Construcción y Manejo de Autómatas Finitos Acíclicos

Jorge Graña Gil

Departamento de Computación
Facultad de Informática
Universidad de La Coruña
Campus de Elviña s/n
15071 - La Coruña
España

grana@dc.fi.udc.es

Resumen

Este trabajo presenta una reflexión sobre la evolución de las diferentes técnicas para la construcción de autómatas de estado finito acíclicos deterministas mínimos a partir de conjuntos finitos de palabras. Se esbozan los métodos más importantes, incluyendo los tradicionales (que consisten en la combinación de dos fases: la inserción de palabras y la minimización del autómata parcial) y los algoritmos incrementales (que añaden las nuevas palabras una a una y minimizan el autómata resultante en cada paso, por lo que son mucho más rápidos y presentan unos requerimientos de memoria mucho menores). Se analizan las características principales de todos estos algoritmos con el fin de proporcionar algunas mejoras para las construcciones incrementales, y se presenta también una arquitectura general necesaria para la implementación de grandes diccionarios en las aplicaciones de procesamiento automático de los lenguajes naturales (PLN). Finalmente, mediante la consideración de otros dos algoritmos, el que inserta palabras en el autómata sin la exigencia del orden lexicográfico, y el que es capaz de eliminar palabras del autómata, se estudia la posibilidad del comportamiento dinámico de este tipo de autómatas.

1 Introducción

Muchas aplicaciones de PLN, tales como la etiquetación de las partes del discurso o el análisis sintáctico de un frase dada, pueden resultar demasiado complejas si se trata directamente el flujo de caracteres de entrada que forman la frase. Normalmente, un paso de procesamiento previo transforma esos caracteres en un flujo de elementos de mayor nivel de significado (llamados *tokens* y que típicamente son las palabras de la frase), y obtiene rápida y cómodamente las etiquetas candidatas para esas palabras. Este paso previo se denomina análisis léxico o *scanning*.

El uso de autómatas de estado finito para implementar *scanners* eficientes es una técnica comúnmente aceptada [1]. Las principales razones para querer comprimir un gran diccionario de palabras bajo la forma de un autómata de estado finito son que la representación del conjunto de palabras es muy compacta, y que la búsqueda de una palabra en el diccionario es muy rápida (proporcional a la longitud de la palabra) [6]. Para las aplicaciones de PLN, resultan de particular interés los autómatas finitos acíclicos mínimos, los cuales reconocen conjuntos finitos de palabras.

Los autómatas de este tipo se pueden construir de varias maneras [10]. Este trabajo esboza los métodos más importantes y analiza sus características principales, con el fin de proponer algunas mejoras para los algoritmos de construcción incremental. La motivación inicial de este trabajo fue construir una arquitectura general para manejar adecuadamente dos grandes diccionarios del español: el diccionario GALENA (291.604 palabras con 354.007 posibles etiquetaciones) y el diccionario ERIAL (775.621 palabras con 993.703 posibles etiquetaciones)[1].

La sección 2 describe el modelo general de diccionario y permite entender el papel de los autómatas finitos aquí. En la sección 3, se proporcionan las definiciones formales y se explica cómo establecer un *hashing* perfecto entre las palabras y sus respectivas posiciones en el diccionario, simplemente mediante la asignación de un *peso* a cada estado del autómata [7]. La sección 4 recuerda un algoritmo de minimización debido a Revuz [8], que está basado en otra propiedad de los estados: la *altura*. La sección 5 presenta la construcción incremental de Daciuk [3], la cual realiza a la vez las inserciones y las minimizaciones, almacenando en un *registro* los estados que formarán el autómata final. En la sección 6, se combinan los *pesos* y las *alturas* para mejorar el acceso al *registro*, y se compara esta implementación con las anteriores. La sección 7 analiza el comportamiento dinámico de los autómatas acíclicos, considerando los algoritmos que resuelven las necesidades de introducir nuevas palabras o de eliminar palabras introducidas previamente. La sección 8 presenta las conclusiones extraídas después del análisis de los datos obtenidos.

2 Modelización Compacta de un Diccionario

Muchas de las palabras presentes en un diccionario son manualmente insertadas por los lingüistas para cubrir exhaustivamente el núcleo invariable de un determinado idioma (artículos, preposiciones, conjunciones, etc.) o la terminología de un campo de aplicación específico. Pero muchas otras palabras pueden ser capturadas a partir de textos anotados, lo cual hace posible la obtención de información adicional, como puede ser la frecuencia de una palabra o la probabilidad con respecto a cada una de sus posibles etiquetas. Esta información es esencial en algunas aplicaciones, por ejemplo, en la etiquetación estocástica o en el análisis sintáctico probabilístico.

Por tanto, la primera visión de un diccionario es simplemente un fichero de texto, con el siguiente formato para cada línea: `palabra etiqueta lema probabilidad`. Las palabras ambiguas utilizan una línea diferente para cada posible etiqueta. Sin pérdida de generalidad, las palabras podrían estar ordenadas alfabéticamente. Entonces, en el caso del diccionario GALENA, el punto en el que aparece la ambigüedad de la palabra `sobre` tiene este aspecto[2]:

```
sobre P sobre 0.113229
sobre Scms sobre 0.0012629
sobre Vysps0 sobrar 0.0117647
```

Para una discusión posterior, diremos que el diccionario GALENA tiene $M = 291.604$ palabras diferentes, con $L = 354.007$ posibles etiquetaciones. Este último número es precisamente el número de líneas del fichero de texto. La primera etiquetación de `sobre` aparece en la línea 325.611, pero la palabra ocupa la posición 268.249 en el conjunto de las M palabras diferentes ordenadas lexicográficamente.

[1] GALENA viene de *Generador de Analizadores para LEnguajes NAturales* y ERIAL viene de *Extracción y Recuperación de Información Aplicando conocimiento Lingüístico*. Véase `http://coleweb.dc.fi.udc.es` para más información sobre ambos proyectos.

[2] Las etiquetas vienen del juego de etiquetas del proyecto GALENA, el cual tiene un cardinal de $T = 373$ etiquetas. El significado de las etiquetas es el siguiente: `P` es preposición; `Scms` es sustantivo, común, masculino, singular; y `Vysps0` es verbo, primera o tercera persona, singular, presente, subjuntivo.

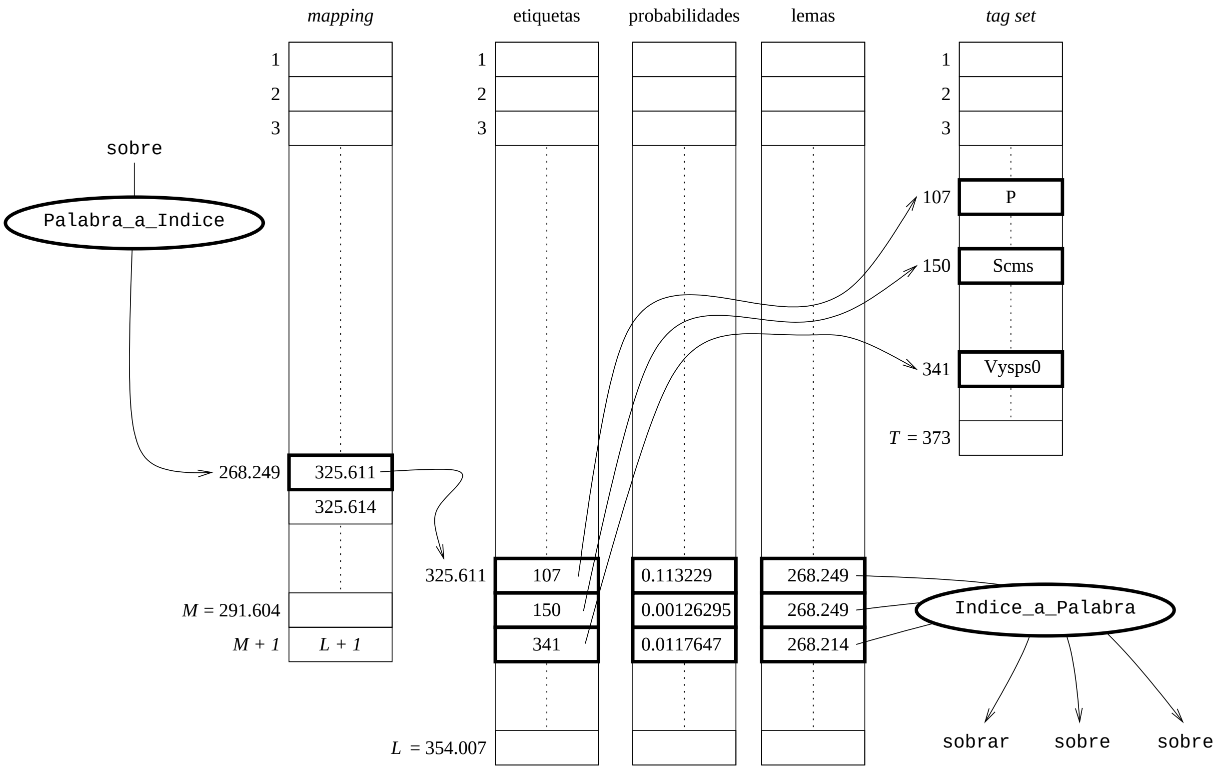

Figura 1: Modelización compacta de un diccionario

Por supuesto, ésta no es una versión operativa de un diccionario. Por tanto, lo importante ahora es proporcionar una versión compilada que compacte esta gran cantidad de datos, y que garantice un acceso eficiente con ayuda de los autómatas. La versión compilada se muestra en la figura 1, y sus elementos principales son:

- La función `Palabra_a_Indice` (que se explicará más tarde) transforma una palabra en su posición relativa dentro del diccionario (por ejemplo, `sobre` en 268.249).

- Mediante un tablero de *mapping* de tamaño $M + 1$, este número se transforma en la posición absoluta de la palabra (por ejemplo, 268.249 en 325.611).

- Este nuevo número es el que se utiliza para acceder a los tableros de *etiquetas*, *lemas* y *probabilidades*, todos ellos de tamaño L.

- El tablero de *etiquetas* almacena números, los cuales son más compactos que los nombres de las etiquetas. Cuando sea necesario, esos nombres se pueden recuperar a partir del tablero *tag set*, de tamaño T.

  El orden lexicográfico garantiza que las etiquetas de una palabra dada están contiguas, pero necesitamos saber cuántas hay. Para ello, es suficiente con restarle la posición absoluta de la palabra al valor de la siguiente celda (por ejemplo, $325.614 - 325.611 = 3$ etiquetas). Este procedimiento es válido también para acceder correctamente a los tableros de *lemas* y *probabilidades*.

- El tablero de *lemas* también almacena números. Un lema es una palabra que también debe estar en el diccionario. El número obtenido por la función `Palabra_a_Indice` para esta palabra es el número que se almacena aquí, ya que es más compacto que el lema en sí mismo. El lema original se puede recuperar mediante la función `Indice_a_Palabra` (que se explicará más tarde).

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

Esta es una de las arquitecturas más compactas que se puede diseñar para almacenar toda la información léxica de las palabras presentes en un diccionario, cuando esta información involucra características tan específicas de cada palabra como son sus probabilidades. Además, esta arquitectura es muy flexible: es fácil incorporar nuevos tableros para otros datos adicionales (como por ejemplo las frecuencias), o eliminar los que no se utilicen (ahorrando así su correspondiente espacio).

Para completar este modelo, sólo necesitamos la implementación de las funciones `Palabra_a_Indice` e `Indice_a_Palabra`. Ambas funciones operan sobre un tipo especial de autómatas: los autómatas finitos acíclicos mínimos numerados, que se describen en la siguiente sección.

3 Autómatas Finitos Acíclicos Mínimos Numerados

Un *autómata finito* se define mediante una 5-tupla $A = (Q, \Sigma, \delta, q_0, F)$, donde:

- Q es un conjunto finito de estados (los nodos del grafo subyacente),
- Σ es un alfabeto finito de los símbolos de entrada que forman las palabras (las etiquetas de las transiciones del grafo),
- δ es una función de $Q \times \Sigma$ en 2^Q que define las transiciones del autómata,
- q_0 es el estado inicial (la entrada del grafo),
- y F es el subconjunto de estados finales o de aceptación de Q (normalmente marcados con círculos más gruesos).

El estado o conjunto de estados alcanzados por la transición de etiqueta a del estado q se denota mediante $q.a = \delta(q, a)$. Cuando esto es siempre sólo un estado, es decir, cuando δ es una función de $Q \times \Sigma$ en Q, se dice que el autómata es *determinista*. Esta notación es transitiva: si w es una palabra, entonces $q.w$ denota el estado que se alcanza utilizando las transiciones etiquetadas con cada letra w_1, w_2, ... ,w_n de w. Una palabra w es aceptada por el autómata si $q_0.w$ está en F. Definimos también $\mathcal{L}(A)$, el *lenguaje reconocido* por el autómata A, como el conjunto de palabras w tales que $q_0.w \in F$. Un autómata es *acíclico* si su correspondiente grafo es acíclico.

Los autómatas acíclicos deterministas son quizás las estructura más compacta que existe para reconocer conjuntos finitos de palabras. Los niveles de compresión son excelentes, y los tiempos de reconocimiento son lineales con respecto a la longitud de la palabra que va a ser analizada. El resto de secciones de este trabajo presentan varios métodos para obtener el autómata acíclico determinista mínimo correspondiente a un conjunto finito de palabras dado.

Sin embargo, esto no es suficiente para nuestro modelo de diccionario. Necesitamos un mecanismo que transforme cada palabra en una clave numérica unívoca y viceversa. Esta transformación de puede realizar fácilmente si el autómata incorpora un *peso* para cada estado, siendo este peso el cardinal del *lenguaje derecho* del estado, es decir, el número de subpalabras que se aceptan a partir de ese estado [7]. Nos referiremos a estos autómatas como *autómatas finitos acíclicos deterministas mínimos numerados*.

La figura 2 muestra el autómata mínimo numerado que reconoce todas las formas de los verbos ingleses `discount`, `dismount`, `recount` y `remount`[3]. La asignación de los pesos de indexación se puede realizar mediante un simple recorrido recursivo del autómata, una vez que éste haya sido correctamente construido y minimizado.

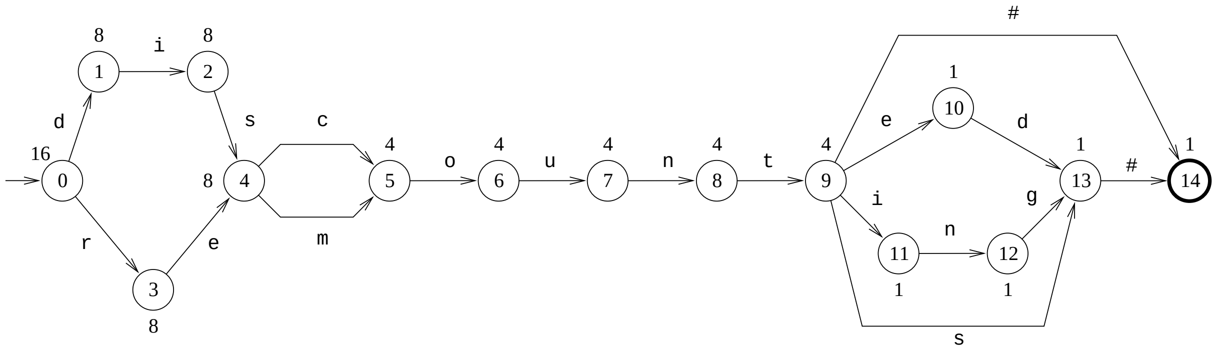

Figura 2: Autómata finito acíclico mínimo numerado para las formas de los verbos ingleses `discount`, `dismount`, `recount` y `remount`

Ahora ya podemos dar los detalles de las funciones que realizan el *hashing* o correspondencia perfecta entre las palabras del diccionario y los números 1 a M (el tamaño del diccionario). La función `Palabra_a_Indice`, que se muestra en la figura 4 del apéndice A, comienza a trabajar con un índice igual a 1 y navega sobre el autómata utilizando las letras de la palabra a analizar. En cada estado del camino recorrido, el índice es incrementado con el peso de indexación del estado destino de todas las transiciones lexicográficamente precedentes a la transición utilizada. Si se procesan todas las letras de la palabra y se alcanza un estado final, el índice contendrá la clave numérica de la palabra. En cualquier otro caso, la función devuelve un valor que indica que la palabra es desconocida.

La función `Indice_a_Palabra`, que se muestra en la figura 5 del apéndice A, comienza trabajar con el índice y realiza los pasos análogos a los de la función `Palabra_a_Indice` con el fin de deducir cuáles son las transiciones que producen ese índice, y obtiene las letras de la palabra buscada a partir de las etiquetas de esas transiciones.

En el autómata de la figura 2, la correspondencia individual de cada palabra es como se indica a continuación:

```
 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
```

Es importante destacar que M, en este caso 16, es el peso de indexación del estado inicial y corresponde al número total de palabras reconocidas por el autómata.

4 Minimización Basada en la Propiedad de la Altura

En esta sección empezamos el estudio de los métodos más eficientes para construir autómatas acíclicos mínimos. La primera estructura que podríamos considerar para implementar un analizador léxico para un conjunto finito de palabras es un *árbol de letras*, el cual es en sí mismo

[3]El símbolo `#` denota el fin de cadena.

un autómata donde el estado inicial es la raíz y los estados finales son las hojas. Sin embargo, los requerimientos de memoria de un árbol son excesivos para los diccionarios grandes[4]. Por tanto, debemos aplicar un proceso de minimización para reducir el número de estados y transiciones.

El proceso de minimización siempre puede ser realizado sobre cualquier autómata finito determinista, y el resultado es otro autómata equivalente, es decir, que reconoce el mismo lenguaje que el autómata original [6]. Además, si el autómata es acíclico, este proceso es más sencillo, tal y como veremos a lo largo del resto de este trabajo.

Por otra parte, y debido también a esos mismos requerimientos de memoria, no es conveniente construir un diccionario insertando todas las palabras en un árbol y después obteniendo el autómata mínimo correspondiente a ese árbol. En lugar de esto, es mucho más aconsejable realizar varias fases de inserción y minimización[5].

En cualquier caso, para introducir formalmente la base de los algoritmos tradicionales de minimización [8], necesitamos las siguientes definiciones. Dos autómatas son *equivalentes* si reconocen el mismo lenguaje. Dos estados p y q son *equivalentes* si el subautómata que tiene a p como estado inicial y el que empieza en q son equivalentes. El concepto opuesto es que dos estados son no equivalentes o *distinguibles.* Si A es un autómata, existe un único autómata $\mathcal{M}$ mínimo en el número de estados que reconoce el mismo lenguaje, es decir, tal que $\mathcal{L}(A) = \mathcal{L}(\mathcal{M})$. Un autómata que no tiene ningún par de estados equivalentes es mínimo. Ahora, para un estado s, definimos su *altura* $h(s) = max\,\{|w| \mid s.w \in F\}$, es decir, la altura del estado s es la longitud del camino más largo que comienza en s y termina en un estado final. Esta función proporciona una partición Π de Q. Π_i denota el conjunto de estados de altura i. Se dice que el conjunto Π_i es *distinguible* si ningún par de estados de Π_i es equivalente.

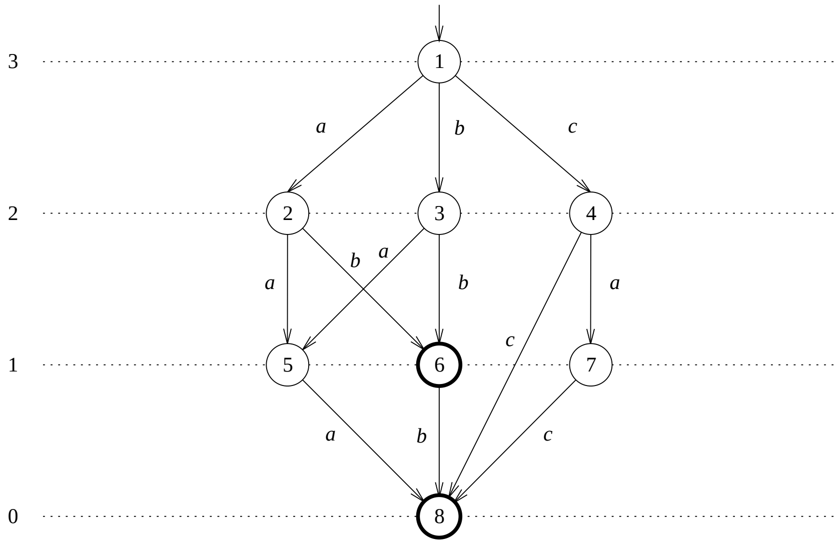

Figura 3: Un autómata finito determinista acíclico no mínimo

La figura 3 muestra un autómata que reconoce el lenguaje $\mathcal{L} = \{aaa,\ ab,\ abb,\ baa,\ bb,\ bbb,\ cac,\ cc\}$. Los estados de la misma altura aparecen dibujados sobre la misma línea punteada.

[4]El diccionario GALENA necesitaría un árbol con más de un millón de nodos (estados) para reconocer sus 291.604 palabras diferentes.

[5]En [4] se describe cómo introducir adecuadamente nuevas palabras en un autómata parcial ya minimizado, evitando las posibles inconsistencias que pueden surgir. La idea básica es clonar los estados conflicitivos que pueden dar lugar a inserciones no intencionadas de palabras no presentes en el diccionario original. Además, se proporciona también un razonamiento empírico sobre el tamaño máximo que debe tener el autómata para obtener un equilibrio razonable entre el número de fases de inserción-minimización y sus duraciones.

Este autómata no es mínimo. Los estados 2 y 3 de altura 2 son equivalentes. Podemos colapsar estos estados eliminando uno de ellos, por ejemplo el estado 2, y reemplazando el destino de sus transiciones entrantes por el otro estado, es decir, cambiando $1 \xrightarrow{a} 2$ por $1 \xrightarrow{a} 3$.

Ahora podemos establecer la *propiedad de la altura*: si cada Π_j con $j < i$ es distinguible, entonces dos estados p y q de Π_i son equivalentes si y sólo si para cada letra a de Σ la igualdad $p.a = q.a$ se verifica. El algoritmo de minimización de Revuz [8] se deduce a partir de la propiedad de la altura. Primeramente, crea una partición por alturas que se calcula mediante un recorrido recursivo sobre el autómata, tarea para la cual la complejidad temporal es $\mathcal{O}(t)$, donde t es el número de transiciones. Si el autómata no es un árbol, se pueden acelerar algunos cálculos mediante un indicador que señale que la altura de un estado ya ha sido calculada, y durante el recorrido se pueden también eliminar aquellos estados inútiles, que no tendrán ninguna altura asignada. Entonces, se procesa cada Π_i, desde $i = 0$ hasta la altura del estado inicial, ordenando los estados según sus transiciones, y colapsando los que resulten ser equivalentes.

Utilizando un esquema de ordenación con complejidad temporal $\mathcal{O}(f(e))$, donde e es el número de elementos a ordenar, el algoritmo de Revuz minimiza un autómata acíclico en

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

lo cual es menos que la complejidad del algoritmo de minimización de Hopcroft para autómatas finitos genéricos: $\mathcal{O}(n \times \log n)$, donde n es el numero de estados [5].

Este proceso necesita 10 fases de inserción-minimización para construir el autómata acíclico mínimo correspondiente al diccionario Galena (11.985 estados y 31.258 transiciones), y consume un tiempo de 29 segundos en un Pentium II a 300 MHz. bajo sistema operativo Linux.

5 Algoritmo de Construcción Incremental

Como hemos visto, los métodos tradicionales para construir autómatas acíclicos mínimos a partir de un conjunto finito de palabras constan de dos fases: la primera construye un árbol o un autómata parcial, y la segunda lo minimiza. Sin embargo, existen métodos de construcción incremental capaces de realizar la minimización en línea, es decir, al mismo tiempo que se insertan las palabras en el autómata [3]. Estos métodos son mucho más rápidos y consumen mucha menos memoria.

Para construir el autómata mínimo palabra por palabra, necesitamos combinar el proceso de añadir nuevas palabras con el proceso de minimización. Para ello, hay dos preguntas cruciales que debemos responder:

1. ¿Qué estados son susceptibles de cambiar cuando se añaden nuevas palabras?

2. ¿Existe alguna manera de añadir nuevas palabras de forma que minimicemos el número de estados que es necesario cambiar durante la inserción de una palabra?

Si los datos de entrada están lexicográficamente ordenados, al añadir una nueva palabra, sólo pueden cambiar los estados que necesitamos atravesar para aceptar la palabra añadida previamente. El resto del autómata permanece invariable, ya que, la nueva palabra:

- O bien comienza con un símbolo diferente al primer símbolo de todas las palabras que ya están en el autómata (en este caso, el símbolo de comienzo de la nueva palabra se sitúa lexicográficamente después de esos símbolos).

- O bien comparte algunos símbolos iniciales con la palabra añadida previamente (en este caso, el algoritmo localiza el último estado en el camino del prefijo común y crea una nueva rama para ese estado, ya que el símbolo de la etiqueta de la nueva transición debe estar situado lexicográficamente después de los otros símbolos de todas las otras transiciones salientes de 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, que se muestra en las fuguras 6 y 7 del apéndice A, 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.

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 *Ultimo_Estado*). Si *Ultimo_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 *Ultimo_Estado* una cadena de nuevos estados capaz de reconocer el *Sufijo_Actual*.

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.

6 Mejoras en el Acceso al Registro

Durante la construcción incremental, los estados del autómata están o bien en el registro, o bien en el camino de la última palabra añadida. Todos los estados del registro son estados que formarán parte del autómata final. Así pues, el autómata temporal correspondiente a cada paso de construcción tiene siempre menos estados que la suma de los que tendrá el automata final más la longitud de la palabra más larga. Como resultado de esto, la complejidad espacial es $\mathcal{O}(n)$, es decir, la cantidad de memoria que necesita el algoritmo es proporcional a n, el número de estados del autómata mínimo. Esta es una importante ventaja del algoritmo.

Respecto al tiempo de ejecución, el algoritmo presenta dos puntos críticos que aparecen marcados con un recuadro en la figura 7 del apéndice A. Esto significa que la complejidad temporal dependerá de la estructura de datos implementada para realizar las búsquedas de estados equivalentes y las inserciones de nuevos estados representantes en el registro. En [3], los autores sugieren que, utilizando una tabla *hash* para implementar el registro y sus relaciones de equivalencia, la complejidad temporal de esas operaciones es casi constante e igual a $\mathcal{O}(\log n)$.

Desafortunadamente, dicha estructura de *hashing* no aparece descrita, aunque podría ser deducida directamente desde la implementación C++ del algoritmo que los autores han puesto libremente disponible en `http://www.pg.gda.pl/~jandac/fsa.html`. Esta implementación consume 3,4 segundos para la construcción del autómata acíclico mínimo correspondiente al diccionario Galena (11.985 estodos y 31.258 transiciones) y 11,2 segundos para el del diccionario Erial (52.861 estados y 159.780 transiciones), en un Pentium II a 300 MHz. bajo Linux.

Aquí, en lugar de realizar un estudio de ese código, preferimos detallar nuestra propia implementación, ya que integra algunas características necesarias para la arquitectura general de diccionarios presentada en la sección 2, y además hemos verificado que es más rápida, tal y como veremos después.

Cuando un estado dado es susceptible de ser reemplazado o registrado, debe ser comparado con los estados que ya están en el registro. Por supuesto, no podemos compararlo con todos esos estados, ya que el registro es cada más y más grande a medida que se insertan nuevas palabras en el autómata. Así pues, debemos reflexionar de nuevo sobre lo siguiente: ¿cuándo dos estados son equivalentes? Y encontramos las siguientes respuestas para esta pregunta, donde cada una de ellas constituye un nuevo filtro que deja más y más estados fuera del proceso de comparación:

- Para que dos estados sean equivalentes, sus alturas tienen que ser iguales. La altura no es específicamente necesaria ni para el algoritmo incremental, ni para el esquema general del diccionarios, pero constituye no obstante un filtro muy efectivo.

  Además, la altura es un número relativamente bajo (que está en el rango que va desde 0 hasta la longitud de la palabra más larga), y puede ser calculado en línea sin ningún recorrido extra del autómata (la altura de un estado es la máxima altura de los estados destino de sus transiciones salientes más uno).

- Para que dos estados sean equivalentes, su número de transiciones salientes también tiene que ser igual para ambos. Este número es necesario para construir el autómata correctamente, y es también un número relativamente bajo (que está entre 1 y el tamaño del alfabeto utilizado).

- Para que dos estados sean equivalentes, sus pesos de indexación también tienen que ser iguales. El peso es necesario para el esquema general de diccionarios, por lo que es una buena idea calcularlo durante la construcción del autómata (esto es posible ya que el peso de un estado es la suma de los pesos de los estados destino de sus transiciones salientes).

  Por supuesto, el rango de posibles valores para el peso de un estado dado puede ser muy grande (entre 1 y el tamaño del diccionario), pero mediante comprobaciones empíricas hemos observado que los pesos más frecuentes son también números bajos.

Por tanto, nuestra implementación del registro es un *array* de tres dimensiones al que se accede mediante altura, número de transiciones salientes, y peso. Cada celda de este *array* contiene la lista de estados que comparten estas tres características[6]. Cuando un estado es susceptible

[6] Realmente, esto es así sólo para los estados con pesos entre 1 y 15, que son los pesos empíricamente más frecuentes. Los estados con pesos más grandes están almacenados en un conjunto de listas diferente. No obstante, las listas de este último conjunto están también ordenadas por peso.

de ser reemplazado o registrado, consideramos sus características y lo comparamos sólo con los estados de la lista correspondiente. Sólo entonces verificamos los símbolos de las etiquetas de las transiciones salientes y sus estados destino, que deben ser iguales si es que los estados son equivalentes.

Utilizando nuestra implementación del algoritmo incremental, el tiempo necesario para construir el autómata del diccionario Galena se reduce a 2,5 segundos. La incorporación de la información relativa a etiquetas, lemas y probabilidades consume 4,6 segundos, lo que da un tiempo total de compilación de 7,1 segundos. En el caso del diccionario Erial, los tiempos correspondientes son 9,2 + 15,6 = 24,8 segundos.

Finalmente, debemos destacar que la velocidad de reconocimiento de nuestros autómatas es de aproximadamente 80.000 palabras por segundo. Esta cifra supone también una mejora respecto a la obtenida por [3], que alcanza sólo alrededor de 35.000 palabras por segundo. La única explicación que encontramos para esta mejora es el haber encontrado una arquitectura interna para los autómatas que también es más eficiente.

7 Comportamiento Dinámico de los Autómatas Acíclicos

En el trabajo [3], los autores proponen también un método incremental para la construcción de autómatas finitos acíclicos deterministas mínimos a partir de conjuntos de palabras no ordenados. Este método se apoya 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.

Sin embargo, esa no es la única aplicación posible. El hecho de disponer de un algoritmo de esas características también hace viable la tarea de insertar nuevas palabras en un autómata que ya es mínimo, independientemente de la posición que ocuparían esas palabras dentro del orden lexicográfico de las que ya están en el autómata, y garantiza que sólo se verán afectadas las partes del autómata relacionadas con la nueva palabra. Así pues, una vez más, estamos ante un algoritmo cuya complejidad temporal es lineal respecto a la longitud de la palabra a insertar.

Si además consideramos la posibilidad de eliminar palabras que ya estén en el autómata, y si somos capaces de construir un algoritmo cuya complejidad temporal sea también lineal respecto a la longitud de la palabra a eliminar, habremos dotado a nuestros autómatas de un comportamiento dinámico que podría ser de gran utilidad en algunas aplicaciones.

Una de estas aplicaciones es, por ejemplo, el mecanismo de indexación de un sistema de recuperación de información. La gestión de este tipo de sistemas puede presentar situaciones como las siguientes:

- Cuando se incorpora un nuevo documento a la colección de documentos disponibles, y se detectan en él los términos que son relevantes, puede surgir la necesidad de indexar términos o palabras que son nuevos, es decir, que todavía no estaban introducidos en el índice.

- Y si por alguna razón es necesario eliminar algún documento de la colección, y si existe en el índice algún término que sólo era relevante para este documento, dicho término o palabra debe ser eliminado del índice.

Tradicionalmente, los índices de los sistemas de recuperación de indexación utilizan la tecnología que ofrecen los sistemas de bases de datos. En este trabajo no pretendemos realizar una comparación exhaustiva entre esos sistemas y nuestros autómatas acíclicos. No obstante, queremos señalar que los autómatas aportan algunas mejoras, y que en el futuro podrían sustituir a los sistemas de bases de datos en lo que se refiere a la realización de las tareas de indexación involucradas en los sistemas de recuperación de información. Entre estas mejoras, destaca por un lado el elevado grado de compactación que presentan los autómatas, lo que haría posible que todo el mecanismo de indexación residiera en la memoria principal del ordenador, algo que no está garantizado en un sistema de base de datos. Y por otra parte, los algoritmos encargados de las principales tareas de mantenimiento, básicamente la inserción y el borrado de palabras, pueden efectivamente implementarse de manera muy eficiente.

Como ya se ha indicado, la operación de inserción se describe en [3]. Respecto a la eliminación de palabras, un esbozo de las ideas principales se puede encontrar en [2] y [9]. A continuación presentamos una descripción más detallada. El algoritmo consta de tres fases:

1. *Fase de clonado.* Para eliminar una palabra es necesario aislar su camino en el autómata. Para hacer esto, transitamos con los caracteres de la palabra que se va a eliminar. Si al transitar llegamos a un estado confluencia, clonamos este estado y cambiamos la transición con la que llegamos al estado, para que se dirija ahora hacia el nuevo estado clon. Continuamos este proceso de aislamiento siguiendo por los estados que se van clonando hasta llegar al estado final (evitando clonarlo aunque éste sea un estado confluencia).

2. *Fase de borrado.* Una vez terminada la fase de clonación, en el camino que hemos aislado hay que eliminar las transiciones y los estados desde el estado final hasta encontrar un estado fuente, es decir, un estado con más de una transición saliente. Tanto el estado final como ese estado fuente no son eliminados del autómata, y solamente se eliminan los estados y las transiciones que hay entre ellos.

3. *Fase de reemplazo o registro.* A partir de este estado fuente hacia atrás, hay que ir reemplazando o registrando los estados clones que se han ido creando en la primera fase hasta que llegamos a un estado original del autómata, es decir, un estado que existía antes de comenzar el borrado de la palabra en cuestión, y que por lo tanto no ha sido creado durante la fase de clonación. Una vez ahí, se debe ir eliminando del registro y reemplazando o registrando nuevamente los estados hacia atrás hasta que alguno de ellos no sea reemplazado (sea registrado) o se llegue al estado inicial.

Estas fases recopilan toda la posible casuística que puede surgir, con el fin de minimizar el número de operaciones de registro o reemplazo, las cuales, como hemos visto, constituyen el punto crítico de este tipo de algoritmos. Además de esto, nuestra implementación de ambas operaciones, inserción y borrado, considera también la necesidad de actualizar los pesos de indexación para que el *hashing* perfecto siga funcionando correctamente.

8 Conclusiones

A través de un profundo estudio de los diferentes métodos existentes para la construcción de autómatas finitos acíclicos, hemos presentado varias aportaciones que hacen más viable el manejo de grandes conjuntos de palabras en el dominio del PLN. La primera ha sido el diseño de una arquitectura general para los diccionarios, que es capaz de almacenar la gran cantidad de información léxica relacionada con las palabras. Hemos demostrado que es una de las representaciones más compactas que se pueden diseñar cuando se necesita manejar información tan específica de esas palabras como son sus probabilidades, lo cual es particularmente útil para las aplicaciones de PLN estocástico.

De manera natural, la segunda contribución completa nuestro modelo de diccionario mediante la mejora de los métodos incrementales para la construcción de autómatas mínimos. En las construcciones incrementales, dado que las partes del diccionario que ya están construidas (es decir, los estados del *registro*) no son susceptibles de cambiar en el futuro, podemos utilizar en paralelo otras características específicas de los estados. Esas características están inspiradas unas veces en los mecanismos de trabajo de nuestra arquitectura de diccionarios (por ejemplo, los *pesos* de indexación), y otras veces en la base de otros algoritmos (por ejemplo, las *alturas*). Todas ellas permiten mejorar el acceso a las partes registradas y verificar de forma muy rápida las equivalencias con los nuevos estados. En consecuencia, el tiempo de construcción total de estos autómatas mínimos es menor que el de los algoritmos previos.

Por último, hemos visto que también existen algoritmos muy eficientes para las tareas básicas de mantenimiento de los autómatas (inserción y eliminación de palabras). Esto dota a los autómatas acíclicos de un carácter dinámico que puede ser de gran utilidad en multitud de aplicaciones, como es el mecanismo de indexación de un sistema de recuperación de información.

Referencias

[1] Aho, A.V.; Sethi, R.; Ullman, J.D. (1985). Compilers: principles, techniques and tools. *Addison-Wesley*, Reading, MA.

[2] Carrasco, R.C.; Forcada, M.L. (2002). Incremental Construction and Maintenance of Minimal Finite-State Automata. *Computational Linguistics*, vol. 28(2), pp. 207-216.

[3] Daciuk, J.; Mihov, S.; Watson, B.W.; Watson, R.E. (2000). Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics*, vol. 26(1), pp. 3-16.

[4] Graña Gil, J. (2000). Robust parsing techniques for natural language tagging (in Spanish). *PhD. Thesis*, Departamento de Computación, Universidad de La Coruña (Spain).

[5] Hopcroft, J.E. (1971). An $n \log n$ algorithm for minimizing the states in a finite automaton. *The Theory of Machines and Computations*, Academic Press, NY, pp. 189-196.

[6] Hopcroft, J.E.; Ullman, J.D. (1979). Introduction to automata theory, languages and computations. *Addison-Wesley*, Reading, MA.

[7] Lucchesi, C.L.; Kowaltowski, T. (1993). Applications of finite automata representing large vocabularies. *Software - Practice and Experience*, vol. 23(1), pp. 15-30.

[8] Revuz, D. (1992). Minimization of acyclic deterministic automata in linear time. *Theoretical Computer Science*, vol. 92(1), pp. 181-189.

[9] Revuz, D. (2000). Dynamic acyclic minimal automaton. In *Proc. of CIAA-2000*, pp. 226-232. London, Ontario, Canada.

[10] Watson, B.W. (1993). A taxonomy of finite automata construction algorithms. Computing Science Note 93/43, Eindhoven University of Technology (The Netherlands).

A Pseudo-Código de los Principales Algoritmos

En este apéndice se incluyen los detalles de los principales algoritmos que han sido citados en este trabajo.

```
function Palabra_a_Indice (Palabra) =
  begin
    Indice ← 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
              Indice ← Indice + Estado_Actual[c].Número;
          Estado_Actual ← Estado_Actual[Palabra[i]];
        end
      else
        return palabra desconocida;
    if (Es_Estado_Final (Estado_Actual)) then return Indice
    else return palabra desconocida
  end;
```

Figura 4: Pseudo-código de la función `Palabra_a_Indice`

```
function Indice_a_Palabra (Indice) =
  begin
    Estado_Actual ← Estado_Inicial;   Número ← Indice;
    Palabra ← Palabra_Vacía;   i ← 1;
    repeat
      for c ← Primera_Letra to Ultima_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;
```

Figura 5: Pseudo-código de la función `Indice_a_Palabra`

function *Construcción_Incremental* (*Diccionario*) =
  begin
    Registro ← ∅;
    while (*hay otra palabra en Diccionario*) **do**
      begin
        Palabra ← *siguiente palabra de Diccionario en orden lexicográfico*;
        Prefijo_Común ← *Prefijo_Común* (*Palabra*);
        Ultimo_Estado ← q_0.*Prefijo_Común*;
        Sufijo_Actual ← *Palabra*[(*Longitud* (*Prefijo_Común*) + 1) ... *Longitud* (*Palabra*)];
        if (*Tiene_Hijos* (*Ultimo_Estado*)) **then**
          Registro ← *Reemplazar_o_Registrar* (*Ultimo_Estado*, *Registro*);
        Añadir_Sufijo (*Ultimo_Estado*, *Sufijo_Actual*);
      end;
    Registro ← *Reemplazar_o_Registrar* (q_0, *Registro*);
    return *Registro*
  end;

Figura 6: Pseudo-código de la función `Construcción_Incremental`

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

Figura 7: Pseudo-código de la función `Reemplazar_o_Registrar`