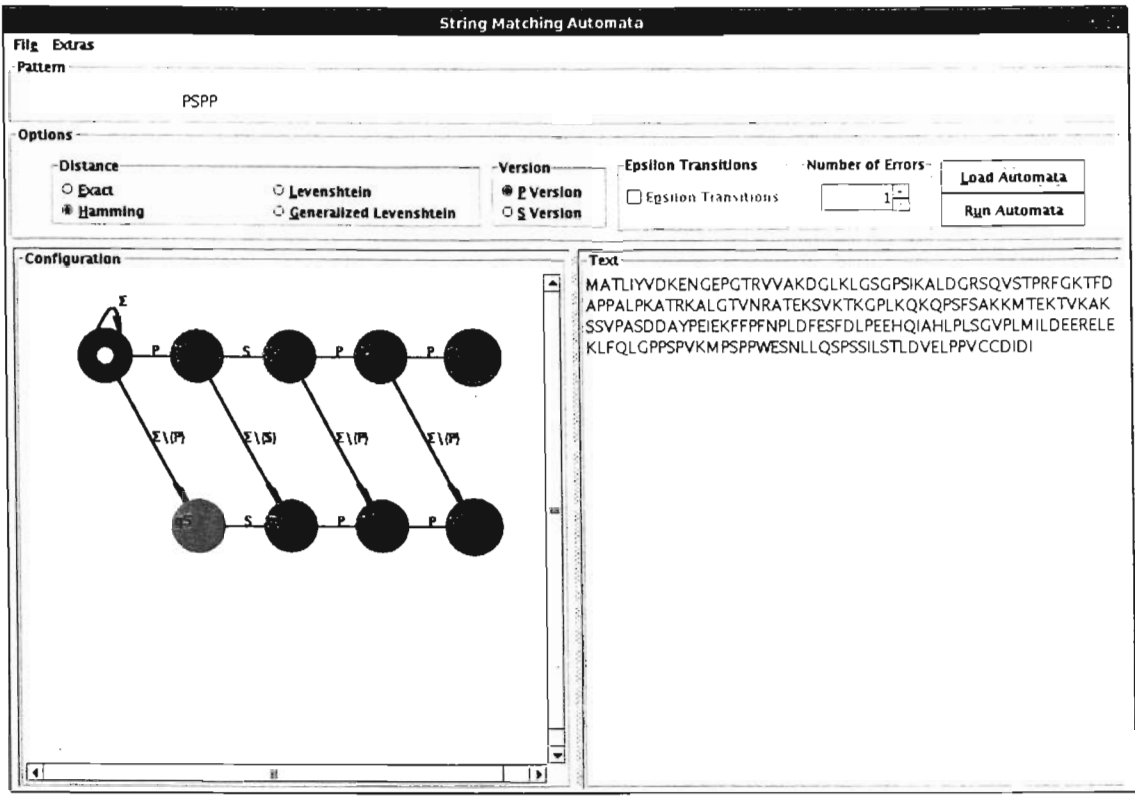


Figura 8.1: Ventana principal

- Un componente [*Options*] que contiene las diferentes opciones de búsqueda del patrón; las opciones se habilitan de acuerdo al tipo de distancia seleccionada.

  El componente [*Options*] contiene:

  - Un componente [*Distance*] con las diferentes métricas de distancia para encontrar la ocurrencia de un patrón.
  - Un componente [$Version$] con las diferentes versiones de autómatas generados (versión-P, versión-Σ).
  - Un componente[$Epsilon Transitions$] que permite elegir la creación del autómata con o sin transiciones-ϵ.

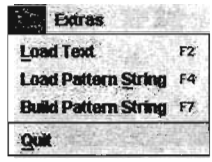


Figura 8.2: Menú [*File*]

Figura 8.3: Menú [*Extras*]

- Un componente [*Number of Errors*] que establece el número de errores permitidos en el texto de acuerdo al tipo de distancia.
- Un botón[*Load Automata*] que permite mostrar la configuración gráfica del autómata, generado de acuerdo al patrón cargado.
- Un botón[*Run Automata*] que permite ejecutar el autómata y decidir si el patrón buscado se encuentra o no dentro del texto.

- Un componente [*Configuration*] que muestra la representación gráfica del autómata construido utilizando el siguiente código:
  - Los estados están representados mediante círculos etiquetados.
  - Los estados iniciales tienen un círculo interno.
  - Los estado finales tienen la circunferencia pintada de diferente color.
  - Las transiciones del autómata están representadas por flechas que parten del estado de salida y apuntan al estado destino.
- Un componente [*Text*] que despliega el contenido del texto dentro del cual se busca el patrón. Las presencias del patrón encontradas dentro del texto se mostrarán iluminando la sección apareada.

8.3. Codificación

La codificación del programa está implementada con el lenguaje de programación *Java*, con el compilador de Sun Microsystems para la plataforma de Java 2 versión 1.4.2.

8.3 Codificación

Se decidió utilizar el lenguaje de programación Java por diversas razones, entre ellas:

- Independencia de plataforma.
- Cuenta con una gran variedad de componentes.
- Permite definir nuevos componentes de manera fácil.

Como base para su desarrollo se utilizó el sistema *Jaguar (Java Automaton and Grammar User Application Resources)* escrito por Iván Hernández Serrano. Jaguar es una herramienta de *software libre* escrita en Java, la cual brinda una infraestructura de bibliotecas y aplicaciones gráficas que soporta la mayoría de los modelos y algoritmos básicos de la Teoría de Autómatas y Lenguajes Formales.

Jaguar se puede obtener en la dirección electrónica:

http://sourceforge.net/projects/ijaguar

De Jaguar se importaron todas las bibliotecas que definen los autómatas finitos no deterministicos y se extendieron algunas clases para modelar autómatas finitos no deterministicos con transiciones-ϵ.

Entre las clases que extienden a los AFN se tienen:

EpsilonSymbol. Extiende la clase *Symbol* para representar el símbolo ϵ dentro de las transiciones-ϵ.

NfaDeltaEpsilon. Extiende la clase *NfaDelta* para representar la función de transición δ de un autómata finito no determinístico con transiciones-ϵ, de manera que se permitieran transiciones entre los estados del autómata sin tener que leer símbolos de la cadena de entrada.

NDFAExtend. Extiende la clase *NDFA* para representar un autómata finito no determinístico con transiciones-ϵ.

Se creó un almacén que contiene la implementación de los algoritmos para el apareamiento de cadenas presentados en este trabajo. La clase que contiene dichos algoritmos es `AutomataMatcherFactory`.

Para la búsqueda de las presencias del patrón dentro del texto se conserva la ejecución del autómata dentro de una estructura de árbol, de manera que los recorridos sobre el árbol hacia nodos que contengan un estado final determinan el lugar exacto de la presencia. La definición de la estructura de árbol se encuentra en la clase `Tree`.

La principal justificación para conservar el árbol de ejecución es mantener la noción de las transiciones-ϵ realizadas durante la ejecución del autómata.

Dentro del código fuente se encuentra un archivo con instrucciones para la herramienta *Apache Ant*.

Las reglas que contiene son:

compile. Compila el código fuente y lo guarda dentro del directorio `build`.

javadoc. Genera la documentación del código fuente y lo guarda dentro del directorio `build/docs`.

cleanBytecode. Borra archivos binarios y temporales.

clean. Borra archivos binarios, temporales y archivos para distribución como son los archivos *JAR* de Java.

pack. Crear un archivo de distribución de Java llamado `aupam-1.01.jar`. dentro del directorio `dist`.

install. Instala el programa dentro de un directorio dado.

run. Ejecuta directamente el programa.

8.4. Publicación

El programa contiene un código fuente totalmente comentado y empaquetado según los estándares de codificación de java.

Además contiene un manual donde se explica el funcionamiento del programa y un archivo README con las instrucciones rápidas para ejecutar el programa.

Dado que es una aplicación distribuida bajo la licencia GPL (*General Public Lincense*) para *software libre*, que da al usuario derechos tales como acceso al código fuente y libertad para modificarlo, el código fuente se puede obtener de la dirección electrónica:

http://sourceforge.net/projects/aupam

8.5. Instalación

8.5.1. Requisitos

Es necesaria una implementación completa de la edición estándar del entorno de tiempo de ejecución de Java 2 (JRE) versión 1.4.2 o superior.

8.5.2. Plataformas

Debido a que AUPAM es una aplicación Java, es independiente de plataforma y debería correr sin ningún problema en cualquier plataforma con una implementación completa del JRE versión 1.4 o superior. Entre las plataforma sobre las que se ha ejecutado de forma satisfactoria se encuentran:

- GNU/Linux, utilizando la implementación de Sun Microsystems de la edición estándar de la plataforma Java 2 (J2SE) 5.0 o JRE.
- GNU/Linux, utilizando la implementación de Sun Microsystems del JRE versión 1.4.2.
- Windows 98/ME/XP utilizando la implementación de Sun Microsystems de J2SE 5.0 JRE.
- Windows 98/ME/XP utilizando la implementación de Sun Microsystems de JRE versión 1.4.2.

8.5.3. Descarga

A continuación se muestra cómo instalar AUPAM en las distintas plataformas en las que ha sido ejecutado.

Preparación. Es necesaria una implementación completa de la edición estándar del entorno de tiempo de ejecución de Java 2 (JRE) versión 1.4 o superior. Si no se cuenta con dicha implementación será necesario conseguir una. La utilizada durante el desarrollo del sistema fue la de Sun Microsystems, que puede encontrarse en la página `http://java.sun.com`.

Instalación. Una vez que se cuenta con una implementación del JRE versión 1.4 o superior, siga los pasos para instalar el programa:

- Descargue el paquete binario llamado *aupam.1-01.bin.tgz* del sitio `http://aupam.sourcefourge.net`
- Expanda el paquete con la instrucción `tar xvfz aupam.1-01.bin.tgz` y ponga los archivos extraídos donde desee que quede instalada la herramienta.

Ahora debe poder ejecutar el programa utilizando la instrucción `java -jar aupam.1-01.jar` desde el directorio en el que se instaló el programa.

Si se está ejecutando el programa desde la plataforma Windows también se puede iniciar el programa dando dos clicks en el archivo `aupam.1-01.jar`.

8.6. Archivos y directorios

8.6.1. Paquete binario

El paquete binario contiene los siguientes archivos y directorios.

Figura 8.4: Contenido del paquete binario

examples. Directorio con algunos ejemplos para probar el programa.

aupam.1-01.jar. Archivo JAR ejecutable.

COPYING. Archivo con la licencia GPL.

MANUAL. Archivo con las instrucciones sobre cómo ejecutar el programa.

README. Archivo con una descripción rápida del programa.

8.6.2. Paquete de código fuente

El paquete con el código fuente contiene los siguientes archivos y directorios.

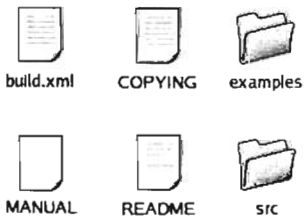


Figura 8.5: Contenido del paquete de código fuente

examples. Directorio con algunos ejemplos para probar el programa.

src. Directorio con el código fuente.

build.xml. Archivo utilizado para compilar el programa con las herramienta *Apache Ant.*

COPYING. Archivo con la licencia GPL.

MANUAL. Archivo con las instrucciones sobre cómo ejecutar el programa.

README. Archivo con una descripción rápida del programa.

8.7. Ejecución

Se describirá la funcionalidad del programa, así como las diferentes opciones que proporciona, utilizando uno de los archivos de ejemplo que viene con la distribución.

Para ejecutar el programa siga estos pasos:

1. Dentro del menú `File` utilice la opción `Open Text` y abra el archivo `PTTG.txt` que se encuentra en el directorio `examples`. El área de texto dentro del panel `Text` desplegará el contenido del archivo como se muestra en la figura 8.6.

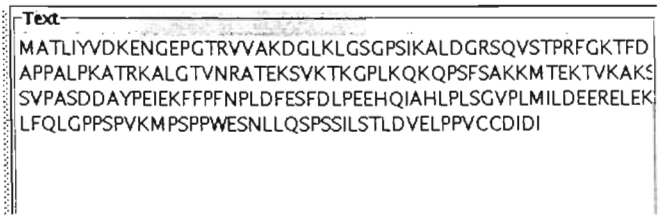


Figura 8.6: Area de texto plano

2. Dentro del menú `File` utilice la opción `Open Pattern` y abra el archivo `PSPP_pat.xml` que se encuentra en el directorio `examples`. El área de texto del panel `Pattern` desplegará el contenido del archivo como se muestra en la figura 8.7.

Figura 8.7: Area del patrón

También se pueden crear diferentes patrones, utilizando la opción `Build Pattern` del menú `File` como se muestra en la figura 8.8

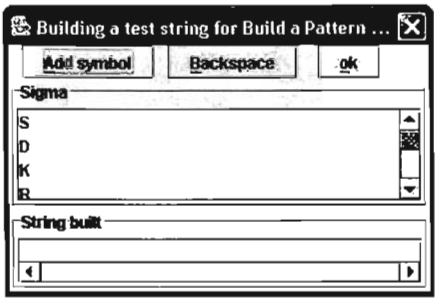


Figura 8.8: Ventana para crear un nuevo patrón

3. Del área `Options`, seleccione las siguientes opciones:

   - De `Distance`, seleccione la opción `Hamming`.
   - De `Version`, seleccione la opción `P Version`.
   - En `Number of errors`, escriba el número `1`.

   Una vez seleccionadas estas opciones, el área `Options` debe verse como en la figura 8.9.

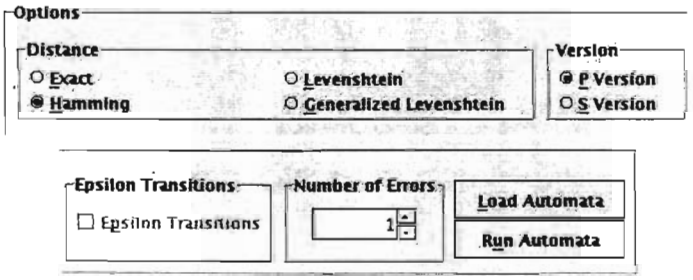


Figura 8.9: Area de Opciones

4. Presione el botón `Load Automata` del área `Options`.

   Una vez realizado esto, el área dentro del panel `Configuration` desplegará la configuración gráfica del autómata generado a partir del patrón buscado, como se muestra en la figura 8.10.

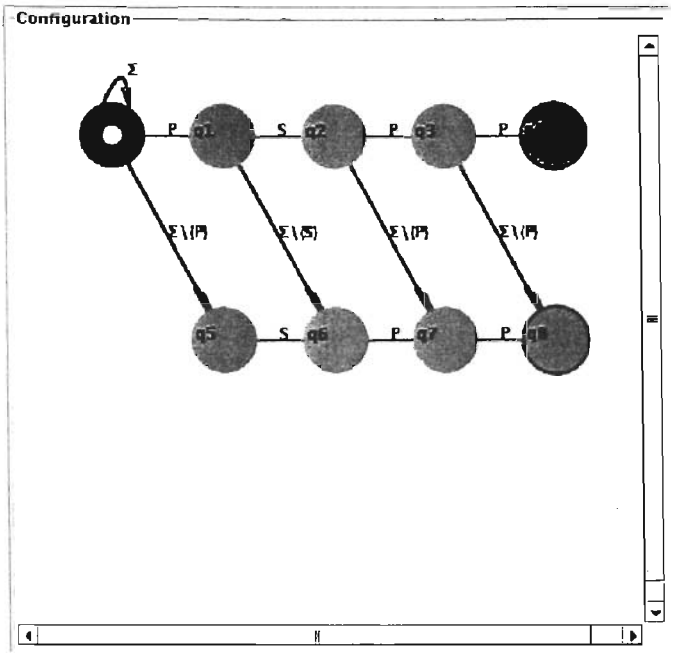


Figura 8.10: Configuración gráfica del autómata

5. Para ejecutar el autómata, presione el botón `Run Automata`, del área `Options`.

   Una vez realizado esto, se desplegará un diálogo con la información acerca del número de presencias del patrón buscado en el texto, como se muestra en la figura 8.11.

   También se sombrearán las cadenas encontradas dentro del texto, como se muestra en la figura 8.12

   Es importante señalar que el número de presencias mostradas en el diálogo de resultados puede no coincidir con el número de cadenas sombreadas en el texto. Esto se debe a que pueden existir presencias traslapadas dentro del texto.

6. Utilice la opción `Formal Definition`, dentro del menú `Extras`, para desplegar el diálogo con la definición formal del autómata como en la figura 8.13.

7. Utilice la opción `Automata Description`, dentro del menú `Extras`, para desplegar el diálogo con la descripción del autómata como en la figura 8.14.

El archivo PTTG.txt contiene la secuencia de aminoácidos de la proteína hPTTG1

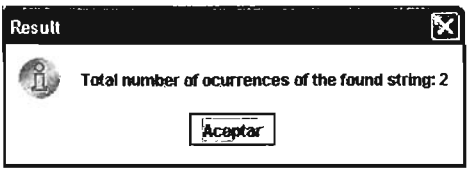


Figura 8.11: Presencias del patrón buscado dentro del texto

MATLIYVDKENGEPGTRVVAKDGLKLGSGPSIKALDGRSQVSTPRFGKTF
DAPPALPKATRKALGTVNRATEKSVKTKGPLKQKQPSFSAKKMTEKTVK
AKSSVPASDDAYPEIEKFFPFNPLDFESFDLPEEHQIAHLPLSGVPLMILD
EERELEKLFQLGPPSPVKMPSPPVVESNLLQSPSSILSTLDVELPPVCCD
IDI

Figura 8.12: Cadenas encontradas dentro del texto

que corresponde a la proteína que codifica el gen de tumores, en la glándula Pituitaria humana.

Se sabe que todas las proteínas humanas contienen dos cadenas de aminoácidos PSPP con una variación en alguna de las posiciones.

Al ejecutar el programa con dichas entradas, se puede ver que la proteína PTTG en efecto contiene una cadena de aminoácidos PSPP y otra PSPV con la variación de V con P.

De esta forma, AUPAM resulta ser una herramienta de fácil uso que logra mostrar la relación que existe entre los autómatas finitos y el apareamiento de patrones en texto.

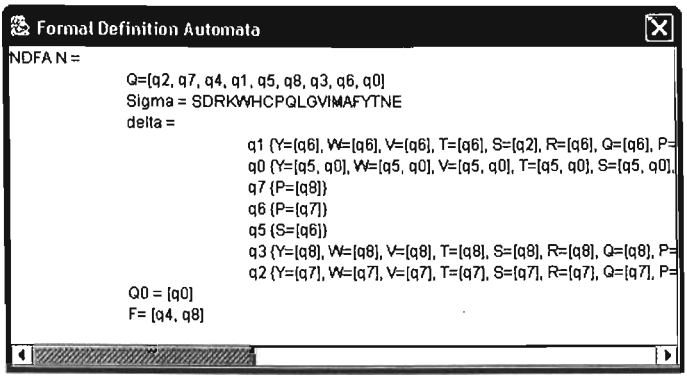


Figura 8.13: Diálogo con la definición formal del autómata

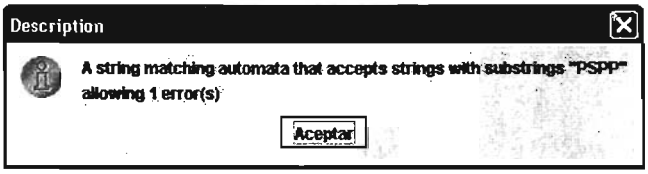


Figura 8.14: Diálogo con la descripción del autómata

Capítulo 9

Conclusiones

La búsqueda de texto en presencia de errores se realiza de manera exhaustiva probando todas las posibilidades, lo que resulta ser un proceso lento, frustante y sin garantía de éxito.

Los algoritmos para búsquedas con errores presentados resuelven dicho problema y realizan la búsqueda de manera robusta.

Se muestran autómatas finitos no determinísticos (AFN) para el apareamiento exacto de cadenas y el apareamiento aproximado de cadenas permitiendo un cierto grado de error en el texto, utilizando las distancias de Hamming, Levenshtein y Levenshtein generalizada.

Para el apareamiento aproximado de cadenas se muestran dos versiones de algoritmos: una llamada versión-$\bar{p}$, la cual utiliza transiciones con el complemento del símbolo p del alfabeto de entrada, y otra llamada versión-Σ, la cual utiliza transiciones con el alfabeto de entrada completo.

Ambas versiones de autómatas finitos no determinísticos evitan las presencias triviales, causadas al agregar símbolos incorrectos detrás de una presencia.

La teoría de los autómatas finitos puede ser utilizada en el diseño y construcción de sistemas para analizar grandes archivos de texto y encontrar la presencia de palabras, frases o incluso patrones con errores.

Dichos sistemas se pueden encontrar todo el tiempo, en uno de un número finito de estados. El propósito de un estado es poder recordar la porción relevante ya leída del texto. Debido a que sólo hay un número finito de estados el texto completo no puede ser cargado completamente la mayoría de las veces; de modo que el sistema debe recordar los fragmentos importantes que se apareen con el patrón y olvidar aquellos que no son importantes.

La ventaja de tener un número finito de estados es que el sistema puede ser implementado con un número fijo de recursos permitiendo tomar decisiones con tan solo ver una limitada parte de los datos de entrada, en este caso el texto.

Con base en lo anterior, se creó una herramienta que permite automatizar el proceso de búsqueda de patrones en texto, además de mostrar de manera visual el autómata finito que se genera a partir del patrón, lo que facilita el entendimiento de la relación que existe entre los autómatas finitos y el apareamiento de patrones.

Más allá de la utilidad que pudiera tener una herramienta de este estilo, se tienen características deseables en cualquier programa:

- Es fácil de modificar, pues el código fuente es modular y está completamente comentado.
- Utiliza formatos estándar en los archivos de texto que carga, lo que facilita su manejo por parte del usuario.
- Es independiente de plataforma por ser una aplicación Java.

Esperamos haber contribuido a una mejor comprensión, tanto de los autómatas finitos como del apareamiento de cadenas de texto.

9.1. Trabajo Futuro

Debido al gran número de posibles problemas que involucra el apareamiento de cadenas, el trabajo a futuro sobre el tema y el estudio de las diferentes soluciones resulta igual de extenso.

Entre los posibles estudios futuros se podría seguir cualquiera de las siguientes líneas:

Transformar los AFN en sus AFD equivalentes. Los autómatas finitos no pueden ser utilizados directamente debido a su no determinismo, de manera que es posible transformarlos en sus autómatas finitos determinísticos equivalentes (AFD), aunque se genera una explosión en el número de estados del AFD.

Hacer una simulación de la ejecución del autómata. Otra forma de utilizar los autómatas finitos es haciendo una simulación de la ejecución del autómata; entre ellos se encuentran el método de *Programación Dinámica* y el algoritmo de *Shift-Or*.

Construcción de autómatas finitos que acepten partes de cadenas. Entre los autómatas finitos que aceptan partes de cadenas se pueden encontrar los autómatas de prefijo, autómatas de sufijo, autómatas de factor y autómatas de subsecuencias.

Reducciones sobre el autómata finito. El estudio en la reducción del número de estados del autómata finito no determinístico para el apareamiento aproximado de cadenas, así como la reducción de tamaño en el alfabeto de entrada y la función de transición producen un impacto significativo en los algoritmos de simulación y transformación del autómata.

Apareamiento de expresiones regulares. La generalización del problema de apareamiento de cadenas resulta ser el apareamiento de expresiones regulares. Así, es posible extender la definición de las distancias de Hamming y Levenshtein entre dos cadenas utilizadas para el apareamiento aproximado de cadenas, de modo que las distancias puedan ser utilizadas en el apareamiento aproximado de expresiones regulares.

Construcción de estructuras de datos persistentes. Las razones típicas para evitar el guardar estructuras de datos persistentes, tales como índices, son los requerimientos de espacio extra, la volatilidad del texto (la construcción de los índices es costosa y debe ser amortizada sobre muchas búsquedas) y resultan muchas veces inadecuados (la velocidad que proporcionan los índices no siempre es satisfactoria); sin embargo existen muchos estudios relacionados con el tema.

Construcción de algoritmos para otro tipo de búsquedas. Algoritmos tales como algoritmos paralelos o probabilísticos; búsquedas sobre textos multidimensionales, en gráficas o búsquedas sobre multipatrones.

Bibliografía

[1] Baeza-Yates, Ricardo. A Unified View to String Matching Algoritms.Technical Report, Dept. of Computer Science, University of Chile, 1995.

[2] Cormen, Thomas and Leiserson, Charles and Rivest, Ronald and Stein, Clifford. Introduction to algorithms. Cambridge, massachusetts : MIT : New York : McGraw-Hill, Chapter 34, 1990.

[3] Friedl, Jeffrey. Mastering regular expressions : powerful techniques for Perl and other tools.Cambridge O'Reilly, c1997.

[4] Gusfield, Dan. Algorithms on strings, trees and sequences: Computer science and computational biology. Cambridge University Press. 1997.

[5] Holub, J. Approximate string matching in text. Master's thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, 1996.

[6] Holub, Jan. Reduced nondeterministic finite automata for approximate string matching. Proceedings of the Prague Stringology Club Workshop 1996, pages 19-27, Czech Technical University, Prague, Czech Republic, 1996.

[7] Holub, Jan. Simulation of NFA in approximate string and sequence matching. Proceedings of the Prague Stringology Club Workshop 1997, pages 39-46, Czech Technical University, Prague, Czech Republic, 1997.

[8] Holub, Jan. Simulation of nondeterministic Finite Automata in Pattern Matching. Dissertation Thesis. Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, 2000.

[9] Hopcroft, J. and Ullman, J. Introduction to automata theory, languages and computation. Addison-Wesley, 2000.

[10] Kozen, D. Automata and Computability. Springer-Verlag, Berlin 1997.

[11] Manber, Udi. Introduction to algorithms: A creative aproach, Addison Wesley Publishing Co., p234-244, 1989.

[12] Manber, Udi. Wu, Sun. Fast text searching allowing errors. Commun. ACM, pages 83-91, 1992.

[13] Melichar, B. 6D classification of pattern matching problems. Proceedings of the Prague Stringology Club Workshop 1997, pages 24-32, Czech Technical University, Prague, Czech Republic, 1997.

[14] Melichar, B. and Holub, Jan. and Polcar, Tomas. Text Searching algorithms: Tutorial for Athens course. Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, 2004..

[15] Navarro, Gonzalo. A Guide Tour to Approximate String Matching. University of Chile.

[16] Sipser, Michael. Introduction to the theory of computation. PWS Publishing Company, 1996.

[17] Skiena, Steven. The algorithm design manual. Springer-Verlag, New York. 1997.

[18] Ukkonen, E. Algorithms for approximate string matching. Inf. Control 64, pages 100-118. 1985.

[19] Viso, Elisa. Autómatas y lenguajes formales. Notas para el curso Teoría de la computación. Facultad de ciencias, UNAM. Vinculos matemáticos No. 223, 2000.