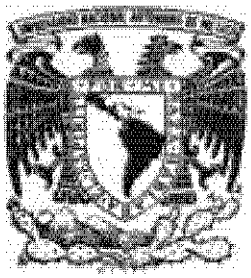
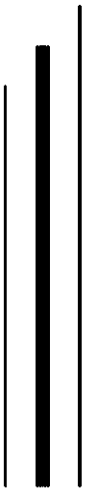


UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS



Búsquedas en Texto con Expresiones Regulares

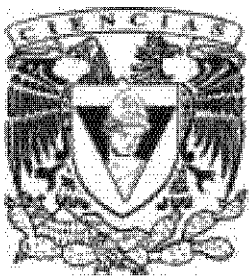
T E S I S

QUE PARA OBTENER EL TÍTULO DE

Licenciada en Ciencias de la Computación

P R E S E N T A:

Fabiola Zárate Mendoza



Tutora:
Dra. Elisa Viso Gurovich

2006

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Hoja de Datos del Jurado

1 Datos del alumno
Zárate
Mendoza
Fabiola
57435150
Universidad Nacional Autónoma de México
Facultad de Ciencias
Lic. en Ciencias de la Computación
098298090

2 Datos del tutor
Dra
Elisa
Viso
Gurovich

3 Datos de sinodal 1
Dr
Sergio
Rajsbaum
Gorodezky

4 Datos de sinodal 2
Dr
José de Jesús
Galaviz
Casas

5 Datos de sinodal 3
L. en C.C.
Iván
Hernández
Serrano

6 Datos de sinodal 4
L. en C.C.
Francisco Lorenzo
Solsona
Cruz

7 Datos del trabajo escrito
Búsquedas en Texto con Expresiones Regulares
76 p
2006

A mis padres: *Juana y Victoriano*

A mi hermano: *Victor Humberto*

Agradecimientos

Doy gracias a Dios por permitirme cerrar una etapa más de mi vida.

A mis padres, muchas gracias por todo, no hay palabras para agradecerles todo su amor.

Muchas gracias a la Dra. Elisa Viso por su ayuda, paciencia y comprensión durante toda la carrera, en especial durante la realización de este trabajo.

Gracias a todos los que me han brindado de su ayuda, apoyo, amistad y cariño.

Y a todos los que lean este trabajo espero lo disfruten, es el fruto de muchos días y noches de esfuerzo.

Gracias.

Índice general

Capítulo 1

Motivación

1.1. Herramientas

Uno de los orígenes de las *ciencias de la computación* es la búsqueda de la automatización y sistematización del pensamiento. El concepto de *algoritmo* es uno de los frutos más antiguos y más importantes de dicha búsqueda. Algoritmo es un conjunto finito de pasos discretos bien definidos que resuelven un *problema*. Es un instrumento abstracto de coordinación que describe procedimientos para varios fines. En palabras más sencillas: una receta, una guía, un código. Los algoritmos son artefactos humanos, cuyos objetos teóricos fundamentales son los *símbolos*. Curiosa y convenientemente todo lo que se puede representar en una computadora moderna se hace a través de *cadenas* o secuencias de símbolos. Entonces los *lenguajes*, que no son sino colecciones de cadenas, han llegado a ser el centro de las ciencias de la computación.

La formalización del concepto de algoritmo dio origen a nuevas preguntas y con ello a nuevos campos de estudio. Uno de ellos es la *complejidad computacional*. Su objetivo es medir o acotar la complejidad de un algoritmo principalmente en cuanto al tiempo o espacio que se requiere para ejecutarlo. Se ha logrado clasificar los problemas en función de la complejidad para obtener su solución. Un problema que puede ser resuelto en tiempo polinomial pertenece a la clase de complejidad **P** (orden polinomial), es decir, que el número de pasos que toma ejecutar el algoritmo que lo resuelve está acotado por un polinomio en función del tamaño de su *entrada*. Por otro lado, dentro de la clase de complejidad **NP** (no polinomial) están los problemas cuyas posibles soluciones pueden ser *verificadas* en tiempo polinomial pero no necesariamente encontradas en tiempo polinomial. **P** representa a los problemas tratables y **NP** a aquellos problemas que no son tratables; las relaciones que hay entre estas dos clases de complejidad siguen siendo una

maravillosa fuente de investigación.

Hoy en día el análisis y diseño de algoritmos es una de las áreas más importantes en ciencias de la computación, que de una u otra forma no se puede dejar de lado pues de ella dependen la *eficiencia* y el buen funcionamiento de todo programa de computadora, sin importar su área de aplicación. Aunque la capacidad de memoria y velocidad de procesamiento se incrementan día con día, al mismo tiempo surgen nuevos problemas, en diversas disciplinas, que requieren más y más poder de cómputo para ser resueltos, pero el poder de cómputo siempre estará limitado, es un recurso finito. Es por ello la importancia de la búsqueda de algoritmos *eficientes* donde, como ya mencioné, la eficiencia se considera en función de la complejidad en tiempo, ya que hay problemas que sin ser tratables (elementos de **NP**) son eficientes, esto debido a que no existen algoritmos *equivalentes* que tomen menor tiempo en llegar a la solución (elementos de **P**).

1.2. Objeto de estudio

Nuestro lenguaje natural tiene reglas que lo generan que para la mayoría de nosotros suelen pasar desapercibidas. Seguramente nadie antes de pronunciar una frase se esfuerza por poner un sujeto, un verbo y un complemento. Éste es el ejemplo más simple y conocido. Sin embargo hay variedad de lenguajes: en computación los lenguajes de programación, en matemáticas el lenguaje algebraico, en biología las estructuras moleculares o de ADN, en física las atómicas, y aún en las artes como la música. Un lenguaje tiene que ser construido sobre elementos (símbolos) de un conjunto llamado *alfabeto* y así como los lenguajes son distintos unos de otros, los alfabetos también.

A los lenguajes que pueden ser generados por un conjunto de reglas se les llama *lenguajes formales*. Los lenguajes formales son fruto también de la sistematización matemática.

Hay una clase muy especial dentro de ésta, los *lenguajes regulares*, que son aquellos que pueden ser generados por *expresiones regulares*, *autómatas finitos* o por *gramáticas regulares*; resultan ser el conjunto más restringido dentro de la Jerarquía de Chomsky.

1.3. Búsquedas en texto con expresiones regulares

La aplicación del análisis y diseño de algoritmos puede ser tan vasto como se quiera. Este trabajo pretende enfocarse en los algoritmos para procesamiento de texto:

búsquedas con expresiones regulares.

La búsqueda con expresiones regulares en texto es una generalización de la búsqueda de cadenas y mientras para la segunda los algoritmos son de orden lineal para la primera los algoritmos se elevan a *orden exponencial* dependiendo de los operadores permitidos para su generación.

1.4. Una aplicación importante.

Dos ramas de la ciencia, biología y ciencias de la computación, antes ajenas, se han unido principalmente por el descubrimiento del *código genético*. La biología no contaba con los conocimientos teóricos, mucho menos con las herramientas para enfrentarse por sí sola a tan gran reto. El algoritmo vino a ser el punto de unión, justo la herramienta que buscaba la biología.

El proyecto del genoma humano ha dado lugar a diversidad de problemas de cómputo. La información generada es tanta que aún las computadoras más poderosas no se dan abasto para procesarla. Se ha vuelto un problema fundamental encontrar algoritmos que reduzcan el tiempo de procesamiento y que permitan encontrar patrones dentro de las largas secuencias de ADN. Esto ha permitido el surgimiento de una nueva área de investigación: *la biología computacional*.

La biología computacional es el uso de las ciencias de la computación para resolver problemas biológicos, enfocándose en el desarrollo de algoritmos y métodos computacionales específicos. Los esfuerzos más importantes de investigación en el campo incluyen: análisis y alineación de secuencias, búsqueda de genes (proteínas y ARN), montaje del genoma (assembly), alineación y predicción de estructuras de proteínas, predicción de expresiones de genes, interacciones proteína-proteína y otros. Se vale de herramientas matemáticas para extraer información útil a partir de datos sin significado aparente producidos por técnicas biológicas de súper-procesamiento.

El ADN mismo se puede ver como un algoritmo codificado que describe el proceso de vida de todo organismo viviente. Las bases **A** (adenina), **T** (*timina*), **G** (*guanina*) y **C** (*citosina*) funcionan como símbolos; los *aminoácidos* y las *proteínas* son los elementos, palabras o expresiones a buscar, son el mensaje expresado en código. Las proteínas son *moléculas* complejas compuestas de varios aminoácidos. Hay 20 tipos posibles de aminoácidos en total, y algunas proteínas son cadenas de hasta 250 residuos de aminoácidos. Por lo tanto hay aproximadamente 20^{250} tipos de proteínas de donde escoger. Un

algoritmo es un esquema para la manipulación de símbolos y todos estos símbolos son instrumentos que transmiten información.

Capítulo 2

Introducción

El objetivo principal de este trabajo es estudiar algunos de los algoritmos más importantes para búsquedas con expresiones regulares, poniendo mucho mayor interés en el algoritmo de Glushkov.

Comenzamos por analizar las definiciones formales de *alfabeto*, *cadena* y *lenguaje*, básicas en el estudio de los lenguajes formales, así como las operaciones entre éstos. Por ejemplo, en el caso de los alfabetos sería la potencialización; para las cadenas la concatenación y para los lenguajes la unión, entre otras.

⋆ **Alfabeto:** Conjunto no nulo y finito de símbolos, generalente representado por Σ.

⋆ **Cadena:** Secuencia finita de elementos de un alfabeto. Si la secuencia es nula, se trata de una cadena especial, la *cadena vacía*, representada por ε (algunos autores usan λ). ε no puede pertenecer al alfabeto de la cadena.

⋆ **Lenguaje:** Conjunto finito de cadenas, representado por letras mayúsculas, por ejemplo, L.

El interés de analizar estas definiciones es dejar en claro los elementos necesarios para definir un lenguaje. Un lenguaje se compone de cadenas y las cadenas toman los símbolos de un alfabeto dado.

Las operaciones con lenguajes son también importantes porque por medio de ellas obtenemos nuevos lenguajes. Son tres las más importantes: *unión*, *concatenación* y *cerradura*. Más adelante veremos la relación de cada una de ellas con las expresiones regulares.

El *apareamiento de cadenas* o *búsqueda de cadenas*, es el problema de encontrar todas las presencias de un *patrón* (una o más palabras) en un *texto*.

Cuando queremos búscar una palabra en un texto, podemos pensar en dos tipos de búsqueda, la exacta y la aproximada. Si nuestro texto es totalmente confiable, es decir, no hay datos corrompidos, nos iremos por la *búsqueda exacta*. En otro caso, si hay interferencia en la transmisión de la información y no recibimos los datos completos, o bien hay ruido, y por lo tanto, datos corrompidos, nos conviene usar la *búsqueda aproximada*.

Las búsquedas con expresiones regulares son una generalización de las búsquedas de una cadena o de un conjunto de cadenas. Para ver más claramente dicha generalización, daremos un breve paseo por algunos algoritmos que trabajan sobre una cadena: fuerza bruta, Karp-Rabin, Knuth-Morris-Pratt y Boyer-Moore; y los de un conjunto de cadenas: Aho-Corasick.

En ese momento pasaremos a la definición de *expresión regular*. Para ello, dedicamos un capítulo completo al estudio de las expresiones regulares. Las expresiones regulares surgen de la necesidad de poder representar de manera sencilla los elementos de un lenguaje, es decir, una expresión regular es una expresión algebraica que puede describir un conjunto de cadenas sin listar todos sus elementos.

Además de los lenguajes, hay otra área de computación teórica que está muy relacionada con las expresiones regulares, se trata de los *autómatas finitos*. Un autómata finito es un dispositivo abstracto que tiene un conjunto finito de *estados* y un *control* que se mueve de un estado a otro como respuesta a entradas externas. El autómata es *determinístico* si en cada unidad de tiempo el control se encuentra en un sólo estado. Y un autómata es *no determinístico* si en cada unidad de tiempo el control se encuentra en uno o más estados. Cada cambio de estado está definido por transiciones y se produce al leer un símbolo (entrada) del alfabeto asociado al autómata. Hay un tipo especial de autómatas que pueden cambiar de estado sin necesidad de leer un símbolo del alfabeto, en este caso decimos que el autómata tiene *transiciones vacías*.

Una vez que entendemos todos los términos relacionados con lenguajes, expresiones regulares y autómatas finitos, estamos listos para entrar al tema central de este trabajo: el algoritmo de búsqueda con expresiones regulares.
El algoritmo consiste en construir el autómata equivalente a nuestra expresión regular y después usar dicho autómata para procesar el texto en el que se desea hacer la búsqueda.

Para construir un autómata equivalente a nuestra expresión regular veremos y analizaremos dos métodos: el de Glushkov y el de Thompson. El autómata de Glushkov, aunque toma más tiempo en la construcción, resulta más efectivo al realizar la búsqueda, porque no tiene transiciones vacías, y en algunos casos puede resultar determinístico.
La ventaja del autómata de Thompson es que tiene una construcción más directa y toma menos tiempo construirlo. La desventaja es que tiene transiciones vacías, lo que lo hace menos eficiente, en tiempo y espacio, al realizar la búsqueda.

En otro capítulo analizamos los algoritmos para contruir autómatas de Thompson a partir del algoritmo de Glushkov, y para simplificar autómatas de Thompson para obtener un autómata de Glushkov. Esto con el fin de quede más claro por qué un autómata de Glushkov es una simplificación de uno de Thompson. Concluimos pues que resulta más eficiente en ejecución el autómata de Glushkov.

Observando toda la información, finalizamos con el desarrollo de una herramienta de software, RESS (*Regular Expression Search System*), que sirve para hacer búsquedas con expresiones regulares. RESS usa el autómata de Glushkov como motor de búsqueda. Dicha herramienta pretende materializar toda la teoría presentada a lo largo este trabajo.

Capítulo 3

Fundamentos

Siempre que en ciencias de la computación buscamos la solución a un problema, primero tenemos que conocer los conceptos básicos que son necesarios para poder representar matemáticamante el problema y poder encontrar los distintos entornos de desarrollo de las soluciones.

Comenzaremos este capítulo con algunas definiciones referentes a lenguaje, cadena, símbolo, alfabeto, operaciones con cadenas, y autómatas. Para profundizar más en estos temas ver [13] y [17].

Como la intuición nos indica, un lenguaje es un conjunto de *palabras* (*secuencias* de *símbolos* que pertenecen a un *alfabeto*) que obedecen un conjunto de *reglas gramaticales*. Las reglas gramaticales de las que hablamos se refieren a la *sintaxis* del lenguaje, es decir, a la estructura del lenguaje y al orden que deben seguir los símbolos en las palabras. El significado de las palabras y las interpretaciones del significado basadas en la estructura son la *semántica* del lenguaje. En este trabajo estaremos más interesados en la estructura de un lenguaje. Por lo tanto pondremos más interés en la sintaxis que en la semántica. Consideremos el lenguaje español, que es una combinación de (1) un diccionario el cual da significado a cada una de las *palabras* en el lenguaje y (2) un conjunto de reglas que determina cuales combinaciones de palabras y signos de puntuación forman un enunciado en el lenguaje.

3.1. Alfabetos

Un *alfabeto* o *vocabulario* es un conjunto finito de símbolos con al menos un elemento. Para denotar a un alfabeto usaremos la letra griega Σ. Por ejemplo,

1. $\Sigma = \{0, 1\}$, el alfabeto binario.
2. $\Sigma = \{a, b, ..., z\}$, nuestro alfabeto español en minúsculas.
3. $\Sigma = \{A, T, C, G\}$, el conjunto de las bases del ADN.
4. El conjunto de todos los carácteres ASCII.

3.2. Cadenas

Una *cadena* (también llamada *palabra*) es una secuencia finita de símbolos que pertencen a algún alfabeto. Si consideramos el alfabeto del ADN, $\Sigma = \{A, T, C, G\}$, las tripletas GCT, GAG, CGA y TCA son ejemplos de cadenas. 101 es otro ejemplo de cadena pero sobre el alfabeto binario, $\Sigma = \{0, 1\}$.

La cadena vacía
La *cadena vacía* es la cadena con cero símbolos. Esta cadena, que denotaremos ε, no pone restricciones al alfabeto y por lo tanto el alfabeto sobre el cual es construida puede ser cualquiera.

Longitud de una cadena
El conjunto de posiciones de una cadena u resulta de enumerar los símbolos de la cadena de izquierda a derecha $u = u_1 u_2 \cdots u_n$. Por ejemplo la cadena $GCTGAG$ se enumera $G_1 C_2 T_3 G_4 A_5 G_6$ y su conjunto de posiciones es $\{1, 2, 3, 4, 5, 6\}$.
La *longitud* de una cadena es el *número de símbolos*[1] que la forman. Por ejemplo, la longitud de $GCTGAG$ es 6. La longitud de una cadena u se denota $|u|$.
Ejemplos: $|\varepsilon| = 0$, $|GCTGAG| = 6$ y $|10111| = 5$.

Potencias de un alfabeto
Para representar el conjunto de todas las cadenas sobre un alfabeto Σ que tienen la misma longitud k, usaremos la notación exponencial Σ^k.

Para cualquier alfabeto, $\Sigma^0 = \{\varepsilon\}$.
Si $\Sigma = \{A, T, C, G\}$ entonces

1. $\Sigma^1 = \{A, T, C, G\}$

[1] Este enunciado no es necesariamente equivalente a *número de símbolos distintos*, ya que la longitud de $GCTGAG$ es 6 y no 4.

2. $\Sigma^2 = \{AA, AT, AC, AG, TA, TT, TC, TG, CA, CT, CC, CG, GA, GT, GC, GG\}$
3. $\Sigma^3 = \{AAA, ATA, ACA, AGA, ATA, ATT, ATC, ATG, ACA, ACT, ACC, ACG, AGA, AGT, AGC, AGG, TAA, TTA, TCA, TGA, TTA, TTT, TTC, TTG, TCA, TCT, TCC, TCG, TGA, TGT, TGC, TGG, CAA, CTA, CCA, CGA, CTA, CTT, CTC, CTG, CCA, CCT, CCC, CCG, CGA, CGT, CGC, CGG, GAA, GTA, GCA, GGA, GTA, GTT, GTC, GTG, GCA, GCT, GCC, GCG, GGA, GGT, GGC, GGG\}$

La diferencia entre Σ y Σ^1 es que los elementos del primero son símbolos y los del segundo son cadenas de longitud 1 sobre el alfabeto Σ.

El conjunto de todas las cadenas sobre un alfabeto Σ se denota Σ^* y podemos expresarlo en términos de potencias como

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

Cuando queremos excluir a la cadena vacía usamos

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$$

que denota al conjunto de cadenas no vacías sobre un alfabeto Σ.
Y la relación entre estos dos conjuntos está dada por

$$\Sigma^* = \Sigma^+ \cup \{\varepsilon\}.$$

Concatencación de cadenas
La *concatenación* de dos cadenas u y v, denotada con uv, es la cadena que resulta de copiar u seguida por una copia de v.
Por ejemplo, $u = TCA$ y $v = TAG$ son cadenas sobre el alfabeto $\Sigma = \{A, T, C, G\}$, la concatenación de u y v es $uv = TCATAG$. La longitud de uv es la suma de las longitudes de u y v; por ejemplo $|uv| = |u| + |v| = 6$.
Para cualquier cadena u, las expresiones $\varepsilon u = u\varepsilon = u$ son equivalentes. Es decir, ε es la identidad para la concatenación pues al concatenarla con cualquier cadena el resultado es la otra cadena. Por ejemplo, si $u = ATC$, $u\varepsilon = ATC$[2].

Prefijos, factores y sufijos de cadenas Dadas tres cadenas u, v y w, decimos que u es un prefijo de uv, u es un factor de vuw y u es un sufijo de wu. Pero si además $v \neq \varepsilon$, u es un prefijo propio de uv, es factor propio de vuw si $v \neq \varepsilon$ ó $w \neq \varepsilon$; finalmente, si $w \neq \varepsilon$, u es sufijo propio de wu.
Por ejemplo, si u=TCATT, entonces

[2] ε es un símbolo que representa la ausencia de símbolos del alfabeto. En un lenguaje de programación sería equivalente a la cadena `ep="";`

- u es un prefijo de u;
- ε es un prefijo propio de u;
- TC es un prefijo propio de u;
- u es un factor de u;
- ε es un factor propio de u;
- CAT, A, AT, ATT son factores de u;
- u es sufijo de u;
- ε es sufijo propio de u;
- CATT, ATT, TT y T son sufijos propios de u.

3.3. Lenguajes

Un *lenguaje* es un conjunto de cadenas tomadas de Σ^*, donde Σ es un alfabeto particular. Si Σ es un alfabeto y L es un subconjunto de Σ^*, entonces L es un *lenguaje sobre* Σ. Notemos que el alfabeto Σ podría contener más símbolos de los que aparecen en las cadenas de L, así que una vez que se ha establecido que L es un alfabeto sobre Σ podemos también afirmar que L es un lenguaje sobre cualquier conjunto que contenga a Σ.
Ejemplos comunes de lenguajes son los lenguajes de programación, como C o Java, y los lenguajes naturales español e inglés, donde los alfabetos son para los primeros el conjunto de carácteres ASCII y para los segundos el conjunto de todas las letras.
Los siguientes son ejemplos más abstractos de lenguajes:

1. $\varnothing$ es el lenguaje vacío sobre cualquier alfabeto.
2. $\{\varepsilon\}$ es un lenguaje con una cadena, la cadena vacía, y también es un lenguaje sobre cualquier alfabeto.
3. Σ^* es un lenguaje para cualquier alfabeto Σ.
4. Σ^3, con $\Sigma = \{A, T, C, G\}$ es el lenguaje de las tripletas de bases de ADN.
5. El conjunto las cadenas binarias con n ceros seguidas por el mismo número de unos, para $n \geq 0$:
$$\{\varepsilon, 01, 0011, 000111, \ldots\}.$$
6. Las cadenas binarias con un número par de ceros y un número impar de unos:
$$\{\varepsilon, 1, 11, 001, 010, 100, ...\}$$

Recordemos que los alfabetos son conjuntos finitos; sin embargo, los lenguajes pueden ser conjuntos infinitos. La restricción que tienen es que las cadenas tienen que estar formadas sólo por símbolos del alfabeto.

A continuación veremos tres de las operaciones con lenguajes que nos serán útiles en secciones posteriores.

Unión
La *unión* de dos lenguajes L y M, que se denota $L \cup M$, es el conjunto de cadenas que pertenecen a L o a M, o ambos. Por ejemplo, $L = \{\varepsilon\}$, $M = \{\varepsilon, 1, 11, 111....\}$ y $L \cup M = \{\varepsilon, 1, 11, 111\}$, la cadena 11 pertence a $L \cup M$ porque 11 pertenece a M.

Concatenación
La *concatenación* de dos lenguajes L y M se forma tomando cualquier cadena de L y concatenándola con cualquier cadena de M. Todas las cadenas resultantes son los elementos de la concatenación. Si $L = \{A, T\}$ y $M = \{C, G\}$, $LM = \{AC, AG, TC, TG\}$. Una cadena u pertenece a la concatenación LM si u es la concatenación de dos cadenas v y w, $u = vw$, con v en L y w en M. Ejemplos:

1. $u = 01$ pertenece a LM con donde L es el conjunto de todas las cadenas con ceros y M es el conjunto de cadenas formadas con unos. $v = 0 \in L$, $w = 1 \in M$.
2. $u = ATTGC$ pertenece a LM, L es el conjunto de las tripletas y M el de las parejas de bases. $v = ATT \in L$ y $w = GC \in M$.

Cerradura
La *cerradura* (o estrella de Kleene) de un lenguaje L se denota L^* y representa al conjunto que se forma concatenando cualquier número de cadenas, diferentes o repetidas, de L. Más formalmente L^* es la unión infinita

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

donde $L^0 = \{\varepsilon\}$, $L^1 = L$, y para $i > 1$ es $LL \cdots L$ (la concatenación de i copias de L). Para calcular L^* es necesario calcular L^i para toda $i \geq 0$, y tomar la unión de todos estos lenguajes. L^i tiene $|L|^i$ elementos.
Ejemplos:

1. Para un lenguaje finito $L = \{AT, TA, CG, GC\}$,
   - $L^0 = \{\varepsilon\}$, esto es para cualquier lenguaje; es la selección de cero cadenas del lenguaje L.
   - $L^1 = \{AT, TA, CG, GC\}$, resulta de escoger una cadena.
   - $L^2 = \{ATAT, ATTA, ATCG, ATGC, \ldots, GCGC\}$, tomamos dos cadenas, posiblemente la misma, y las concatenamos. Como $|L| = 4$, el número de cadenas que contiene este conjunto es $4^2 = 16$.

- $L^3 = \{ATATAT, ATATTA, ATATCG...GCGCGC\}$ con $4^3 = 64$ cadenas.
- $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cdots$ es el conjunto infinito que resulta de unir conjuntos finitos.

2. Para $L = \{0, 10\}$ también finito,
   - $L^0 = \{\varepsilon\}$ con $|L|^0 = 2^0 = 1$ elementos.
   - $L^1 = \{0, 10\}$ con 2^1 elementos.
   - $L^2 = \{00, 010, 100, 1010\}$ con 2^2 elementos.
   - $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cdots$ es el conjunto infinito que resulta de unir conjuntos finitos.
3. El lenguaje $L = \{0, 00, 000, 0000, ...\}$ es un conjunto infinito ($|L| = \infty$) de todas las cadenas de ceros.
   - $L^0 = \{\varepsilon\}$ con 1 elementos.
   - $L^1 = \{0, 00, 000, 0000, ...\}$ con un número infinito de elementos.
   - $L^2 = \{00, 000, 0000, ...\}$
   - L^* es el conjunto infinito que resulta de unir, en su mayoría, conjuntos infinitos. Notemos que $L^* = \{\varepsilon\} \cup L$, el único elemento que realmente es agregado a L^* es ε.
4. $L = \varnothing$ entonces
   - $L^0 = \{\varepsilon\}$
   - $L^1 = \varnothing$
   - $L^2 = \varnothing$, note que $\varnothing^i = \varnothing$ para $i \geq 1$
   - $L^* = \{\varepsilon\}$ es un conjunto finito de un elemento. De hecho, $\varnothing$ es uno de los dos lenguajes cuya cerradura no es infinita; el otro lenguaje es $\{\varepsilon\}$.

Usualmente denotamos a los elementos de un alfabeto con letras minúsculas, pero en biología computacional un alfabeto común, que representa el conjunto de las bases del ADN, contiene las letras mayúsculas: A, T, C, G. También es usual un alfabeto de 20 letras mayúsculas que denotan a los veinte aminoácidos esenciales que conforman las proteínas.

Hablamos de cadenas y alfabetos siempre finitos debido a que no tiene sentido considerar cadenas o alfabetos *infinitos*; todos los algoritmos que vamos a presentar necesitarían memoria infinita para almacenar este tipo de cadenas, además de no poder garantizar que el proceso terminaría en tiempo finito. Aunque los textos en áreas como biología computacional tienden a ser cadenas mucho muy largas y cada día tienden a aumentar, siempre podrán ser acotados por un entero.

3.4. Autómatas Finitos

Otra de las bases para el desarrollo de este trabajo es la teoría de autómatas. Un autómata finito es un modelo matemático que puede ser simulado con un programa de computadora. Dicho modelo surge en situaciones físicas en las que se procesan señales que conllevan información.

Un *autómata finito no determinístico* (AFN) es una 5-tupla $M = (Q, \Sigma, \delta, I, F)$ donde:

- Q es un *conjunto finito de estados*,
- Σ es un alfabeto,
- $I \subseteq Q$ es el conjunto de estados iniciales,
- $F \subseteq Q$ es el conjunto de estados finales,
- $\delta : Q \times \Sigma \rightarrow Q$ es la función de transición que asigna un nuevo conjunto de estados a cada pareja de la forma *(estado, símbolo)*.

Un autómata $M = (Q, \Sigma, \delta, I, F)$ es *determinístico* (AFD) si y sólo si

- hay un único estado inicial: $I = \{q_0\}$, y
- para todo $(q, a) \in Q \times \Sigma$ hay a lo más un estado p tal que $\delta(q, a) = p$

El funcionamiento consiste en que al pasar un símbolo de la *entrada* (secuencia de símbolos) al autómata, éste produce un cambio de estado o estados, que depende exclusivamente de:

1. El símbolo que está procesando.
2. Los estados en los que se encuentra el autómata (*estados actuales*).
3. La función de transición del autómata es la que determina el nuevo conjunto de estados actuales del autómata.

La diferencia entre AFN y AFD es el número de estados en los que pueden encontrarse en cada instante de tiempo. Un AFD sólo puede encontrarse en un estado, mientras que un AFN puede estar en más de un estado.

En el capítulo 5 extenderemos este tema y en capítulos posteriores daremos el contexto del apareamiento con expresiones regulares, que es el tema central en este trabajo.

3.5. Apareamiento de cadenas

El apareamiento de cadenas es el problema de encontrar todas las presencias de un *patrón* (una o más cadenas) dentro de un *texto*. Tanto el patrón como el texto son cadenas

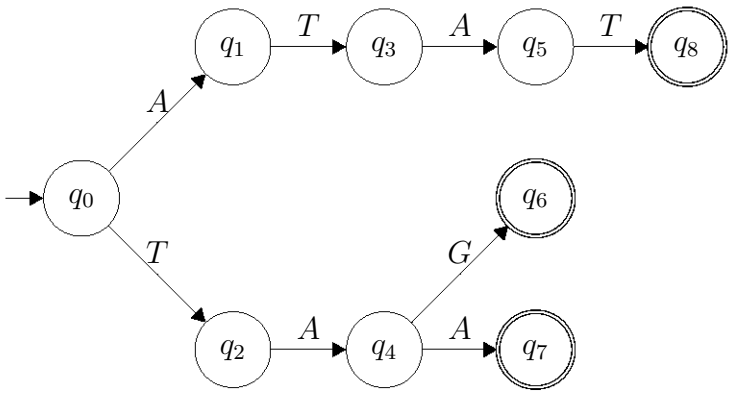


Figura 3.1: Representación gráfica de un autómata

sobre el mismo alfabeto y la búsqueda tiene sentido si el texto es más largo que la cadena.

Las soluciones para el apareamiento de cadenas se pueden clasificar de distintas formas dependiendo de las características del patrón y del texto, así como del método de análisis del texto en función del patrón. Continuaremos describiendo dos de los enfoques más importantes en el apareamiento de cadenas.

3.6. Apareamiento exacto contra apareamiento aproximado

Dependiendo del estado del texto y del patrón se pueden usar dos tipos de algoritimos para el apareamiento de cadenas según convenga. Si el texto t no está completo o es inexacto, la búsqueda del patrón u dentro del texto tiene que permitir errores, es decir, se deben buscar cadenas *parecidas* a u. Este tipo de búsqueda, en la que el patrón o el texto están incompletos o han sufrido algún tipo de corrupción, es el *apareamiento aproximado de cadenas* y tiene aplicaciones muy importantes en biología computacional, procesamiento de señales, recuperación de texto y minería de datos.
En cambio, si el texto y el patrón están íntegros, la búsqueda es exacta y no se permiten errores; éste es el caso del *apareamiento exacto de cadenas*. Por ejemplo, cuando el texto se trata de una base de datos o un diccionario, generalmente se parte de que los datos son exactos ó están en rangos predeterminados.

3.7. Apareamiento exacto

Dependiendo del patrón que se quiere buscar podemos identificar tres grupos: cuando el patrón es una cadena, cuando es un *conjunto finito de cadenas* y cuando es una *expresión regular*. Más adelante trabajaremos extensivamente con expresiones regulares.

3.7.1. Algoritmos con una cadena o un conjunto de cadenas

En este tipo de algoritmos el patrón que se va a buscar en el texto es una cadena o un conjunto de cadenas con al menos dos elementos, para que podamos distiguir un caso del otro. Estos casos son los más sencillos de tratar. Las primeras ideas de búsqueda que surgieron en el área de procesamiento de textos, fueron estas dos y las soluciones fueron surgiendo de manera intuitiva, y de lo más sencillo a lo más sofisticado. Nuevas preguntas han surgido: cómo *optimizar* los algoritmos; reducir el tiempo de búsqueda; reducir el espacio o memoria requeridos por el algoritmo; el sí puede usarse tiempo constante o espacio constante para la búsqueda, independientemente del alfabeto, el texto y/o los patrones; el sí ayuda en algo hacer un preproceso del patrón o del texto. Muchas respuestas se han obtenido, y muchas otras se siguen buscando.

En muchos de los algoritmos que vamos a revisar en esta sección se usa una *ventana de búsqueda* de la misma longitud que el patrón, que se desliza a través del texto de izquierda a derecha o de derecha a izquierda. Sirve para delimitar el segmento de texto que se está procesando en cada paso del algoritmo, ya que el patrón se busca en la ventana.

Para los algoritmos de apareamiento exacto de una cadena o un conjunto de cadenas hay tres métodos generales de búsqueda.

Búsqueda de prefijos
La búsqueda se lleva a cabo de adelante hacia atrás en la ventana de búsqueda, leyendo todos los caracteres del texto uno después de otro. Para cada posición de la ventana se busca el prefijo más largo de la ventana que sea al mismo tiempo prefijo del patrón.
Este método lo usan entre otros, los algoritmos de: fuerza bruta, Karp-Rabin, Knuth-Morris-Pratt y Aho Corasick, entre otros.

Búsqueda de sufijos
La búsqueda se hace de atrás hacia adelante en la ventana de búsqueda, leyendo el sufijo más largo de la ventana que sea también sufijo del patrón. Con este método podemos, en promedio, evitar leer algunos carácteres del texto. Los algoritmos que usan este método son de orden sublineal en el caso promedio.

Considerado como el algoritmo más eficiente para el apareamiento de cadenas en aplicaciones usuales, el algoritmo más famoso que usa esta técnica es el de Boyer-Moore, así como todas las variantes y simplificaciones que se le han hecho. Se implementa en editores de texto para los comandos búsqueda y sustitución.

Búsqueda de factor
La búsqueda se hace de atrás para adelante en la ventana de búsqueda considerando el sufijo más largo de la ventana que es factor del patrón. Al igual que en la búsqueda de sufijos, para los algoritmos que usan ésta búsqueda se espera en promedio tiempo sublineal de ejecución.

La eficiencia de estos tres métodos está en función de la longitud del patrón y del texto, así como del tamaño del alfabeto y la periodicidad del patrón.
Comparando la longitud del patrón contra la del texto, el patrón tiende a ser mucho menor. Muchos algoritmos aprovechan esta propiedad para ganar eficiencia en el tiempo total de ejecución al hacer un preproceso del patrón, que en los mejores casos representa un costo despreciable por estar en función de la longitud del patrón, y almacenar la información que se requiere para el apareamiento.
En todos los métodos antes explicados, es de gran ayuda conocer de antemano los prefijos, sufijos y/o factores del patrón que se espera aparezcan en la ventana de búsqueda durante el apareamiento. La diferencia de complejidad entre los algoritmos que hacen el preproceso y los que no, es de un orden. Los primeros llegan a tiempo cuadrático, mientras que los segundos son de orden lineal, gracias al preproceso. Y como podemos estar hablando de textos con millones de carácteres, esta diferencia es significativa. Entonces el preproceso es un elemento clave para reducir el tiempo de ejecución del algoritmo, aunque en la mayoría de los algoritmos que usan esta técnica necesitan $O(m)$ espacio adicional para almacenar la información obtenida del patrón, donde m es el tamaño del patrón. Se reduce el tiempo y aumenta el espacio.

En la siguiente tabla (3.1) se hace un análisis de algunos de los algoritmos más conocidos y se define de manera breve cuáles son los casos promedio y los peores casos para cada tipo de algoritmo, así como la complejidad en cada caso.

3.7.2. Algoritmos con expresiones regulares

Éste es el caso general que contiene a los dos anteriores. Los patrones pueden no ser numerables y se representan usualmente por medio de *expresiones regulares*. Debido al incremento en la complejidad de los algoritmos, sólo deben usarse cuando el patrón no se puede representar como una cadena simple o un conjunto finito de cadenas.

Algoritmo	Cadenas	Método	Preproceso	Tiempo
Fuerza bruta	1	Prefijos	0	$O(nm)$, dado que es una búsqueda exhaustiva, no influye la forma del texto o del patrón.
Karp-Rabin	1	Prefijos	$O(m)$	Caso promedio: $O(n + m)$, cuando el texto es aleatorio. Peor caso: $O(nm)$, cuando hay presencias del primer carácter del patrón en todo el texto.
Knuth-Morris-Pratt	1	Prefijos	$O(m)$	$O(n)$, no repite la lectura de carácteres en el texto.
Boyer-Moore	1	Sufijos	$O(m)$	Caso promedio: $O(n/m)$ cuando el tamaño del alfabeto es grande comparado con la longitud del patrón y las presencias de los carácteres en el texto son aleatorias. Peor caso: $O(n)$, en alfabetos reducidos y repeticiones consecutivas de los carácteres.
Aho-Corasick	$r > 1$	Prefijos	$O(m)$, m=suma de las longitudes de las r cadenas	$O(m+n+k)$, k es el número de presencias de las cadenas en el texto, extensión del algoritmo de Knuth-Morris-Pratt para conjunto de cadenas.

Cuadro 3.1: Algoritmos de una cadena y de un conjunto de cadenas ([2]).

Los métodos usados para resolver este tipo de apareamiento difieren en mucho de los mencionados anteriormente. El método más común es construir a partir de una expresión regular dada, un autómata finito no determinístico (AFN) que la reconozca. Kleene se encargó de demostrar que esto siempre es posible ([17]); y varios autores han dado su

propia variante de la construcción ([9] [10] [16]).

Es muy importante mencionar que la investigación hecha acerca de los autómatas que reconocen una expresión regular ha hecho posible la optimización de estos algoritmos. Los más famosos y más usados son el de *Glushkov* [10] y el de *Thompson* [16].

Algunos de los algoritmos para apareamiento con expresiones regulares son multietapas. En la primera etapa se obtiene la representación en árbol y apartir de ésta se construye el autómata finito no determinístico que la acepta.

Si tomamos el método clásico que ya mencionamos, para buscar un expresión regular de longitud m en un texto de longitud n, obtendremos un *autómata finito no determinístico* (AFN) con $O(m)$ *estados*. En el peor caso, hacer la búsqueda usando el autómata, tiene una complejidad en tiempo de $O(mn)$. El costo viene del hecho de que en cada paso se puede activar más de un estado del AFN y por lo tanto podría ocurrir que todos los estados estén activados simultáneamente. Un método más eficiente en tiempo, pero no en espacio, es convertir el AFN en un *autómata finito determinístico* (AFD), que tiene un estado activado en cada paso, lo cual reduce el tiempo de búsqueda a $O(n)$. El problema con este método es que el AFD tiene $O(2^m)$ estados, lo cual implica un costo de preprocesamiento y espacio adicional exponencial en m.

Las expresiones regulares se usan en aplicaciones como recuperación de texto y biología computacional para representar patrones de búsqueda que son más complicados que una cadena o un conjunto de cadenas.

Posteriormente estudiaremos las propiedades de las expresiones regulares, los autómatas finitos y los lenguajes regulares, así como sus transformaciones y equivalencias.

3.8. Apareamiento aproximado

El apareamiento aproximado también conocido como apareamiento de cadenas con errores es el problema de encontrar un patrón p en un texto t permitiendo un número acotado k de *diferencias* entre el patrón y sus presencias en el texto.

La definición de diferencia más común es la de *distancia de Levenshtein* o *distancia de edición*. Dadas dos cadenas x, y, la distancia de edición $ed(x, y)$ se define como el mínimo número de operaciones de edición necesarias para convertir x en y o viceversa. Las operaciones de edición son:
* Inserción de un carácter.

* Borrar un carácter.
* Sustituir un cáracter por otro.
Por ejemplo, ed(`computacion`,`computable`)=4.

`computacion`	
`computabion`	*substituir* `c` *por* `b`
`computablon`	*substituir* `i` *por* `l`
`computablen`	*substituir* `o` *por* `e`
`computable`	*borrar* `n`

El apareamiento aproximado de cadenas para un texto t, un patrón p y un entero k, consiste en encontrar todas las cadenas p' que cumplen $ed(p, p') \leq k$, con $0 < k < m$ donde $m = |p|$.

Este trabajo no pretende profundizar en este tema. Para concluir sólo mencionaremos cuatro de los métodos principales para los algoritmos de apareamiento aproximado. El primero, que es también el más flexible, implementa un algoritmo de programación dinámica para calcular la distancia de edición. El segundo método usa un autómata finito no determinístico para representar el problema y lo resuelve simulándolo. El tercero, uno de los más exitosos, está basado en la simulación por paralelismo de bits de otros métodos, principalmente del segundo. El último consiste en filtrar el texto, eliminando grandes porciones del mismo, y después ejecutar un algoritmo que busque en las regiones del texto que no fueron descartadas.

Capítulo 4

Expresiones regulares

En este capítulo enfocaremos nuestra atención en las *expresiones regulares*, una de las herramientas principales en el desarollo de este trabajo. Primero daremos la definición formal para posteriormente analizar las propiedades algebraicas de dichas expresiones.

El origen de las expresiones regulares reside en la teoría de autómatas y en la teoría de los lenguajes formales, de la necesidad teórica y práctica de poder representar de manera sencilla los elementos de un lenguaje. Stephen Kleene y Ken Thompson formalizaron el concepto matemático y su notación respectivamente.

Una expresión regular puede describir un conjunto de cadenas sin listar todos sus elementos. Si recordamos que los lenguajes son conjuntos de cadenas, entonces las expresiones regulares son una descripción algebraica para algunos lenguajes; más adelante demostraremos formalmente a qué tipo de lenguajes son capaces de describir.

4.1. Construcción

El álgebra de las expresiones regulares está formada por expresiones elementales: constantes y variables; y un conjunto de operaciones que se aplican a éstas para construir más expresiones. Las constantes y las variables denotan lenguajes, y las operaciones son: unión, concatenación y cerradura (ver sección 3.3). Usaremos los paréntesis para agrupar operadores con sus operandos.

Vamos a definir recursivamente a las expresiones regulares como sigue. En esta definición, además de describir lo que son las expresiones regulares, para cada expresión

E describimos el lenguaje que representa y lo denotamos $L(E)$.

- **BASES:** Las bases consisten de tres partes:
  1. Las constantes $\boldsymbol{\varepsilon}$ y $\boldsymbol{\varnothing}$ son expresiones regulares, denotan a los lenguajes $\{\varepsilon\}$ y $\varnothing$, respectivamente. Esto es, $L(\boldsymbol{\varepsilon}) = \{\varepsilon\}$, y $L(\boldsymbol{\varnothing}) = \varnothing$.
  2. Si a es un símbolo, entonces $\boldsymbol{a}$ es una expresión regular. Esta expresión denota al lenguaje $\{a\}$. Usaremos letras negritas para diferenciar una expresión de un símbolo.
  3. Una variable, usualmente mayúscula y con letra cursiva como L, representa a algún lenguaje.
- **INDUCCIÓN:** Hay cuatro partes para el paso inductivo, una para cada uno de los tres operadores y uno para la introducción de paréntesis.
  1. Si E y F son expresiones regulares, entonces $(E) + (F)$ es una expresión regular que denota la unión de $L(E)$ y $L(F)$. Es decir, $L((E) + (F)) = L(E) \cup L(F)$.
  2. Si E y F son expresiones regulares, entonces $(E)(F)$ es la expresión regular que denota la concatenación de $L(E)$ y $L(F)$. Es decir, $L((E)(F)) = L(E)L(F)$.
  3. Si E es un expresión regular, entonces $(E)^*$ es una expresión regular que denota la cerradura de $L(E)$. Es decir, $L(E^*) = (L(E))^* = \bigcup_{i=0}^{\infty} L^i(E)$.
  4. Si E es una expresión regular, entonces (E), que es la parentización de E, es también una expresión regular y denota el mismo lenguaje que E. Formalmente $L((E)) = L(E)$.

Veamos algunos ejemplos. Vamos a construir la expresión para el lenguaje que consiste de un número impar de ceros, $\{0, 000, 00000, \ldots\}$. Observemos que la longitud de cada cadena corresponde a $2n+1$ para alguna $n \geq 0$ y, que $\mathbf{0}$ y $\mathbf{00}$ son las expresiones regulares que corresponden a los conjuntos de cadenas $\{0\}$ y $\{00\}$ respectivamente. $\mathbf{00}$ se obtiene de concatenar las expresiones $\mathbf{0}$ y $\mathbf{0}$; y para obtener $2n$ para cualquier n usaremos el operador $*$; por lo que $(\mathbf{00})^*\mathbf{0}$ es la expresión regular cuyo lenguaje es el conjunto de cadenas con un número impar de ceros.

Ahora, para obtener la expresión regular que corresponde al lenguaje de las tripletas de ADN, el alfabeto es $\Sigma = \{A, C, G, T\}$. $\mathbf{A}$, $\mathbf{C}$, $\mathbf{G}$ y $\mathbf{T}$ son las expresiones regulares que denotan a los conjuntos $\{A\}$, $\{C\}$, $\{G\}$ y $\{T\}$, respectivamente. Si para cada posición hay 4 posibles expresiones a considerar, entonces $\mathbf{A} + \mathbf{C} + \mathbf{G} + \mathbf{T}$ denota la unión de los cuatro lenguajes básicos anteriores. La expresión completa se obtiene concatenando tres veces la expresión regular anterior, es $(\mathbf{A} + \mathbf{C} + \mathbf{G} + \mathbf{T})(\mathbf{A} + \mathbf{C} + \mathbf{G} + \mathbf{T})(\mathbf{A} + \mathbf{C} + \mathbf{G} + \mathbf{T})$.

	Lenguaje	Expresión regular
1.	$\{1\}$	$(\mathbf{1}) = \mathbf{1}$
2.	$\{1\} \cup \{0\}$	$(\mathbf{1}) + (\mathbf{0}) = \mathbf{1} + \mathbf{0}$
3.	$\{10\}$	$(\mathbf{1})(\mathbf{0}) = \mathbf{10}$
4.	$\{\varepsilon, 1, 11, ...\}$	$(\mathbf{1})^* = \mathbf{1}^*$
5.	$\{\varepsilon, 1, 11, ...\} \cup \{\varepsilon, 0, 00, ...\}$	$((\mathbf{1})^*) + ((\mathbf{0})^*) = \mathbf{1}^* + \mathbf{0}^*$
6.	$\{\varepsilon, 1, 0, 11, 10, 01, 00,$ $111, 110, 100, 011, 001, ...\}$	$((\mathbf{1}) + (\mathbf{0}))^* = (\mathbf{1} + \mathbf{0})^*$
7.	$\{\varepsilon, 1, 0, 11, 00, 10,$ $111, 000, 110, 100, ...\}$	$((\mathbf{1})^*)((\mathbf{0})^*) = \mathbf{1}^*\mathbf{0}^*$
8.	$\{\varepsilon, 10, 1010, 101010, ...\}$	$((\mathbf{1})(\mathbf{0}))^* = (\mathbf{10})^*$
9.	$\{1, 10, 100, 1000, 10000, ...\}$	$(\mathbf{1})((\mathbf{0})^*) = \mathbf{10}^*$
10.	$\{0, 00, 21, 211, 210, 000, 2110,$ $2100, 2111, 0000, ...\}$	$((\mathbf{2})((\mathbf{1})((\mathbf{1})^*))((\mathbf{0})^*)) + ((\mathbf{0})((\mathbf{0})^*))$ $= \mathbf{211}^* + \mathbf{0}^* + \mathbf{00}^*$

Cuadro 4.1: Expresiones regulares

El uso de paréntesis es necesario para asegurar la agrupación apropiada de los operadores.

4.2. Precedencia de operadores

Surge un punto importante para discutir, la precedencia de los operadores. Estos deben conservar el orden de las operaciones sobre lenguajes: concatenación, unión y cerradura. El orden de precedencia de un conjunto de operaciones se refiere a reglas que indican el orden en el que dichas operaciones serán evaluadas en una expresión, es decir, el orden en el que los operadores son asociados con sus operandos.

El orden de precedencia para expresiones regulares es el siguiente:

1. El operador estrella es el de mayor precedencia. Significa que será aplicado a la secuencia de símbolos más corta que esté a su izquierda. La secuencia de símbolos debe ser una expresión regular que cumpla con la definición recursiva.
2. La siguiente operación es la concatencación. Después de agrupar todos los operadores estrella con sus operandos, agrupamos las concatenaciones con sus operandos. Como el operador concatenación no tiene un símbolo explícito[1], basta con

[1] En algunos textos se usa $\cdot$ (punto) para denotar a la concatencación

agrupar las expresiones que son adyacentes, es decir, las expresiones que no tienen un operador entre ellas. La concatencación es un operador asociativo, esto hace que no importe el orden en que agrupamos las concatenaciones consecutivas, pero podemos acordar que la agrupación sea de izquierda a derecha. Por ejemplo ATG se agrupa $(AT)G$.

3. Por último, agrupamos todas las uniones (operador +) con sus operandos. Como la unión también es asociativa, no importa el orden en el agrupemos uniones consecutivas, pero también lo haremos empezando por la izquierda.

Habrá ocasiones en las que no queremos la agrupación que nos da la precedencia de los operadores. Si es así, podemos usar los paréntesis para agrupar los operandos como sea requerido. Por supuesto que, debido a la definición de expresión regular, no puede haber errores al agregar paréntesis, ni siquiera cuando la agrupación deseada esté implicada por las reglas de precedencia.

En los ejemplos del cuadro 2 borramos los paréntesis cuando el orden de precedencia nos da la agrupación deseada, pero los conservamos cuando es necesario modificar el orden de agrupación para obtener el conjunto requerido en cada caso.

Comparando los conjuntos de los ejemplos 8 y 9 vemos claramente que $(\mathbf{10})^* \neq \mathbf{10}^*$. Por ejemplo, 10 está en los dos conjuntos, pero 101010 sólo pertenece al primero.

La expresión $\mathbf{01}^* + \mathbf{1}$ se agrupa $(\mathbf{0}(\mathbf{1})^*) + \mathbf{1}$. El operador estrella se agrupa primero. Como el símbolo $\mathbf{1}$ que está inmediatamente a su izquierda es una expresión regular válida, es el operando de la estrella. La siguiente operación es la concatenación entre $\mathbf{0}$ y $\mathbf{1}^*$, dada por $(\mathbf{0}(\mathbf{1}^*))$. Finalmente, el operador unión conecta esta última expresión y la expresión $\mathbf{1}$ de más a la derecha.
Notemos que el lenguaje denotado por $\mathbf{01}^* + \mathbf{1}$, obedece la agrupación de la reglas de precedencia y representa al conjunto de la cadena 1 más todas las cadenas formadas por un cero seguido de cualquier número de unos. Sin embargo, si agregamos paréntesis que modifiquen el orden de precedencia, obtenemos lenguajes distintos.

- $(\mathbf{01})^* + \mathbf{1}$ es el lenguaje de la cadena 1 mas todas las cadenas que intercalan 01, cero o más veces.
- $\mathbf{0}(\mathbf{1}^* + \mathbf{1})$ es el lenguaje de todas la cadenas que comienzan con 0 y que enseguida tienen cualquier número de unos.

Expresión regular	Evaluación
$\varnothing^*$	$\varnothing$
ε^*	ε
$\mathbf{0}$	0
$\mathbf{0+1}$	$0, 1$
$\mathbf{01}$	01
$\mathbf{1}^*$	ε, 1, 11, 111, 1111 ...
$(\mathbf{A+T+C+G})(\mathbf{A+T+C+G})(\mathbf{A+T+C+G})(\mathbf{A+T+C+G})$	$AAAA$, $AAAT$, $AAAC$, $AAAC$, $AAAG$, $AATA$, $AATT$, $AATC$, $AATG$, $AACA$, $AACT$, $AACC$, $AACG$, ..., $GAAA$, $GAAT$, $GAAC$, $GAAC$, $GAAG$, $GATA$, $GATT$, $GATC$, $GATG$, $GACA$, $GACT$, $GACC$, $GACG$, ..., $GGGG$
$(\mathbf{0+1})^*$	ε, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, ..., 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 ...

Cuadro 4.2: Ejemplos de evaluación de expresiones regulares

4.3. Leyes algebraicas para expresiones regulares

Ahora surge la pregunta, ¿dada una expresión regular, será posible encontrar otra equivalente? Pensemos que la expresión regular con la que estamos trabajando es muy compleja, es decir muy larga, y quisiéramos poder simplificarla de manera que el lenguaje que denota no se modifique, pero que la longitud de la expresión disminuya para poder manipularla más fácilmente. En esta sección mostraremos por qué una expresión puede ser reemplazada por otra. El problema fundamental es saber cuándo dos expresiones son *equivalentes* en el sentido de que describan el mismo lenguaje. Veremos un conjunto de leyes algebraicas que nos ayudarán a definir formalmente la equivalencia entre expresiones regulares. En vez de analizar expresiones regulares específicas, vamos a considerar expresiones con variables como argumentos. Dos expresiones con variables son equivalentes si al sustituir las variables por lenguajes cualesquiera el resultado de las dos expresiones es el mismo lenguaje.

4.3.1. Asociatividad y conmutatividad

Conmutatividad es la propiedad de un operador que nos indica que podemos cambiar el orden de sus operandos y que vamos a obtener el mismo resultado. Por ejemplo para las expresiones aritméticas tenemos que $x + y = y + x$. *Asociatividad* es la propiedad de

un operador que permite reagrupar los operandos cuando el operador es aplicado más de una vez. Para las expresiones aritméticas, las leyes asociativas de la multiplicación son $(x \times y) \times z = x \times (y \times z)$.
Leyes conmutativas y asociativas para expresiones regulares:

I. $L + M = M + L$. *Ley conmutativa para la unión*: podemos tomar la unión de dos lenguajes en cualquier orden.
II. $(L + M) + N = L + (M + N)$. *Ley asociativa para la unión*: para obtener la unión de tres lenguajes, podemos comenzar por tomar la unión de los dos primeros, o la de los dos últimos. Notemos que a partir de la ley conmutativa para la unión, concluimos que podemos tomar la unión de cualquier colección de lenguajes sin importar el orden y la agrupación, y el resultado será el mismo lenguaje.
III. $(LM)N = L(MN)$. *Ley asociativa para concatencación*: para concatenar tres lenguajes podemos comenzar concatenando los dos primeros o los dos últimos.

Observemos que faltó la ley de conmutatividad para concatenación, $LM = ML$. Esta ley no siempre se cumple para expresiones regulares. Por ejemplo, las expresiones regulares **01** y **10** denotan a los conjuntos $\{01\}$ y $\{10\}$ respectivamente; estos lenguajes son diferentes, y $LM \neq ML$.

4.3.2. Identidad y elemento nulo

La *identidad* para un operador es un valor tal, que cuando el operador se aplica a cualquier valor, el resultado es el otro valor. En aritmética, el operador identidad para la suma es 0 puesto que $x + 0 = 0 + x = x$, y 1 es la identidad para la multiplicación porque $1 \times x = x \times 1 = x$. Un *elemento nulo* para un operador es un valor tal que cuando el operador es aplicado al elemento nulo y algún valor, el resultado es el elemento nulo. Siguiendo con las operaciones aritméticas, 0 es un elemento nulo para la multiplicación, porque $0 \times x = x \times 0 = 0$. No hay elemento nulo para la suma.
Hay tres leyes para expresiones regulares relacionadas con estos conceptos:

I. $\varnothing + L = L + \varnothing = L$. El elemento identidad para la unión es la expresión regular $\varnothing$, que denota al lenguaje vacío.
II. $\varepsilon L = L\varepsilon = L$. La identidad para la concatencación es ε.
III. $\varnothing L = L\varnothing = \varnothing$. El elemento nulo para la concatencación es $\varnothing$.

No hay elemento nulo para la unión. Estas leyes son de gran utilidad en la simplificación de expresiones. Por ejemplo, si tenemos la unión de varias expresiones, alguna de las cuales es, o ha sido simplificada a $\varnothing$, dicha expresión puede ser borrada de la unión.

$$E_1 \cup E_2 \cup E_3 \cup \cdots \cup E_n = E_1 \cup \varnothing \cup E_3 \cup \cdots \cup E_n = E_1 \cup E_3 \cup \cdots \cup E_n$$

O bien, si tenemos la concatencación de varias expresiones, alguna de las cuales es, o ha sido simplificada a ε, podemos borrar dichas expresiones de la concatenación. Finalmente, si tenemos la concatenación de cualquier número de expresiones, y alguna de ellas es $\varnothing$, entonces la concatenación completa puede ser reemplazada por $\varnothing$.

$$E_1E_2E_3\cdots E_n = E_1\varnothing E_3\cdots E_n = \varnothing$$

4.3.3. Leyes distributivas

En una *ley distributiva* intervienen dos operadores, uno de los cuales puede ser distribuido al ser aplicado a cada argumento del otro operador individualmente. El ejemplo más común en aritmética es la ley distributiva de la multiplicación sobre la suma, que es, $x \times (y + z) = x \times y + x \times z$. Como la multiplicación es conmutativa, no importa si la multiplicación está a la derecha o a la izquierda de la suma. Sin embargo, debido a que la concatenación no es conmutativa, en la ley análoga para expresiones regulares tenemos que considerar dos casos. Las leyes distributivas para expresiones regulares son:

- $N(L + M) = NL + NM$. *Ley distributiva por la izquierda de la concatenación sobre la unión.*
- $(L + M)N = LN + MN$. *Ley distributiva por la derecha de la concatenación sobre la unión.*

4.3.4. Ley idempotente

Un operador es *idempotente* si al aplicarlo a dos valores iguales el resultado es el mismo valor. En aritmética, los operadores de suma y multiplicación no son idempotentes. Los ejemplos comunes de idempotencia son la unión y la intersección. Para expresiones regulares tenemos la siguiente ley:

- $L + L = L$. *Ley de idempotencia para la unión*. Si tomamos la unión de dos expresiones idénticas, podemos reemplazarla por una copia de la expresión.

4.3.5. Equivalencia de expresiones regulares

A partir de las leyes que ya hemos visto podríamos proponer una infinidad de equivalencias, las cuales una vez que se hayan demostrado, serán a su vez nuevas leyes sobre expresiones regulares.
Para demostrar nuevas leyes, por ejemplo $E = M$, lo que se tiene que demostrar en realidad es la equivalencia entre $L(E)$ y $L(M)$. Dos expresiones regulares son iguales si el lenguaje que describen son el mismo. Para ver la equivalencia entre dos lenguajes

basta con verificar que tengan los mismos elementos.

Ejemplos:

1. $(L^*)^* = L^*$. Lo primero que haremos es sustituir la variable L por una expresión regular concreta, digamos $\boldsymbol{a}$, entonces obtenemos la siguiente expresión: $(\boldsymbol{a}^*)^* = \boldsymbol{a}^*$. $L((\boldsymbol{a}^*)^*)$ son todas las cadenas formadas por concatenar cadenas en el lenguaje $L(\boldsymbol{a}^*)$. Pero estas cadenas son a su vez compuestas por cadenas de $L(\boldsymbol{a})$. Entonces, las cadenas de $L((\boldsymbol{a}^*)^*)$ se forman concatenando cadenas de $L(\boldsymbol{a})$ y por lo tanto están en $L(\boldsymbol{a}^*)$.
2. $(\boldsymbol{\varepsilon}+L)^* = L^*$. Sustituimos L por $\boldsymbol{b}$, $(\boldsymbol{\varepsilon}+\boldsymbol{b})^* = \boldsymbol{b}^*$. Como $\varepsilon b = b\varepsilon = b$, es decir que, al concatenar cualquier cadena con ε no se afecta la cadena original, entonces $\boldsymbol{\varepsilon}$ se puede omitir de la expresión $(\boldsymbol{\varepsilon} + \boldsymbol{b})^*$ y sólo falta verificar que $L(\boldsymbol{b}^*)$ conteniene a ε, pero por definición $L(\boldsymbol{b})^0 = \varepsilon$. Por lo tanto $L((\boldsymbol{\varepsilon} + \boldsymbol{b})^*) = L(\boldsymbol{b}^*)$.
   O bien usando la definición de cerradura y desarrollando la expresión:
$$\begin{aligned}(\boldsymbol{\varepsilon} + L)^* &= (\boldsymbol{\varepsilon} + L)^0 + (\boldsymbol{\varepsilon} + L)^1 + (\boldsymbol{\varepsilon} + L)^2 + \cdots \\ &= \boldsymbol{\varepsilon} + (\boldsymbol{\varepsilon} + L) + (\boldsymbol{\varepsilon} + L)^2 + \cdots \\ &= \boldsymbol{\varepsilon} + L + L^2 + \cdots \\ &= L^*\end{aligned}$$

4.4. Árboles de parseo

La definición de un árbol está dada por las siguientes consideraciones:

Definiciones

- ⋆ Un *nodo* es la base sobre la que se construye un árbol y puede tener cero o más nodos hijos conectados a él.
- ⋆ Un nodo a es *padre* o *ancestro* de un nodo b, si existe un enlace desde a hasta b (en ese caso, también decimos que b es un nodo *hijo* o *descendiente* de a).
- ⋆ Un *árbol* es:
  a) Un nodo que no tiene hijos (es al mismo tiempo raíz del árbol y hoja).
  b) Un nodo n que tiene como hijos a $k \geq 1$ árboles. Supongamos que $n_1, \ldots, n_k$ son los nodos ráiz de los árboles $A_1, \ldots, A_k$ respectivamente, el nodo n está conectado a $A_1, \ldots, A_k$ a través de $n_1, \ldots, n_k$. Cada uno de los árboles A_i es un *subárbol* de la *raíz* n.

La característica principal de un árbol es que un nodo puede tener cualquier número de hijos, pero cada nodo tiene a lo más un padre, es decir, que cada nodo tiene a un único nodo apuntando a él, pero puede apuntar a cero o más nodos hijos. Otra característica es que existe un único camino entre cualesquiera dos nodos.

Los nodos de un árbol se pueden clasificar en:

- *Raíz*. Un árbol sólo puede tener una raíz. La raíz es un nodo que no tiene *padre*.
- *Nodos internos*. Cualquier nodo que tiene padre y uno o más hijos es un nodo interno.
- *Hojas*. Son los nodos que no tienen hijos.

Definición de árbol de parseo
Parsear una expresión regular significa separarla en sus componentes (símbolos del alfabeto, símbolos que denotan operaciones y paréntesis), y recordar de alguna manera el orden de evaluación de los operandos.

Una expresión regular debe ser entendida por un *programa* no sólo como una cadena, sino como una secuencia de operaciones y operandos bien diferenciados unos de otros. Es decir que los símbolos que representan operaciones no pueden pertenecer al alfabeto y que los elementos del alfabeto son totalmente distiguibles unos de otros. Por ejemplo, para facilitar la lectura de las expresiones regulares hemos preferido no usar un símbolo de concatenación en nuestros ejemplos, pero para fines de programación sí suele usarse, esto con el fin de facilitar el *parseo* de la expresión regular.

Mostraremos cómo *parsear* una expresión regular para obtener su representación en árbol, la cual generalmente no es única, esto debido a que algunos operadores son conmutativos y/o asociativos. En el árbol, cada hoja está etiquetada por un símbolo de $\Sigma \cup \{\varepsilon\}$ y los nodos internos por un operador en el conjunto $\{+, \cdot, *\}$.
Hay herramientas clásicas de Unix como *Lex* y *Yacc*, o de Gnu como *Flex* y *Bison* para generar el autómata a partir de la gramática, y que reconocen la expresión regular y la transforman en árbol. Pero la gramática para las expresiones regulares es demasiado compleja para usar sólo un analizador léxico, y demasiado simple para usar un parser especializado. El mejor método es construir un analizador simple, y eso es lo que haremos. Damos el algoritmo en el Cuadro 4.3.

Supondremos que el símbolo $\boxed{\cdot}(v_l, v_r)$ representa un árbol de concatenación con ráiz "$\cdot$" e hijos v_l y v_r. Similarmente, $\boxed{+}(v_l, v_r)$ es el árbol con raíz "$+$". El símbolo $\boxed{*}(v_*)$ representa un nodo "$*$" con un hijo único v_*.

Usando la función **Parse** podremos convertir una expresión regular en un árbol que la represente. Esto nos será util en las secciones posteriores para los algortimos de búsqueda, los cuales recibirán como entrada un árbol de parseo.

Parse$(p = p_1 p_2 \dots p_m, last)$
1. $v \leftarrow \theta$
2. **While** $p_{last} \neq \$$ **Do**
3. **If** $p_{last} \in \Sigma$ OR $p_{last} = \varepsilon$ **Then** /* símbolos de $\{\Sigma \cup \varepsilon\}$ */
4. $v_r \leftarrow$ Crear un nodo con p_{last}
5. **If** $v \neq \theta$ **Then** $v \leftarrow \boxed{\cdot}\,(v, v_r)$
6. **Else** $v \leftarrow v_r$
7. $last \leftarrow last + 1$
8. **Else If** $p_{last} =' +'$ **Then** /* unión */
9. $(v_r, last) \leftarrow$ **Parse**$(p, last + 1)$
10. $v \leftarrow \boxed{+}\,(v, v_r)$
11. **Else If** $p_{last} =' *'$ **Then** /* cerradura */
12. $v \leftarrow \boxed{*}\,(v)$
13. $last \leftarrow last + 1$
14. **Else If** $p_{last} =' ('$ **Then** /* abre paréntesis */
15. $(v_r, last) \leftarrow$ **Parse**$(p, last + 1)$
16. $last \leftarrow last + 1$
17. **If** $v \neq \theta$ **Then** $v \leftarrow \boxed{\cdot}\,(v, v_r)$
18. **Else** $v \leftarrow v_r$
19. **Else If** $p_{last} =')'$ **Then** /* cerrar paréntesis */
20. **Return** $(v, last)$
21. **End of if**
22. **End of while**
23. **Return** $(v, last)$

Cuadro 4.3: Algoritmo para obtener el árbol de parseo de una expresión regular bien escrita [11]. θ es el árbol vacío.

Capítulo 5

Autómatas finitos

Todas las máquinas en la actualidad pueden tener sólo un número finito de componentes. Por lo tanto éstas pueden tener sólo un número finito de estados. Entonces cada una de ellas es una máquina finita de estados.

La teoría de autómatas es el estudio de dispositivos de cómputo abstractos, o también llamados *máquinas*. Antes del surgimiento de las computadoras, en 1930, A. Turing estudió una máquina abstracta que tenía todas las capacidades de una computadora de hoy en día, al menos en lo que una y otra pueden computar. Después, entre 1940 y 1950, algunos investigadores comenzaron a estudiar máquinas más simples que las de Turing, las que hoy en día llamamos *autómatas finitos*. Entre 1950 y 1960 se comenzó a trabajar en el problema de la traducción mecánica de lenguajes *naturales*, tales como el Inglés, Ruso, Chino, y Alemán. Antes de que éstos pudieran ser traducidos, tenía que estudiarse la estructura de los enunciados en dichos lenguajes. Por eso a finales de 1950, el lingüista N. Chomsky comenzó el estudió de las *gramáticas formales*, desarrollando varios modelos para la estructura ligüística. Algunos de estos modelos se usan para resolver el problema de la traducción de programas de computadora.

El estudio de los autómatas finitos es una parte importante de este trabajo debido a la equivalencia que existe entre los lenguajes que puede describir un autómata finito y los que describen las expresiones regulares.

Dado que los autómatas finitos tienen aplicaciones en diversas áreas de ciencias de la computación y de ingeniería, existen distintas definiciones formales. Todas estas definiciones tienen en común aceptar que los autómatas finitos pretenden modelar instrumentos de cómputo que tienen memoria finita y que leen sucesiones de símbolos construidas a partir de un alfabeto específico.

Un autómata finito tiene un conjunto finito de estados y su *control* se mueve de un estado a otro en respuesta a *entradas* externas. Una de las diferencias más importantes entre clases de autómatas finitos es que el control puede ser *determinístico* o *no determinístico*. Más adelante en esta sección mostraremos la equivalencia entre estas dos clases.

5.1. Autómata finito determinístico

El término *determinístico* se refiere al hecho de que por cada entrada hay uno y sólo un estado al cual el autómata puede cambiar desde su *estado actual*. La definición precisa se encuentra en la sección 3.4.

Para representar un AFD usaremos la notación en la que listamos los 5 elementos del autómata, la notación de quíntupla:

$$M = (Q, \Sigma, \delta, q_0, F)$$

donde M es el nombre del AFD, Q es el conjunto de estados, Σ es el alfabeto con los símbolos de entrada, δ es la función de transición, q_0 es el estado inicial, y F es el conjunto de los estados de aceptación.

Ejemplo 1: $M_1 = (Q_1, \Sigma_1, \delta, q_0, F_1)$ donde:

- $Q_1 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$
- $\Sigma_1 = \{A, C, G, T\}$
- La función de transición δ:
  a) $\delta(q_0, A) = q_1$
  b) $\delta(q_0, C) = q_6$
  c) $\delta(q_0, G) = q_7$
  d) $\delta(q_0, T) = q_2$
  e) $\delta(q_1, T) = q_3$
  f) $\delta(q_2, A) = q_4$
  g) $\delta(q_3, A) = q_5$
  h) $\delta(q_4, G) = q_6$
  i) $\delta(q_5, T) = q_8$
  j) $\delta(q_5, A) = q_7$
  k) $\delta(q_6, C) = q_7$
- $q_0 = q_0$
- $F_1 = \{q_6, q_7, q_8\}$

5.1.1. Procesamiento de cadenas

El lenguaje del AFD es el conjunto de todas las cadenas que acepta dicho AFD. Veremos cómo el AFD decide si *acepta o no acepta* una secuencia de símbolos de entrada. Supongamos que $a_1a_2\cdots a_n$ es una secuencia de símbolos de entrada. El AFD se encuentra en su estado inicial q_0. Consultamos la función de transición δ, $\delta(q_0, a) = q_1$ para saber el estado al que el AFD entra despueś de procesar el primer símbolo de entrada a_1. Procesamos el siguiente símbolo de entrada, a_2, evaluando $\delta(q_1, a_2)$; y supongamos que este nuevo estado es q_2. Continuamos de esta manera, encontrando los estados $q_1, q_2, \ldots, q_n$ tal que $\delta(q_{i-1}, a_i) = q_i$ para cada i. Si q_n es un elemento de F, entonces la entrada $a_1a_2\cdots a_n$ es *aceptada*, en cualquier otro caso es *rechazada*.

Por ejemplo, queremos saber si el AFD M_1 acepta la cadena $ATAT$:

1. M_1 está en q_0, el estado inicial, y lee el símbolo A, por la transición a) pasa al estado q_1.
2. Estando en q_1 lee T y por la transición e) pasa al estado q_3.
3. En q_3 lee A y por la transición g) pasa al estado q_5.
4. En q_5 lee T, el último símbolo de entrada, y por la transición i) termina en el estado q_8.
5. La secuencia de estados es: q_0, q_1, q_3, q_5, q_8.
6. Como $q_8 \in F$, M_1 acepta $ATAT$.

5.1.2. Notación

Definir el autómata por la quíntupla puede resultar tedioso en algunas ocasiones, sobre todo por la función de transición. Hay 2 notaciones opcionales que resultan más practicas para describir autómatas:

1. *Diagrama de transición*. Es una gráfica en la se representan las transiciones de δ por medio de nodos y arcos.
2. *Tabla de transición*. Es una tabla en la que se listan las transiciones de δ.

Diagramas de transición

Un *diagrama de transición* para un AFD $M = (Q, \Sigma, \delta, q_0, F)$ es una gráfica dirigida definida de la siguiente manera:

a) Para cada estado en Q hay un nodo.
b) Sean q en Q, a en Σ, y $\delta(q, a) = p$; entonces el digrama de transición tiene un arco (arista dirigida) del nodo q al nodo p, etiquetada con a. Si hay varios símbolos de

entrada que causen una transición de q a p, el diagrama de transición puede tener sólo un arco etiquetado por la lista de estos símbolos.

c) Hay un arco que entra al estado inicial q_0, este arco no se origina en ningún nodo.

d) Los nodos correspondientes a los estados de aceptación (en F) son marcados con doble círculo. Los estados que no están en F son marcados con un círculo simple.

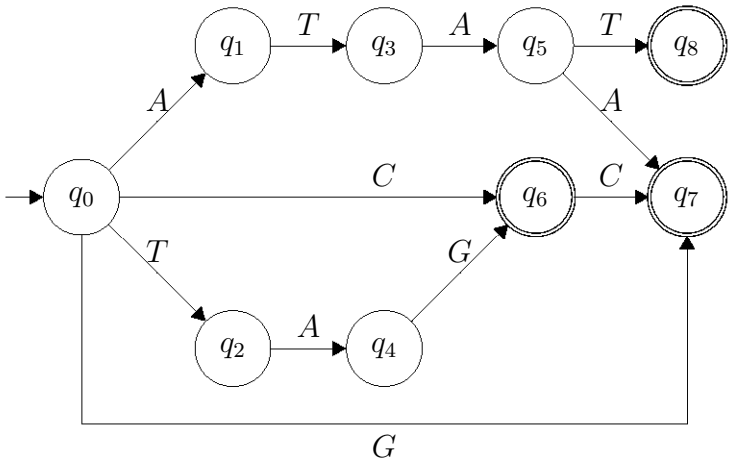


Figura 5.1: Diagrama de transición del AFD M_1

Tablas de transición

Una *tabla de transición*, por convención, es la función δ representada por una tabla. Los renglones de la tabla corresponden a los estados y las columnas corresponden a los símbolos de Σ. La entrada para el renglón correspondiente al estado q y la columna correspondiente al símbolo a es el estado que resulta de $\delta(q, a)$. La tabla de transición correspondiente a M_1 se encuentra en el Cuadro 5.1.

5.1.3. El lenguaje aceptado por un AFD

Como ya habíamos mencionado, un AFD define un lenguaje: el conjunto de todas las cadenas que resultan en una secuencia de transiciones de estados de su estado inicial a alguno de los estados de aceptación. En términos del diagrama de transición, el lenguaje de un AFD es el conjunto de etiquetas de todos los caminos que comienzan en el estado inicial y terminan el algún estado de aceptación.

	A	*C*	*G*	*T*
q_0	q_1	q_6	q_7	q_2
q_1				q_3
q_2	q_4			
q_3	q_5			
q_4			q_6	
q_5	q_7			q_8
q_6		q_7		
q_7				
q_8				

Cuadro 5.1: Tabla de transiciones del AFD M_1

Extendiendo la función de transición

Vamos a definir formalmente el lenguaje que acepta un AFD. Para ello, necesitamos definir una *función de transición extendida* que describa lo que pasa cuando empezamos en el estado inicial seguido de cualquier secuencia de entradas. Si δ es la función de transición, denotaremos con $\hat{\delta}$ la función de transición extendida. La función de transición extendida es una función que toma un estado q y una cadena w y regresa un estado p - el estado que el autómata alcanza empezando desde el estado q y procesando la secuencia de símbolos w. Definimos $\hat{\delta}$ recursivamente, como sigue:

$\hat{\delta}(q, \varepsilon) = q$ Esto es, cuando estamos en el estado q sin leer entradas, permanecemos en el estado q.

$\hat{\delta}(q, xa)$ Supongamos que w es una cadena de la forma xa; es decir que, a es el último símbolo de w, y x es la cadena que consiste de todos los símbolos menos el último. Por ejemplo, si $w = ATGAT$, $x = ATGA$ y $a = T$. Entonces

$$\hat{\delta}(q, w) = \hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$$

El lenguaje de un AFD

El lenguaje de un AFD $M = (Q, \Sigma, \delta, q_0, F)$, denotado $L(M)$, se define como

$$L(M) = \{w | \hat{\delta}(q_0, w) \in F\}$$

El lenguaje de M es el conjunto de cadenas w que del estado inicial q_0 llegan a algún estado de aceptación. Si L es $L(M)$ para algún autómata finito M, entonces L es un

lenguaje regular.

Del ejemplo 1, el lenguaje aceptado por M_1 es un lenguaje finito. Analizando el diagrama de transiciones vemos que la gráfica no tiene ciclos (dirigidos), por lo que M_1 es un AFD *acíclico*. El lenguaje aceptado por un AFD acíclico es siempre finito. Y todos los lenguajes finitos son regulares.

$$L(M_1) = \{ATAT, ATAAA, CC, TAGC, G, C, TAG\}$$

5.2. Autómata finito no determinístico

Un autómata finito *no determinístico* (AFN) tiene la ventaja de poder estar en varios estados a la vez. Esta capacidad se interpreta comúnmente como la facultad que tienen estas máquinas de *adivinar* algo acerca de su entrada. Más adelante explicaremos la utilidad de tal capacidad en los algoritmos de búsqueda.

Su definición precisa se encuentra en la seción 3.4. Un AFN se representa igual que un AFD:

$$M = (Q, \Sigma, \delta, q_0, F)$$

donde el significado de cada elemento está también en dicha sección.

Aunque la diferencia fundamental entre un AFN y un AFD está en la definición de la función de transición, los tipos de notación para los AFD también aplicarán para los AFN. Para los diagramas de transición, sólo habría que considerar que, si $a \in \Sigma$ y $q \in Q$, por cada estado en $\delta(a, q)$ se agrega un arco a la gráfica. En la tabla de transiciones cada entrada, en vez de ser un único estado (AFD), ahora es un conjunto de estados (AFN). Y cuando no esté definida la transición para un estado y un símbolo dados, la entrada apropiada es $\varnothing$, el conjunto vacío.

Ejemplo 2: $M_2 = (Q_2, \Sigma_2, \delta, q_0, F_2)$ donde:

- $Q_2 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$
- $\Sigma_2 = \{A, C, G, T\}$
- La función de transición δ:
  a) $\delta(q_0, A) = \{q_1, q_7\}$
  b) $\delta(q_0, T) = \{q_2, q_6\}$
  c) $\delta(q_1, T) = \{q_3\}$
  d) $\delta(q_2, A) = \{q_4\}$

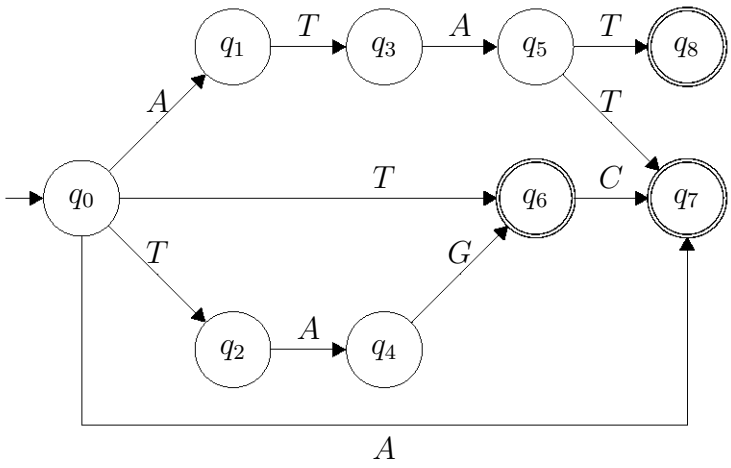


Figura 5.2: Diagrama de transición del AFN M_2

	A	*C*	*G*	*T*
q_0	$\{q_1, q_7\}$	$\varnothing$	$\varnothing$	$\{q_2, q_6\}$
q_1	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_2	$\{q_4\}$	$\varnothing$	$\varnothing$	$\varnothing$
q_3	$\{q_5\}$	$\varnothing$	$\varnothing$	$\varnothing$
q_4	$\varnothing$	$\varnothing$	$\{q_6\}$	$\varnothing$
q_5	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_7, q_8\}$
q_6	$\varnothing$	$\{q_7\}$	$\varnothing$	$\varnothing$
q_7	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
q_8	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Cuadro 5.2: Tabla de transiciones del AFN M_2

e) $\delta(q_3, A) = \{q_5\}$
f) $\delta(q_4, G) = \{q_6\}$
g) $\delta(q_5, T) = \{q_7, q_8\}$
h) $\delta(q_6, C) = \{q_7\}$

- $q_0 = q_0$
- $F_2 = \{q_6, q_7, q_8\}$

El diagrama de transiciones de M_2 está dado por la Figura 5.2, y la tabla de transiciones por el Cuadro 5.2.

5.2.1. El lenguaje aceptado por un AFN

Así como fue necesario extender la función de transición de un AFD, ahora necesitamos extender la función de transición δ, de un AFN, a una función $\hat{\delta}$ que tome un estado q y una cadena de símbolos w, y regrese el conjunto de estados en los que el AFN termina después de leer w desde el estado q.

La función de transición extendida

Dada una función de transición δ de un AFN, definimos formalmente $\hat{\delta}$ como:

$\hat{\delta}(q, \varepsilon) = \{q\}$ Mientras no se lean símbolos de la cadena de entrada, permanecemos en el mismo estado.

$\hat{\delta}(q, xa)$ Sea w tal que $w = xa$, donde a es el símbolo final de w y x es el resto de w. Supongamos que $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$ y que

$$\hat{\delta}(q, xa) = \bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$$

Entonces $\hat{\delta}(q, w) = \{r_1, r_2, \ldots, r_m\}$. Considerando el diagrama de transiciones, para calcular $\hat{\delta}(q, w)$ primero tenemos que calcular $\hat{\delta}(q, x)$ y seguir todos los arcos que van de cualquiera de estos estados y están etiquetados con el símbolo final a.

El lenguaje de un AFN

Un AFN acepta una cadena w si el conjunto de estados en el que queda después de leer w contiene al menos un estado de F. Es decir, existe al menos una secuencia de estados que va del estado inicial a un estado de aceptación.

Formalmente, si $M = (Q, \Sigma, \delta, q_0, F)$ es un AFN, entonces

$$L(M) = \{w | \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$$

Lo que quiere decir que $L(M)$ es el conjunto de cadenas $w \in \Sigma^*$, tales que $\hat{\delta}(q_0, w)$ contiene al menos un estado de aceptación.

M_2 es un autómata acíclico (Figura 3) y $L(M_2)$, así como $L(M_1)$ en el ejemplo 1, es finito:

$$L(M_2) = \{ATAT, T, TAG, A, TC, TAGC\}$$

5.3. Autómata finito con transiciones vacías

Ahora mostraremos otra extensión del autómata finito. La nueva *característica* consiste en permitir transiciones con ε, la cadena vacía. Esto significa que un AFN podrá hacer una transición espontánea sin necesidad de recibir un símbolo de entrada. Así como el no determinismo, esta nueva capacidad no expande la clase de lenguajes que acepta un autómata finito, aunque si da ciertas ventajas de *diseño*. Veremos también la relación que existe entre las expresiones regulares y los AFN con transiciones ε, a los cuales llamaremos ε-AFN. Esta relación nos servirá para demostrar la equivalencia entre la clase de lenguajes aceptada por un autómata finito y la aceptada por las expresiones regulares.

Podemos representar un ε-AFN igual que como lo hicimos con un AFN, con una excepción: la función de transición debe considerar el caso de las transiciones con ε. Formalmente, representamos un ε-AFN M por $M = (Q, \Sigma, \delta, q_0, F)$, donde todos los componentes tienen la misma interpretación que con un AFN, excepto que δ es ahora una función que toma como argumentos:

1. Un estado en Q, y
2. Un elemento de $\Sigma \cup \{\varepsilon\}$, es decir que puede ser un símbolo del alfabeto o bien la cadena vacía. Para evitar confusiones, ε, el símbolo que denota a la cadena vacía, no puede pertenecer al alfabeto Σ.

Ejemplo 3: $M_3 = (Q_3, \Sigma_3, \delta, q_0, F_3)$ es un ε-AFN, donde:

- $Q_3 = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$
- $\Sigma_3 = \{A, C, G, T\}$
- La función de transición δ:
  a) $\delta(q_0, A) = \{q_1\}$
  b) $\delta(q_0, G) = \{q_2\}$
  c) $\delta(q_1, T) = \{q_3\}$
  d) $\delta(q_2, C) = \{q_4\}$
  e) $\delta(q_3, A) = \{q_5\}$
  f) $\delta(q_3, \varepsilon) = \{q_0\}$
  g) $\delta(q_4, \varepsilon) = \{q_0\}$
  h) $\delta(q_5, T) = \{q_6\}$
  i) $\delta(q_5, \varepsilon) = \{q_1\}$
- $q_0 = q_0$
- $F_3 = \{q_4, q_6\}$

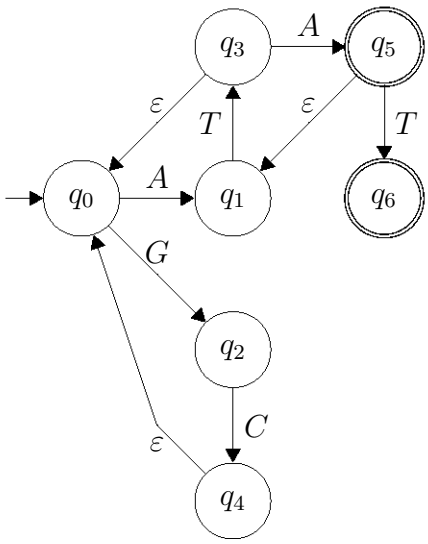


Figura 5.3: Diagrama de transiciones de M_3

	A	C	G	T	ε
q_0	$\{q_1\}$	$\varnothing$	$\{q_2\}$	$\varnothing$	$\varnothing$
q_1	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$
q_2	$\varnothing$	$\{q_4\}$	$\varnothing$	$\varnothing$	$\varnothing$
q_3	$\{q_5\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_0\}$
q_4	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_0\}$
q_5	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_6\}$	$\{q_1\}$
q_6	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Cuadro 5.3: Tabla de transiciones de M_3

5.3.1. El lenguaje aceptado por un ε-AFN

Antes de dar las definiciones formales de la función de transición extendida y de el lenguaje aceptado por un ε-AFN, necesitamos analizar otra función central en el estudio de los ε-AFN, llamada la ε-cerradura de un estado.

ε-cerradura

La idea de esta función es, dado un estado $q \in Q$, obtener todos los estados que se alcanzan siguiendo los arcos que salen de q y que están etiquetados con ε. Sin embargo, cuando obtenemos otros estados siguiendo a ε, tenemos que seguir las ε-transiciones que salen de estos estados, y así sucesivamente; eventualmente encontraremos todos los estados que pueden alcanzarse desde q siguiendo un camino cuyos arcos están etiquetados con ε.
Formalmente, definimos recursivamente la ε-cerradura ECERRADURA(q), como sigue:

- El estado q está en ECERRADURA(q).
- Si un estado p está en ECERRADURA(q), y hay una transición del estado p al estado r etiquetada con ε, entonces r está en ECERRADURA(q). Más precisamente, si δ es la función de trasición del ε-AFN en cuestión, y p está en ECERRADURA(q), entonces la ECERRADURA(q) también contiene todos los estados en $\delta(p, \varepsilon)$.

Por ejemplo, para el ε-AFN M_3:
ECERRADURA(q_0)= $\{q_0\}$, ECERRADURA(q_3)= $\{q_3, q_0\}$, ECERRADURA(q_4)= $\{q_4, q_0\}$, etc.

La función de transición extendida

Supongamos que $E = (Q, \Sigma, \delta, q_0, F)$ es un ε-AFN. Definiremos la función de transición $\hat{\delta}$ para ver qué pasa cuando el autómata lee una secuencia de símbolos de entrada. La idea es que $\hat{\delta}(q, w)$ es el conjunto de estados que se pueden alcanzar siguiendo un camino cuyas etiquetas, al ser concatenadas, forman la cadena w.
La definición formal de $\hat{\delta}$ es:

- $\hat{\delta}(q, \varepsilon)$ =ECERRADURA(q). Es decir que, si la etiqueta del camino es ε, entonces podemos seguir sólo los arcos etiquetados con ε que se extienden desde el estado q.
- Supongamos que w es de la forma xa, donde a es el último símbolo de w. Nótese que $a \in \Sigma$ y por lo tanto no puede ser ε. $\hat{\delta}(q, w)$ se calcula como sigue:
  1. Sea $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$. Esto quiere decir que, el conjunto de las p_is son los estados que podemos alcanzar desde q al seguir caminos etiquetados con x. Este camino podría contener arcos etiquetados con ε.
  2. Sea $\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$. Esto es, seguir todas las transiciones con etiqueta a desde los estados que podemos alcanzar desde q al leer la cadena x. Los r_js son algunos de los estados que podemos alcanzar desde q siguiendo caminos etiquetados con w. Tenemos que considerar también los estados que se alcanzan desde los r_js por medio de ε-transiciones.

3. Entonces $\hat{\delta}(q, w) = \bigcup_{j=1}^{m}$ECERRADURA($r_j$). Este paso adicional incluye todos los caminos desde q etiquetados con w, considerando la posibilidad de que haya arcos adicionales etiquetados con ε que pueden ser alcanzados después de hacer las transiciones con el último símbolo de w.

El lenguaje de un ε-AFN

El lenguaje de un ε-AFN $E = (Q, \Sigma, \delta, q_0, F)$ es:

$$L(E) = \{w|\hat{\delta}(q_0, w) \cap F \neq \varnothing\}$$

El lenguaje de E es el conjunto de todas las cadenas w que van del estado inicial hasta al menos un estado de aceptación.

Analizando la Figura 5.3 correspondiente al diagrama de transiciones de M_3, observamos que tiene 3 ciclos, es decir que M_3 es un ε-AFN *cíclico*. Estos ciclos están relacionados con la cerradura de las expresiones regulares. Gracias a esta propiedad podemos identificar rápidamente que $L(M_3)$ es un conjunto infinito.

$$L(M_3) = \{AT\,AT, AT\,AT\,AT, AT\,AT\,AT\,AT, \ldots, GC, GCGC, GCGCGC, \ldots\}$$

Capítulo 6

Autómata de Glushkov

Está bien establecido el hecho de que una expresión regular puede ser transformada en un AFN con o sin transiciones vacías. Y entre los primeros autores, que dieron una variante de la construcción sin transiciones vacías, se encuentra V. M. Glushkov [10].

En este capítulo estudiaremos una clase particular de AFN sin transiciones vacías: *el autómata de Glushkov*. El autómata de Glushkov es producto de la investigación sobre la síntesis de autómatas, o mejor dicho, de la conversión de expresiones regulares en autómatas.

La construcción de Glushkov fue popularizada por Berry y Sethi [9], al demostrar que dicha construcción es una representación natural de la expresión regular, debido a que se basa en las derivadas de Brzozowski [5] de la expresión.

En la construcción de el autómata de Glushkov nos auxiliaremos de las *expresiones marcadas*. Una expresión marcada se obtiene al asignar uno o más índices a los símbolos del alfabeto que aparecen en la expresión regular. Un mismo símbolo puede tener más de una posición debido a que puede aparecer más de una vez en la expresión regular. Los índices de posiciones materializan la idea del orden de los símbolos en los elementos de $L(E)$, ya que dado un símbolo del alfabeto, se trata de obtener el conjunto de símbolos que pueden sucederle en alguna palabra del lenguaje. A partir de estas expresiones marcadas construiremos el autómata de Glushkov; y finalmente, al borrar las posiciones de los símbolos en las transiciones, obtendremos el autómata de Glushkov de la expresión original.

La idea general del algoritmo consiste en ver una expresión regular como un autómata M_E cuyos estados corresponden a las posiciones o presencias de símbolos en E y cuyas

transiciones conectan posiciones que pueden ser consecutivas en los elementos de $L(E)$.

El algoritmo consta de tres etapas, la primera obtener el árbol de parseo de la expresión regular, calcular a partir del árbol las variables de Glushkov de la expresión y por último construir el autómata a partir de las variables. Recordemos que en la sección 4.4 explicamos cómo obtener el árbol de parseo de una expresión regular.
En esta sección mostraremos que el autómata de Glushkov puede contruirse en tiempo cúbico sobre el tamaño de la expresión, lo cual es óptimo en nuestro caso ya que el tamaño de la expresión es relativamente mucho menor que el texto.

6.1. Expresiones marcadas

Sea Σ un alfabeto; E, F y G denotan expresiones regulares sobre Σ. Para distinguir posiciones diferentes o presencias del mismo símbolo en una expresión, marcaremos los símbolos de Σ con *subíndices* (enteros positivos) de izquierda a derecha. A la *expresión marcada* de una expresión regular E la denotaremos con $\overline{E}$, y su lenguaje, donde cada símbolo incluye su índice, se denotará $L(\overline{E})$.
Por ejemplo:

$E =$`(AT|GA)((AG|AAA)*)`	$L(E) =$ {`AT, GA, ATAG, GAAG, ATAAA,...`}
$\overline{E} = (A_1T_2\|G_3A_4)((A_5G_6\|A_7A_8A_9)^*)$	$L(\overline{E}) = \{A_1T_2, G_3A_4, A_1T_2A_5G_6, G_3A_4A_5G_6, A_1T_2A_7A_8A_9, \dots\}$

El alfabeto con los símbolos marcados según las posiciones de $\overline{E}$ se denotará $\overline{\Sigma}$. En el ejemplo anterior $\Sigma = \{A, G, T\}$ y $\overline{\Sigma} = \{A_1, T_2, G_3, A_4, A_5, G_6, A_7, A_8, A_9\}$.

Llamaremos *posiciones* a los índices; y usaremos las letras x, y y z para representarlos. El conjunto de posiciones para una expresión E se denotará $Pos(E)$. Del ejemplo anterior, con $E =$`(AT|GA)((AG|AAA)*)`, el conjunto de posiciones es $Pos(E) = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
Si F es una subexpresión de E, denotaremos $Pos_E(F)$ al subconjunto de posiciones de E que son posiciones de F. Para subexpresiones se cumplen las siguientes propiedades:

1. Si $E = F + G$ o $E = FG$ entonces $Pos_E(F) \cap Pos_E(G) = \varnothing$
2. $Pos(E^*) = Pos(E)$

Por ejemplo, $E =$`(AT|GA)((AG|AAA)*)`$= FG$ con $F =$`AT|GA` y $G =$`(AG|AAA)*`:

- $Pos_E(F) = \{1, 2, 3, 4\}$
- $Pos_E(G) = \{5, 6, 7, 8, 9\}$

- Observemos que $Pos_E(F) \cap Pos_E(G) = \varnothing$

χ es la función que asocia a cada posición el símbolo de Σ que le corresponde en E.

$$\chi : Pos(E) \rightarrow \Sigma$$

Si $\overline{E} = (A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)^*)$, $\chi(4) =$A, $\chi(6) =$G, $\chi(1) =$A y $\chi(2) =$T.

6.2. Algoritmo

El autómata de Glushkov se construye primero sobre la expresión marcada $\overline{E}$ y reconoce $L(\overline{E})$. Entonces, borramos los índices de posiciones de todos los símbolos y así derivaremos el autómata de Glushkov que reconoce $L(E)$.
El conjunto de posiciones se toma como una referencia convirtiéndose en el conjunto de estados del autómata resultante, además de un estado inicial 0. Así que, si $Pos(E) = \{1, \ldots, m\}$, construiremos $m + 1$ estados etiquetados de 0 a m. Cada estado j representa el hecho de haber leído una cadena, a cuyo último símbolo de entrada le corresponde el índice j. Si leemos un nuevo símbolo α, necesitamos conocer las posiciones que podemos alcanzar desde j por α. Para cada posición j el algoritmo de Glushkov calcula todos los estados accesibles desde j.
Para ello, definiremos tres funciones antes de explicar el algoritmo completo.
Denotamos α_y al símbolo indexado de $\overline{E}$ que está en la posición y.

Definición $First(\overline{E}) = \{x \in Pos(E) \mid \exists u \in \overline{\Sigma}^*,\ \alpha_x u \in L(\overline{E})\}$.

El conjunto $First(\overline{E})$ representa al conjunto de posiciones iniciales de $L(\overline{E})$, es decir, el conjunto de posiciones desde las cuales puede iniciar la lectura. En nuestro ejemplo, $First((A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)^*)) = \{1, 3\}$.

Definición $Last(\overline{E}) = \{x \in Pos(E) \mid \exists u \in \overline{\Sigma}^*,\ u\alpha_x \in L(\overline{E})\}$.

El conjunto $Last(\overline{E})$ representa el conjunto de posiciones finales de $L(\overline{E})$, que es el conjunto de posiciones en las que una cadena leída puede ser aceptada. En nuestro ejemplo, $Last((A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)^*)) = \{2, 4, 6, 9\}$.

Definición $Follow(\overline{E}, x) = \{y \in Pos(E) \mid \exists u, v \in \overline{\Sigma}^*,\ u\alpha_x\alpha_y v \in L(\overline{E})\}$.

El conjunto $Follow(\overline{E})$ representa todas las posiciones en $Pos(E)$ accesibles desde x. Volviendo a nuestro ejemplo, si consideramos la posición 6, el conjunto de posiciones accesibles es $Follow((A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)^*), 6) = \{7, 5\}$.

Para darnos una idea más clara de estos conjuntos, las funciones pueden también definirse inductivamente sobre la expresión, es decir, considerando todos las casos posibles. Para esto definimos otras funciones auxiliares, la primera, $Empty_E$ que indique si la cadena vacía ε está en $L(E)$; y la segunda $\mathcal{I}_X$ que indique si una posición pertenece o no al conjunto X.

Definición Definimos recursivamente la función $Empty_E$, cuyo valor es $\{\varepsilon\}$ si ε pertenece a $L(E)$ y $\varnothing$ en otro caso.

$$\begin{array}{lcl} Empty_{\varnothing} & = & \varnothing \\ Empty_{\varepsilon} & = & \{\varepsilon\} \\ Empty_{a\in\Sigma} & = & \varnothing \\ Empty_{F+G} & = & Empty_F \cup Empty_G \\ Empty_{FG} & = & Empty_F \cap Empty_G \\ Empty_{F^*} & = & \{\varepsilon\} \end{array}$$

Definición Para cada conjunto X, definimos $\mathcal{I}_X$ la función de X a $\{\{\varepsilon\}, \varnothing\}$ como:

$$\mathcal{I}_X(x) = \begin{cases} \varnothing & si\ x \notin X \\ \varepsilon & si\ x \in X \end{cases}$$

Es muy importante analizar con un poco más de detalle las funciones $First(\overline{E})$, $Last(\overline{E})$ y $Follow(\overline{E}, x)$, pues de ellas se deducen los componentes de el autómata de Glushkov. El autómata de Glushkov está basado fundamentalmente en la idea de las posiciones; el conjunto $First(\overline{E})$ servirá para definir las transiciones que salen del estado inicial 0, $Last(\overline{E})$ va a determinar el conjunto de estados finales y $Follow(\overline{E}, x)$ las transiciones internas del autómata.

Proposición $First(\overline{E})$ se calcula recursivamente por casos como sigue:

$$\begin{array}{lcl} First(\varnothing) & = & \varnothing \\ First(\varepsilon) & = & \varnothing \\ First(\alpha_x) & = & \{x\} \\ First(\overline{F+G}) & = & First(\overline{F}) \cup First(\overline{G}) \\ First(\overline{FG}) & = & First(\overline{F}) \cap Empty_F First(\overline{G}) \\ First(\overline{F^*}) & = & First(\overline{F}) \end{array}$$

Demostración:

`caso 1 y 2:` $\overline{E} = \varnothing$ y $\overline{E} = \varepsilon$
$\Rightarrow Pos(E) = \varnothing$
$\Rightarrow First(\overline{E}) = \varnothing$

`caso 3:` $\overline{E} = \alpha_x$
$Pos(E) = \{x\}$, $L(\overline{E}) = \{\alpha_x\}$
Con $u = \varepsilon$, tenemos que $\alpha_x u \in L(\overline{E})$
$\Rightarrow First(\overline{E}) = \{x\}$

`caso 4:` $\overline{E} = \overline{F + G}$
Como $L(\overline{F + G}) = L(\overline{F}) \cup L(\overline{G})$
$\Rightarrow First(\overline{F + G}) = First(\overline{F}) \cup First(\overline{G})$

`caso 5:` $\overline{E} = \overline{FG}$
Claramente $First(\overline{F}) \subseteq First(\overline{FG})$
sólo falta considerar el caso en el que $First(\overline{G}) \subseteq First(\overline{FG})$
y esto pasa sí y solo si $\varepsilon \in L(F)$
$\Leftrightarrow Empty_F = \{\varepsilon\}$
$\Rightarrow First(\overline{FG}) = First(\overline{F}) \cup Empty_F \cdot First(\overline{G})$

`caso 6:` $\overline{E} = \overline{F^*}$
Por la definición de cerradura este caso es equivalente a la unión
$First(\overline{F^*}) = First(\overline{F})$

Proposición $Last(\overline{E})$ se calcula recursivamente por casos como sigue:

$$
\begin{array}{lcl}
Last(\varnothing) & = & \varnothing \\
Last(\varepsilon) & = & \varnothing \\
Last(\alpha_x) & = & \{x\} \\
Last(\overline{F + G}) & = & Last(\overline{F}) \cup Last(\overline{G}) \\
Last(\overline{FG}) & = & Last(\overline{G}) \cap Empty_G Last(\overline{F}) \\
Last(\overline{F^*}) & = & Last(\overline{F})
\end{array}
$$

Demostración: análoga a la demostración de $First(\overline{E})$.

Proposición $Follow(\overline{E}, x)$ se calcula recursivamente por casos como sigue:

$$
\begin{array}{lcl}
Follow(\varnothing, x) & = & \varnothing \\
Follow(\varepsilon, x) & = & \varnothing \\
Follow(a_y, x) & = & \varnothing \\
Follow(\overline{F + G}, x) & = & \mathcal{I}_{Pos_{F+G}(F)}(x) \cdot Follow(\overline{F}, x) \cup \mathcal{I}_{Pos_{F+G}(G)}(x) \cdot Follow(\overline{G}, x) \\
Follow(\overline{FG}, x) & = & \mathcal{I}_{Pos_{FG}(F)}(x) \cdot Follow(\overline{F}, x) \cup \\
 & & \mathcal{I}_{Pos_{FG}(G)}(x) \cdot Follow(\overline{G}, x) \cup \\
 & & \mathcal{I}_{Last(\overline{F})}(x) \cdot First(\overline{G})
\end{array}
$$

$Follow(\overline{F^*}, x) \quad = \quad Follow(\overline{F}, x) \cup \mathcal{I}_{Last(\overline{F})}(x) \cdot First(\overline{F})$

Demostración:

`caso 1, 2 y 3:` $\overline{E} = \varnothing$, $\overline{E} = \varepsilon$ y $\overline{E} = \alpha_y$
$|Pos(E)| \leq 1$ no hay posiciones que puedan ser consecutivas a x.
$\Rightarrow Follow(\overline{E}, x) = \varnothing$

`caso 4:` $\overline{E} = \overline{F + G}$
Como $Pos_{F+G}(F) \cap Pos_{F+G}(G) = \varnothing$
$\Rightarrow x \in Pos_{F+G}(F)$ ó $x \in Pos_{F+G}(G)$
Si $x \in Pos(F)$ entonces $Follow(\overline{F}, x) \subseteq Follow(\overline{E})$
Si $x \in Pos(G)$ entonces $Follow(\overline{G}, x) \subseteq Follow(\overline{E})$
Por lo tanto, $Follow(\overline{E}) = \mathcal{I}_{Pos_{F+G}(F)}(x) \cdot Follow(\overline{F}, x) \cup$
$\mathcal{I}_{Pos_{F+G}(G)}(x) \cdot Follow(\overline{G}, x)$

`caso 5:` $\overline{E} = \overline{FG}$
Tenemos tres subcasos.
Los dos primeros, como en $\overline{E} = \overline{F + G}$, son:
$x \in Pos_{FG}(F)$ ó $x \in Pos_{FG}(G)$
$\Rightarrow \mathcal{I}_{Pos_{FG}(F)}(x) \cdot Follow(\overline{F}, x) \subseteq Follow(\overline{FG}, x)$
y $\mathcal{I}_{Pos_{FG}(G)}(x) \cdot Follow(\overline{G}, x) \subseteq Follow(\overline{FG}, x)$
El último caso ocurre cuando $x \in Last(\overline{F})$
$\Rightarrow First(\overline{G}) \subseteq Follow(\overline{FG}, x)$
Por lo tanto, $Follow(\overline{E}, x) = \mathcal{I}_{Pos_{FG}(F)}(x) \cdot Follow(\overline{F}, x) \cup$
$\mathcal{I}_{Pos_{FG}(G)}(x) \cdot Follow(\overline{G}, x) \cup \mathcal{I}_{Last(\overline{F})}(x) \cdot First(\overline{G})$

`caso 6:` $\overline{E} = \overline{F^*}$
Si $x \in Last(\overline{F})$ entonces
$First(\overline{F}) \subseteq Follow(\overline{E}, x)$.
En otro caso bastará incluir al conjunto $Follow(\overline{F}, x)$
Por lo tanto,
$Follow(\overline{E}, x) = Follow(\overline{F}, x) \cup \mathcal{I}_{Last(F)}(x) \cdot First(\overline{F})$

La definición del autómata de Glushkov $\overline{M}$ que reconoce el lenguaje $L(\overline{E})$ se deduce a partir de estas funciones y se construye de la siguiente manera.

$$\overline{M} = (Q, \overline{\Sigma}, q_0, F, \overline{\delta})$$

donde:

(I) $Q = \{q_0\} \cup Pos(E) = \{0, 1, \ldots, m\}$, donde $Pos(E) = \{1, \ldots, m\}$ y el estado inicial $q_0 = 0$.

(II) $F = Last(\overline{E}) \cup Empty_E \cdot \{0\}$. Informalmente, un estado (posición) i es final si está en $Last(\overline{E})$. El estado inicial 0 es final si la cadena vacía pertenece a $L(\overline{E})$, en

cuyo caso $Empty_E = \{\varepsilon\}$ y por lo tanto $Empty_E \cdot \{0\} = \{0\}$. Si no es el caso, entonces $Empty_E \cdot \{0\} = \varnothing$.

(III) $\overline{\delta}$, la función de transición de el autómata, está definida por

$$\overline{\delta}(x, \alpha_y) = \{y\},\ \forall x \in Pos(E),\ \forall y \in Follow(\overline{E}, x)$$

Informalmente, hay una transición del estado x a y por α_y si y sucede a x. Las transiciones del estado inicial están definidas por

$$\overline{\delta}(0, \alpha_y) = \{y\},\ \forall y \in First(\overline{E})$$

El autómata

$$M_E = (Q, \Sigma, q_0, F, \delta)$$

se obtiene a partir de $\overline{M}$ de la siguiente manera: la etiqueta α_x de un arco de $\overline{M}$ se reemplaza por $\chi(x)$ en M_E.

$$\delta = \{(q, a, q') \mid q \in Q,\ q' \in Q,\ a \in \Sigma,\ \exists\ (q, \alpha_x, q') \in \overline{\delta}\ \text{y}\ \chi(x) = a\}$$

Simplemente borramos los índices de posiciones en el autómata marcado. En este paso, el autómata usualmente se convierte en no determinístico. El nuevo autómata reconoce el lenguaje $L(E)$.

El algoritmo de Glushkov está basado en la representación de árbol T_E (*árbol de parseo*) de la expresión regular. Cada nodo v de este árbol represente una subexpresión E_v de E. Asociaremos las siguientes variables a v:

- $first(v)$: lista de posiciones que representa el conjunto $First(\overline{E_v})$.
- $last(v)$: lista de posiciones que representa el conjunto $Last(\overline{E_v})$.
- $empty_v$: conjunto $\{\varepsilon\}$ si $L(E_v)$ contiene a la cadena vacía ε, y $\varnothing$ en otro caso.

Estas variables se calculan en orden *postfijo* para cada nodo, es decir que, primero se calculan para los hijos de v y al último para v. Denotamos a los dos hijos de v como v_l y v_r si v es "$+$" o "$\cdot$", y en el caso de la cerradura, if v representa "$*$", denotamos al único hijo de v como v_*.
El conjunto $follow(x)$ es una variable global. Para cada nodo v actualizamos $follow(x)$ de acuerdo a las posiciones en la subexpresión $\overline{E_v}$.
El algoritmo recursivo **Glushkov_variables**$(v_E, lpos)$ está dado en la Figura 5.

Este algoritmo calcula los valores de $first(V)$, $last(v)$, $follow(x)$ y $empty_v$ para cada nodo v del árbol de parseo de la expresión regular E. Los nodos se visitan en orden postfijo. Los valores del nodo v_E se calculan a partir de los valores obtenidos de sus hijos. La posición de cada carácter se calcula al vuelo (línea 9).
El algoritmo completo de Glushkov consta de tres etapas:

Glushkov_variables$(v_E, lpos)$
/* Orden postfijo, calculamos recursivamente el primer hijo */
If $v = \boxed{+}\,(v_l, v_r)$ OR $v = \boxed{\cdot}\,(v_l, v_r)$ **Then**
  $lpos \leftarrow$ **Glushkov_variables**$(v_l, lpos)$
  $lpos \leftarrow$ **Glushkov_variables**$(v_r, lpos)$
Else If $v = \boxed{*}\,(v_*)$ **Then** $lpos \leftarrow$ **Glushkov_variables**$(v_*, lpos)$
End of If
  /* Fin de la parte recursiva, calculamos los valores para el nodo actual */
If $v = (\varepsilon)$ **Then**
  $first(v) \leftarrow \varnothing,\ last(v) \leftarrow \varnothing,\ empty_v \leftarrow \{\varepsilon\}$
Else If $v = (\alpha),\ \alpha \in \Sigma$ **Then**
  $first(v) \leftarrow \{lpos\},\ last(v) \leftarrow \{lpos\},\ empty_v \leftarrow \varnothing,\ follow(lpos) \leftarrow \varnothing$
Else If $v = \boxed{+}\,(v_l, v_r)$ **Then**
  $first(v) \leftarrow first(v_l) \cup first(v_r)$,
  $last(v) \leftarrow last(v_l) \cup last(v_r)$,
  $empty_v \leftarrow empty_{v_l} \cup empty_{v_r}$
Else If $v = \boxed{\cdot}\,(v_l, v_r)$ **Then**
  $first(v) \leftarrow first(v_l) \cup (empty_{v_l} \cdot first(v_r))$
  $last(v) \leftarrow (empty_{v_r} \cdot last(v_l)) \cup last(v_r)$
  $empty_v \leftarrow empty_{v_l} \cap empty_{v_r}$
  For $x \in Last(v_l)$ **Do** $follow(x) \leftarrow follow(x) \cup follow(v_r)$
Else If $v = \boxed{*}\,(v_*)$ **Then**
  $first(v) \leftarrow first(v_*),\ last(v) \leftarrow last(v_*),\ empty_v \leftarrow \{\varepsilon\}$
  For $x \in Last(v_*)$ **Do** $follow(x) \leftarrow follow(x) \cup follow(v_*)$
End of if
Return $lpos$

Figura 6.1: Parte recursiva del algoritmo de Glushkov [11]. Esta función calcula los valores de $first(v)$, $last(v)$, $follow(x)$ y $empty_v$ para cada nodo v de la representación en árbol de la expresión regular $\overline{E}$.

- Transformar la expresión regular E en un árbol v_E.
- Calcular las variables de v_E con **Glushkov_variables** $(v_E, 0)$;
- Construir el autómata de Glushkov a partir de las variables de la raíz v_E del árbol, siguiendo su definición.

Glushkov(E)
/* Obtener el árbol de parseo de la expresión regular */
1. $v_E \leftarrow$ **Parse** $(E\$, 1)$
/* Construir la variables a partir del árbol */
2. $m \leftarrow$ **Glushkov_variables** $(v_E, 0)$
/* Construir el autómata */
3. $\Delta \leftarrow \varnothing$
4. **For** $i \in \{0, \cdots, m\}$ **Do** crear el estado i
5. **For** $x \in first(v_E)$ **Do** $\Delta \leftarrow \Delta \cup \{(0, \alpha_x, x)\}$
6. **For** $i \in \{0, \cdots, m\}$ **Do**
7. **For** $x \in follow(i)$ **Do** $\Delta \leftarrow \Delta \cup \{(i, \alpha_x, x)\}$
8. **End of for** 9. **For** $x \in last(v_E) \cup (empty_{v_E} \cdot \{0\})$ **Do** marcar x como estado final

---

Figura 6.2: El algoritmo completo de Glushkov [11]. La función de transición está denotada con Δ y el estado inicial es 0.

6.3. Propiedades del autómata de Glushkov

Dos propiedades son de mayor interés para nosotros.

Propiedad El autómata de Glushkov no tiene transiciones vacías.
Aunque el algoritmo de Glushkov no se sigue tan directamente de la expresión regular como otros algoritmos, tiene la ventaja de estar libre de transiciones vacías. Esto resulta mucho más atractivo en la práctica, debido a que hay casos en los que incluso resulta determinístico, lo que reduce el tiempo de búsqueda a orden lineal sobre el tamaño del texto. Cuando esto sucede se dice que la expresión regular que genera el autómata de Glushkov es *determinística*, así como el lenguaje que describe. Los lenguajes determinísticos tienen, entre otras, aplicaciones en el estándar ISO de SGML (Lenguaje Estándar Generalizado de Marcas), el cual provee metalenguajes sintácticos para la definición de sistemas de marcado textual.

Propiedad Todos los arcos que llegan a un estado y están etiquetados por el mismo símbolo.
Ésta propiedad se sigue de la definición de la función de transición

$$\forall x \in Pos(E),\ \forall y \in Follow(\overline{E}, x),\ \overline{\delta}(x, \alpha_y) = \{y\}$$

Cuando borramos los índices de los símbolos nos queda únicamente α, por lo que todos los arcos que lleguen a y tienen que estar etiquetados por α.

6.4. Complejidad

El peor caso del algoritmo completo está dominado por la función **Glushkov_variables**. En esta función, todas las uniones de conjuntos, excepto por la cerradura, son disjuntas y pueden calcularse en tiempo $O(1)$.
El ciclo del **For** de la línea 19 en su peor caso es $O(m)$. El peor caso de complejidad es debido a la línea 22, esto es, el cálculo de la cerradura (estrella). Como $follow(x)$ y $first(v_*)$ pueden intersectarse, el peor caso de la unión es de $O(m)$. Como esto es dentro de un ciclo de **For**, pueden desarrollarse hasta $O(m)$ iteraciones, la complejidad del ciclo completo es de $O(m^2)$. La complejidad total de el algoritmo es entonces en el peor caso de $O(m^3)$, debido a que pueden existir $O(m)$ estrellas en la expresión.

Capítulo 7

Autómatas de Thompson

Thompson [16] [8] desarrolló esta construcción en 1968; su objetivo era compilar una expresión regular de tal forma que se obtenga una representación útil en búsquedas en texto. El autómata es una traducción directa del árbol de parseo. Usa trancisiones vacías para facilitar la traducción.
La idea general consiste en recorrer el árbol de parseo T_E de la expresión regular E y producir por cada nodo v un autómata $Th(v)$ que reconoce el lenguaje E_v representado por el subárbol cuya raíz es v. Una construcción específica del autómata se asocia a cada tipo de nodo y hoja en el árbol.

7.1. Construcción

La construcción se define recursivamente y produce un AFND con trancisiones vacías. Se usa frecuentemente para demostrar la equivalencia entre AFND y expresiones regulares.

(a) $E = \varnothing$. Un estado inicial y uno final, no hay trancisiones definidas.

(b) $E = \varepsilon$. El autómata consiste de sólo dos estados unidos por una transición vacía.

(c) $E = \mathbf{a},\ a \in \Sigma$. Para un símbolo del alfabeto Σ la construcción es similar a la anterior, excepto que la transición está etiquetada con el símbolo en vez de la expresión regular vacía.

(d) $E = (F + G)$. La construcción para la unión requiere transiciones vacías. La idea es representar el hecho de que podemos entrar a cualquiera de los dos autómatas $Th(v_l)$ o $Th(v_r)$ que corresponden a los dos hijos. Agregamos dos nuevos estados, uno inicial I con dos transiciones vacías a los dos estados iniciales de $Th(v_l)$ y $Th(v_r)$, y un estado final F que puede alcanzarse desde los dos estados finales de $Th(v_l)$ y $Th(v_r)$. Un camino de I a F tiene que pasar por uno de los dos autómatas, por lo

tanto el lenguaje que reconoce es $E_{v_l} + E_{v_r}$

(e) $E = (FG)$. Los dos autómatas de Thompson de los dos hijos v_l y v_r se mezclan a traves del estado inicial del primer autómata y el estado final del segundo.

(f) $E = (F^*)$. La construcción para la cerradura usa la misma idea que la unión. Primero el lenguaje E_{v_*}, donde v_* es el único hijo de v, puede repetirse tantas veces como se desee. Se crea una transición vacía de retorno del estado final del autómata $Th(v_*)$ al inicial. Pero la cerradura también significa que el autómata puede ser ignorado, entonces aquí se crean dos nuevos estados, un inicial I y un final F, unidos por una transición vacía. Con otras dos transiciones vacías unimos I con el estado inicial de $Th(v_*)$, y el estado final de $Th(v_*)$ con F.

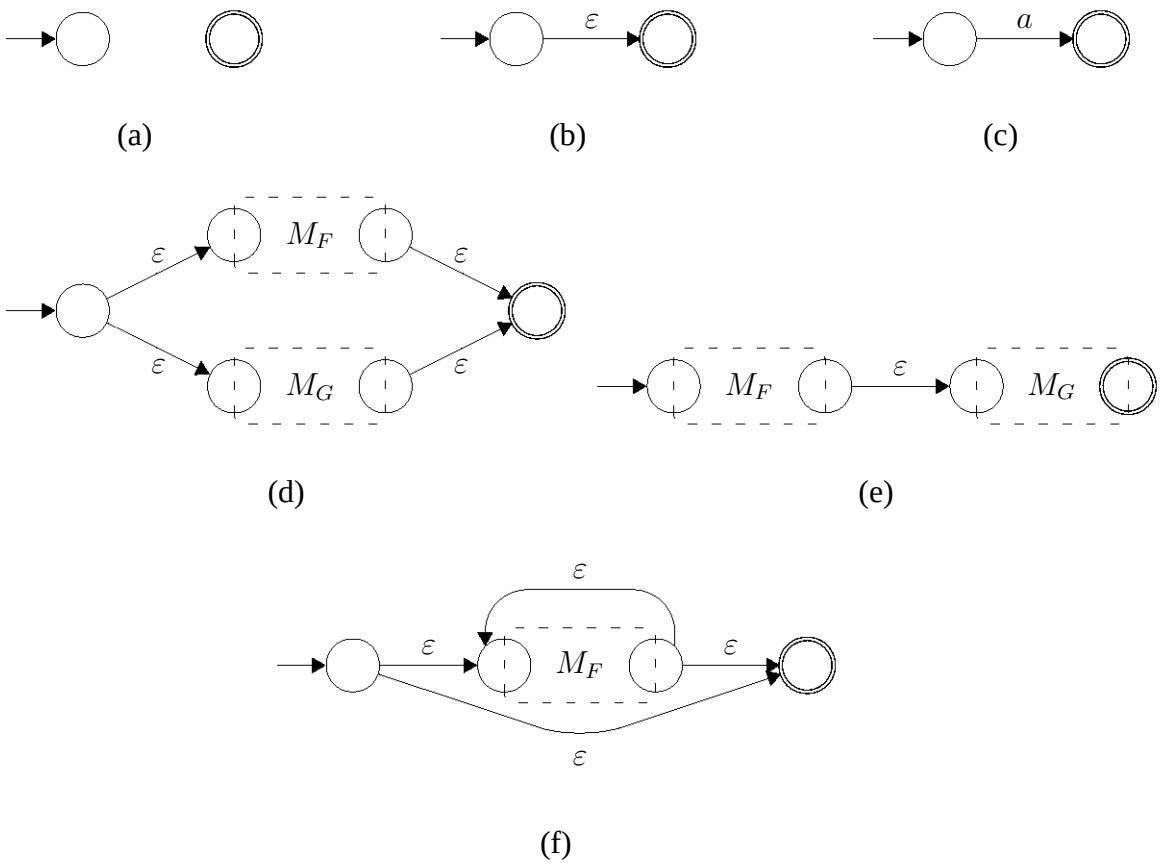


Figura 7.1: Construcción de Thompson ([16], [11], [8]).

7.1.1. Ejemplo

Construiremos el autómata de Thompson para la expresion regular $(AT|GA)((AG|AAA)*)$. En la figura comenzamos por los casos base y después unimos los autómatas según sea el caso: unión, concatenación o cerradura.

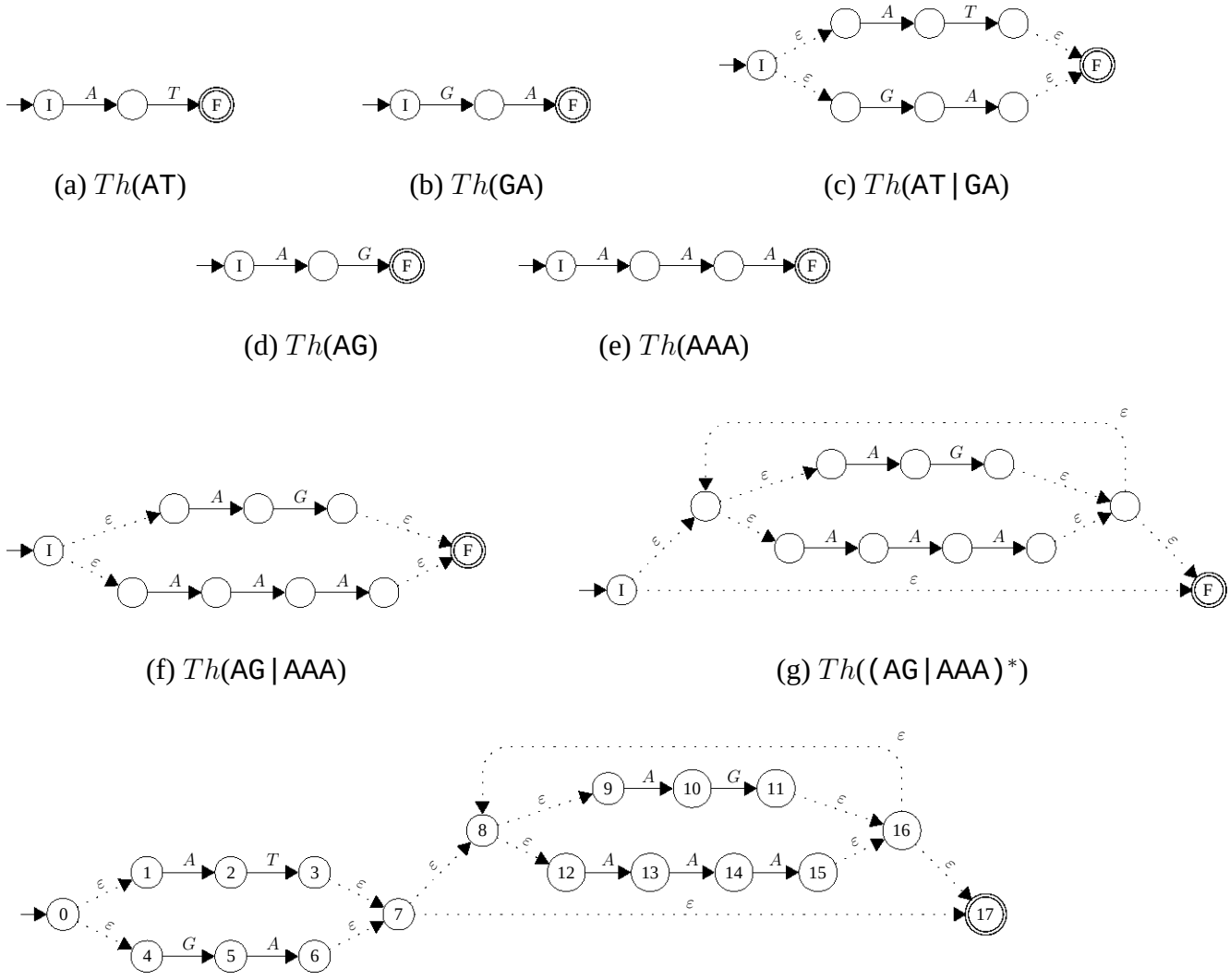


Figura 7.2: Autómata de Thompson correspondiente a $(AT|GA)((AG|AAA)^*)$.

Según la construcción dada en la sección 7, desarrollamos el ejemplo de la figura anterior como sigue:

(a) El autómata de Thompson correspondiente a la concatenación de A y T.
(b) La concatenación de G y A.
(c) La unión de los autómatas (a) y (b) reconoce la expresión regular $AT|GA$.
(d) La concatenación de A con G.
(e) El que reconoce la expresión regular AAA.
(f) La unión del autómata (d) y (e).
(g) La cerradura del autómata (f).
(h) El autómata final corresponde a la concatenación de (c) y (g).

7.2. Propiedades del autómata de Thompson

Algunas propiedades del autómata de Thompson que se derivan de la construcción son:

1. Hay un único estado inicial.
2. Hay un único estado final.
3. Cada estado tiene a lo más dos transiciones que entran y dos que salen.
4. El tamaño de la máquina es a lo más tres veces el tamaño de la expresión regular dada.

7.3. Complejidad

Como para cada nodo del árbol podemos crear la construcción en tiempo constante, la complejidad en tiempo del algoritmo es lineal.

Capítulo 8

Equivalencia entre autómatas de Glushkov y de Thompson

En esta sección analizaremos las relaciones que existen entre las construcciones de Thompson y Glushkov. Primero probaremos que podemos usar la construcción de Glushkov para construir las máquinas obtenidas por la construcción de Thompson. Preprocesaremos la expresión regular dada y postprocesaremos la máquina resultante.
Segundo, usaremos la construcción de Thompson para construir máquinas obtenidas por la construcción de Glushkov. Postprocesaremos las máquinas resultantes.

8.1. Construcción de Champarnaud-Glushkov

Champarnaud [12] [7] probó que la construcción de Glushkov puede hacerse por casos, con el mismo estilo que la construcción de Thompson. Los casos base se dan en la Figura 8.1.

La construcción de la máquina M_{F+G} combina las máquinas M_F y M_G identificando o mezclando sus estados iniciales. El paso inductivo de FG identifica el estado inicial de M_G separadamente con los estados finales de M_F.

El resultado de la identificación es que todos los estados, antes finales, de M_F tienen transiciones a todos los estados a los que el estado, antes inicial, de M_G tenía transiciones, de tal modo que el estado inicial de M_G desaparece. Y si el estado inicial de M_G era un estado final, entonces los estados finales de M_F seguirán siendo finales, en otro caso, serán no finales.
El paso inductivo para F^* es similar al de la concatenación. En este caso identificamos

por separado el estado inicial de M_F con cada uno de los estados finales de M_F. La diferencia es que el estado inicial no desaparece, se convierte en estado final (debido a la cadena vacía) y todos los otros estados finales de M_F permanecen como finales. El resultado de la identificación es que cada estado final en M_F tiene transiciones a todos aquellos estados a los que el estado inical de M_F tiene trancisiones.

8.2. La construcción de Glushkov de una máquina de Thompson

Como los autómatas de Glushkov no tienen transiciones vacías, la idea de la construcción es introducir un nuevo símbolo en la expresión regular, que simule a la cadena vacía ε. De acuerdo al método de Thompson, dicho símbolo se introducirá convenientemente en la expresión regular de tal modo que represente las transiciones vacías que se agregan al autómata de Thompson en cada caso.

Sea τ un nuevo símbolo que no pertenece a Σ. Definimos la τ**-expansión** de una expresión regular E sobre Σ como una expresión regular E_τ sobre $\Sigma \cup \{\tau\}$ que se construye inductivamente como sigue:
$E_\tau = \varnothing$, si $E = \varnothing$.
$E_\tau = \tau$, si $E = \varepsilon$.
$E_\tau = a$, si $E = a$.
$E_\tau = (((\tau \cdot F_\tau) + (\tau \cdot G_\tau)) \cdot \tau)$, si $E = F + G$.
$E_\tau = ((F_\tau \cdot \tau) \cdot G_\tau)$, si $E = FG$.
$E_\tau = (((\tau \cdot F_\tau)^*) \cdot \tau)$, si $E = F^*$.
$E_\tau = (F_\tau)$, si $E = (F)$.

A la expresión resultante le aplicamos la construcción de Glushkov (ver Figura 8.2), y si en la máquina M_τ que obtenemos remplazamos el símbolo τ por la cadena vacía ε obtenemos la máquina de Thompson para la expresión regular E.
Definimos la función proyección $h_\tau : (\Sigma \cup \{\tau\})^* \longrightarrow \Sigma^*$ como: $h_\tau(a) = a$, para toda $a \in \Sigma$, y $h_\tau(\tau) = \varepsilon$. Para un autómata finito $M = (Q, \Sigma \cup \{\tau\}, \delta, q_0, F)$, $h_\tau(M) = (Q, \Sigma, h_\tau(\delta), q_0, F)$, donde $h_\tau((p, a, q)) = (p, h_\tau(a), q)$, para todo $(p, a, q) \in \delta$.

8.3. Conversión de un autómata de Thompson a un autómata de Glushkov

El segundo resultado de las construcciones de Thompson y Glushkov es que todo autómata de Glushkov está oculto en su correspondiente autómata de Thompson. Para esta conversión usaremos una transformación que borra las transiciones vacías y preserva ciertas propiedades de los autómatas. Cuando la aplicamos consecutivamente a un autómata de Thompson para eliminar todas las transiciones vacías, obtenemos su correspondiente autómata de Glushkov.
Brüggemann-Klein y Wood[1] fueron los primeros en probar este resultado.

En la Figura 8.3 ilustramos la transformación específica que elimina las transiciones vacías.

Denotamos con $elim(M, q)$ el autómata finito que se obtiene al borrar todas las transiciones vacías que entran en el estado q de un autómata M. En la mayoría de los casos elimina todas las transiciones que salen y llegan de q. Definimos formalmente $elim(M, q)$ como sigue: Primero, el estado q debe satisfacer la **precondición para eliminación de vacíos**:

> *Todas las transiciones que entran al estado q son vacías y hay al menos una de dichas transiciones*.

Segundo, si q tiene un ciclo vacío, lo borramos. Tercero, supongamos que hay k transiciones vacías entrando en q, con $k \geq 1$, y m transiciones que salen de q, donde $m \geq 0$. Sean (p_i, ε, q), con $1 \leq i \leq k$, las transiciones que entran a q y (q, a_j, r_j), $0 \leq j \leq m$, las transiciones que salen de q. Entonces, definimos $elim(M, q)$ como el autómata finito $(Q', \Sigma, \delta', q_0, F')$, donde

$$Q' \quad = \quad \begin{cases} Q, & \text{si } q = q_0, \\ Q - \{q\}, & \text{en otro caso,} \end{cases}$$

[1] A. Brüggemann-Klein y D. Wood.The validation of SGML content models. Mathematical and Computer Modelling, 25:73-84, 1997.

$$
\begin{aligned}
\overline{\delta} &= \delta - \{(p_i, \varepsilon, q) : 1 \leq i \leq k\} \\
&\quad \cup \{(p_i, a_j, r_j) : 1 \leq i \leq k \; y \; 0 \leq j \leq m\}
\end{aligned}
$$

$$
\delta' = \begin{cases} \overline{\delta}, & \text{si } q = q_0, \\ \overline{\delta} - \{(q, a_j, r_j) : 0 \leq j \leq m\}, & \text{en otro caso} \end{cases}
$$

$$
\overline{F} = \text{F} \cup \{p_i : 1 \leq i \leq k\},
$$

$$
F' = \begin{cases} \overline{F}, & \text{si } q \text{ es final e inicial} \\ \overline{F} - \{q\}, & \text{si } q \text{ es final y no es inicial} \\ F, & \text{en otro caso.} \end{cases}
$$

Notemos que la transformación borra el estado q excepto cuando q es el estado inicial. Observemos que cuando no salen transiciones de q, la transformación borra el estado q y todas las transiciones vacías que entran a q y que vienen de algún estado p_i. En este caso, si q es un estado final, entonces los estados p_i se convierten en estados finales.

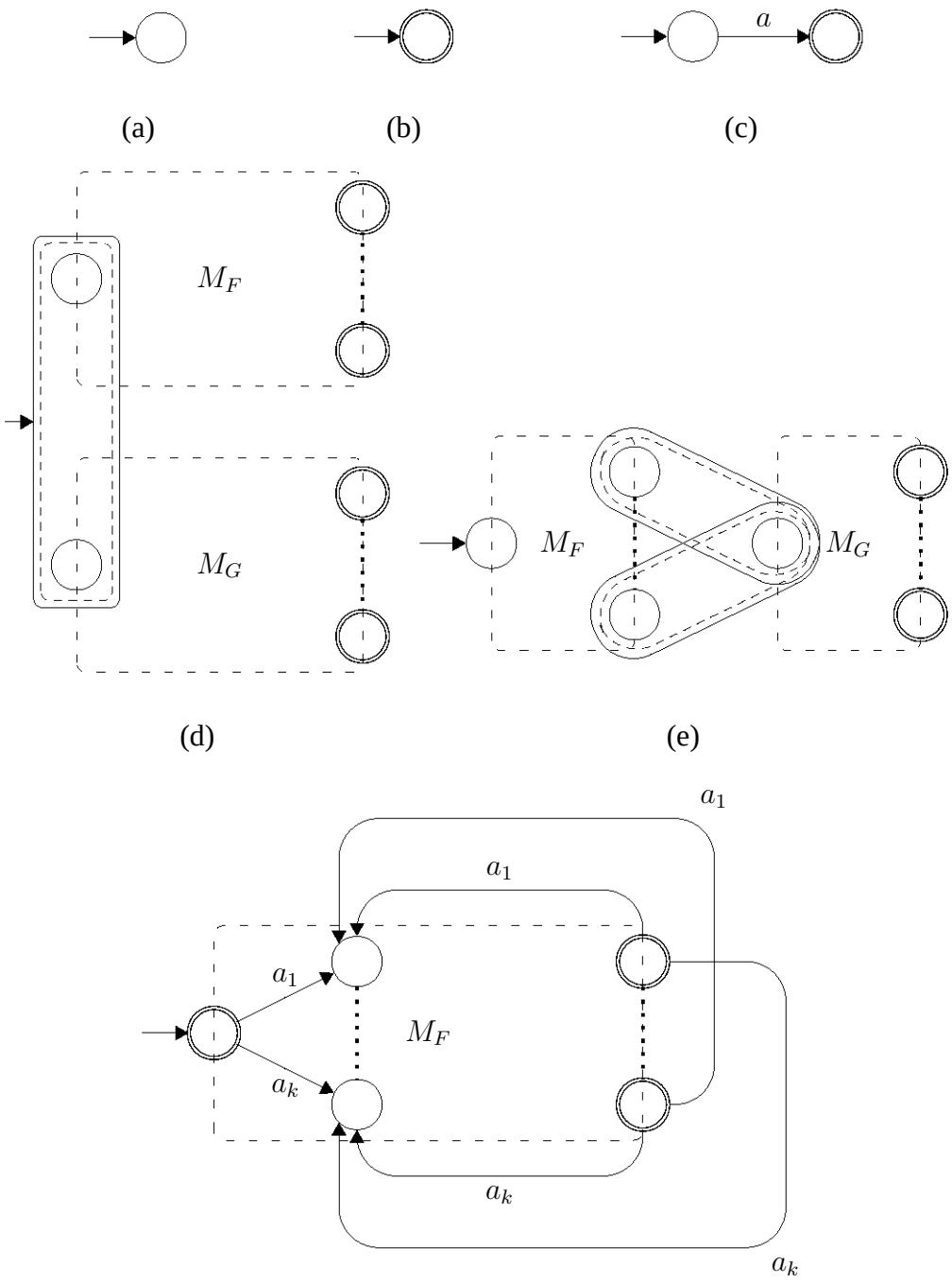


Figura 8.1: Construcción inductiva de Champarnaud de una máquina de Glushkov ([8]).

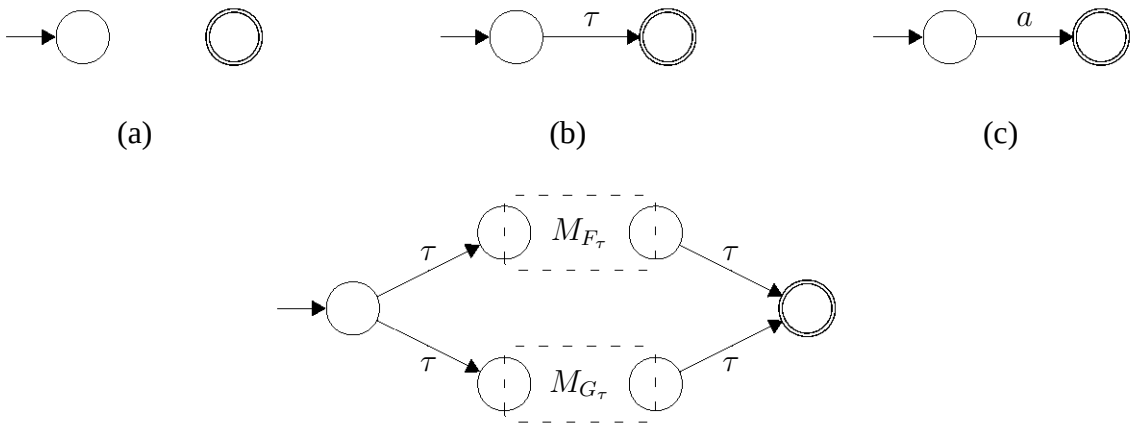


Figura 8.2: La construcción por casos de Champarnaud-Glushkov para expresiones regulares τ-expandidas. Los autómatas finitos corresponden a expresiones regulares: a. $E = \varnothing$. b. $E = \varepsilon$. c. $E = a$. d. $E = (F + G)$ ([8]).

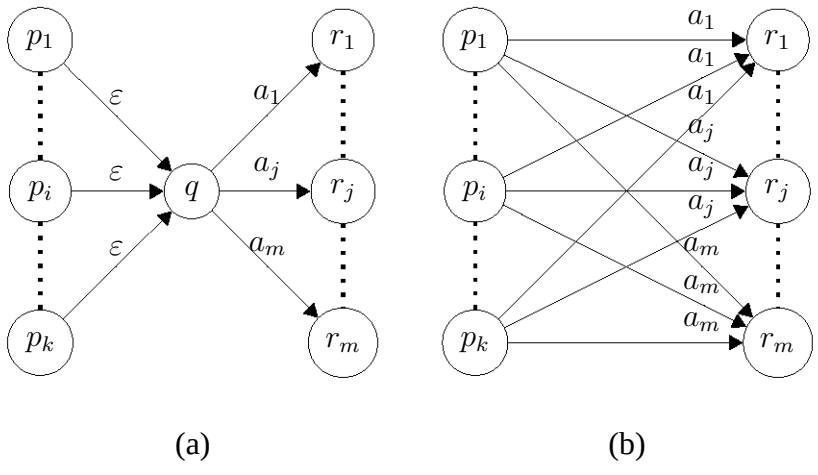


Figura 8.3: La transformación que elimina transiciones vacías se usa para transformar un autómata de Thompson en un autómata de Glushkov ([8]).

Capítulo 9

RESS

Una de las partes fundamentales de este trabajo es la aplicación que implementa el algoritmo desarrollado en las secciones anteriores. *RESS* (Regular Expression Search System) es un sistema de búsqueda con expresiones regulares que usa el autómata de Glushkov como motor de búsqueda.

Iniciamos el capítulo con el diseño e implementación de RESS y concluimos con un manual de usuario. Es importante asociar las clases con las estructuras y algoritmos desarrollados en este trabajo.

9.1. Diseño e implementación

9.1.1. Características de programación

Dentro de la características de programación están las siguientes:

a. *Paradigma de programación:* POO
   Nos permite modelar de forma independiente los componentes del sistema.

b. *Lenguaje de Programación:* Java
   Algunas de las ventajas que tenemos al usar Java son la portabilidad y el gran número de bibliotecas con las que cuenta.

c. *Motor gráfico para AFND:* Jaguar
   Jaguar (*Java Automaton and Grammar User Aplication Resources*) es una herramienta de software libre escrita en Java que brinda una infraestructura de bibliotecas y aplicaciones gráficas que soportan la mayoría de los modelos y algoritmos relacionados con autómatas y lenguajes formales.
   Para mayores informes de Jaguar ver `http://www.sf.net/projects/ijaguar`

o comunicarse con Iván Hernández Serrano a *ivanx@users.sourceforge.net*. En RESS se importan todas las bibliotecas con las estructuras necesarias para definir un AFND.

9.1.2. Componentes del sistema

Para explicar con mayor detalle el diseño de RESS, dividimos el sistema en tres partes principales:

I. **El parser o analizador:** A partir de la expresión regular obtiene el árbol de parseo.
II. **El algoritmo de Glushkov:** Recibe un árbol de parseo y contruye el autómata de Glushkov que reconoce el lenguaje de la expresión regular representada en el árbol.
III. **La interfaz gráfica:** Proporciona al usario un medio más amigable para hacer búsquedas en texto.

I. El parser

El algoritmo para obtener el árbol de parseo de una expresión regular está en el Cuadro 4.3 de la sección 4.4.

El parser consta de tres partes:

1. Las estructuras. Son las clases que representan los componentes de un árbol de parseo. Algunas clases son propias de RESS:
   - `Operation.java`: Operación válida para una expresión regular. Elementos de los nodos internos del árbol.
   - `ParseTree.java`: Estructura de árbol binario con características propias de un árbol de parseo, como por ejemplo identificar si un nodo es símbolo u operación.

   Y otras son importadas de Jaguar:
   - `Symbol`: Símbolo de un alfabeto.
   - `Alphabet`: Conjunto de símbolos.
2. El algoritmo para parsear la expresión regular. Consta de las siguientes clases:
   - `ResultParser.java`: Compuesta de un árbol de parseo y de un entero que indica la última posición leída de la expresión regular.
   - `RegularExpressionParser.java`: El algoritmo de parseo para la expresión regular.
3. Las estructuras gráficas para el árbol de parseo. RESS muestra una ventana con el árbol de parseo, los nodos representados por círculos con etiquetas y la relación de padres-hijos representada por líneas.
   - `JParseTree.java`: Un nodo se representa con un círculo. Un árbol se pinta recursivamente en el `JParseTreeCanvas` de la siguiente manera: Se pinta el nodo actual y manda pintar a sus hijos, si los hijos existen, pinta la

línea que lo une con ellos. La posición de los hijos depende de la del nodo padre.
- `JParseTreeCanvas.java`: Es el lienzo en el que se pinta el árbol.

II. El algoritmo de Glushkov
En la sección 6.2 analizamos el algoritmo de Glushkov. Después de construir el árbol de parseo de la expresión regular, la siguiente etapa consiste en calcular las variables *first, last, follow* y *empty* del árbol, y finalmente construir el autómata a partir de dichas variables.

- La clase `GlushkovAutomatonMatcher.java` es la encargada de implementar la función que calcula las variables y la que construye el autómata.

Las clases importadas de Jaguar que implementan un AFND son:
- `NDFA.java`
- `NDfaDelta.java`

Algunas de las clases que implementan la interfaz gráfica para un AFND son:
- `JNDFA.java`
- `JNDfaCanvas.java`

III. La interfaz gráfica
La interfaz gráfica principal consta de un área de texto no editable, una caja de texto para introducir la expresión regular y un botón que inicia el proceso de búsqueda.
El usuario puede abrir un archivo y el contenido se muestra en el área de texto. A partir de este archivo construimos el alfabeto, mismo que usamos para validar la expresión regular.
La ventana, con el árbol de parseo, el autómata de Glushkov y los resultados de la búsqueda, se abre al finalizar todo el proceso de construcción y ejecución del autómata.

Para marcar en el área de texto las presencias encontradas por el autómata de Glushkov usamos un *árbol de ejecución* en el que guardamos el estado del autómata asociado con el índice de inicio de la presencia. Las ramas de árbol crecen hasta que no se encuentran transiciones definidas o se llega a un estado final. Una vez que se termina de ejecutar el autómata se hace una búsqueda a profundidad en el árbol de ejecución para identificar todos los caminos que terminan en un estado final. El índice final de la presencia está dado por el índice inicial más el tamaño del camino.

9.2. Manual Usuario

9.2.1. Instalación

Para instalar RESS es necesaria la instalación de Java 2 (JRE) 1.5.0 o superior. Para descarga o información ir a `http://java.sun.com`.

9.2.2. Cómo usar RESS

Para realizar una búsqueda con RESS se siguen los siguientes pasos:

1. Ejecutar `ress-1.0.jar`. Puede ser simplemente dando doble clic sobre el archivo.
2. Abrir el archivo con el texto en el que se quiere realizar la búsqueda.
3. Teclear la expresión regular en la caja de texto correspondiente. Para ingresar la expresión regular ε dar clic en *Write* ε.
4. Dar clic en el botón *Glushkov Method*.
5. En caso de que la expresión regular sea válida se tienen los siguientes resultados:
   a) Una ventana principal con los resultados de la búsqueda. La ventana contiene a su vez tres ventanas internas (JInternalFrames), la primera con la información obtenida de la búsqueda, la segunda con el árbol de parseo y la tercera con la representación gráfica del autómata de Glushkov correspondiente a la expresión regular.
   b) En el área de texto se indican las presencias sombreandolas con color.
6. Eso es todo.

Capítulo 10

Colofón

10.1. Búsquedas en texto con expresiones regulares

Muchos de los algoritmos de apareamiento de cadenas usan autómatas finitos como motor de búsqueda. Los algoritmos para expresiones regulares no son la excepción.
Las expresiones regulares pueden verse como la generalización de los casos particulares como es el apareamiento de cadenas y de un conjunto de cadenas.

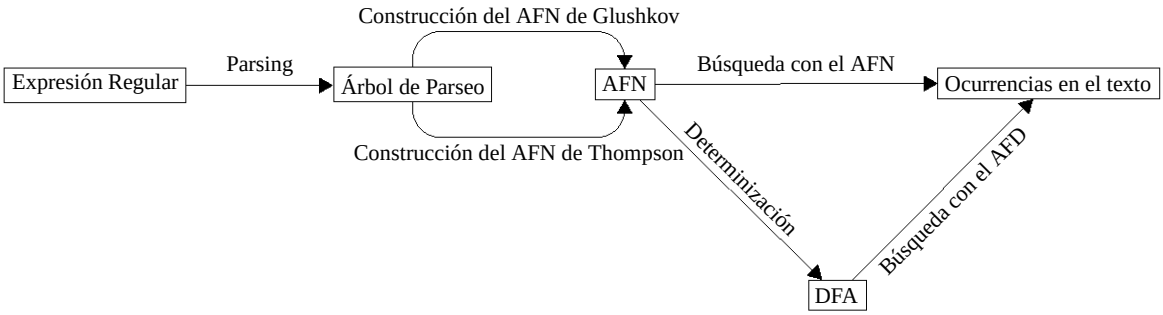


Figura 10.1: Método clásico para busquedas en texto con expresiones regulares.

En el método clasico de búsquedas en texto para expresiones regulares, lo primero es el parseo de la expresión regular en el *árbol* que la representa. Lo siguiente, es usar dicho aŕbol en la construcción del AFN equivalente.
Para la transformación del árbol en AFN hay varias maneras. Entre los métodos principales y que han sido de mayor interés en la investigación están:

- El algoritmo de Glushkov

- El algoritmo de Thompson

Apesar de que el primero es $O(m^3)$ y el segundo $O(m)$ sobre el tamaño de la expresión regular, en la práctica el autómata de Glushkov es preferido sobre el de Thompson por varias razones:

⋆ No tiene transiciones vacías.
⋆ En algunos casos resulta determinístico.
⋆ Es una simplificación del autómata de Thompson, que se obtiene al eliminar las transiciones vacías.

Sin embargo tambien comparten algunas propiedades. Para cualquier estado p, todos los arcos que entran a p tienen la misma etiqueta. Es decir son *Autómatas homogéneos*.
El número de estados está acotado por el tamaño de la expresión regular. El número de estados del autómata de Glushkov es el tamaño de la expresión regular más uno.
Mientras que para el autómata de Thompson depende del tipo de operaciones que contenga la expresión, pero a lo más, es tres veces el tamaño de la expresión.

10.2. Expresiones regulares y autómatas de Glushkov

Las expresiones regulares juegan un papel importante en aplicaciones prácticas. En especificaciones sintácticas de lenguajes de programación describen elementos (*tokens*) léxicos, y en sistemas de manipulación de texto describen conjuntos de patrones.

Los autómatas de Glushkov tienen aplicaciones recientes en recuperación de texto, en biología computacional y en sistemas de procesamiento de documentos: por ejemplo los que siguen el estandar SGML (*Standard Generalized Markup Languaje*) y las definiciones de documentos XML (DTD, *document-type-definition*).
En estos dos últimos casos la investigacón en síntesis de expresiones regulares ha sido de mayor interés debido a que el lado derecho de las producciones son expresiones regulares *no ambigüas*, y como resultado se obtienen autómatas finitos determinísticos.

Me hubiera gustado hablar mucho más de las optimizaciones del algoritmo original, pero por falta de tiempo sólo daremos un breve resumen.
La construcción del autómata de Glushkov fue formalizada por Glushkov [10] en 1961, posteriormente fue analizada y optimizada por Gerard Berry y Ravi Sethi [9] en 1986. Y les siguieron entre otros: C.H. Chang y Robert Paige [6] en 1992; Brügegeman-Klein [4] en 1993; y D. Ziadi, J.L. Ponty y J.M. Champarnaud [12] en 1995; y en 1996 Pascal Caron y D. Ziadi [7]. Las investigaciones más recientes son las relacionadas con biología computacional y con DTDs.

Las optimizacion de Pascal Caron y D. Ziadi [14] se basa en la *normalización* de la expresión regular que toma tiempo lineal en el tamaño de la expresión, y la construcción del autómata baja a $O(m^2)$. También Chang y Paige logran esta cota.

10.3. Otros algoritmos en búsquedas en texto con expresiones regulares

Además del método clásico hay otros algoritmos basados en autómatas que no alcanzamos a tratar en este trabajo. Explicaremos brevemente algunos de los más importantes.

10.3.1. Determinización del autómata

El AFN puede ser usado directamente para hacer la búsqueda, pero podría resultar muy lento ya que varios estados estarían activos al mismo tiempo. Entonces hay quienes prefieren convertirlo en un autómata finito determinístico, el cual sólo tiene un estado activo por unidad de tiempo. El problema es que el tamaño del AFD puede resultar exponencial sobre el tamaño del AFN, lo cual hace mucho más favorable su uso únicamente con expresiones regulares pequeñas.

10.3.2. Paralelismo de bits

Otra forma es simular el AFN usando *paralelismo de bits* en vez de convertirlo en AFD. El paralelismo de bits fue propuesto en búsquedas en texto por Gonzalo Navarro [11]. Esta técnica consiste en aprovechar el paralelismo intrínseco de las operaciones de bits según el tamaño de palabra en una computadora. Podemos empaquetar varios valores en una sola palabra, y actualizarlos todos con una sola operación. Usando este hecho, el número de operaciones a desarrollar en un algoritmo puede reducirse en un factor de a lo más w, donde w es el número de bits por palabra en la computadora. Como en las arquitecturas actuales w es de 32 o 64 bits, la rápidez es considerable en la práctica.

10.3.3. Problema abierto

Algunas veces requerimos de búquedas con patrones complejos y que además permitan errores.

El apareamiento aproximado con expresiones regulares ha sido tratado escasamente, sin embargo existen unos pocos algoritmos basados en programación dinámica.

Conclusiones

El estudio de las expresiones regulares y de los autómatas finitos es central en el estudio de las Ciencias de la Computación, dada su gran variedad de aplicaciones. En este trabajo únicamente rascamos la superficie de este tema, y aún así, presentamos resultados interesantes.

Por lo amplio del tema no es posible, en un trabajo de este nivel, agotarlo. A pesar de ello, esperamos haber despertado la curiosidad por seguir explorando a los autómatas finitos y las expresiones regulares. En mi caso, quedé con esta curiosidad y las decisión de seguir aprendiendo del tema.

Bibliografía

[1] A. Brüggemann-Klein y D. Wood. Deterministic Regular Languages. En A. Finkel y M. Jantzen, editor, *9th Annual Symposium on Theoretical Aspects of Computer Science. Proceedings*, páginas 173–184, Cachan, France, February 1992. Springer-Verlag Inc.

[2] M. J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[3] David Berlinski. *The Advent of the Algorithm*. Harcourt, Inc., New York, 2000.

[4] Anne Brüggemann-Klein. Regular Expressions into Finite Automata. En Imre Simon, editor, *Proceedings of Latin American Symposium on Theoretical Informatics (LATIN '92)*, páginas 87–98. Springer, 1992.

[5] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[6] Chia-Hsiang Chang y Robert Paige. From regular expressions to dfa's using compressed nfa's. En *CPM '92: Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, páginas 90–110, London, UK, 1992. Springer-Verlag.

[7] D. Ziadi, J.-L. Ponty y Jean-Marc Champarnaud. Passage d'une expression rationnelle à un automate fini non-déterministe. *Bulletin of the Belgian Mathematical Society Simon Stevin*, 4(2):117–203, 1997.

[8] Dora Giammarresi, Jean-luc Ponty y Derick Wood. A Reexamination of the Glushkov and Thompson Constructions, 1998.

[9] G. Berry y R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

[10] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

[11] G.Navarro y M.Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[12] J.-M. Champarnaud, J.-L. Ponty y D. Ziadi. From regular expressions to finite automata. *International Journal of Computer Mathematics*, 72:415–431, 1999.

[13] John E. Hopcroft,Rajeev Motwani, Rotwani y Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edición, 2000.

[14] Pascal Caron y Djelloul Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 233(1-2):75–90, 2000.

[15] Rafael C. Carrasco, Alejandro Bia, et al. Turning DTDs into specialized tree-automata-based schemata to match a collection of marked-up documents, 2001.

[16] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

[17] Elisa Viso. *Autómatas y Lenguajes Formales. Notas para el curso Teoría de la Computación*. Vinculos Matemáticos, Facultas de Ciencias, UNAM, 2000.