Capítulo 3

Análisis léxico de grandes diccionarios

El problema de la correcta etiquetación o del análisis sintáctico de una frase dada puede resultar complejo si se aborda tratando directamente el flujo de los caracteres de entrada que conforman esa frase. Para evitar dicha complejidad, normalmente existirá un paso de procesamiento previo, cuya misión es la de transformar el flujo de caracteres de entrada en un flujo de elementos de más alto nivel de significado[1], que típicamente serán las palabras de la frase en cuestión, y la de obtener rápida y cómodamente todas las etiquetas candidatas de esas palabras. Dicho paso previo se denomina análisis léxico[2].

Este capítulo está destinado a esbozar las distintas técnicas existentes para la realización de esta tarea. No obstante, comenzaremos presentando en detalle la visión particular que se ha utilizado a lo largo de este trabajo para modelizar los diccionarios, y la técnica concreta con la que se han implementado.

3.1 Modelización de un diccionario

Si bien es cierto que muchas de las palabras que aparecen en un diccionario se pueden capturar automáticamente a partir de los textos etiquetados, otras muchas serán introducidas manualmente por los expertos lingüistas, con el fin de cubrir de manera exhaustiva algunas categorías de palabras poco pobladas, que constituyen el núcleo invariable de un idioma (artículos, preposiciones, conjunciones, etc.), o alguna terminología particular correspondiente a un determinado ámbito de aplicación. Sin embargo, cuando nos enfrentamos a lenguajes naturales que presentan un paradigma de inflexión de gran complejidad, resulta impensable que el usuario tenga que introducir en el diccionario todas y cada una de las formas derivadas de un lema dado[3]. En lugar de esto, resulta mucho más conveniente realizar un estudio previo que identifique los diferentes grupos de inflexión (género, número, irregularidad verbal, etc.), de manera que a la hora de introducir posteriormente un nuevo término en el diccionario, el usuario sólo tenga que hacer mención de las raíces involucradas en él y de los grupos de inflexión mediante los cuales se realiza la derivación de formas desde cada una de esas raíces [Graña *et al.* 1994].

Por supuesto, una interfaz gráfica que genere temporalmente las formas correspondientes a la operación de inserción que se va a realizar, tal y como se muestra en la figura 3.1, puede resultar de gran ayuda para el usuario al permitirle comprobar si efectivamente está realizando

[1] Normalmente denominados *tokens*.

[2] O también *scanning*.

[3] Por ejemplo, el paradigma de conjugación verbal del español puede utilizar hasta 118 formas distintas para un mismo verbo.

la inserción de las raíces en el grupo correcto o no [Vilares *et al.* 1997]. Pero tal y como vimos en la sección 2.2.2, en un primer momento la representación inicial de un diccionario consiste en una base de datos que almacena la información léxica correspondiente sólo a las raíces y a sus grupos de inflexión correspondientes.

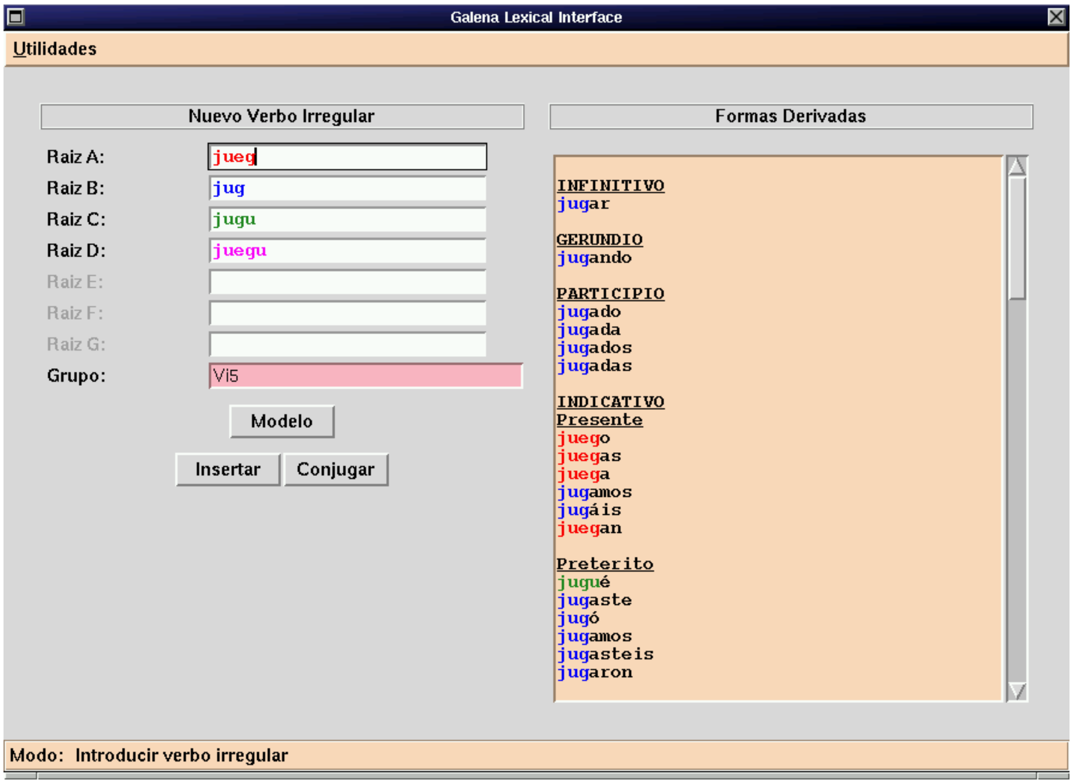

Figura 3.1: Interfaz gráfica para la introducción de términos en el diccionario

Sin embargo, en este punto surgen dos problemas:

1. El primero de ellos es un problema inherente al uso de bases de datos. Las bases de datos se presentan como herramientas válidas para el almacenamiento de gran cantidad de información estructurada, ya que resultan muy flexibles a la hora de realizar tareas de gestión y mantenimiento: las inserciones, actualizaciones, borrados y consultas se realizan mediante operaciones relacionales muy potentes que pueden implicar a uno o varios campos de información, de la misma o de diferentes tablas de datos. Pero cuando se procesan textos grandes, quizás de millones de palabras, no sólo interesa un acceso flexible a los datos, sino también un acceso muy rápido, y una base de datos no es el mejor mecanismo a la hora de recuperar este tipo de información, y menos aún si en ella residen las raíces y no las palabras concretas que aparecen en los textos. Existen mecanismos mucho más eficientes para esta última tarea, como pueden ser los autómatas finitos que describiremos en la siguiente sección.

2. El segundo de los problemas es que muchas aplicaciones necesitan incorporar información adicional relativa a las palabras, no a las raíces. Tal es el caso de determinados paradigmas de etiquetación estocástica o de análisis sintáctico estocástico, que necesitan asociar una probabilidad a cada una de las combinaciones posibles palabra-etiqueta. Por ejemplo, en el contexto de los modelos de Markov ocultos, que veremos más adelante, esa probabilidad representa la *probabilidad de emisión* de la palabra dentro del conjunto de

todas las palabras que tienen esa misma etiqueta. O bien, dentro del marco del análisis sintáctico estocástico, esa misma probabilidad puede verse como la probabilidad de la regla gramatical *etiqueta* $\rightarrow$ *palabra*. Así que una vez más, la información de las raíces no es suficiente. Necesitamos disponer de todas las palabras que se pueden generar a partir de esas raíces. El mismo procedimiento que aparece en la interfaz gráfica de introducción de raíces se puede también utilizar aquí para generar todas las palabras y, posteriormente, a través de otro procedimiento para la estimación de las probabilidades, que veremos con detalle en el próximo capítulo, podemos integrar toda la información necesaria, de manera que resida junta en un mismo recurso.

Por lo tanto, en este segundo momento, nuestra visión de un diccionario o lexicón es simplemente un fichero de texto, donde cada línea tiene el siguiente formato:

```
palabra etiqueta lema probabilidad
```

Las palabras ambiguas, es decir, con varias etiquetaciones posibles, utilizarán una línea diferente para cada una de esas etiquetas.

Ejemplo 3.1 Sin pérdida de generalidad, las palabras podrían estar ordenadas alfabéticamente, de tal manera que, en el caso del diccionario del sistema Galena [Vilares *et al.* 1995], el punto donde aparece la ambigüedad de la palabra `sobre` presenta el siguiente aspecto:

```
...
sobraste V2sei0 sobrar 0.00162206
sobrasteis V2pei0 sobrar 0.00377715
sobre P sobre 0.113229
sobre Scms sobre 0.00126295
sobre Vysps0 sobrar 0.0117647
sobrecarga Scfs sobrecarga 0.00383284
sobrecarga V2spm0 sobrecargar 0.00175131
sobrecarga V3spi0 sobrecargar 0.000629723
sobrecargaba Vysii0 sobrecargar 0.0026455
...
```

Retomando los datos de la sección 2.2.2, recordemos que el diccionario del sistema Galena tiene 291.604 palabras diferentes, con 354.007 etiquetaciones posibles. Esta última cifra es precisamente el número de líneas del fichero anterior. Para una discusión posterior, diremos ahora que la primera etiquetación de la palabra `sobre`, es decir, como preposición,

```
sobre P sobre 0.113229
```

aparece en la línea 325.611 dentro de ese fichero. Y diremos también que, en el conjunto de todas las 291.604 palabras diferentes ordenadas alfabéticamente, la palabra `sobre` ocupa la posición 268.249. □

Por supuesto, ésta tampoco es todavía la versión operativa final de un diccionario. El problema del acceso eficiente a los datos sigue aún presente, pero éste es un problema que, como ya hemos dicho, resolveremos en la siguiente sección. Lo realmente importante ahora es obtener una versión compilada que represente de una manera más compacta todo este gran volumen de información. Esta versión compilada y su forma de operar se muestran en la figura 3.2, donde se pueden identificar cada uno de los siguientes elementos:

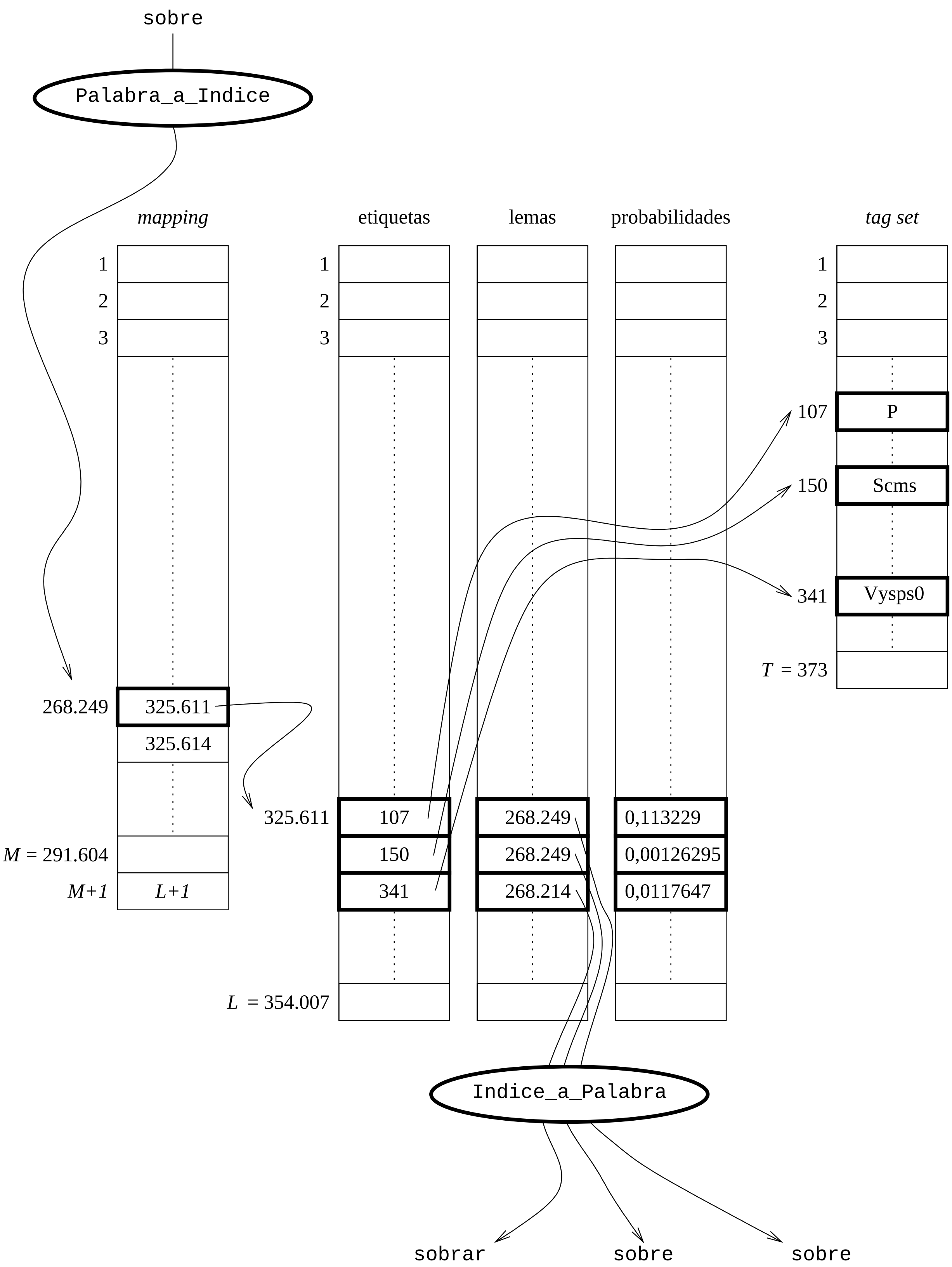

Figura 3.2: Modelización compacta de un diccionario

- La función `Palabra_a_Índice`, que explicaremos con detalle en la siguiente sección, es capaz de convertir una palabra en un número que representa la posición que ocupa esa palabra dentro del conjunto de todas las palabras diferentes ordenadas alfabéticamente. Por ejemplo, esta función transforma `sobre` en el número 268.249.

- Ese número sirve para indexar un tablero de correspondencia[4] de tamaño $M + 1$, que transforma la posición relativa de cada palabra en la posición absoluta dentro del lexicón original. En el caso del diccionario del sistema Galena, M es igual a 291.604, el número de palabras distintas. En el caso de `sobre`, la posición relativa 268.249 se transforma en la posición absoluta 325.611.

- Ese número sirve para indexar los tableros de *etiquetas*, *lemas* y *probabilidades*. Todos estos tableros son de tamaño L. En el caso del diccionario del sistema Galena, L es igual a 354.007, el número de etiquetaciones distintas.

- El tablero de *etiquetas* almacena números. Una representación numérica de las etiquetas es más compacta que los nombres de las etiquetas en sí. Las etiquetas originales se pueden recuperar indexando con esos números el tablero del juego de etiquetas[5]. El tablero del juego de etiquetas tiene un tamaño T. En el caso del diccionario del sistema Galena, T es igual a 373, el número de etiquetas distintas.

  Dado que hemos partido de una ordenación alfabética, está garantizado que las etiquetas de una misma palabra aparecen contiguas. No obstante, necesitamos saber de alguna manera dónde terminan las etiquetas de una palabra dada. Para ello, es suficiente con restarle el valor de la posición absoluta de la palabra al valor de la siguiente casilla en el tablero de correspondencia. En nuestro caso, la palabra `sobre` tiene $325.614 - 325.611 = 3$ etiquetas. Esta operación es también válida para acceder correctamente a la información de los tableros de *lemas* y *probabilidades*.

- El tablero de *lemas* almacena también números. Un lema es una palabra que debe estar también presente en el diccionario. El número que la función `Palabra_a_Índice` obtendría para esa palabra es el número que se almacena aquí, siendo esta representación mucho más compacta que el lema en sí. El lema se puede recuperar aplicando la función `Índice_a_Palabra`, que explicaremos con detalle en la siguiente sección.

- El tablero de *probabilidades* almacena directamente las probabilidades. En este caso no es posible realizar ninguna compactación.

Ésta es por tanto la representación más compacta que se puede diseñar para albergar toda la información léxica relativa a las palabras presentes en un diccionario. Es además una representación muy flexible en el sentido de que resulta particularmente sencillo incorporar nuevos tableros, si es que se necesita algún otro tipo de información adicional. Por ejemplo, algunas aplicaciones de lexicografía computacional que realicen estudios sobre el uso de un idioma podrían utilizar un tablero de números enteros que almacene la frecuencia de aparición de las palabras en un determinado texto. De igual manera, aquellos tableros que no se utilicen se pueden eliminar, con el fin de ahorrar su espacio correspondiente[6].

Por otra parte, ideando un nuevo método de acceso o una nueva disposición de los elementos de los tableros, es también sencillo transformar esta estructura en un generador de formas, es

[4] O tablero de *mapping*.

[5] O tablero de *tag set*.

[6] Por ejemplo, no todas las aplicaciones hacen uso del lema y de la probabilidad, pudiendo ser suficiente sólo con las etiquetas.

decir, en un mecanismo bidireccional que no sólo reconozca palabras, sino que también, dado un lema y una etiqueta, genere la forma flexionada correspondiente. Esta funcionalidad es indispensable en aplicaciones de generación de lenguaje, es decir, en aquellas aplicaciones donde el flujo de información que va desde el sistema hacia el usuario se realiza utilizando lenguaje natural [Vilares *et al.* 1996a].

Otro aspecto más de la flexibilidad de esta representación es el que se describe a continuación. Cuando al procesar un texto aparece una palabra que no está presente en nuestro diccionario, el tratamiento normal será asumir que es desconocida y posteriormente intentar asignarle la etiqueta que describe su papel en la frase, igual que si se tratara de una palabra normal. La diferencia radica en que para la palabra desconocida no podemos obtener un conjunto inicial de etiqueta candidatas, de manera que habrá que definir una serie de categorías *abiertas* que permitan inicializar dicho conjunto. Sin embargo, hay aplicaciones en las que puede ser más conveniente suponer que todas las categorías están *cerradas* y que por tanto esa palabra es realmente una palabra del diccionario, pero que presenta algún error lexicográfico que hay que corregir. Ocurre entonces que el hecho de tener la información de etiquetación totalmente separada del mecanismo de reconocimiento de las palabras en sí facilita la posterior incorporación de algoritmos de corrección automática de este tipo de errores lexicográficos [Vilares *et al.* 1996b].

Finalmente, para completar la modelización de diccionarios que hemos presentado, sólo nos resta detallar la implementación de las funciones `Palabra_a_Índice` e `Índice_a_Palabra`. Ambas funciones trabajan sobre un tipo especial de autómatas finitos, los autómatas finitos acíclicos deterministas numerados, que se describen a continuación.

3.2 Autómatas finitos acíclicos deterministas numerados

La manera más eficiente de implementar analizadores léxicos[7] es quizás mediante el uso de autómatas finitos [Hopcroft y Ullman 1979]. La aplicación más tradicional de esta idea la podemos encontrar en algunas de las fases de construcción de compiladores para los lenguajes de programación [Aho *et al.* 1985]. El caso del procesamiento de los lenguajes naturales es cuantitativamente diferente, ya que surge la necesidad de representar diccionarios léxicos que muchas veces pueden llegar a involucrar a cientos de miles de palabras. Sin embargo, el uso de los autómatas finitos para las tareas de análisis y reconocimiento de las palabras sigue siendo una técnica perfectamente válida.

Definición 3.1 Un *autómata finito* es una estructura algebraica que se define formalmente como una 5-tupla $A = (Q, \Sigma, \delta, q_0, F)$, donde:

- Q es un conjunto finito de estados,
- Σ es un alfabeto finito de símbolos de entrada, es decir, el alfabeto de los caracteres que conforman las palabras,
- δ es una función del tipo $Q \times \Sigma \rightarrow P(Q)$ que define las transiciones del autómata,
- q_0 es el estado inicial del autómata, y
- F es el subconjunto de Q al que pertenecen los estados que son finales.

El estado o conjunto de estados que se alcanza mediante la transición etiquetada con el símbolo a desde el estado q se denota como $q.a = \delta(q, a)$. Cuando este estado es único, es decir, cuando la función δ es del tipo $Q \times \Sigma \rightarrow Q$, se dice que el autómata finito es *determinista*. □

[7] También denominados *scanners*.

La anterior notación es transitiva, es decir, si w es una palabra de n letras, entonces $q.w$ denota el estado alcanzado desde q utilizando las transiciones etiquetadas con cada una de las letras w_1, w_2, ..., w_n de w. Una palabra w es aceptada por el autómata si $q_0.w$ es un estado final.

Definición 3.2 Se denota como $L(A)$ el *lenguaje reconocido por el autómata A*, es decir, el conjunto de todas las palabras w tales que $q_0.w \in F$. □

Definición 3.3 Un autómata finito es *acíclico* cuando su grafo subyacente es acíclico. Los autómatas finitos acíclicos reconocen lenguajes formados por conjuntos finitos de palabras. □

3.2.1 Construcción del autómata

La primera estructura que nos viene a la mente para implementar un reconocedor de un conjunto finito de palabras dado es un *árbol de letras*.

Ejemplo 3.2 El árbol de letras de la figura 3.3 reconoce todas las formas de los verbos ingleses `discount`, `dismount`, `recount` y `remount`, es decir, el conjunto finito de palabras[8]:

```
discount    discounted    discounting    discounts
dismount    dismounted    dismounting    dismounts
recount     recounted     recounting     recounts
remount     remounted     remounting     remounts
```

Esta estructura es en sí misma un autómata finito acíclico determinista, donde el estado inicial es el 0 y los estados finales aparecen marcados con un círculo más grueso. □

Como en todo autómata finito, la complejidad de reconocimiento de un árbol de letras es lineal respecto a la longitud de la palabra a analizar, y no depende para nada ni del tamaño del diccionario, ni del tamaño de dicho autómata.

Sin embargo, los requerimientos de memoria de esta estructura de árbol pueden llegar a ser elevadísimos cuando el diccionario es muy grande. Por ejemplo, el dicionario del sistema Galena necesitaría un árbol de más de un millón de nodos para reconocer las 291.604 palabras diferentes. Por tanto, en lugar de utilizar directamente esta estructura, le aplicaremos un proceso de minimización para obtener otro autómata finito con menos estados y menos transiciones. Los autómatas finitos tienen una propiedad que garantiza que este proceso de minimización siempre se puede llevar a cabo, y que además el nuevo autómata resultante es equivalente, es decir, reconoce exactamente el mismo conjunto de palabras que el autómata original [Hopcroft y Ullman 1979]. Por otra parte, en el caso de los autómatas finitos acíclicos deterministas, este proceso de minimización es particularmente sencillo, tal y como veremos más adelante.

Ejemplo 3.3 El autómata finito acíclico determinista mínimo correspondiente al árbol de letras de la figura 3.3 es el que se muestra en la figura 3.4. □

Por otra parte, y debido una vez más a esos mismos requerimientos de memoria, no resulta conveniente construir un diccionario insertando primero todas y cada una de las palabras en un árbol de letras y obteniendo después el autómata mínimo correspondiente a dicho árbol. En lugar de esto, es mucho más aconsejable realizar varias etapas de inserción y minimización. Por tanto, la construcción del autómata se realiza de acuerdo con los pasos básicos del siguiente algoritmo.

[8]El símbolo `#` que aparece en las figuras denota el fin de palabra.

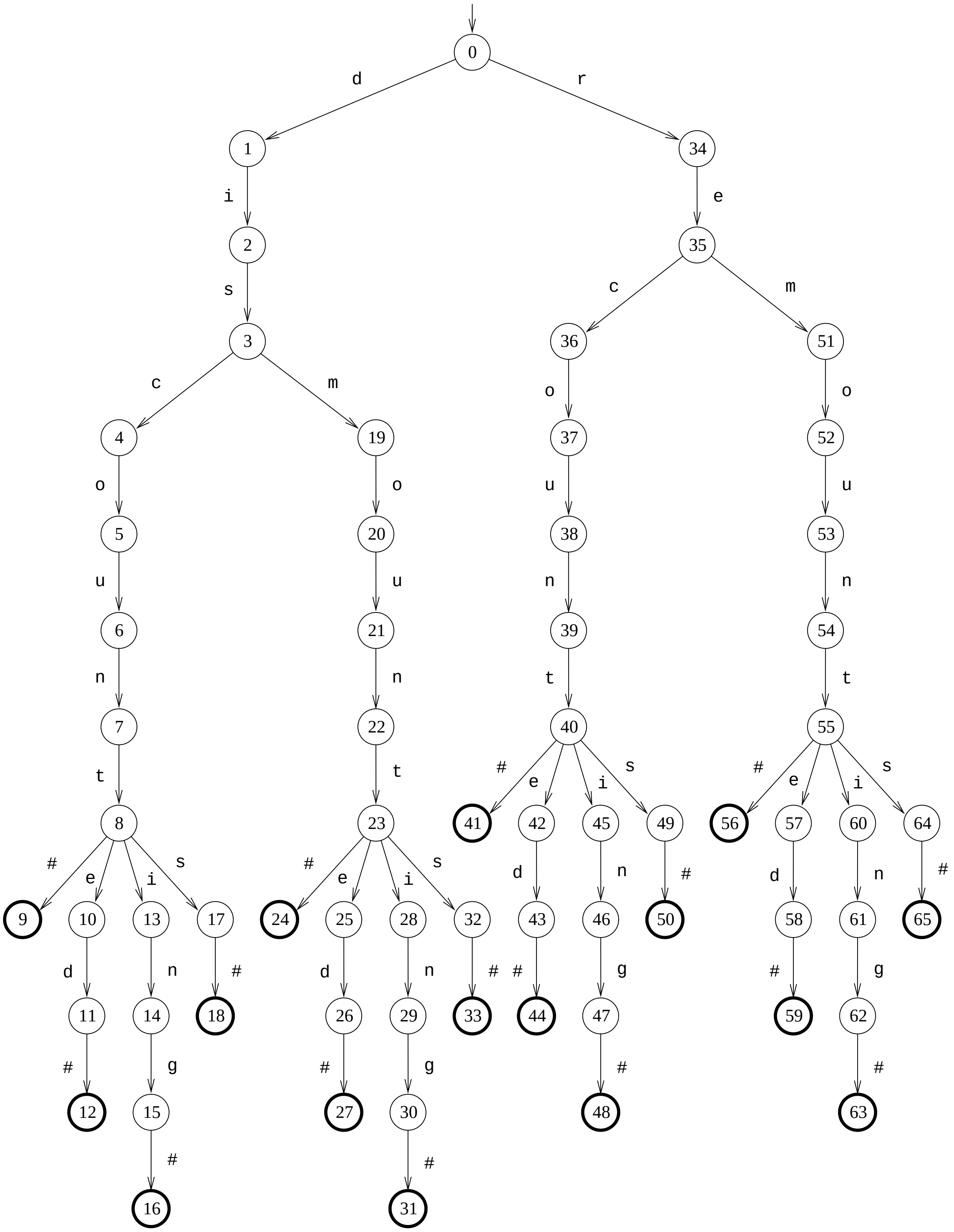

Figura 3.3: Árbol de letras para las formas de los verbos `discount`, `dismount`, `recount` y `remount`

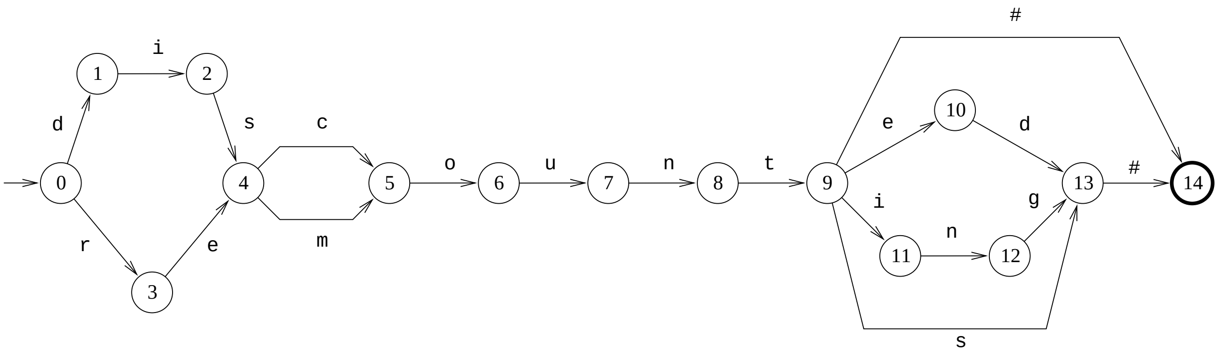

Figura 3.4: Autómata finito acíclico determinista mínimo para las formas de los verbos `discount`, `dismount`, `recount` y `remount`

Algoritmo 3.1 Algoritmo para la construcción de un autómata finito acíclico determinista a partir de un lexicón:

> **function** *Construir_Autómata* (*Lexicón*) =
>   **begin**
> $A \leftarrow$ *Autómata_Vacío*;
>
>   **while** (*queden palabras del Lexicón por insertar*) **do**
>   **begin**
>   **if** (*A está lleno*) **then**
> $A \leftarrow$ *Minimizar_Autómata* (A);
>   *Insertar la siguiente palabra del Lexicón en A*
>   **end**;
>
> $A \leftarrow$ *Minimizar_Autómata* (A);
>   **return** A
>   **end**;

Eligiendo un tamaño máximo previo para el autómata, este algoritmo de construcción permite que los consumos tanto de memoria como de tiempo por parte de los procesos de inserción y minimización sean muchísimo más moderados. □

Ejemplo 3.4 Fijando el tamaño máximo en 65.536 estados[9], el proceso de construcción del autómata finito acíclico determinista mínimo para el diccionario del sistema GALENA necesitó 10 etapas de inserción y minimización. La evolución de dicho proceso se puede observar en la tabla 3.1. □

En este momento es importante indicar cómo se debe realizar la correcta inserción de nuevas palabras en un autómata durante el proceso de construcción del mismo. Dicha inserción se puede llevar a cabo mediante una sencilla operación recursiva que hace uso de las transiciones que ya han aparecido, con el fin de compartir los caminos ya existentes en el grafo. Pero ocurre que este procedimiento *estándar* de inserción sólo es válido para los árboles de letras, y realmente nuestro

[9]La razón para elegir este número como cota superior del tamaño del autómata es que es el número máximo de enteros diferentes que se pueden almacenar en dos *bytes* de memoria, y además se ha comprobado empíricamente que resulta adecuado ya que con cotas mayores los pasos de minimización se alargan excesivamente, y con cotas menores el número de palabras que se pueden insertar en cada etapa de construcción es muy pequeño y como consecuencia el número de etapas necesarias crece considerablemente.

Etapa de construcción	Palabras insertadas	Antes de la minimización Estados	Transiciones	Después de la minimización Estados	Transiciones
1	34.839	65.526	100.363	2.277	5.219
2	68.356	65.527	101.986	4.641	11.431
3	100.325	65.526	104.285	6.419	16.163
4	132.094	65.530	107.043	7.377	18.560
5	163.426	65.532	108.047	8.350	21.094
6	193.368	65.525	108.211	9.858	24.877
7	223.743	65.530	110.924	10.646	27.166
8	253.703	65.527	112.007	11.221	28.850
9	283.281	65.528	112.735	11.779	30.617
10	291.604	26.987	54.148	11.985	31.258

Tabla 3.1: Evolución del proceso de construcción del autómata finito acíclico determinista mínimo para el diccionario del sistema GALENA

autómata sólo es un árbol de letras antes de la primera minimización. Si el procedimiento de inserción estándar se aplica después de minimizar, es decir, cuando el autómata ya no es un árbol de letras, las inserciones pueden dar lugar a errores de construcción.

Ejemplo 3.5 Tal es el caso de la inserción que se plantea en la figura 3.5. Nuestra intención aquí es incorporar la palabra `removal` en el autómata de la figura 3.4. Si añadimos esta nueva palabra mediante un procedimiento de inserción estándar, éste aprovecha la existencia del camino $0 \xrightarrow{\texttt{r}} 3 \xrightarrow{\texttt{e}} 4 \xrightarrow{\texttt{m}} 5 \xrightarrow{\texttt{o}} 6$ para añadir el menor número posible de nuevos estados y transiciones, y la operación degenera recursivamente en la inserción de la subpalabra `val` en el subautómata que comienza en el estado 6. Aparentemente todo es correcto, pero si nos fijamos bien observaremos que de repente hemos hecho válida no sólo la palabra `removal` sino también las palabras `discoval`, `dismoval` y `recoval`, lo cual no era nuestra intención, y además dichas palabras no existen en inglés. □

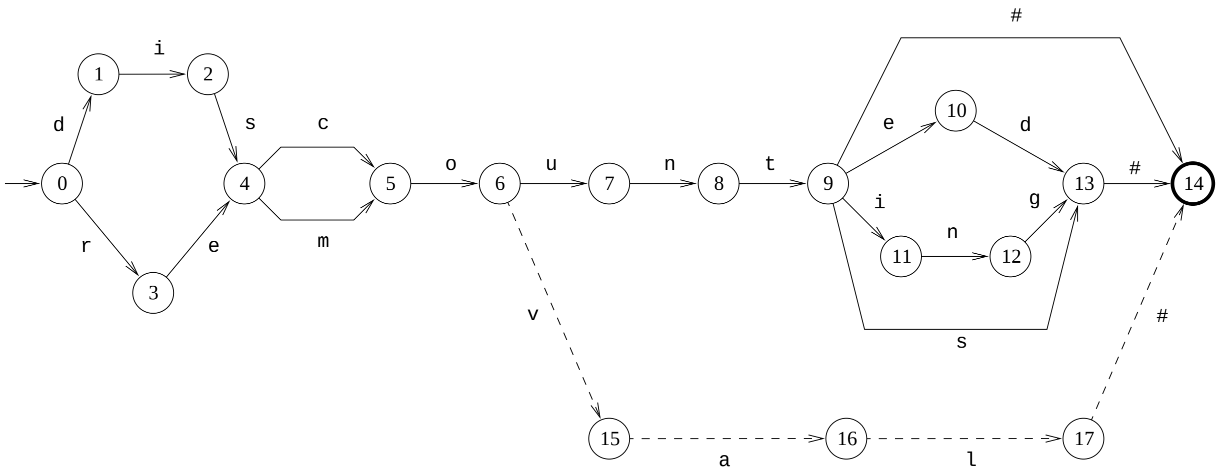

Figura 3.5: El problema de la inserción de nuevas palabras en un autómata finito acíclico determinista

La solución al problema planteado pasa por diseñar un nuevo procedimiento de inserción *especial*. Dicho procedimiento comprueba en todo momento que está haciendo uso de transiciones cuyo estado destino tiene una única transición entrante. Si se llega a un estado con más de una transición entrante, entonces se trata de un estado al que se puede llegar también al menos por otro camino distinto al que está recorriendo el proceso de inserción en ese momento. Si esto ocurre, entonces ese estado problemático se duplica, el estado destino de la transición implicada se cambia por el estado copia, y se continúa recursivamente la inserción de la palabra en el subautómata que comienza en este nuevo estado.

Ejemplo 3.6 Como se puede observar en la figura 3.6, esta operación de duplicado de un estado podría ser necesario realizarla más de una vez durante el proceso de inserción de una misma palabra, ya que es posible que se alcancen varias veces estados con más de una transición entrante, en nuestro caso, los estados 4, 5 y 6. □

Aparentemente, esta operación de rotura y duplicación de determinadas partes del autómata contradice el objetivo perseguido, que es el de obtener el autómata mínimo. Pero a medida que se inserten más y más palabras irán apareciendo nuevos fenómenos léxicos subceptibles de ser compartidos durante la siguiente aplicación del proceso de minimización.

En cualquier caso, lo que sí es importante señalar es que este procedimiento de inserción especial es más complejo que el procedimiento de inserción estándar. Pero ocurre que si insertamos las palabras en el autómata según su orden alfabético, entonces está garantizado que sólo es necesario aplicar el procedimiento de inserción especial a la primera palabra que aparece después de cada minimización, pudiéndose realizar el resto de inserciones con el procedimiento estándar. La demostración de esta afirmación es sencilla de razonar: si después de una inserción especial, al intentar insertar una nueva palabra, aparece algún estado problemático con más de una transición entrante, necesariamente dicha inserción ha de corresponder a una palabra lexicográficamente menor a la última palabra insertada en el autómata, y dado que las palabras se insertan en orden alfabético dicha palabra no puede aparecer.

Ejemplo 3.7 Si observamos otra vez la figura 3.6, es fácil comprobar intuitivamente que la inserción de cualquier palabra lexicográficamente mayor que `removal` caerá por debajo del camino de líneas punteadas y no pasará por ningún estado problemático, y por tanto se puede insertar normalmente en el autómata como si de un árbol de letras se tratara. □

En el momento en que el autómata se llena de nuevo, se realiza una minimización, una inserción especial, y se continúa con el proceso hasta que todas las palabras del lexicón hayan sido insertadas.

3.2.2 El algoritmo de minimización de la altura

Para definir formalmente el algoritmo de minimización, es necesario introducir primero los siguientes conceptos.

Definición 3.4 Se dice que dos autómatas son *equivalentes* si y sólo si reconocen el mismo lenguaje. Se dice también que dos estados p y q de un autómata dado son *equivalentes* si y sólo si el subautómata que comienza con p como estado inicial y el que comienza con q son equivalentes. O lo que es lo mismo, si para toda palabra w tal que $p.w$ es un estado final, entonces $q.w$ es también un estado final, y viceversa. □

Definición 3.5 El concepto contrario es que dos estados p y q se dicen *distinguibles* o *no equivalentes* si y sólo si existe una palabra w tal que $p.w$ es un estado final y $q.w$ no lo es, o viceversa. □

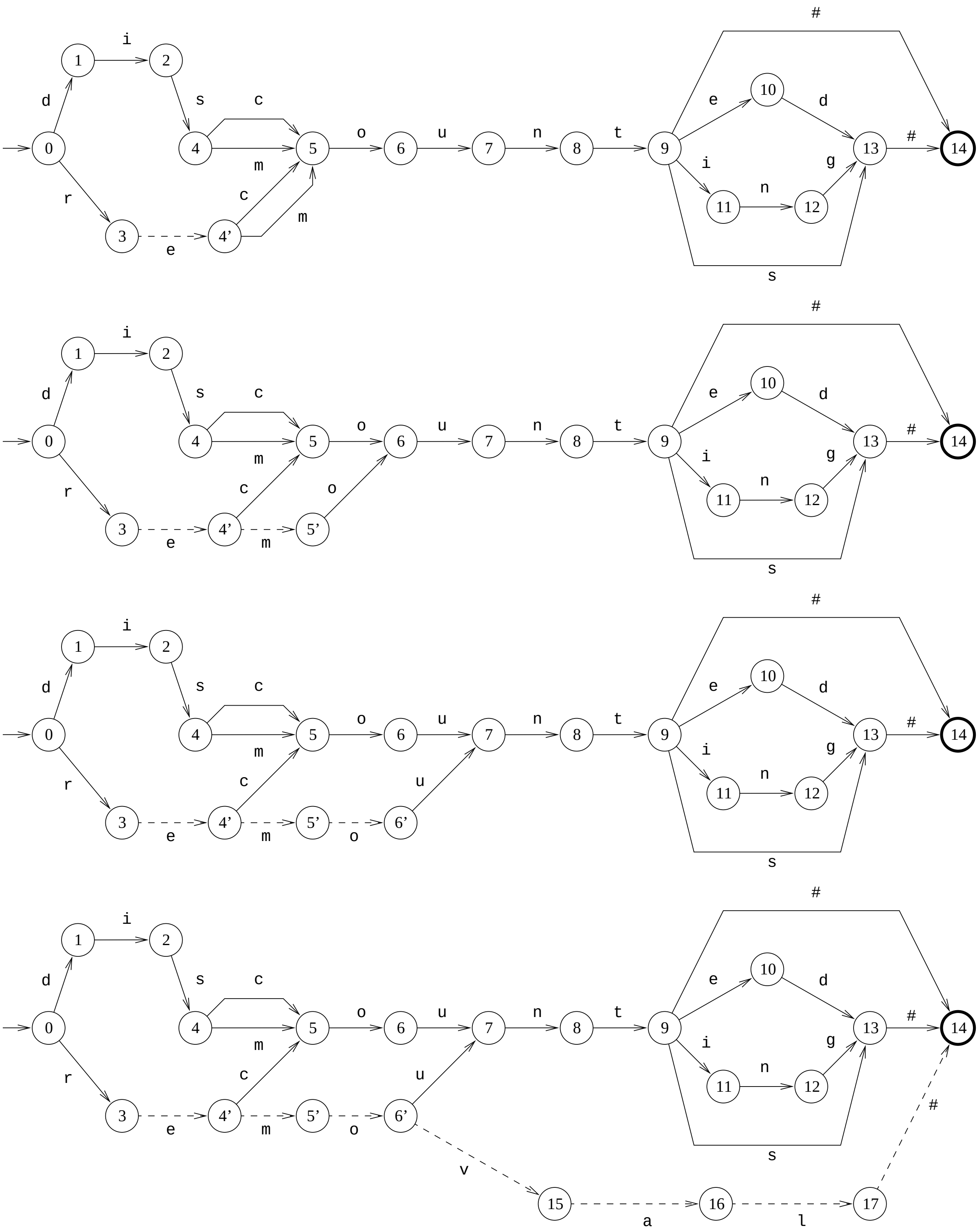

Figura 3.6: Inserción correcta de nuevas palabras en un autómata finito acíclico determinista

Definición 3.6 Si A es un autómata, existe un único autómata M, mínimo en el número de estados, que reconoce el mismo lenguaje, es decir, tal que $L(A) = L(M)$. Un autómata *mínimo* es aquél que no contiene ningún par de estados equivalentes. Y un autómata mínimo para un lenguaje dado L es por tanto aquél que contiene el menor número de estados posible de entre todos los autómatas que reconocen L. □

Definición 3.7 Si denotamos la *altura* de un estado s como $h(s)$, entonces $h(s) = max\,\{|w| \text{ tal que } s.w \in F\}$. Es decir, la altura de un estado s es la longitud del camino más largo de entre todos los que empiezan en dicho estado s y terminan en alguno de los estados finales. Esta función de altura establece una partición Π sobre Q, de tal manera que Π_i denota el conjunto de todos los estados de altura i. Diremos que el conjunto Π_i es *distinguible* si todos sus estados son *distinguibles*, es decir, si no contiene ningún par de estados equivalentes. □

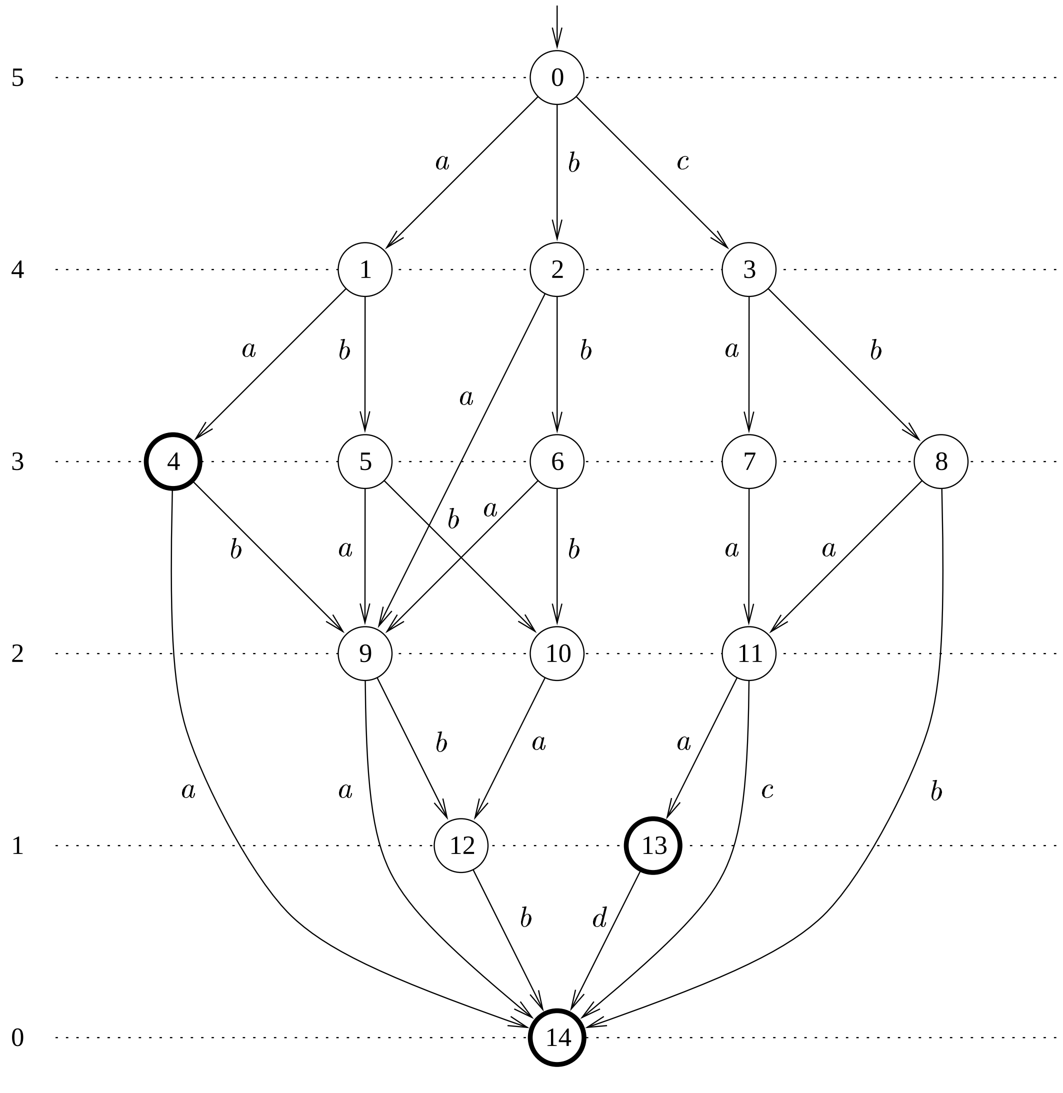

Figura 3.7: Un autómata finito acíclico determinista no mínimo

Ejemplo 3.8 La figura 3.7 muestra un autómata finito acíclico determinista, donde el estado inicial es el 0 y los estados finales son el 4, el 13 y el 14, y que por tanto reconoce el lenguaje $L = \{aa$, aaa, $aaba$, $aabbb$, $abaa$, $ababb$, $abbab$, baa, $babb$, $bbaa$, $bbabb$, $bbbab$, $caaa$, $caaad$, $caac$, *cbaa*, *cbaad*, *cbac*, *cbb*}. Los estados de la misma altura aparecen conectados por una línea punteada horizontal. Este autómata no es mínimo, ya que los estados 5 y 6 de altura 3 son equivalentes. Podemos colapsar dichos estados en uno solo eliminando uno de ellos, por ejemplo el 5, y cambiando el estado destino de sus transiciones entrantes por el otro estado, es decir, cambiando la transición $1 \xrightarrow{b} 5$ por $1 \xrightarrow{b} 6$. □

Una vez que ha sido definida la altura de un estado, estamos en condiciones de introducir el siguiente teorema.

Teorema 3.1 Si todos los Π_j con $j < i$ son distinguibles, entonces dos estados p y q pertenecientes a Π_i son equivalentes si y sólo si para cualquier letra $a \in \Sigma$ la igualdad $p.a = q.a$ se verifica.

Demostración: Si dicha igualdad se verifica, los estados son equivalentes, por la propia definición de estado equivalente. Así que para la demostración de esta propiedad, basta estudiar los casos en los que dicha igualdad no se verifica, y comprobar que efectivamente los estados no son equivalentes. Entonces, dados p y q dos estados de Π_i con $p.a \neq q.a$, tenemos dos posibilidades:

1. Cuando $p.a$ y $q.a$ pertenecen al mismo Π_j. Dado que $j < i$, que el autómata es acíclico, y que por hipótesis todos los estados de Π_j son distinguibles, entonces p y q también son distinguibles.

2. Cuando $p.a \in \Pi_j$ y $q.a \in \Pi_k$, $j \neq k$. Supongamos sin pérdida de generalidad que $k < j$. Entonces, por la definición de Π, existe una palabra w de longitud j tal que $(p.a).w$ es final y $(q.a).w$ no lo es. Por tanto, los estados $p.a$ y $q.a$ son distinguibles, y entonces p y q también son distinguibles.

Este resultado se conoce también como la *propiedad de la altura*. □

El algoritmo de minimización se puede deducir ahora de manera sencilla a partir de la propiedad de la altura [Revuz 1992].

Algoritmo 3.2 Los pasos básicos del algoritmo de minimización de un autómata finito acíclico determinista son los siguientes:

```
procedure Minimizar_Autómata (Autómata) =
  begin
    Calcular Π;

    for i ← 0 to h(q0) do
      begin
        Ordenar los estados de Πi según sus transiciones;
        Colapsar los estados equivalentes
      end
  end;
```

Primero creamos la partición del conjunto de estados según su altura. Esta partición se puede calcular mediante un recorrido recursivo estándar sobre el autómata, cuya complejidad temporal es $\mathcal{O}(t)$, donde t es el número de transiciones del autómata. No obstante, si el autómata no es un árbol, se puede ganar algo de velocidad mediante una marca que indique si la altura de un estado ha sido ya calculada o no. De igual manera, los estados no útiles no tendrán ninguna altura asignada y ya se pueden eliminar durante este recorrido. Posteriormente, se procesa cada uno de los Π_i, desde $i = 0$ hasta la altura del estado inicial, ordenando los estados según sus transiciones y colapsando los estados que resulten ser equivalentes. □

Utilizando un esquema de ordenación de complejidad temporal $\mathcal{O}(f(e))$, donde e es el número de elementos a ordenar, el algoritmo 3.2 presenta una complejidad

$$\mathcal{O}(t + \sum_{i=0}^{h(q_0)} f(|\Pi_i|))$$

que en el caso de los autómatas finitos acíclicos deterministas es menor que la complejidad del algoritmo general de Hopcroft para cualquier tipo de autómatas finitos deterministas: $\mathcal{O}(n \times \log n)$, donde n es el número de estados [Hopcroft y Ullman 1979].

3.2.3 Asignación y uso de los números de indexación

Hemos visto que los autómatas finitos acíclicos deterministas son la estructura más compacta que se puede diseñar para el reconocimiento de un conjunto finito de palabras dado. Los resultados de compresión son excelentes, y el tiempo de reconocimiento es lineal respecto a la longitud de la palabra a analizar, y no depende ni del tamaño del diccionario, ni del tamaño del autómata.

Sin embargo, si detenemos el proceso de construcción del autómata en este punto, dispondremos de una estructura que solamente es capaz de indicarnos si una palabra dada pertenece o no al diccionario, y esto no es suficiente para el esquema de modelización de diccionarios que hemos desarrollado anteriormente. Dicha modelización necesita un mecanismo que transfome cada palabra en una clave numérica unívoca, y viceversa.

Definición 3.8 Esta transformación se puede llevar a cabo fácilmente si el autómata incorpora, para cada estado, un entero que indique el número de palabras que se pueden aceptar mediante el subautómata que comienza en ese estado [Lucchesi y Kowaltowski 1993]. Nos referiremos a este autómata como *autómata finito acíclico determinista numerado.* □

Ejemplo 3.9 La versión numerada del autómata de la figura 3.4 es la que se muestra en la figura 3.8. □

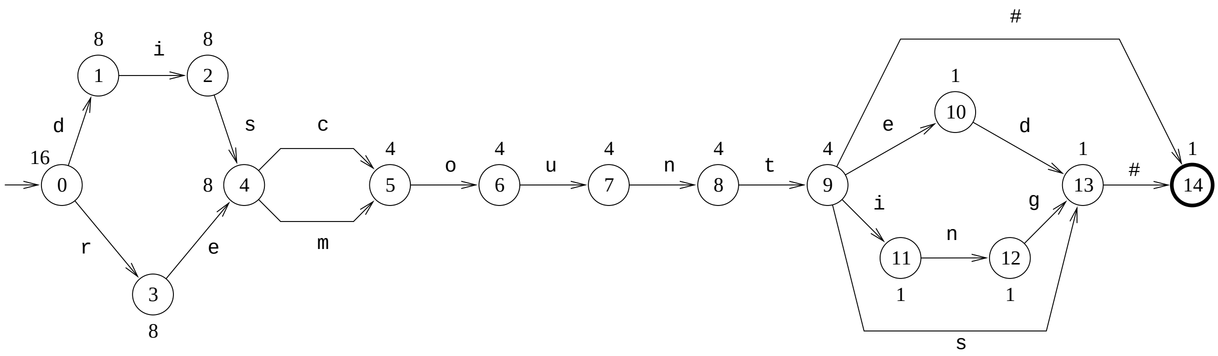

Figura 3.8: Autómata finito acíclico determinista mínimo numerado para las formas de los verbos `discount`, `dismount`, `recount` y `remount`

La asignación de los números de indexación a cada estado se puede realizar mediante un sencillo recorrido recursivo sobre el autómata, una vez que éste ha sido correctamente construido y minimizado. Por tanto, la versión definitiva de la función `Construir_Autómata` es la que se muestra a continuación.

Algoritmo 3.3 Algoritmo para la construcción de un autómata finito acíclico determinista a partir de un lexicón:

```
function Construir_Autómata (Lexicón) =
  begin
    A ← Autómata_Vacío;

    while (queden palabras del Lexicón por insertar) do
      if (A está lleno) then
        begin
          A ← Minimizar_Autómata (A);
          Inserción especial de la siguiente palabra del Lexicón en A
        end
      else
        Inserción estándar de la siguiente palabra del Lexicón en A;

    A ← Minimizar_Autómata (A);
    Asignar los números de indexación a los estados de A;
    return A
  end;
```

Este nuevo proceso de construcción del autómata completa y substituye al visto anteriormente en el algoritmo 3.1. □

Una vez que el autómata ha sido numerado, podemos ya escribir las funciones `Palabra_a_Índice` e `Índice_a_Palabra`, que son las que realizan la correspondencia uno a uno entre las palabras del diccionario y los números 1 a M, donde M es el número de total de palabras distintas aceptadas por el autómata.

Algoritmo 3.4 Pseudo-código de la función `Palabra_a_Índice`:

```
function Palabra_a_Índice (Palabra) =
  begin
    Índice ← 1;
    Estado_Actual ← Estado_Inicial;

    for i ← 1 to Longitud (Palabra) do
      if (Transición_Válida (Estado_Actual, Palabra[i])) then
        begin
          for c ← Primera_Letra to Predecesor (Palabra[i]) do
            if (Transición_Válida (Estado_Actual, c)) then
              Índice ← Índice + Estado_Actual[c].Número;
          Estado_Actual ← Estado_Actual[Palabra[i]];
        end
      else
        return palabra desconocida;

    if (Es_Estado_Final (Estado_Actual)) then
      return Índice
    else
      return palabra desconocida
  end;
```

Esta función parte con un índice igual a 1 y va transitando por el autómata desde el estado inicial utilizando cada una de las letras de la palabra a analizar. En cada uno de los estados por los que pasa el camino correspondiente a dicha palabra, el índice se va incrementando con el número de indexación del estado destino de aquellas transiciones que son lexicográficamente precedentes a la transición utilizada. Si después de procesar todos los caracteres de la palabra llegamos al estado final, entonces el índice contendrá la clave numérica de la palabra. En caso contrario, la palabra no pertenece al lexicón que se está manejando. Como consecuencia, el valor del índice no es correcto, y en su lugar la función devuelve un valor que indica que la palabra es desconocida. □

Algoritmo 3.5 Pseudo-código de la función `Índice_a_Palabra`:

```
function Índice_a_Palabra (Índice) =
  begin
    Estado_Actual ← Estado_Inicial;
    Número ← Índice;
    Palabra ← Palabra_Vacía;
    i ← 1;

    repeat
      for c ← Primera_Letra to Última_Letra do
        if (Transición_Válida (Estado_Actual, c)) then
          begin
            Estado_Auxiliar ← Estado_Actual[c];
            if (Número > Estado_Auxiliar.Número) then
              Número ← Número − Estado_Auxiliar.Número
            else
              begin
                Palabra[i] ← c;
                i ← i + 1;
                Estado_Actual ← Estado_Auxiliar;
                if (Es_Estado_Final (Estado_Actual)) then
                  Número ← Número − 1;
                exit forloop
              end
          end
    until (Número = 0);

    return Palabra
  end;
```

Esta función parte del índice y realiza las operaciones análogas a las del algoritmo 3.4, para deducir cuáles son las transiciones que dan lugar a ese índice, y a partir de esas transiciones obtiene las letras que forman la palabra que se está buscando. □

Ejemplo 3.10 En el caso del autómata numerado de la figura 3.8, la correspondencia individual de cada palabra con su índice es como sigue:

```
 1 <-> discount   2 <-> discounted    3 <-> discounting   4 <-> discounts
 5 <-> dismount   6 <-> dismounted    7 <-> dismounting   8 <-> dismounts
 9 <-> recount   10 <-> recounted    11 <-> recounting   12 <-> recounts
13 <-> remount   14 <-> remounted    15 <-> remounting   16 <-> remounts
```

Se puede observar que M, en este caso 16, corresponde efectivamente con el número de indexación del estado inicial, que es el que indica el número total de palabras que puede reconocer el autómata. □

Los requerimientos de almacenamiento y la complejidad temporal extra en el proceso de reconocimiento implicada por la incorporación de los números de indexación resulta modesta. Así que finalmente sólo nos queda hacer una pequeña referencia a las prestaciones de nuestro analizador léxico, que simplemente confirma algunos de los resultados ya comentados en la sección 2.2.2: el autómata finito acíclico determinista numerado correspondiente al diccionario GALENA consta de 11.985 estados y 31.258 transiciones; el tamaño del fichero compilado correspondiente a dicho autómata es de 3.466.121 *bytes*; el tiempo de compilación es de aproximadamente 29 segundos; y la velocidad de reconocimiento en una máquina con un procesador Pentium II a 300 MHz bajo sistema operativo Linux es de aproximadamente 40.000 palabras por segundo.

3.2.4 Algoritmos de construcción incrementales

Como hemos visto, los métodos tradicionales para la construcción de autómatas finitos acíclicos deterministas mínimos a partir de un conjunto de palabras consisten en combinar dos fases de operación: la construcción de un árbol o de un autómata parcial, y su posterior minimización. Sin embargo, existen métodos de construcción incremental, capaces de realizar las operaciones de minimización en línea, es decir, al mismo tiempo que se realizan las inserciones de las palabras en el autómata [Daciuk *et al.* 2000]. Estos métodos son mucho más rápidos, y sus requerimientos de memoria son también significativamente menores que los de los métodos descritos anteriormente.

Para construir un autómata palabra a palabra de manera incremental es necesario combinar el proceso de minimización con el proceso de inserción de nuevas palabras. Por tanto, hay dos preguntas cruciales que hay que responder:

1. ¿Qué estados, o clases de equivalencia de estados, son subceptibles de cambiar cuando se insertan nuevas palabras en el autómata?

2. ¿Existe alguna manera de minimizar el número de estados que es necesario cambiar durante la inserción de una palabra?

Como ya sabemos, si las palabras están ordenadas lexicográficamente, cuando se añade una nueva, sólo pueden cambiar los estados que se atraviesan al aceptar la palabra insertada previamente. El resto del autómata permanece inalterado, ya que la nueva palabra:

- O bien comienza con un símbolo diferente del primer símbolo de todas las palabras ya presentes en el autómata, con lo cual el símbolo inicial de la nueva palabra es situado lexicográficamente después de todos esos símbolos.

- O bien comparte algunos de los símbolos iniciales de la palabra añadida previamente. En este caso, el algoritmo localiza el último estado en el camino de ese prefijo común y crea una nueva rama desde ese estado. Esto es debido a que el símbolo que etiqueta la nueva transición será lexicográficamente mayor que los símbolos del resto de transiciones salientes que ya existen en ese estado.

Por tanto, cuando la palabra previa es prefijo de la nueva palabra a insertar, los únicos estados que pueden cambiar son los estados del camino de reconocimiento de la palabra previa que no están en el camino del prefijo común. La nueva palabra puede tener una finalización igual a la de otras palabras ya insertadas, lo cual implica la necesidad de crear enlaces a algunas partes del autómata. Esas partes, sin embargo, no se verán afectadas.

A continuación describimos el algoritmo de construcción incremental a partir de un conjunto de palabras ordenado lexicográficamente. Dicho algoritmo se apoya en una estructura denominada *Registro* que mantiene en todo momento un único representante de cada una de las clases de equivalencia de estados del autómata. Es decir, el *Registro* constituye en sí mismo el autómata mínimo en cada instante.

Algoritmo 3.6 El algoritmo de construcción incremental de un autómata finito acíclico determinista, a partir de un conjunto de palabras ordenado lexicográficamente, consta básicamente de dos funciones: la función principal *Construir_Autómata_Incremental* y la función *Reemplazar_o_Registrar*. Los pasos de la función principal son los siguientes:

function *Construir_Autómata_Incremental* (*Lexicón*) =
  begin
    Registro $\leftarrow \emptyset$;

    while (*queden palabras del Lexicón por insertar*) **do**
      begin
        Palabra $\leftarrow$ *siguiente palabra del Lexicón en orden lexicográfico*;
        Prefijo_Común $\leftarrow$ *Prefijo_Común* (*Palabra*);
        Último_Estado $\leftarrow q_0$.*Prefijo_Común*;
        Sufijo_Actual $\leftarrow$ *Palabra*[(*Longitud* (*Prefijo_Común*) $+ 1) \ldots$ *Longitud* (*Palabra*)];
        if (*Tiene_Hijos* (*Último_Estado*)) **then**
          Registro $\leftarrow$ *Reemplazar_o_Registrar* (*Último_Estado*, *Registro*);
        Añadir_Sufijo (*Último_Estado*, *Sufijo_Actual*);
      end;

    Registro $\leftarrow$ *Reemplazar_o_Registrar* (q_0, *Registro*);
    return *Registro*
  end;

El esquema de la función *Reemplazar_o_Registrar* es como sigue:

function *Reemplazar_o_Registrar* (*Estado*, *Registro*) =
  begin
    Hijo $\leftarrow$ *Último_Hijo* (*Estado*);
    if (*Tiene_Hijos* (*Hijo*)) **then**
      Registro $\leftarrow$ *Reemplazar_o_Registrar* (*Hijo*, *Registro*);
    if ($\exists\, q \in Q : q \in$ *Registro* $\wedge\ q \equiv$ *Hijo*) **then**
      begin
        Último_Hijo (*Estado*) $\leftarrow q$;
        Eliminar (*Hijo*)
      end
    else
      Registro $\leftarrow$ *Registro* $\cup$ *Hijo*;
    return *Registro*
  end;

El lazo principal del algoritmo lee las palabras y establece qué parte de cada palabra está ya en el autómata, es decir, el *Prefijo_Común*, y qué parte no está, es decir, el *Sufijo_Actual*. Un paso importante es determinar cuál es el último estado en el camino del prefijo común, que en el algoritmo se denota como *Último_Estado*. Si *Último_Estado* ya tiene hijos, quiere decir que no todos los estados del camino de la palabra añadida previamente están en el camino del prefijo común. En ese caso, mediante la función *Reemplazar_o_Registrar*, hacemos que el proceso de minimización trabaje sobre los estados del camino de la palabra añadida previamente que no están en el camino del prefijo común. Posteriormente, añadimos desde el *Último_Estado* una cadena de nuevos estados capaz de reconocer el *Sufijo_Actual*.

La función *Prefijo_Común* busca el prefijo más largo de la palabra a insertar que es prefijo de alguna palabra ya insertada. La función *Añadir_Sufijo* crea una nueva rama que extiende el autómata, la cual representa el sufijo no encontrado de la palabra que se va a insertar. La función *Último_Hijo* devuelve una referencia al estado alcanzado por la última transición en orden lexicográfico que sale del estado argumento. Dado que los datos de entrada están ordenados, dicha transición es la transición añadida más recientemente en el estado argumento, durante la inserción de la palabra previa. La función *Tiene_Hijos* es cierta si y sólo si el estado argumento tiene transiciones salientes.

La función *Reemplazar_o_Registrar* trabaja efectivamente sobre el último hijo del estado argumento. Dicho argumento es el último estado del camino del prefijo común, o bien el estado inicial del autómata en la última llamada de la función principal. Es necesario que el estado argumento cambie su última transición en aquellos casos en los que el hijo va a ser reemplazado por otro estado equivalente ya registrado. En primer lugar, la función se llama recursivamente a sí misma hasta alcanzar el final del camino de la palabra insertada previamente. Nótese que cada vez que se encuentra un estado con más de un hijo, siempre se elige el último. La longitud limitada de las palabras garantiza el fin de la recursividad. Por tanto, al volver de cada llamada recursiva, se comprueba si ya existe en el registro algún estado equivalente al estado actual. Si es así, el estado actual se reemplaza por el estado equivalente encontrado en el registro. Si no, el estado actual se registra como representante de una nueva clase de equivalencia. Es importante destacar que la función *Reemplazar_o_Registrar* sólo procesa estados pertenecientes al camino de la palabra insertada previamente, y que esos estados no se vuelven a procesar. □

Durante la construcción, los estados del autómata o están en el registro o están en el camino de la última palabra insertada. Todos los estados del registro son estados que formarán parte del autómata mínimo resultante. Así pues, el número de estados durante la construcción es siempre menor que el número de estados del autómata resultante más la longitud de la palabra más larga. Por tanto, la complejidad espacial del algoritmo es $\mathcal{O}(n)$, es decir, la cantidad de memoria que precisa el algoritmo es proporcional a n, el número final de estados del autómata mínimo. Respecto al tiempo de ejecución, éste es dependiente de la estructura de datos implementada para manejar las búsquedas de estados equivalentes y las inserciones de nuevos estados representantes en el registro. Utilizando criterios de búsqueda e inserción basados en el número de transiciones salientes de los estados, en sus alturas y en sus números de indexación, los cuales se pueden también calcular dinámicamente, la complejidad temporal se puede rebajar hasta $\mathcal{O}(\log n)$.

Con este algoritmo incremental, el tiempo de construcción del autómata correspondiente al diccionario del sistema GALENA se reduce considerablemente: de 29 segundos a 2,5 segundos, en una máquina con un procesador Pentium II a 300 MHz bajo sistema operativo Linux. La incorporación de la información relativa a las etiquetas, lemas y probabilidades consume un tiempo extra aproximado de 4,5 segundos, lo que hace un tiempo de compilación total de 7 segundos, tal y como se había indicado en la sección 2.2.2.

En el mismo trabajo, los autores proponen también un método incremental para la construcción de autómatas finitos acíclicos deterministas mínimos a partir de conjuntos de palabras no ordenados [Daciuk *et al.* 2000]. Este método se apoya también en la clonación o duplicación de los estados que se van volviendo conflictivos a medida que aparecen las nuevas palabras. Por esta razón, la construcción incremental a partir de datos desordenados es más lenta y consume más memoria que la construcción incremental a partir de esos mismos datos ordenados. No obstante, el método puede ser de gran utilidad en situaciones donde el tamaño de los diccionarios es tan elevado que el propio proceso de ordenación podría resultar problemático.

3.3 Otros métodos de análisis léxico

Los procesos productivos o derivativos presentes en todos los idiomas constituyen una gran fuente de complicaciones para el análisis morfológico. Debido a ellos, siempre existirán multitud de palabras que no estarán incluidas en un diccionario morfológico estático, sin importar lo grande o preciso que éste sea. En otras palabras, se podría decir que el tamaño del lexicón de cualquier lengua es virtualmente infinito. Inevitablemente, éste es un problema al que deben enfrentarse los etiquetadores. Tal y como veremos con detalle en los capítulos siguientes, todo etiquetador debe incluir un módulo de tratamiento de palabras desconocidas, independientemente de los mecanismos que tenga integrados para el análisis léxico y para el acceso al diccionario.

Pero efectivamente, muchas de las palabras desconocidas serán formas derivadas a partir de una raíz y de una serie de reglas de inflexión. Por ejemplo, es muy probable que la palabra `reanalizable` no esté incluida en un diccionario dado, pero parece claro que esta palabra es parte del lenguaje, en el sentido de que puede ser utilizada, es comprensible y podría ser fácilmente deducida a partir de la palabra `analizar`[10] y de reglas de derivación tales como la incorporación de prefijos o la transformación de los verbos en sus respectivos adjetivos de calidad.

Además, no siempre es necesario manejar una información tan particular para cada palabra concreta como puede ser su frecuencia o su probabilidad. Fuera de los paradigmas de aproximación estocástica al NLP, la mayoría de las aplicaciones utilizan sólo la etiqueta, o como mucho la etiqueta y el lema. En esta última sección, por tanto, enumeramos brevemente otros posibles métodos de análisis léxico que han sido desarrollados.

La aproximación quizás más importante, y que ha servido de base para multitud de trabajos relacionados con la construcción automática de analizadores léxicos a partir de la especificación de las reglas morfológicas de flexión y derivación de una determinada lengua, es el modelo de *morfología de dos niveles*[11] [Koskenniemi 1983]. Este modelo se basa en la distinción tradicional que hacen los lingüistas entre, por un lado, el conjunto teórico de morfemas y el orden en el cual pueden ocurrir, y por otro lado, las formas alternativas que presentan dichos morfemas dentro del contexto fonológico en el que aparecen. Esto establece la diferencia entre lo que se denomina la *cadena superficial* del lenguaje (por ejemplo, la palabra `quiero`), y su correspondiente *cadena léxica* o *teórica* (en este caso, `quero` = `quer` + `o`). Por tanto, `quer` y `quier` se consideran alomorfos o formas alternativas del mismo morfema. El modelo de Koskenniemi es de dos niveles en el sentido de que cualquier palabra se puede representar mediante una correspondencia directa letra a letra entre su cadena de superficie y la cadena teórica subyacente.

Ejemplo 3.11 Para la palabra inglesa `skies`, podemos asumir que existe una raíz `sky` y una terminación de plural `-es`, salvo que, como ocurre en inglés con todos los sustantivos terminados en `y`, la `y` se realiza como una `i`, y por tanto las cadenas teórica y superficial serían, respectivamente:

```
s k y - e s
s k i 0 e s
```

En este ejemplo, el `-` inidica el límite de morfemas, y el `0` un caracter nulo o no existente. □

[10] Suponiendo que ésta sí está presente en el lexicón.

[11] *Two-level morphology.*

Posibles representaciones de las correspondencias letra a letra son `s:s` (no necesaria, porque por defecto cada letra se corresponde consigo misma, y por tanto dicha relación se puede expresar simplemente como `s`), o bien `-:0`, o también `y:i`. Respecto a esta última relación habría que especificar lo siguiente:

- No es cierto que cualquier `y` se corresponda con una `i`. Esto sólo es válido para el proceso de formación de plural que estamos considerando.
- En este caso concreto, la `y` se corresponde sólo con una `i`, y con ninguna otra letra, ni siquiera consigo misma.

El modelo de morfología de dos niveles incorpora un componente más que permite expresar este tipo de restricciones mediante reglas de transformación. Por ejemplo, esta relación *si y sólo si* podría denotarse mediante una regla de doble flecha como la siguiente:

```
y:i <=> _ -: e: s: ;
```

Un conjunto dado de reglas de este tipo se puede transformar en un traductor de estado finito [Karttunen y Beesley 1992]. Un *traductor de estado finito* no es más que un autómata finito que, en lugar de caracteres simples, incorpora correspondencias letra a letra como etiquetas de sus transiciones, permitiendo transformar la cadena superficial en la cadena teórica, y viceversa.

Ejemplo 3.12 La figura 3.9 representa un traductor de estado finito capaz de asociar cada una de las formas verbales del verbo `leave` con su lema y su correspondiente etiqueta, es decir: `leave` con `leave+VB`, `leaves` con `leave+VBZ` y `left` con `leave+VBD`[12]. □

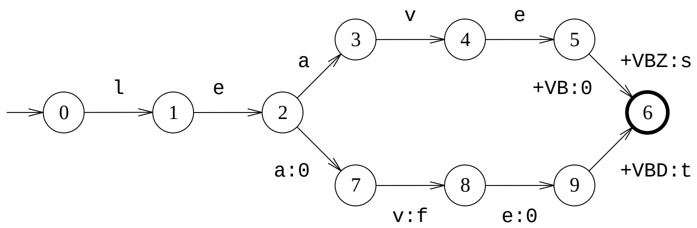

Figura 3.9: Traductor de estado finito para el verbo `leave`

Para cada palabra, existe en el traductor un camino que contiene la forma flexionada de la cadena superficial y el par lema-etiqueta de la cadena teórica. Dicho camino se puede encontrar utilizando como clave de entrada cualquiera de las dos cadenas. Esta característica de procesamiento bidireccional de los traductores finitos hace que el modelo de morfología de dos niveles resulte válido tanto para el análisis morfológico, como para la generación de formas.

Sin embargo, para los traductores, la noción de determinismo normalmente tiene sentido si se considera sólo una de las direcciones de aplicación. Si un traductor presenta este tipo de determinismo unidireccional, se dice que es o bien *secuencial por arriba*[13] si es determinista en la dirección de la cadena superficial, o bien *secuencial por abajo*[14] si es determinista en la dirección de la cadena teórica. Por ejemplo, el traductor de la figura 3.9 es secuencial por arriba, pero no secuencial por abajo, ya que desde el estado 2 con el símbolo de entrada `a` se puede transitar a dos estados diferentes, el 3 y el 7.

[12] La notación de las etiquetas es una adaptación del juego de etiquetas utilizado en el corpus Brown [Francis y Kučera 1982]: `VB` significa verbo en infinitivo, `VBZ` es verbo en tercera persona del singular y `VBD` es verbo en tiempo pasado.

[13] *Sequential upward.*

[14] *Sequential downward.*

No obstante, al procesar una cadena dada, es probable que uno de los caminos termine con éxito y el otro no. Es decir, esta ambigüedad local podría ser resuelta algunas transiciones más tarde, y de hecho en este caso es así, ya que la relación que establece el traductor de la figura 3.9 es no ambigua en ambas direcciones. Si la relación es no ambigua en una dirección dada, y si todas las ambigüedades locales en esa dirección se pueden resolver a sí mismas en un número finito de pasos, se dice que el traductor es *secuenciable*, es decir, se puede construir un traductor secuencial equivalente en la misma dirección [Roche y Schabes 1995, Mohri 1995].

Ejemplo 3.13 La ambigüedad local del traductor de la figura 3.9 se resuelve en el estado 5. Por tanto, se puede incorporar un camino desde el estado inicial hasta el estado 5 que transforma la secuencia `ave` en la cadena vacía, lo cual, tal y como se muestra en la figura 3.10, permite construir un traductor secuencial por abajo, pero sólo en esa dirección, ya que ahora existen dos arcos con `a` en la cadena superior partiendo del estado 5. □

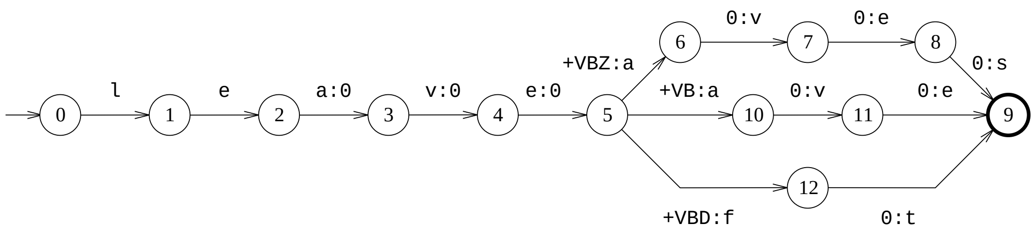

Figura 3.10: Traductor de estado finito secuencial por abajo para el verbo `leave`

Los traductores secuenciales se pueden extender para permitir que emitan desde los estados finales una cadena de salida adicional (*traductores subsecuenciales*) o un número finito p de cadenas de salida (*traductores p-subsecuenciales*). Estos últimos son los que permiten realizar el manejo de las ambigüedades léxicas presentes en los lenguajes naturales [Mohri 1995].

Y todos ellos, mediante la incorporación de una información numérica extra en cada estado, pueden pasar de ser simples conversores de una cadena en otra[15] a incorporar también la funcionalidad de conversión de una cadena en un peso numérico[16]. Típicamente, ese nuevo dato numérico podría ser una probabilidad logarítmica, de tal manera que la probabilidad de emisión de una palabra dada se calcularía simplemente sumando el dato de todos los estados por los que pasa el camino de procesamiento de dicha palabra en el traductor. Ésta es la forma de integrar probabilidades de emisión para las palabras en los traductores finitos [Mohri 1997].

De entre los múltiples trabajos basados en el modelo de Koskenniemi, destaca la herramienta MMORPH [Russell y Petitpierre 1995], un compilador de reglas morfológicas de dos niveles de libre distribución, desarrollado en el marco del proyecto MULTEXT[17].

Otros métodos de diseño de analizadores léxicos, que también han sido aplicados con éxito al idioma español, incluyen las aproximaciones basadas en unificación [Moreno 1991, Moreno y Goñi 1995, González Collar *et al.* 1995] y las basadas en árboles de decisión [Triviño 1995, Triviño y Morales 2000].

[15] Traductores *string-to-string*.

[16] Traductores *string-to-weight*.

[17] *Multilingual Text Tools and Corpora* es una iniciativa de la Comisión Europea (bajo los programas *Investigación e Ingeniería Lingüística*, *Copernicus*, y *Lenguas Regionales y Minoritarias*), de la Fundación Científica Nacional de los Estados Unidos, del Fondo Francófono para la Investigación, del CNRS (Centro Nacional francés para la Investigación Científica) y de la Universidad de Provenza. MULTEXT comprende una serie de proyectos cuyos objetivos son el desarrollo de estándares y especificaciones para la codificación y el procesamiento de textos, y para el desarrollo de herramientas, *corpora* y recursos lingüísticos bajo esos estándares. Los trabajos han sido realizados también en estrecha colaboración con los proyectos EAGLES y CRATER, e incluyen herramientas y recursos para la mayoría de los idiomas europeos.