# Estimación eficiente de orden en modelos Markovianos

Instituto de computación
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

7 de Febrero de 2014

Proyecto de grado.

**Luciana Vitale**
Tutor: **Dr. Álvaro Martín**

**Resumen**

En este proyecto construimos un estimador eficiente de orden de procesos de Markov de máxima verosimilitud penalizada. Es frecuente el uso de árboles de sufijos en la implementación de este tipo de estimadores. Los mismos se ven limitados por el consumo de memoria cuando crece el tamaño de la entrada. Investigaciones recientes obtuvieron una cota para el orden estimado, reduciendo las opciones posibles que debe tomar en cuenta el estimador. Haciendo uso de esta cota, propusimos usar la estructura de árboles de sufijos truncados en la implementación, incorporando la ganancia de espacio que representan frente a los árboles de sufijos. Utilizamos para su construcción una adaptación del algoritmo de Ukkonen y la representación de Kurtz.

Usualmente los árboles de sufijos truncados se etiquetan mediante punteros a la cadena original. En este trabajo desarrollamos un mecanismo de etiquetado de aristas que reemplaza la cadena original por una cadena, más corta, formada a partir de ella. Los ahorros de memoria que aporta son equivalentes a la diferencia de tamaño entre el árbol de sufijos y el árbol de sufijos truncado. Es posible construirla junto con el árbol sin comprometer el tiempo de procesamiento. Evaluamos el consumo de memoria de este mecanismo sobre un conjunto de imágenes y obtuvimos diferencias significativas en comparación con el etiquetado clásico.

# Agradecimientos

Quiero agradecer sinceramente a todos los que formaron parte de este proceso.

A mi tutor, Álvaro Martín, por su comprensión, apoyo y dedicación. Siempre con una visión interesante y una pregunta justa ha guiado mi trabajo. Hoy se ha convertido en un referente para mí. A Gabriel Illanes, Sebastián Gedanke y Gadiel Seroussi por por su lectura y sus aportes valiosos.

A Geju quien me acompañó en este y todos los caminos que he elegido. Con su cariño ha sido mi fortaleza. A mi madre, mi hermana y mi padre que supieron motivarme y entusiasmarse con mis logros a lo largo de toda mi vida. A Leo y a Lu que estuvieron cerca en todo este proceso. A mi familia, mi familia elegida y mis amigos por su contención y aliento. A mis compañeros de estudio de matemática y de ingeniería que hicieron linda a esta carrera.

Esto no hubiera sido posible sin todos ellos. Muchas gracias.

# Índice general

# Lista de algoritmos

# Capítulo 1

# Introducción

En este trabajo estudiamos estimadores de orden de procesos de Markov de tiempo discreto, de tipo máxima verosimilitud penalizada. Este tipo de estimadores, dada una cadena $x$ sobre un alfabeto finito $\mathcal{A}$, determina cuál es el orden del modelo de Markov que mejor la modelaría. A cada posible orden $k$ le asocia un costo que depende de la probabilidad de máxima verosimilitud (de $x$ como realización de un modelo de Markov de orden $k$), $\hat{P}_k(x)$ y de un término de penalización que es creciente con $k$. Específicamente, el costo asociado al orden $k$ para la cadena $x$ de largo $n$ es

$$\mathrm{C}_k(x) = -\log \hat{P}_k(x) + f(n)\alpha^k \,, \tag{1.1}$$

donde $\alpha$ es la cantidad de letras en el alfabeto $\mathcal{A}$ y $f(n)$ es una función de penalización que satisface las siguientes condiciones: es positiva, no decreciente, $f(n) \to \infty$ y $\frac{f(n)}{n} \to 0$. El orden estimado resulta de comparar estos costos para los distintos valores posibles de $k$:

$$\hat{k}(x) = \arg\min_{k \geq 0} \mathrm{C}_k(x) \,. \tag{1.2}$$

En este proyecto nos proponemos construir una implementación eficiente de dichos estimadores. Es decir que a partir de una secuencia $x$ y fijada la función de penalización, calcule eficientemente el orden estimado $\hat{k}(x)$.

Obtener $\hat{k}(x)$ requiere calcular $\hat{P}_k(x)$ para distintos valores de $k$, y para esto es necesario conocer información estadística sobre la ocurrencia de patrones en $x$. Los árboles de sufijos[1] presentados por primera vez en [3] son una estructura de datos que permite contar patrones en $x$ de manera

[1] En este tipo de árboles, las aristas están etiquetadas por cadenas y cada hoja representa un sufijo de $x$ (al que resulta de concatenar las etiquetas de las aristas que forman el camino desde la raíz hasta él). Los prefijos comunes se representan mediante antepasados comunes, si dos sufijos tienen un prefijo común, sus hojas correspondientes descenderán de un mismo nodo que corresponde a dicho prefijo.

eficiente y constituyen una herramienta usualmente utilizada para abordar este tipo de problemas (ver [1] y [2]).

Existen algoritmos ([3], [4], [5])capaces de construir un árbol de sufijos consumiendo tiempo lineal y requiriendo un consumo de memoria del orden de $O(n \log n)$ bits. Sin embargo, los árboles de sufijos adolecen de una limitante en la práctica: para cadenas grandes, el uso de memoria puede convertirse en un cuello de botella. Esto ha motivado incursiones en la reducción de espacio utilizado ([6]).

En [7] se describe una cota superior para $\hat{k}(x)$

$$\frac{n \log \alpha}{f(n)} \geq \alpha^{\hat{k}} - 1, \tag{1.3}$$

que a su vez resulta ser óptima, como se demuestra en [8]. La existencia de la cota (1.3) asegura que solamente es necesario contar patrones de largo $k_n+1$, siendo $k_n$ el mayor de los valores posibles para $\hat{k}$, como veremos luego con mayor detenimiento. Esto permite reemplazar a los árboles de sufijos por árboles de sufijos truncados [10], que son una modificación de los primeros, donde solamente se consideran patrones hasta cierto largo, introduciendo un ahorro de memoria que en la práctica puede ser sustantivo. Esta posibilidad (la de incorporar los árboles de sufijos truncados a partir de la existencia de (1.3)) fue la motivación inicial que originó el trabajo aquí presentado.

A lo largo de este documento veremos los distintos elementos que combinamos para construir el estimador. En el capítulo 2 definimos el concepto de árboles de sufijos truncados, vemos en qué sentido son apropiados para recolectar información estadística sobre los patrones y cómo es posible estimar el órden de Markov una vez que contamos con el árbol de sufijos truncado correpondiente a la cadena original $x$. En el capítulo 3 estudiamos el algoritmo $U$, una adaptación del algoritmo de Ukkonnen para construir árboles de sufijos truncados que elegimos como punto de partida para nuestra implementación. Luego profundizamos en los aspectos claves para entenderlo; introducimos el algoritmo `construirTST` (y su procedimiento auxiliar `agregarFactor`) donde brindamos mayor nivel de detalle, permitiendo abarcar su complejidad siendo más claros. Tradicionalmente, la representación de las etiquetas de las aristas de los árboles de sufijos truncados se realiza en base a referencias a la cadena original $x$, alcanzando los órdenes de consumo de memoria $O(n \log n)$. En este trabajo incorporamos un nuevo mecanismo de etiquetado de aristas que describimos en el capítulo 4. Este resultado, que fue presentado en [9], es sin duda la contribución más importante de este trabajo, ya que aplica a un tipo de estructura de datos general y, por lo tanto, trasciende al objetivo concreto de estimar el orden de un proceso de Markov. También en el capítulo 4 explicamos cómo adaptamos la representación de Kurtz (originalmente propuesta para árboles de sufijos) y exponemos los detalles concretos de la implementación que consideramos

relevantes. Finalmente en el capítulo 5 exponemos los resultados experimentales que obtuvimos al evaluar el consumo de memoria, donde observamos que el mecanismo de etiquetado que desarrollamos introduce una ganancia significativa en los casos estudiados.

# Capítulo 2

# Estimación del orden de Markov a partir de árboles de sufijo truncados

Los árboles de sufijos truncados son árboles donde los distintos factores (noción que definiremos en breve) de una cadena $x$ corresponden a caminos que parten desde la raíz, y que por la forma de disponer la información tienen características deseables a la hora de contar patrones. Hemos anunciado que los usaremos para la construcción del estimador. Entender qué son y cómo es posible estimar el orden de Markov asociado a $x$ a partir de su árbol de sufijos truncado, es el objetivo de este capítulo. Es así que luego de presentar con mayor rigor los conceptos relacionados a los árboles de sufijos truncados, introducimos el algoritmo 1 que estima el órden de Markov.

Recordemos que $\mathcal{A}$ es un alfabeto finito de cardinal $\alpha$. Llamamos letras a sus elementos. Sea $\mathcal{A}^n$ el conjunto de las cadenas de largo $n$ sobre él y $\mathcal{A}^*$ el de las cadenas finitas (también llamadas palabras o secuencias). El símbolo $\lambda$ indica la cadena vacía y $\mathcal{A}^+$ es el conjunto de cadenas finitas no vacías. La yuxtaposición simboliza concatenación; es decir si $x, u, v, w$ pertenecen a $\mathcal{A}^*$, posiblemente vacíos, y $x$ es el resultado de concatenar $u, v$ y $w$ usamos la notación $x = uvw$. Además en este caso decimos que $u$ es prefijo de $x$, $v$ es un factor, $w$ es sufijo y que $v$ ocurre en la posición $|u| + 1$ de $x$, donde $|u|$ es el largo de $u$. Un prefijo, factor o sufijo de $x$ es propio si es distinto de $x$. La notación $u \prec x$ indica que $u$ es prefijo propio de $x$, $u \preceq x$ indica que es prefijo y en ese caso $x \backslash u$ es el sufijo de $x$ que ocurre luego de $u$. Para indicar la $i-$ésima letra de una cadena $x \in \mathcal{A}^*$ usamos $x_i$ y para indicar el factor $x_i \dots x_j$ de $x$ usamos $x_i^j$, con la convención de no exceder los bordes, es decir

$$x_i^j = x_{\text{máx}\{1,i\}} \dots x_{\text{mín}\{j,|x|\}}.$$

Sea $T$ un árbol dirigido cuyas aristas están etiquetadas por cadenas de

$\mathcal{A}^+$. Decimos que $T$ es un $\mathcal{A}^+-$árbol si cumple con las siguientes dos condiciones:

1. Las aristas de los nodos internos hacia sus hijos comienzan con distintas letras. Es decir, ningún padre tiene dos aristas hacia sus hijos cuyas etiquetas comiencen con la misma letra.

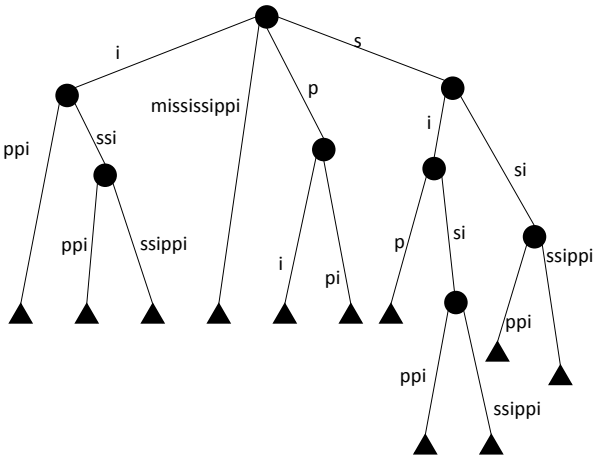
2. Todo nodo interno tiene al menos dos hijos.

Estas propiedades aseguran que la concatenación de las etiquetas de las aristas de los caminos desde la raíz a los nodos del árbol son únicas. De esta forma podemos identificar a cada nodo con la concatenación de las etiquetas de las aristas que conforman el camino desde la raíz hasta él (identificamos la raíz con la cadena vacía $\lambda$), permitiéndonos utilizar la notación $v \in T$ para referirnos a un nodo de $T$, donde $v \in \mathcal{A}^*$ es el resultado de concatenar dichas etiquetas. Además decimos que $u$ ocurre en $T$ si $u$ es un prefijo de un nodo $v \in T$. Denotamos a la arista que une dos nodos $v$ y $w$, padre e hijo respectivamente, mediante $v \to w$.

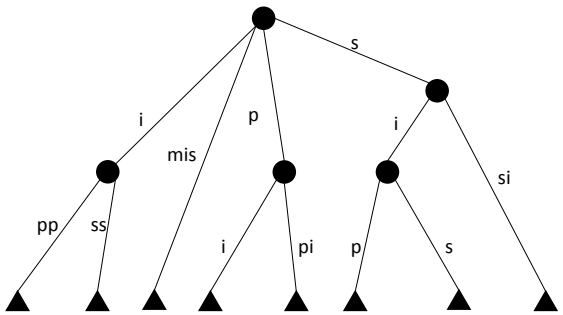
Un árbol de sufijos [4] de una cadena $x$ es un $\mathcal{A}^+$-árbol tal que una cadena $w$ ocurre en $T$ si y solamente si $w$ es un factor de $x$. En la figura 2.1(a) presentamos un ejemplo: de árbol de sufijos de la cadena **mississipi**. Sea un nodo $v \in T$, $|v|$ es su profundidad. Observemos que esta definición no coincide con la definición usual de profundidad en árboles, sino que está relacionado con el largo de las etiquetas de las aristas. El árbol de sufijos truncado de $x$ a profundidad $k$ es simplemente el resultado de truncar el árbol de sufijos a profundidad $k$. Observemos que todo factor de $x$ de largo $k$ o menor ocurre en su árbol de sufijos truncado y viceversa. Usamos $TST_k(x)$ para denotar el árbol de sufijos truncado a profundidad $k$ de $x$ y $ST(x)$ para el árbol de sufijos. Si se deduce del contexto podremos omitir tanto $x$ como $k$, haciendo más concisa la notación. En la figura 2.1 podemos comparar el árbol de sufijos truncado a profundidad 3 de **mississippi**, $TST_3$(**mississippi**), con $ST$(**mississippi**).

Volviendo a los estimadores, modelamos la secuencia $x$ de largo $n$ como la realización de un proceso de Markov de orden $k$ desconocido, $k \leq k_n$, siendo $k_n$ la cota superior para $k$ enunciada en (1.3). Vemos a los elementos de $\mathcal{A}^k$ como estados del proceso de Markov. Para $s \in \mathcal{A}^k$ y $a \in \mathcal{A}$ decimos que $s$ ocurre en $x$ y que emite a $a$ si $sa$ es un factor de $x$. Para determinar los estados iniciales usamos la convención de anteponer a la cadena $x$ una cadena inicial $z$ de largo $k_n$, $z = x_{-k_n+1}^0$. Sea $n_s$ la cantidad de veces que ocurre $s$ como estado en $x$ y $n_s^a$ la cantidad de veces que emite al símbolo $a$. Claramente si $s = x_{i-k}^{i-1}$ y $a = x_i$, se cumple

$$\hat{P}_k(x_i|x_{i-k}\dots x_{i-1}) = \frac{n_s^a}{n_s}, \text{ y } \hat{P}_k(x^n) = \prod_{i=1}^{n} \hat{P}(x_i|x_{i-k}\dots x_{i-1}).$$




(a) El árbol de sufijos de **misssissippi**




(b) $TST_3$(**mississippi**)

Figura 2.1: Comparación entre árbol de sufijos y árbol de sufijos truncado de **mississippi**.

Agrupando en los estados de largo $k$ junto con los símbolos emitidos, podemos expresar la componente estadística de (1.1) como sigue

$$-\log \hat{P}_k(x^n) = -\sum_{s\in\mathcal{A}^k, a\in\mathcal{A}} n_s^a \log \frac{n_s^a}{n_s}\,. \tag{2.1}$$

Como se cumple que $n_s = \sum_{a\in\mathcal{A}} n_s^a$ y, salvo potencialmente en condiciones de borde [1], $n_s^a = n_{sa}$, para calcular $-\log \hat{P}_k(x^n)$ para los distintos valores de $k$ es suficiente conocer $n_{sa}$ para todos los $sa \in \mathcal{A}^{k+1}$ que ocurren en $x$. En otras palabras alcanza con conocer cuantas veces ocurre cada $(k+1)-$factor, siendo estos los factores de largo $k+1$.

Una característica de los árboles de sufijos trucados es que para todo nodo interno del árbol, la cantidad de ocurrencias de éste como factor es igual a la suma de la cantidad de ocurrencias de todos sus hijos. Esta propiedad, a la que llamaremos *de agrupación*, permite contabilizar a la vez todas las ocurrencias de los factores de la cadena en una recorrida del árbol, y por lo tanto calcular simultáneamente la probabilidad de máxima verosimilitud de todos los órdenes menores a la profundidad del árbol.

El centinela, \$, es un símbolo que no pertenece a $\mathcal{A}$ cuyo propósito es forzar en el árbol de sufijos truncado la creación de una hoja para cada sufijo de largo menor o igual a $k$; más precisamente $TST_k(x\$)$ contiene una hoja por cada sufijo de $x$ de largo $k$ o menor. Esto no necesariamente ocurre si no se agrega el símbolo centinela al final de $x$. Por ejemplo, en la Figura 2.1(b), el sufijo **i** de **mississippi** no es una hoja de $TST_3$(**mississippi**). Sea $T = TST_{k_n+1}(zx\$)$ y $v \in T$. Definimos $m_v$ y $\overline{m_v}$ como la cantidad de ocurrencias de $v$ como factor de $zx\$$ y $z$ respectivamente. De estas definiciones se deduce inmediatamente el siguiente lema.

**Lema 2.1.** *[9] Sea $s \in \mathcal{A}^*$, $|s| \le k_n$ y $a \in \mathcal{A}$ tales que $sa$ ocurre en $zx\$$. Entonces existe una única arista $u \to v$ en $T$ tal que $u \prec sa \preceq v$ y se cumple $n_s^a = m_v - \overline{m_v}$. Además se verifica $n_{sa} = n_s^a - \delta$ siendo $\delta = 1$ si $sa = v$ y además $v$ tiene como hijo una hoja cuya arista está etiquetada con \$ y $\delta = 0$ en otro caso.*

Una vez que tenemos el TST (anotado con la cantidad de ocurrencias en cada hoja) es relativamente simple obtener el orden buscado. A continuación mostramos el algoritmo 1 `TSTestim`, tomado de [9], donde se calcula el orden de Markov a partir de $T = TST_{k_n+1}(zx\$)$. El mismo calcula los sumandos de la siguiente igualdad obtenida a partir de (2.1) para distintos valores de $k$

$$-\log \hat{P}_k(x^n) = \sum_{s\in\mathcal{A}^k} n_s \log n_s \; - \sum_{s\in\mathcal{A}^k, a\in\mathcal{A}} n_s^a \log n_s^a\,. \tag{2.2}$$

[1] Estas condiciones se definen formalmente en el Lema 2.1 más abajo.

Los vectores `log_ocurrencias` y `log_emisiones` representan el sumando izquierdo y derecho de (2.2) respectivamente. Aquellos estados que emiten un solo símbolo hacen el mismo aporte al sumando izquierdo y al derecho y por eso no son tenidos en cuenta en el algortimo. Estos estados corresponden a los factores $s$ que están en medio de una arista o, puesto en términos del lema 2.1, $u \prec s$ y $sa \prec v$.

**Algoritmo 1** `TSTEstim` Estimación del orden de Markov a partir de un TST

Entradas: $T = TST_{k_n}(zx\$)$ equipado con los contadores $m_v$ y $\overline{m_v}$ para todo nodo $v$.
Salida: El orden estimado $\hat{k}$.

1: Inicializar $\texttt{log\_ocurrencias}[0] = n$, y $\texttt{log\_ocurrencias}[k] = 0$ para todo $k$, $0 < k \leq k_n$.
2: Inicializar $\texttt{log\_emisiones}[k] = 0$ para todo $k$, $0 \leq k \leq k_n$.
3: **for all** arista $u \to v$ de $T$, **do**
4: Si $|v| \leq k_n$, sumar $(m_v - \overline{m_v} - \delta)\log(m_v - \overline{m_v} - \delta)$ a $\texttt{log\_ocurrencias}\Big[|v|\Big]$, siendo $\delta$ como se definió en el lema 2.1.
5: Sumar $(m_v - \overline{m_v})\log(m_v - \overline{m_v})$ a $\texttt{log\_emisiones}\Big[|u|\Big]$.
6: **end for**
7: Sea $\hat{k} = \arg\min\Big\{\texttt{log\_ocurrencias}[k] - \texttt{log\_emisiones}[k] + f(n)|\mathcal{A}|^k : 0 \leq k \leq k_n\Big\}$.

En la línea 3 del algortimo es donde finalmente encontramos cómo las ocurrencias de factores se traducen a probabilidades empíricas. Observemos que puede hacerse en una recorrida del árbol en profundidad, ya que la cantidad de ocurrencias de los hijos $m_v$ determina la cantidad de ocurrencias de su padre $m_u$. Además si llevamos cuenta de las hojas que corresponden a cada índice de $z$ es sencillo calcular $\overline{m_v}$.

A partir del algoritmo recién presentado, vemos que el problema de estimar eficientemente el órden de Markov se reduce a construir eficientemente TST. En los próximos capítulos veremos los algoritmos y estructuras fundamentales que utilizamos a tales efectos.

# Capítulo 3

# Construcción de árboles de sufijos truncados

## 3.1. El algoritmo de Ukkonen para árboles truncados

Los algoritmos de Mc Creight [4] y Ukkonen[5] son algoritmos de construcción de árboles de sufijos. Fueron adaptados en [10] para construir árboles de sufijos truncados. En este trabajo hemos elegido la adaptación del algoritmo de Ukkonen, que llamaremos algoritmo $U$, porque tiene la virtud de construir el árbol de manera incremental. Es decir, la construcción de $TST_k(x^n)$ se realiza en *fases*; en cada fase $f$ se obtiene el árbol $TST_k(x_1^f)$ mediante modificaciones sobre el árbol obtenido en la fase anterior, $TST_k(x_1^{f-1})$.

Así como en el algoritmo de Ukkonen, el algortimo $U$ construye junto con el árbol la estructura de *suffix links*. Definimos para todo nodo interno del árbol $v \in TST_k(x)$ a su *suffix link*, denotado con $sLink(v)$, al nodo interno $u$ tal que $v = au$, donde $a$ es la primera letra de $v$. Inicialmente puede no ser claro por qué $u$ debe existir. Que $v$ sea un nodo interno implica que existen $a, b \in \mathcal{A}$, $a \neq b$, tales que $va$ y $vb$ son factores de $x$ de largo $k$ o menor. Por lo tanto $ua$ y $ub$ también lo son y en consecuencia $u$ es un nodo interno de $TST_k(x)$. En la figura 3.1 mostramos el árbol de sufijos truncado a profundidad 3 para la cadena **mississippi**; los *suffix links* se representan con flechas punteadas. Estos permiten la navegación rápida dentro del árbol durante la construcción del $TST$ y contribuyen a que sea posible alcanzar un tiempo de ejecución de orden lineal[5][10].

Un $k-$factor de $x$ es un factor de largo $k$ o un sufijo de largo menor que $k$. En cada fase $f$ se busca insertar en $TST_k(x_1^{f-1})$ los $k-$factores de $x_1^f$ en el orden en que ocurren. Estudiemos qué sucede al intentar agregar al árbol el $k-$factor $v$ de $x_1^f$ que ocurre en la posición $p$. Primero observemos

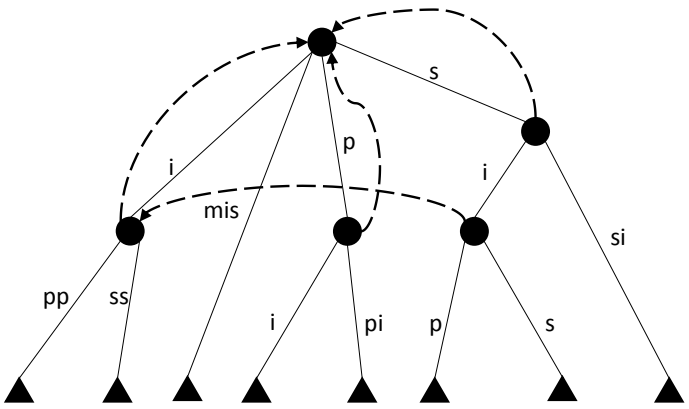



Figura 3.1: $TST_3$(**mississippi**) con *suffix links*

que si $p \leq f - k$, $v$ ocurre en $x_1^{f-1}$ y por lo tanto no es necesario modificar el árbol. Nos concentramos entonces únicamente en el caso $p > f - k$, es decir, cuando $v = x_p^f$ es sufijo de $x_1^f$ de largo $k$ o menor (incluyendo el caso $v = \lambda$). Sea $u$ el prefijo propio más largo de $v$ cuando tiene sentido ($|v| > 0$). Observamos que $u$ es un sufijo de $x_p^{f-1}$. Solamente una de las siguientes opciones es verdadera:

1. $v$ no ocurre en $x_1^{f-1}$, $v \neq \lambda$, y $u$ solamente ocurre una vez (necesariamente como sufijo de $x_1^{f-1}$). Por lo tanto el camino de $u$ en el árbol $TST_k(x_1^{f-1})$ lleva a una hoja. En este caso sólo es necesario agregar una letra a la etiqueta de la arista que lleva a la hoja.

2. $v$ no ocurre en $x_1^{f-1}$, $v \neq \lambda$, y $u$ ocurre varias veces. En este caso hay que agregar una hoja al árbol y, si todavía no existe un nodo interno para $u$ ($u$ terminaba en medio de una arista), también un nodo interno.

3. $v = \lambda$ o $v$ ocurre en $x_1^{f-1}$. En este caso no es necesario modificar el árbol.

Analicemos más en detalle las opciones anteriores. Si $v$ estuviera en la situación (1) y $w$ fuera un $k-$factor en la posición $q$, $f-k < q < p$, ($v$ es un sufijo de $w$ que lo es a su vez de $x_1^f$) entonces $w$ también estaría en la situación (1). Por otro lado si $v$ estuviera en la situación (3) y $\overline{w}$ fuera un $k-$factor en la posición $\overline{q}$, $p < \overline{q}$, ($w$ es un sufijo de $v$) entonces $\overline{w}$ también estaría en la situación (3). Dicho en otras palabras, los $k-$factores en la situación (1) ocurren todos en posiciones contiguas, seguidos por todos los que están en

la situación (2) y estos a su vez seguidos por los que están en la situación (3).

El algoritmo $U$ propone un mecanismo, explicado más adelante, que prolonga automáticamente las etiquetas de las aristas que llevan a hojas de $TST_{k-1}(x_1^{f-1})$ de profundidad menor que $k$. Cuando estamos frente a una situación de este tipo, decimos que ocurre una inserción implícita. Las hojas que son insertadas de forma implícita, corresponden a los sufijos en la situación (1); lo llamaremos *grupo de prolongación*. El segundo grupo es el de los sufijos que son insertados explícitamente: para que el árbol contenga a estos sufijos es necesario modificarlo más allá de la prolongación de las hojas; es el *grupo activo*. El tercer grupo es de los sufijos contenidos, es decir el de sufijos que ya ocurrían en el árbol como prefijos de alguna rama; lo llamamos *grupo resuelto*.

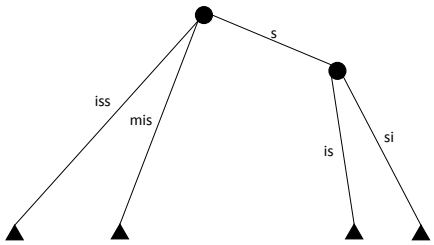
El grupo resuelto tiene al menos al sufijo $\lambda$. La primera posición de los sufijos resueltos, es decir el menor $p$ tal que $x_p^f$ ocurre en $x^{f-1}$, será llamada posición final. Definimos posición inicial como la posición del primer sufijo que no pertenece al grupo de prolongación. Las posiciones inicial y final delimitan el grupo activo (posiblemente vacío) que es el foco principal del algoritmo $U$(ver algoritmo 2). Como no hay lugar a ambigüedad, usamos los términos grupo de prolongación, grupo activo y grupo resuelto para referirnos al conjunto tanto de factores como de posiciones.

Dentro de una fase, el algoritmo itera en el grupo activo (línea 3) agregando todos los $k-$factores (de $x_1^f$) pertenecientes a dicho grupo. Al inicio de la fase se conoce la posición inicial pero no la final. De todas formas, es fácil determinarla sobre la marcha y finalizar la iteración cuando resulta que $x_p^f$ ya está contenido en el árbol. Al final de una fase, se busca la próxima posición inicial (línea 6).

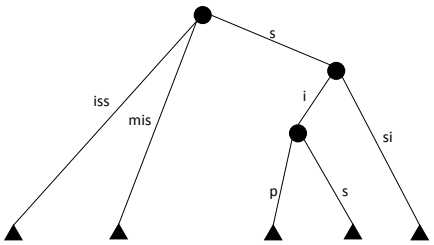
**Algoritmo 2** $U$, el algoritmo de Ukkonen adaptado

1: iniciar el árbol con solamente la raíz y la posición inicial igual a 1.
2: **for** toda fase $f = 1 \dots n$ **do**
3: **while** $p$ es una posición del grupo activo **do**
4: agregar al árbol el $k-$factor de la posición $p$, $x_p^f$, agregando la hoja correspondiente (y el nodo interno $x_p^{f-1}$ si amerita) y completando suffix links.
5: **end while**
6: buscar la nueva posición inicial(correpondiente a fase f+1)
7: **end for**

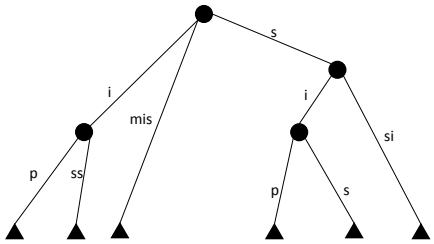
**Ejemplo 3.1.** Para el ejemplo de la cadena $x =$ **mississippi** y $k = 3$ es interesante ver como $U$ modifica el árbol durante la fase 9. Al principio de la fase 9 el árbol es $TST_3$(**mississi**), lo podemos ver en la figura 3.2(a). La



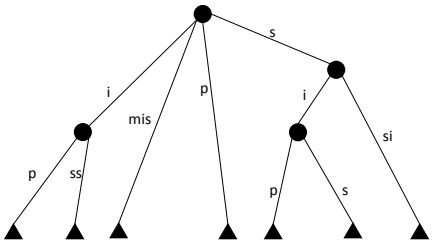

(a) Al inicio de la fase 9




(b) Inserción correspondiente a la posición 7




(c) Inserción correspondiente a la posición 8




(d) Inserción correspondiente a la posición 9

Figura 3.2: Fase 9 en la construcción de $TST_3$(**mississippi**).

posición inicial en la fase 9 es 7. Al agregar el factor $x_7^9 =$ **sip** (línea 4) se parte la arista **s** $\rightarrow$ **sis** creando el nodo interno **si** y la hoja **sip**. Esto puede verse en la figura 3.2(b). Luego, para el siguiente valor de $p$, $p = 8$, las modificaciones en el árbol pueden verse en la figura 3.2(c). Para $p = 9$ insertamos el factor $x_9^9 =$ **p** en el árbol; aquí no es necesario crear un nodo interno y solamente se agrega una hoja al nodo raíz. Por último $x_{10}^9 = \lambda$ es un factor del grupo resuelto y no es necesario modificar el árbol. Se termina la fase y el punto inicial para la fase 10 es 10.

En el algoritmo $U$, el etiquetado de las aristas se realiza mediante pares de índices o punteros a la cadena original que indican el principio y el fin de la etiqueta. El crecimiento automático de las hojas se realiza dejando variable el puntero al fin de la hoja. Más precisamente, el fin de la etiqueta es inferido y no se almacena en ningún lado, lo que hace que la hoja no tenga que modificarse una vez que fue creada.

## 3.2. Una aproximación al procedimiento

En esta sección nos interesamos en cambiar el enfoque sobre el algoritmo $U$, aumentando el nivel de detalle y acercándolo a lo que fue la implementación construida en este trabajo. Así, mostraremos el comportamiento del algoritmo `construirTST` (descrito en el algoritmo 3) y su procedimiento auxiliar `agregarFactor`(descrito en el algoritmo 4).

**Algoritmo 3** `construirTST`. Construcción del árbol de sufijos truncado.

```
 1: inicializar árbol con solo la raíz.
 2: nodo := raíz
 3: p := 1
 4: for f = 1 ... n do
 5:   finFase = false
 6:   while not finFase do
 7:     finFase = agregarFactor()
 8:     if not finFase then
 9:       p := p + 1
10:       nodo := SLink (nodo)
11:     end if
12:   end while
13:   if cola_prolongacion no es vacía then
14:     if f ≥ k then
15:       quitar el primer elemento de cola_prolongacion
16:     end if
17:   else
18:     p := p + 1
19:     nodo := SLink (nodo)
20:   end if
21: end for
```

En `construirTST`, para cada fase $f$, iteramos en la posición $p$ correspondiente al próximo factor a ser insertado. Debemos detener la iteración al encontrar la primera posición que pertenece al grupo resuelto. El procedimiento auxiliar `agregarFactor` devuelve si $p$ es una posición del grupo resuelto (si $x_p^f$ ya ocurría en el árbol), y en caso de que no lo sea, inserta en el árbol el factor $x_p^f$. Usamos el valor devuelto por este para interrumpir la iteración (línea 7).

Si $p$ es una posición del grupo activo para la fase $f$ y $|x_p^{f+1}| \leq k$ entonces $p$ es una posición del grupo de prolongación de la fase $f+1$, ya que $x_p^f$ ocurrió en $x_p^f$ por primera vez en la posición $p$ y $x_p^{f+1}$ no ocurrió en $x_1^f$ (ver definición en la página 22). La cola de prolongación es una cola de hojas donde al principio de una fase cada hoja en la cola corresponde a un factor que pertenece al grupo de prolongación y de ahí su nombre. Las hojas se

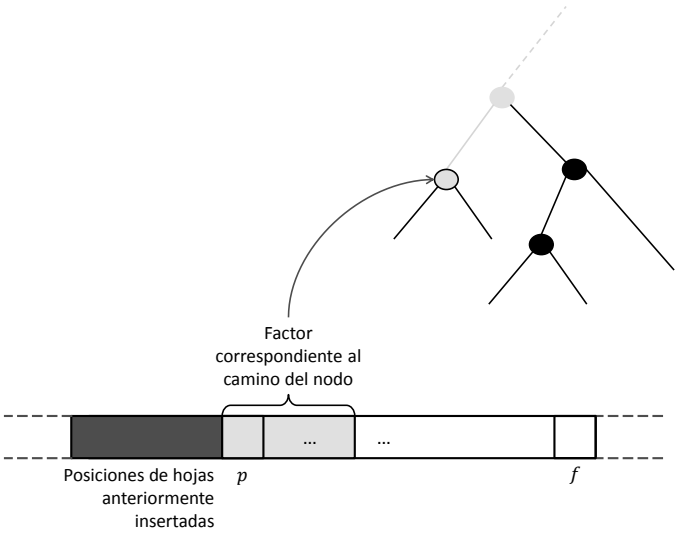



Figura 3.3: Correspondencia entre el nodo y $p$

agregan a medida que son creadas. En las primeras $k$ fases no se quitan hojas de la cola. Al final de la iteración en el grupo activo (líneas 6 a 12) en la cola de prolongación se encuentran las hojas correspondientes a posiciones del grupo de prolongación y del grupo activo. La primera de ellas (si la cola no es vacía) tiene por lo tanto largo $k$ si adicionalmente se cumple $f \geq k$. Para la siguiente fase $f+1$ la primera hoja excedería el largo $k$ (si $f \geq k$) y por eso se quita (línea 15). También al finalizar la iteración del grupo activo, $p$ corresponde a la primera posición del grupo resuelto de la fase $f$. Si la cola de prolongación es vacía quiere decir que el grupo resuelto y el grupo activo son vacíos. Por lo tanto $x_p^f$ es de largo $k$. Entonces la posición $p$ no debe ser considerada para la fase siguiente en que $x_p^{f+1}$ tendrá largo $k+1$. En ese caso la posición inicial de la fase $f+1$ debe ser $p+1$, así en la línea 18 se actualiza el valor de $p$ para asegurar que al principio de la siguiente fase se corresponda con la posición inicial. Si la cola no fuera vacía entonces la posición inicial de la fase $f+1$ debe ser $p$ (las posiciones anteriores a $p$ pertenecen al grupo de prolongación de la fase $f+1$).

Junto con la variable $p$ mantenemos otra variable, `nodo`, que representa un nodo interno del árbol que corresponde a un prefijo de $x_p^f$. La figura 3.3 ilustra esta relación entre $p$ y `nodo`. Cada vez que se incrementa $p$ entonces, es necesario avanzar en el árbol para mantener este sincronismo entre posición en la cadena y nodo. En las líneas 10 y 19 se actualiza el valor de `nodo` con el de su *suffix link*. Esta correlación permite insertar rápidamente en el árbol ya que la variable `nodo` sirve de insumo al procedimiento `agregarFactor` que baja por el árbol partiendo desde `nodo`, como veremos enseguida.

Sea $w$ un factor que ocurre en el árbol, definamos $loc(w)$ (por *location* en inglés) al par $(v, |w|)$ tal que $u \to v$ es la única arista tal que $u \prec w \preceq v$. Observemos que $loc(w)$ representa una suerte de puntero en el árbol que identifica a $w$. El procedimiento `agregarFactor` conoce la posición $p$ donde está el factor $x_p^f$ que se está procesando y la variable `nodo`. En general, todas las variables son compartidas entre `agregarFactor` y `construirTST`. Debe bajar por el árbol hasta encontrar una arista $u \to v$ tal que $u \prec x_p^{f-1} \preceq v$, que existe por construcción, ya que $x_p^{f-1} \in TST_k(x_1^{f-1})$. Para lograr esto es necesario ir eligiendo el camino que coincide con el factor (líneas 3 a 6).

Una vez que encontramos $u$ y $v$, hay que distinguir dos casos. En el primer caso $x_p^{f-1} = v$ (cuando la condición de la línea 7 es cierta), si $v$ tiene un hijo cuya arista empieza con $x_f$ entonces $x_p^f$ pertenece al grupo resuelto, sino hay que agregar una hoja a $v$. En el caso donde $x_p^{f-1} \prec v$, estamos parados en medio de la arista $u \to v$ donde debemos ver si $x_p^f$ ya esta contenido o, equivalentemente, si es prefijo de $v$. Esto es evaluar si se cumple que $x_f$ sea la letra de la arista $u \to v$ a profundidad $f - p + 1$ (la condición de la línea 13); que sea cierto implica que $x_p^f$ pertenece al grupo resuelto y que sea falso implica que hay que crear un nodo partiendo $u \to v$ y agregarle una nueva hoja.

**Algoritmo 4** `agregarFactor`. Modifica el árbol para contemplar un nuevo factor.

```
1:  hijo := nodo
2:  yaExistia := true
3:  while hijo ≠ NULL y prof(hijo)< |x_p^f|  do
4:     nodo := hijo
5:     hijo = el hijo de nodo cuya arista empieza con x_{p+prof(nodo)} o NULL
       si no existe.
6:  end while
7:  if (prof(nodo) = |x_p^{f-1}|)  then
8:     if hijo=NULL then
9:        agregar una hoja a nodo correspondiente a x_p^f
10:       yaExistia=false
11:    end if
12: else
13:    if La etiqueta de nodo→hijo no tiene x_f en la posición |x_p^f|-
       prof(nodo) then
14:       partir nodo→hijo creando un nuevo nodo correspondiente a x_p^{f-1}
          y agregale una hoja correspondiente a x_p^f
15:       yaExistia = false
16:    end if
17: end if
18: return  yaExistia
```

### 3.2.1. Contar ocurrencias de factores

Junto con el árbol de sufijos truncado se cuentan ocurrencias de cada una de sus hojas. Claramente, cuando se crea una hoja el contador que tiene asociado se inicializa en 1. Por otro lado, puede resultar confuso distinguir cuál es el instante apropiado para incrementar el contador de ocurrencias de una hoja. Ciertamente, no es cada vez que se visita durante la construcción del árbol. La cantidad de ocurrencias de una hoja debe incrementarse cuando el $k-$factor asociado vuelve a ocurrir en $x$. Esto sucede cuando en la fase $f$ la cola de prolongación está vacía (entonces $|x_p^f| = k$) y resulta que no hay que modificar el árbol para incluir el factor en la posición $p$, es decir en los términos del algoritmo 3 la actualización se puede hacer justo antes de la línea 18 donde debe aumentarse el contador de la hoja $x_p^f$.

# Capítulo 4

# Una representación eficiente de árboles de sufijos truncados

A grandes rasgos, las herramientas teóricas que utilizamos para la construcción del estimador ya fueron descritas en capítulos anteriores. Construimos el árbol de sufijos truncados a través del algorimto $U$ para luego computar el orden estimado con el algoritmo 1. Es momento ahora de ahondar en detalles más técnicos que conforman la implementación del estimador. El uso de memoria que conllevan los árboles de sufijos ya ha sido una preocupación que ha motivado investigaciones para mitigarlo. Uno de ellos es la representación eficiente de Kurtz [6], que aprovechando redundancias en la estructura de los árboles obtuvo una representación más compacta. En este trabajo la tomamos y la adaptamos doblemente: a los árboles de sufijos truncados (casi inmediato) y al nuevo método de etiquetado que presentaremos enseguida. Este capítulo explica estas adaptaciones, a la vez que recorre la manera en que fueron implementados los distintos aspectos del algoritmo $U$ y cómo se construye la cadena de etiquetado junto con el árbol.

## 4.1. Etiquetado

Originalmente en el algoritmo $U$ las etiquetas del árbol se representan a través de punteros a la cadena original. En [9] se presenta una representación de las etiquetas mediante punteros a otra cadena, más corta, formada a partir de la cadena original. Esto hace que no sea necesario mantener toda la cadena en memoria produciendo un ahorro de memoria que, como veremos, es proporcional a la diferencia en la cantidad de nodos entre $ST(x)$ y $TST(x)$. Esta nueva cadena, que llamaremos de etiquetado, se construye incrementalmente como mostramos a continuación, luego de hacer las definiciones necesarias.

La cabeza de un $k-$factor $v$ que ocurre en la posición $i$ es su prefijo más largo que ya haya ocurrido en la cadena anteriormente. Al sufijo restante lo llamamos cola. Si $v$ hubiera ocurrido en una posición anterior, es claro que su cabeza es $v$, en el caso contrario decimos que $i$ es un *primer punto de ocurrencia*. Es inmediato comprobar que si $i$ es un primer punto de ocurrencia y $u$ es la cabeza de $v$, entonces $i$ pertenece al grupo activo de la fase $i + |u|$. Conviniendo que posiciones negativas son primeros puntos de ocurrencia, definimos la cadena de etiquetado de $x$, $E(x)$, recursivamente como $E(\lambda) = \lambda$ y para todo $f$, $1 \leq f \leq n$

$$E(x^f) = \begin{cases} E(x^{f-1})x_f, & \text{si } f-k+1 \text{ es un primer punto de ocurrencia}, \\ E(x^{f-1}), & \text{en otro caso}. \end{cases} \tag{4.1}$$

Claramente para todo $f$, $f \leq k$ se cumple $E(x^f) = x^f$. En el ejemplo de la cadena **mississipi** tenemos que $E(\mathbf{mississippi}) = \mathbf{missisppi}$. Aunque $E(x^n)$ se obtiene eliminando letras de $x^n$ y, como probaremos más adelante, todas las etiquetas de aristas de $TST(x^n)$ son factores de $E(x^n)$, no se cumple en general que $TST(x^n)$ y $TST(E(x^n))$ coincidan.

### 4.1.1. Representación de etiquetas de aristas mediante $E(x)$

Sea $O(i)$ la cantidad de primeros puntos de ocurrencia hasta la posición $i$, es decir

$$O(i) = |\{j : 1 \leq j \leq i,\ x_j^{j+k-1} \text{ no ocurre en } x_1^{j+k-2}\}|.$$

El teorema siguiente establece la ocurrencia de las etiquetas de las aristas de $TST_k(x)$ en $E(x)$ y un mecanismo para encontrarlas.

**Teorema 4.1 ([9] Representación de aristas con $E(x)$).** *Sea $u \to v$ una arista de $TST_k(x)$ e $i$ la posición en que ocurre $v$ por primera vez como factor de $x$. Entonces la etiqueta de $u \to v$ ocurre en $E(x)$ en la posición $O(i) + |u|$.*

Para probar el teorema usamos los siguientes dos lemas auxiliares.

**Lema 4.2 ([4, 10]).** *Sea $i$ un entero, $1 < i \leq n$, sea $u$ la cabeza del $k-$factor en la posición $i$ y $u'$ la del $k-$factor en la posición $i-1$. Se cumple $u' \preceq x_{i-1}u$.*

**Lema 4.3.** *Sea $i \geq 1$ y $f = i+k-1$. Si $i$ es un primer punto de ocurrencia, entonces la cola del $k-$factor $x_i^f$ es un sufijo de $E(x^f)$ y se cumple*

$$|E(x^f)| = O(i) + |x_i^f| - 1. \tag{4.2}$$

*Recordemos que $|x_i^f|$ es menor que $k$ si $f > n$.*

*Demostración.* En primer lugar el enunciado es cierto para $i = 1$, ya que 1 es un primer punto de ocurrencia, $x_1^k = E(x^k)$ y $O(1) = 1$.

Ahora consideremos $i > 1$; sea $j$ el mayor de los puntos de ocurrencia anteriores a $i$ y asumamos por inducción que el enunciado se cumple para $j$. Para el resto de la demostración nos será útil definir $f' = j + k - 1$.

Primero veamos que se verifica la igualdad (4.2). Si $f \leq n$, entonces los $k-$factores en las posiciones $j$ e $i$ tienen largo $k$ y, por la definición de $j$, se cumple $|E(x^f)| = |E(x^{f'})| + 1$ y $|O(i)| = |O(j)| + 1$, lo cual implica que se cumple (4.2) para este caso. Si por el contrario fuera $f > n$, entonces el $k-$factor en la posición $i$ es un sufijo del de la posición $i - 1$ y, por lo tanto, $i - 1$ también es un primer punto de ocurrencia. Así se deduce que $j$ debe ser $i - 1$; luego $|x_j^f| = |x_i^f| + 1$ y $E(x^f) = E(x^{f'})$, volviendo a obtener (4.2).

Sean $v'$ y $v$ las colas de los $k-$factores en las posiciones $i - 1$ e $i$ respectivamente. Veamos que $v$ es sufijo de $E(x^f)$. El lema 4.2 implica que

$$v \text{ es sufijo de } v'z, \tag{4.3}$$

donde $z = x_f$ si $f \leq n$ y $z = \lambda$ en caso contrario.

Como $i$ es un primer punto de ocurrencia, de acuerdo con la definición de $E$ en (4.1), se verifica que

$$E(x^f) = E(x^{f-1})z. \tag{4.4}$$

Si $i - 1$ también fuera un primer punto de ocurrencia, por hipótesis de inducción $v'$ sería un sufijo de $E(x^{f-1})$; por otro lado si $i - 1$ no lo fuera, tendríamos que $v' = \lambda$. En ambos casos, combinando (4.3) y (4.4), concluimos que $v$ es sufijo de $E(x^f)$ como queríamos demostrar. □

*Demostración del teorema 4.1.* Sea $s$ el $k-$factor en la posición $i$, $s = x_i^{i+k-1}$ y $w$ su cabeza. Por la definición de $i$, es claro que $w \prec v \preceq s$. Además $w$ debe ser prefijo de $u$ porque de otra forma habría un nodo partiendo la arista $u \to v$, correspondiente a la vez anterior que ocurrió $w$ pero no $v$. Por lo tanto $s \backslash u$ es sufijo de de la cola de $s$ que a su vez, por el lema 4.3, es sufijo de $E(x)^{O(i)+|s|-1}$. Por lo tanto, concluimos que la etiqueta de la arista $u \to v$, prefijo de $s \backslash u$, ocurre en $E(x)$ en la posición

$$O(i) + |s| - 1 - (|s| - |u|) + 1 = O(i) + |u|\,.$$

□

### 4.1.2. Ganancias de espacio

Sea $T$ un $\mathcal{A}^+-$árbol. Definimos $L(T)$ como el conjunto de hojas de $T$. El teorema que sigue vincula el largo de $E(x)$ con la cantidad de hojas de $TST_k(x)$. Como corolario obtenemos que utilizar $E(x)$ para el etiquetado produce una ganancia en el uso de memoria proporcional a la diferencia entre la cantidad total de nodos de $ST(x)$ y $TST_k(x)$.

**Teorema 4.4.** *[9] Se cumplen las siguientes desigualdades*

$$|L(\,TST_k(x)\,)| \leq |E(x)| < |L(\,TST_k(x)\,)| + k.$$

*Si adicionalmente* $x_n \notin x^{n-1}$ *entonces* $|L(\,TST_k(x)\,)| = |E(x)|$.

*Demostración.* Si $n < k$, entonces $E(x) = x$ y $TST_k(x)$ es el árbol de sufijos de $x$, siendo trivial la demostración. Consideremos entonces $n \geq k$. Y sea $i$, $k \leq i \leq n$, el mayor primer punto de ocurrencia. Entonces $E(x) = E(x^f)$, donde $f = i + k - 1$. El lema 4.3 implica entonces que el largo de la cadena de etiquetado es $|E(x)| = |E(x^f)| = O(i) + |x_i^f| - 1$. Cada primer punto de ocurrencia da lugar a exactamente una hoja en $TST_k(x)$. Por consiguiente se cumple

$$|E(x)| = |L(TST_k(x))| + |x_i^f| - 1.$$

Finalmente queda observar que $|x_i^f| \leq k$ y que si $x_n \notin x^{n-1}$ entonces $n$ es un primer punto de ocurrencia ($i = n$) y por lo tanto $|x_i^f| = 1$. □

Como comentamos anteriormente, una representación clásica de $TST$ tal como se propone en [10] requiere mantener en memoria las $n$ letras de la cadena $x$. Si en cambio utilizamos $E(x)$, el ahorro de memoria es proporcional a la diferencia $n - |E(x)|$. El siguiente corolario muestra que este ahorro será significativo siempre que $TST_k(x)$ tenga una cantidad de nodos significativamente menor que $ST(x)$.

**Corolario 4.5.** *La cadena de etiquetado* $E(x)$ *satisface*

$$n - |E(x)| > \frac{1}{2}\Big(|ST(x)| - |TST_k(x)|\Big) - k\,. \tag{4.5}$$

*Demostración.* Como en un árbol de sufijos a lo sumo hay una hoja por sufijo, se cumple

$$n - |E(x)| \geq |L(ST(x))| - |E(x)|,$$

de donde, usando el teorema 4.4, obtenemos

$$n - E(x) > |L(ST(x))| - |L(TST_k(x))| - k.$$

Como cada nodo interno de $ST(x)$ tiene al menos dos hijos, al truncar un árbol de sufijos no pueden eliminarse más nodos internos que hojas (de lo contrario alguno de los nodos internos eliminados habría tenido un solo hijo). Por lo tanto se cumple

$$|N(ST(x))| - |N(TST_k(x))| \leq |L(ST(x))| - |L(TST_k(x))|,$$

donde $N(TST_k(x))$ es el conjunto de nodos internos de $TST_k(x)$ y $N(ST(x))$ el de $ST(x)$. Por último, observando que $|ST(x)|-|TST_k(x)| = |N(ST(x))|-|N(TST_k(x))| + |L(ST)| - |L(TST_k(x))|$, obtenemos (4.5). □

## 4.2. La representación

La representación de Kurtz[6] para árboles de sufijos (no truncados) separa a las hojas de los nodos internos y los almacena en tablas separadas. Inserta nodos internos y hojas en las tablas a medida que se van creando. Si $u$ es la $j-$sima hoja creada, decimos que su número de hoja es $j$, análogamente si $v$ es el $i-$simo nodo interno creado decimos que $i$ es su número de nodo interno.

Como en los árboles de sufijos las hojas se crean en el mismo orden en que ocurren los sufijos correspondientes, la posición de una hoja en la tabla de hojas (su número de hoja) coincide con la posición en que ocurre el sufijo. A partir del número de hoja y la profundidad de su padre, es inmediato determinar la etiqueta de la arista de la hoja en la cadena original $x$. Específicamente la $j-$sima hoja creada corresponde al sufijo $x_j^n$ y si $d$ es la profundidad de su padre, la etiqueta de su arista $x_{j+d}^n$ se encuentra en la posición $j + d$ de $x$.

Cambiando al contexto de árboles de sufijos *truncados*, cada primer punto de ocurrencia determina la creación de una hoja. Si una arista $u \to v$ conduce a una hoja $v$, cuyo número de hoja es $j$, entonces $O(i)$ en el teorema 4.1 es igual a $j$. Luego la etiqueta de $u \to v$ es $E(x)_{j+|u|}^{j+|v|}$. En definitiva sigue siendo válida la premisa sobre la que se elabora la representación de las hojas en el artículo de Kurtz donde, almacenando las hojas en orden, se omite el puntero para determinar la etiqueta de su arista. En consecuencia, adaptar este aspecto de la representación de Kurtz a árboles truncados con el nuevo esquema de etiquetado de aristas es inmediato a partir del teorema 4.1.

En lugar de representar las etiquetas de las aristas entrantes a los nodos internos con pares de punteros, Kurtz almacena para cada nodo interno un índice tal que al sumarle la profundidad del padre se encuentra el inicio de la etiqueta de su arista dentro de la cadena $x$. Luego usando su propia profundidad se determina el fin de la etiqueta. Esta representación tiene la virtud de que si una arista $u \to v'$ es partida durante la ejecución del algoritmo insertando un nodo interno $v$, dando lugar a aristas $u \to v \to v'$, el

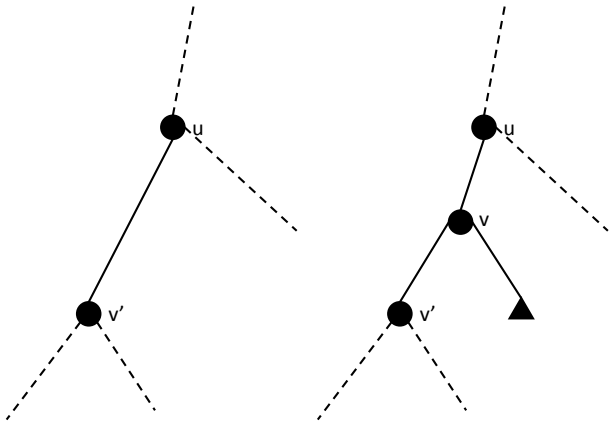



Figura 4.1: Creación del nodo interno $v$

índice almacenado originalmente en $v'$ representa correctamente la etiqueta de la nueva arista $v \to v'$, por lo cual no es necesario modificarlo.

Definimos primer punto de división para un factor $u$, $|u| < k$ como la primera ocurrencia de $ua$, para alguna letra $a$, como factor de $x$ tal que $u$ ocurre en una posición anterior pero $ua$ no. Observemos que si $p$ es el primer punto de división de $u$ entonces en la fase $f = p + |u|$, $p$ es una posición del grupo activo. Además es en la fase $f$ que se crea el nodo interno $u$ (anteriormente ocurría en el árbol como prefijo de otro nodo). Kurtz elige almacenar para cada nodo interno el primer punto de división a fin de representar la etiqueta de su arista. Para usar esta representación, sin embargo, es necesario disponer de la cadena $x$.

Adaptamos la representación de Kurtz al uso de $E(x)$ utilizando un valor *índiceEtiqueta* tal que al sumarle la profundidad del padre, determina el inicio de la etiqueta de la arista en $E(x)$. Durante la creación de un nodo interno $v$ se parte la única arista $u \to v'$ tal que $u \prec v \prec v'$ (ver figura 4.1). Es claro que la etiqueta de la arista $u \to v$ es un prefijo de la de $u \to v'$. Por lo tanto la etiqueta de $u \to v$ se encuentra en $E(x)$ en la misma posición que la de $u \to v'$. De esta forma obtuvimos un mecanismo para determinar *índiceEtiqueta*, siendo para las hojas simplemente su número de hoja (no es necesario almacenar ningún dato extra) y los nodos internos $v$ heredando su valor de $v'$. Observemos que la profundidad del padre solamente puede aumentar luego de la creación del nodo interno (si un nuevo nodo interno partiera la arista entre él y su padre) y en tal caso la etiqueta de su arista seguiría estando en la posición dada por *índiceEtiqueta* más la profundidad de su padre (y en consecuencia no es necesario actualizar su valor).

Por otro lado, la representación de Kurtz se vale de ciertas redundan-

cias para codificar los nodos internos que se crean para insertar sufijos en posiciones consecutivas dentro de una misma fase. En cada una de estas secuencias de nodos internos, $x_q^s, x_{q+1}^s \dots x_r^s$, la profundidad, el primer punto de división y el *suffix link* de los nodos internos $x_q^s \dots x_{r-1}^s$, llamados nodos pequeños, pueden inferirse a partir de los del nodo interno $x_r^s$, que llamamos nodo grande y, por lo tanto, no es necesario almacenarlos explícitamente. Concretamente, si la inserción de dos factores, $x_p^f$ y $x_{p+1}^f$, en la fase $f$ requiere la inserción de sendos nodos internos $x_p^{f-1}$ y $x_{p+1}^{f-1}$ (recordar que por construcción $x_p^{f-1}$ y $x_{p+1}^{f-1}$ ocurren en el árbol) entonces claramente $sLink(x_p^{f-1}) = x_{p+1}^{f-1}$, las profundidades satisfacen $|x_p^{f-1}| = |x_{p+1}^{f-1}| + 1$ y los respectivos primeros puntos de división, $p$ y $p+1$ difieren en 1. En consecuencia, para una secuencia $x_q^s \dots x_{r-1}^s$ de nodos pequeños seguidos por uno grande $x_r^s$, si $x_{q'}^s$ es un nodo pequeño de la secuencia, entonces $|x_{q'}^s| = |x_r^s| + r - q'$; el primer punto de división de $x_{q'}^s$ es $q'$ y el de $x_r^s$ es $r$ y el *suffix link* de $x_{q'}^s$ es el siguiente en la secuencia de nodos, $x_{q'+1}^s$. En nuestro caso, las etiquetas de aristas entrantes a nodos internos no se representan mediante el primer punto de división sino mediante el valor de *índiceEtiqueta*. Sin embargo, como las etiquetas de las aristas de los nodos de una misma secuencia de nodos pequeños seguidos por uno grande son sufijos del primero de la secuencia, vemos que usar *índiceEtiqueta* es compatible con esta representación.

Al buscar un uso eficiente de la memoria, los hijos de un nodo interno se representan con una lista (y no con un vector indizado con letras de $\mathcal{A}$). Por lo tanto cada nodo interno debe almacenar un puntero al primer hijo y al siguiente hermano y cada hoja un puntero al siguiente hermano.

En resumen las estructuras de datos que representan al árbol son las siguientes.

- Una tabla de hojas, donde cada hoja tiene un puntero a su hermano derecho.
- Una tabla de nodos internos, donde cada nodo pequeño contiene
  - Un puntero al primer hijo (29 bits + 1 bit para indicar si es nulo).
  - Un puntero al hermano derecho (29 bits + 1 bit para indicar si es nulo).
  - La distancia al nodo grande de la secuencia, que permite ubicarlo en la tabla de nodos (5 bits).

  y donde cada nodo grande contiene
  - Un puntero al primer hijo (29 bits + 1 bit para indicar si es nulo).
  - Un puntero al hermano derecho (29 bits + 1 bit para indicar si es nulo).

- El *suffix link* (cantidad variable de bits).
- La profundidad .
- Un índice de etiquetado para inferir la etiqueta (27 bits).

En un nodo grande la distribución de bits entre *suffix link* y profundidad depende de la profundidad del nodo. Si es expresable con 10 bits, entonces el *suffix link* ocupa 29 bits y la profundidad 10. De lo contrario, la profundidad ocupa 27 bits y el *suffix link* se codifica en los bits desperdiciados por el puntero al hermano derecho de su último hijo. Los nodos grandes ocupan el doble de palabras que los pequeños y, por lo tanto, dos posiciones en la tabla de nodos internos. Por más detalles sobre la codificación de los nodos referirse a [6].

Adicionalmente, a los efectos de implementar el algoritmo de estimación del órden de Markov (algoritmo 1), en nuestra implementación utilizamos las siguientes estructuras auxiliares.

- Un vector de contadores de ocurrencias, donde puede encontrarse la cantidad de ocurrencias de cada hoja en la posición de su numero de hoja. Esta estructura se usa para determinar los contadores $m_v$ en el algoritmo 1.
- Una lista de primeras $k$ hojas, para determinar los contadores $\overline{m_v}$, $v \in T$.

## 4.3. Secuencias de nodos

Veamos que en cada fase hay (a lo sumo) una única secuencia de nodos pequeños seguidos por uno grande, que corresponde a los primeros factores del grupo activo. Claramente las posiciones de los $k-$factores de $x^f$ que dan lugar a nuevos nodos internos en una fase $f$ pertenecen al grupo activo. Si $x_p^f$ pertenece al grupo activo y no da lugar a la creación de un nodo interno (solamente se agrega una hoja al nodo interno $x_p^{f-1}$, creado en una fase anterior $f'$) entonces la inserción de $x_{p+1}^f$ no puede dar lugar a la creación del nodo $x_{p+1}^{f-1}$ porque debería existir al final de la fase $f'$.

El seguimiento de la secuencia puede hacerse entonces con un puntero al primer nodo interno de la secuencia y una variable booleana que inicia cada fase con valor falso y se marca en verdadero cuando se crea el primer nodo de la fase (si se cumple la condición en la linea 14 del algoritmo 4). Cuando la secuencia esté iniciada y la inserción de $x_p^f$ no requiere la creación de un nodo (tanto si ya existía como si solamente se añade una hoja a un nodo preexistente) el último nodo visitado se marca como *suffix link* del nodo $v_{p-1}$, la secuencia se marca como terminada y se completan los valores de $v_{p-1}$ como nodo grande. Por último observemos que si en una fase $f$ se

creó un nodo interno $x_p^{f-1}$ al insertar el factor $x_p^f$ en el árbol (línea 14 del algoritmo 4), uno podría querer actualizar la referencia de la variable `nodo` al nodo interno $x_p^{f-1}$ ya que es un nodo interno que es prefijo de $x_p^f$ y es más profundo que su padre (al que apunta `nodo`). Sin embargo esto no es deseable porque no podríamos navegar por su *suffix link* (línea 10) ya que podría corresponder al próximo nodo a ser creado (aún no existe).

## 4.4. Construcción de $E(x)$

Volvamos sobre el algoritmo 3. Hemos dicho que al final de una fase $f$ en la cola de prolongación se encuentran las posiciones del grupo activo y de prolongación. Viéndolo de otra forma, podemos decir que se encuentran todas las hojas correspondientes a posiciones mayores o iguales a $f-k+1$ que ya comprobamos que son puntos de primera ocurrencia. Observemos que, si la cola de prolongación es no vacía, la primera hoja se corresponde con la posición $f-k+1$ (remontarse a la observación sobre las posiciones consecutivas de los grupos de prolongación y activo capítulo 3). Por lo tanto si la cola de prolongación no es(está) vacía significa que $f-k+1$ es un primer punto de ocurrencia (que corresponde a la primera de las hojas de la cola) y por lo tanto, por la definición de $E(x)$ en (4.1), tenemos $E(x)^f = E(x)^{f-1}x_f$. Estas observaciones permiten constatar que para construir $E(x)$ alcanza con modificar el algoritmo 3 para agregar $x_f$ al final de la cadena etiquetado si se verifica la condición de la línea 13.

## 4.5. Otras consideraciones sobre la implementación

Aunque la cola de prolongación sea decisiva en varias de las cuestiones propias de la adaptación del algoritmo de Ukkonen a $U$ su implementación es sumamente sencilla. De hecho sólo requiere de un puntero (un índice en la tabla de hojas) a la primera hoja de la cola. Para el tope de la cola se usa el tope de la tabla de hojas. Cuando una hoja es creada, se inserta automáticamente entonces al final de la cola de prolongación. Al finalizar una fase, se incrementa el índice que marca el inicio.

Ahora que conocemos cómo se construye $E(x)$, cómo se relaciona con la cola de prolongación y cómo son las estructuras internas con las que construimos el estimador, puede ser útil traducir algunas de las condiciones descritas en los algoritmos del capítulo anterior a los nuevos términos. Por ejemplo enfoquémonos en el descenso por el árbol del algoritmo 4. Sabiendo que un nodo interno $u'$ de profundidad $d$ pertenece al camino de $x_p^{f-1}$, evaluar si la arista de su hijo $v'$ también lo hace es equivalente a evaluar si $x_{p+d+1} = E(x)_{v'.indiceE+d+1}$, donde $v'.indiceE$ es el índice de etiquetado de $v'$. Obtener el hijo cuya arista empieza con la letra $x_{p+prof(nodo)}$ de la línea

5 equivale a comparar para cada hijo $v'$ de *nodo* la condición que acabamos de describir. Para ser más exactos no es necesario compararlo para *todos* los hijos ya que están ordenados por la letra con la que empieza la arista y por lo tanto se iteraría hasta encontrar el hijo buscado o hasta excederse alfabéticamente. De manera similar se puede traducir la condición de la línea 7 del algoritmo 4 a $E(x)_{\text{hijo}.indiceE+f-p} \neq x_f$.

# Capítulo 5

# Evaluación del consumo de memoria de `TSTEstim`

La cantidad de operaciones que insume construir $TST_k(x)$ junto con la cadena de etiquetado $E(x)$ es prácticamente igual al de construirlo con el mecanismo de etiquetado convencional (usando la propia cadena $x$ para etiquetar las aristas). Es más interesante observar los aportes del uso de $E(x)$ en términos de consumo de memoria, siendo éste el cuello de botella para este tipo de abordajes cuando crece $n$. Un menor consumo de memoria, a su vez, puede redundar en la práctica en menor tiempo de cómputo debido a un mejor aprovechamiento de la jerarquía de memoria, aunque esto no fue evaluado específicamente en este proyecto. En este capítulo analizamos de forma teórica el consumo de memoria asintótico que requiere `TSTestim` para representar el árbol $TST_k$. También evaluamos este consumo experimentalmente, comparando los resultados obtenidos con otras alternativas disponibles: usar el mecanismo convencional de etiquetado y usar una tabla para representar los contadores de los $|\mathcal{A}|^{k+1}$ posibles $k+1-$factores.

## 5.1. Resultados teóricos

En el corolario 4.5 probamos que la ganancia de memoria obtenida al etiquetar las aristas mediante $E(x)$ es proporcional a la diferencia de nodos entre el árbol de sufijos y el árbol de sufijos truncado.

**Teorema 5.1.** *El uso de memoria que conlleva el estimador con el nuevo mecanismo de etiquetado es del orden de* $|\mathcal{A}|^{k_n+1} \log n$ *bits.*

*Demostración.* Recordemos que las estructuras necesarias para representar el árbol durante la construcción fueron descritas en el capítulo 4. La lista de las primeras $k$ hojas puede descartarse ya que no representa almacenamiento

significativo. El largo de la cadena de etiquetado es de orden $O(|L(T)|)$ como lo demostramos en el corolario 4.5. Por otro lado los nodos y los contadores de las hojas pueden almacenarse en registros de tamaño $O(\log n)$. Luego, observando que el árbol tiene a lo sumo $|\mathcal{A}|^{k_n+1}$ hojas y que $|T| \leq 2|L(T)|$ obtenemos que el consumo de memoria es $O(|\mathcal{A}|^{k_n+1} \log n)$ bits. □

Usar el mecanismo tradicional de etiquetado implica un consumo de memoria $O(n \log n)$. Si $f(n)$ es $O(\log n)$ como en el caso del estimador BIC (Bayesian information criterion), el nuevo mecanismo de etiquetado no presenta mejoras asintóticas respecto del consumo de memoria, según se establece a partir de la cota (1.3). Si por el contrario $\log n = o(f(n))$ el consumo de memoria usando el nuevo mecanismo es $o(n)$. En cualquier caso, nunca se empeora el consumo de memoria. A continuación veremos que el nuevo mecanismo de etiquetado en la práctica produce importantes mejoras comparativas.

## 5.2. Resultados experimentales

Para comparar las distintas representaciones, ejecutamos nuestra implementación de `TSTestim` con 7 imágenes binarias en blanco y negro de distintos tamaños. Fijamos la función de penalización $f(n) = \frac{1}{|\mathcal{A}|-1} \log n$, obteniendo el estimador BIC. El cuadro 5.1 muestra el largo de $E(x)$ para el árbol de sufijos truncado $T$ de profundidad $k_n + 1$ construido para cada archivo, donde $n$ es la cantidad de letras (puntos) de la imagen. El largo de $E(x)$ representa en todos los casos menos de 1,5 % del largo de la cadena original $x$. Observemos también en el cuadro 5.1 que la cantidad de nodos de $T$ es en general mucho más pequeña que la cantidad de patrones posibles de largo $k_n + 1$, $|\mathcal{A}|^{k_n+1}$. En consecuencia, construir $T$ puede ser más eficiente en consumo de memoria que construir un árbol completo de profundidad $k_n + 1$, aunque en el peor caso los requerimientos de memoria son asintóticamente equivalentes.

Cuadro 5.1: Tamaño de las estructuras de datos

| Archivo | $n$ | $k_n$ | $\lvert E(x)\rvert$ | $\lvert E(x)\rvert/n$ (%) | $\lvert\mathcal{A}\rvert^{k_n+1}$ | $\lvert T\rvert$ |
|---|---|---|---|---|---|---|
| 1 | 842 752 | 16 | 3 991 | 0,47 | 131 072 | 7 964 |
| 2 | 1 036 288 | 16 | 5 799 | 0,56 | 131 072 | 11 580 |
| 3 | 1 228 800 | 16 | 3 830 | 0,31 | 131 072 | 7 643 |
| 4 | 2 593 472 | 17 | 11 197 | 0,43 | 262 144 | 22 375 |
| 5 | 12 212 224 | 19 | 3 241 | 0,03 | 1 048 576 | 6 461 |
| 6 | 17 915 904 | 20 | 218 077 | 1,22 | 2 097 152 | 436 132 |
| 7 | 21 138 432 | 20 | 5 837 | 0,03 | 2 097 152 | 11 652 |

Los ahorros de memoria generales obtenidos al usar $E(x)$ en lugar de

$x$ dependen de la representación específica del árbol y de las estadísticas asociadas. El cuadro 5.2 compara la memoria necesaria para almacenar $T$ junto con las estructuras auxiliares descritas en el capítulo 4 tanto en la representación convencional como en la nueva representación ($T_{\text{conv}}$ y $T_{\text{nuevo}}$ respectivamente). Observemos que $T_{\text{nuevo}}$ requiere menos de un tercio del almacenamiento requerido por $T_{\text{conv}}$, siendo en muchos casos esta distancia significativamente más pronunciada.

Cuadro 5.2: Uso de memoria

| File | $T_{\text{completo}}$ (Bytes) | $T_{\text{conv}}$ (Bytes) | $T_{\text{nuevo}}$ (Bytes) | $T_{\text{nuevo}}/T_{\text{completo}}$ (%) | $T_{\text{nuevo}}/T_{\text{conv}}$ (%) |
|---|---|---|---|---|---|
| 1 | 524 288 | 901 623 | 73 043 | 13,93 | 8,10 |
| 2 | 524 288 | 1 121 455 | 106 637 | 20,34 | 9,51 |
| 3 | 524 288 | 223 536 | 70 415 | 13,43 | 31,50 |
| 4 | 1 048 576 | 2 758 373 | 205 696 | 19,62 | 7,46 |
| 5 | 4 194 304 | 12 262 793 | 56 454 | 1,35 | 0,46 |
| 6 | 8 388 608 | 20 969 269 | 4 170 084 | 49,71 | 19,89 |
| 7 | 8 388 608 | 21 229 765 | 101 666 | 1,21 | 0,48 |

También es interesante incluir en la comparación un tercer competidor: un árbol completo de profundidad $k_n+1$ (denotado $T_{\text{completo}}$), para el cual es suficiente almacenar una tabla conteniendo los contadores $n_s^a$ (representados como enteros de 4Bytes) para todo patrón $s \in \mathcal{A}_n^k$ y letra $a \in \mathcal{A}$. El uso de memoria asociado a este mecanismo está representado en la tabla bajo la columna $T_{\text{completo}}$. El árbol $T_{\text{nuevo}}$ requiere menos de la mitad del espacio que requiere $T_{\text{completo}}$ para todas las imágenes consideradas.

Por último también hemos ensayado esta misma comparación con un archivo conteniendo la decodificación de material genético con el propósito de ver si se alejaban del otro conjunto de pruebas al incorporar un alfabeto más grande y al variar el tipo de contenido del archivo. Obtuvimos resultados similares a los anteriores.

# Capítulo 6

# Conclusiones

Nos propusimos como objetivo de este proyecto construir un estimador eficiente del orden de procesos de Markov de máxima verosimilitud penalizada. A lo largo de este documento hemos analizado los distintos elementos (tanto teóricos como de implementación) que conjugamos para construirlo. Partimos de la existencia de la cota para el orden óptimo, el algoritmo de construcción de árboles de sufijos truncados $U$ y la representación de Kurtz. Combinamos estos elementos con el nuevo mecanismo de etiquetado y así logramos implementar el estimador propuesto.

A nivel práctico enfrentamos dos grandes desafíos: traducir el algoritmo $U$ a un nivel de detalle con el que pudiera implementarse e incorporar la representación de Kurtz. En el capítulo 3 vimos cómo resolvimos el primero de ellos. Respecto del segundo, pese a que la representación estaba provista, a los efectos de este trabajo requirió tener en cuenta el manejo a muy bajo nivel de la distribución de bits dentro de los registros para poder adaptarla a la implementación que construimos, tarea que resultó sumamente compleja.

Junto con el estimador hemos propuesto un mecanismo de etiquetado que reduce significativamente el consumo de memoria en la práctica, sin comprometer el tiempo de ejecución. Además probamos su correctitud y propusimos una manera sencilla de construirlo junto con el árbol de sufijos truncado e integrarlo a su representación (la de Kurtz). Este mecanismo es además interesante en sí mismo ya que incluso puede adaptarse a otras formas de construcción de árboles de sufijos truncados así como a otras representaciones.

Es de destacar que existen muchas otras aplicaciones de los árboles de sufijos truncados fuera de los estimadores de orden de Markov, como pueden ser algoritmos de compresión[10], de simulación de estadísticas[7] y, en general, búsqueda de patrones sobre texto. El resultado de nuestro trabajo puede ser útil a cualquiera de esos fines ya que provee primitivas para acceder al árbol de sufijos truncado construido durante la ejecución. En definitiva, obtuvimos un mecanismo eficiente para construir árboles de sufijos trun-

cados que, como estructura de datos abstracta, puede encontrar múltiples aplicaciones.

La implementación que construimos contempla registros de 32 bits en los que dispone la representación de los nodos. Una mejora interesante, parcialmente prevista, sería transformarla a registros de 64 bits para permitir entradas con largo mucho mayor. Asimismo podríamos explorar otras representaciones de letras fuera de los caracteres (CHAR) para alfabetos más pequeños como el binario o el usado para representar nucléotidos dentro de las cadenas de ADN, permitiendo mayor flexibilidad en los archivos de entrada.

# Bibliografía


[1] Á. Martín, G. Seroussi, and M. J. Weinberger, "Linear time universal coding and time reversal of tree sources via FSM closure," vol. 50, no. 7, pp. 1442–1468, Jul. 2004.

[2] A. Garivier, "Consistency of the unlimited BIC context tree estimator," vol. 52, pp. 4630–4635, 2006.

[3] P. Weiner, "Linear pattern matching algorithms," in *Proc. 14th Annual Symposium on Switching and Automata Theory*, ser. SWAT '73. Washington, DC: IEEE Computer Society, 1973, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SWAT.1973.13

[4] E. M. McCreight, "A space-economical suffix tree construction algorithm," *J. ACM*, vol. 23, no. 2, pp. 262–272, Apr. 1976.

[5] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[6] S. Kurtz, "Reducing the space requirement of suffix trees," *Software – Practice and Experience*, vol. 29, pp. 1149–1171, 1999.

[7] Á. Martín, N. Merhav, G. Seroussi, and M. J. Weinberger, "Twice-universal simulation of Markov sources and individual sequences," vol. 56, pp. 4245–4255, 2010.

[8] L. Vitale, Á. Martín, and G. Seroussi, "Bounds on estimated Markov orders of individual sequences," in *Proc. ISIT*, 2012, pp. 1102–1106.

[9] ——, "Space-efficient representation of truncated suffix trees, with applications to markov order estimation," in *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, 2013, pp. 1924–1928.

[10] J. C. Na, A. Apostolico, C. S. Iliopoulos, and K. Park, "Truncated suffix trees and their application to data compression," *Theoretical Computer Science*, vol. 304, no. 1–3, pp. 87–101, 2003.

[11] I. Csiszár and Z. Talata, "Context tree estimation for not necessarily finite memory processes, via BIC and MDL," vol. 52, pp. 1007–1016, 2006.