Tarea 2 - Suffix Trees

CC4102 - Diseño y Análisis de Algoritmos
Profesor: Gonzalo Navarro
Auxiliar: Jorge Bahamonde
Ayudantes: Sebastián Ferrada, Willy Maikowski

9 de Noviembre de 2016
Fecha de Entrega:11 de Diciembre de 2016

1 Introducción

Los suffix trees son estructuras que almacenan todos los sufijos de un texto $S[1...n]$ como llaves, y sus posiciones en el texto como valores. Se mencionó en clases que los suffix trees pueden ser construidos en tiempo $O(n)$: exploraremos una forma de hacer esto en esta tarea. Se pide que plantee una hipótesis con respecto al tiempo *amortizado* de construcción de una estructura de este tipo y al tiempo de búsqueda, y la ponga a prueba de forma experimental.

Se espera que se implemente la estructura y los algoritmos correspondientes, y se entregue un informe que indique claramente los siguientes puntos:

1. Las *hipótesis* escogidas antes de realizar los experimentos.
2. El *diseño experimental*, incluyendo los detalles de la implementación de los algoritmos, la generación de las instancias y las medidas de rendimiento utilizadas.
3. La *presentación de los resultados* en forma de una descripción textual, tablas y/o gráficos.
4. El *análisis e interpretación* de los resultados.

2 Suffix trees implícitos

Para explicar mejor el algoritmo de Ukkonen, definiremos lo que son los suffix trees *implícitos*.

Definición 2.1. Suffix tree implícito. Un *suffix tree implícito* para un string S es el árbol obtenido del suffix tree para $S\$$, al que se le han aplicado los siguientes pasos:

- Primero, se remueven todas las instancias del símbolo terminal \$.
- Luego se remueven los arcos que quedaron sin etiquetas.
- Luego se remueven los nodos que no tienen al menos dos hijos.

Un suffix tree implícito para un prefijo $S[1...i]$ de S se define, de la misma forma, tomando el suffix tree para $S[1...i]\$$ y borrando las instancias de \$, los arcos y nodos como se detalló. Denotaremos el suffix tree implício de $S[1...i]$ como I_i, para $i = 1, ..., n$.

3 Algoritmo de Ukkonen

El algoritmo de Ukkonen construye I_n, partiendo de I_1 en n *fases*. En la fase $i+1$, se construye I_{i+1} a partir de I_i. En otras palabras, en cada fase tenemos el suffix tree implícito de un prefijo de S, que vamos "extendiendo" hasta tener el suffix tree implícito de S.

Con esto en mente, cada fase $i+1$ se divide en $i+1$ *extensiones*, una para cada sufijo de $S[1...i+1]$. En la extensión j de la fase $i+1$, el algoritmo encuentra, en I_i, el final del camino correspondiente al substring $S[j...i]$. Luego extiende el substring agregando $S[i+1]$ al final del camino, a menos que $S[i+1]$ ya esté.

Visto de otra forma, en la fase $i+1$, se agrega el string $S[1...i+1]$, luego $S[2...i+1]$, $S[3...i+1]$, etc. en las extensiones 1, 2, 3..., etc. La extensión $i+1$ de la fase $i+1$ extiende el sufijo vacío de $S[1...i]$ (es decir, agrega el caracter $S[i+1]$, a menos que ya esté). El árbol I_1 es simplemente el árbol que posee un único arco con la etiqueta $S[1]$.

Un poco más ordenado:

Algoritmo 1: Algoritmo de Ukkonen, resumido

Input: Un string $S[1...n]$
Output: El suffix tree implícito de S.
$I \leftarrow I_1$;
for $i = 1, ..., n-1$ **do**
    `// Fase` i
    for $j = 1, ..., i+1$ **do**
        `// Extensión` j
        Encontrar el final del camino desde la raíz de I, correspondiente a $S[j...i]$;
        `// Nos aseguramos de que` $S[j...i+1]$ `está en el árbol.`
        Si es necesario, extender ese camino agregando $S[i+1]$;
return I

Ahora, ¿cómo *extendemos* los caminos? Sea $\beta = S[j...i]$ un sufijo de $S[1...i]$. En la extensión j, cuando el algoritmo llega hasta el final de β en el árbol, *extiende* β con $S[i+1]$. Esto se hace siguiendo estas reglas:

Regla 1 Si el camino para β termina en una hoja, se agrega $S[i+1]$ al final de la etiqueta del arco que lleva a esa hoja.

Regla 2 Si al final del camino para β existe al menos un camino, pero ninguno que empiece con $S[i+1]$, debe crearse un nuevo arco a hoja, que comience desde el final del camino de β. Si β se nos terminó a la mitad de un arco, será necesario crear además un nuevo nodo interno. La hoja al final del nuevo arco recibe el número j como valor.

Regla 3 Si algún camino al final de β comienza con $S[i+1]$, recordamos que el árbol es *implícito*, así que no tenemos que hacer nada :), por la misma definición.

¿Cuánto demora esto? Una vez que llegamos al final del camino correspondiente a β, aplicar alguna de las tres reglas mencionadas toma tiempo constante. Entonces, encontrar de forma rápida los finales de estos caminos es lo que importa para hacer que el algoritmo sea eficiente.

La primera idea sería encontrar el final de cada β en tiempo $O(|\beta|)$ (bajando por el árbol como vimos en clases). Eso daría que la extensión j de la fase $i+1$ tomaría tiempo $O(i+1-j)$ con lo que la fase i tomaría tiempo $O(i^2)$, y el algoritmo completo tomaría tiempo $O(n^3)$.

Describiremos una serie de técnicas para reducir este tiempo, comenzando por agregar los llamados suffix links.

4 Suffix Links

Los suffix links son enlaces dentro del mismo suffix tree que si bien no son estrictamente necesarios en su definición, son útiles en varios casos, como el de su construcción. Definiremos un suffix link como sigue:

Definición 4.1. Suffix link. Sea $x\alpha$ un string arbitrario, donde x es un caracter y α un string (potencialmente vacío). Sea un nodo interno v de un suffix tree que corresponde al camino $x\alpha$ y un nodo $s(v)$ que corresponde al camino α, ambos caminos desde la raíz. Un *suffix link* es un puntero de v a $s(v)$.

En otras palabras, un suffix link es un puntero de un nodo que corresponde a un string (visto como un camino desde la raíz) a otro que corresponde al mismo string corrido en un caracter.

Hay dos casos que es mejor aclarar. Primero, en el caso en que α es el string vacío, el suffix link apunta a la raíz del árbol (como se esperaría). Por otro lado, no consideraremos la raíz como un nodo interno para esta definición, con lo que no salen suffix links de ésta.

Para ver cómo los agregaremos, usaremos la siguiente propiedad:

Propiedad 1. *Si un nodo interno nuevo v correspondiente a un camino $x\alpha$ se agrega en la extensión j de la fase $i+1$, entonces una de las siguientes situaciones se da:*

- *El camino correspondiente a α ya termina en un nodo interno del árbol (que corresponde a $s(v)$ por definición).*
- *Un nodo interno al final de α será creado en la extensión $j+1$ en la misma fase $i+1$ (que corresponde a $s(v)$ por definición).*

En otras palabras, cuando agregamos un nodo, en la siguiente extensión tendremos disponible el nodo destino de su suffix link. Luego agregamos el siguiente paso al algoritmo, posibilitado por la propiedad que ya demostramos:

Si en la extensión j se crea un nodo interno nuevo v correspondiente a un camino $x\alpha$, en la extensión $j+1$ (si existe) se crea el suffix link de v a $s(v)$.

Para que este paso sea posible, sólo tenemos que recordar el nodo que agregamos en una extensión j durante la siguiente extensión.

Corolario 1. *Todo nodo interno que se crea con este algoritmo tendrá un suffix link desde él, al final de la siguiente extensión.*

Como conclusión de esta sección, entonces, tenemos que podemos agregar suffix links para todos los nodos; ahora veremos cómo usarlos.

5 Usando los suffix links

Nuestra primera idea para cada paso de extensión era encontrar el final de cada sufijo $S[j...i]$ bajando en el árbol. Los suffix links pueden ayudar a acortar este descenso. Explicaremos la extensión $j=1$ y $j=2$, para ayudar a entender cómo.

Para $j=1$, sabemos que $S[1...i]$ debe terminar en una hoja, porque $S[1...i]$ es la cadena más larga que está almacenada hasta el momento. Una forma de hacer esto más rapido es mantener siempre un puntero hacia la hoja correspondiente al string completo $S[1...i]$ actual. Luego la extensión $j=1$ tomaría tiempo constante, aplicando la Regla 1.

Para $j=2$, tenemos que encontrar $S[2...i]$ en el árbol.

- Sea $S[1...i]=x\alpha$, como en la definición de los suffix links. Con esto, la segunda extensión nos pide buscar α en el árbol.
- Sea v el último nodo no-hoja por el que pasamos en la extensión anterior: si es interno, posee un suffix link $s(v)$.

- $s(v)$ corresponderá a un camino que es un prefijo de α, por la misma definición de $s(v)$ (no necesariamente corresponde a α, pues tuvimos que subir un poco en el árbol para llegar a v). De esta forma, podemos empezar a buscar el resto de α (que llamaremos γ) desde $s(v)$.

Sea γ la etiqueta en el arco entre un nodo v y la hoja a la que llegamos buscando $x\alpha$. Para llegar al final de α, se sube desde la hoja al nodo v; se sigue el suffix link de v a $s(v)$, y se baja desde $s(v)$ por el camino correspondiente a γ (desde este punto). El final de este camino es el final del camino correspondiente a α desde la raíz, por lo que aquí se aplican las reglas de extensión.

Las siguientes extensiones ($j > 2$) se realizan de la misma forma. Al final de la extensión anterior, se llegó al final del camino $S[j-1...i]$. En el caso general, el nodo que está al final de $S[j-1, i]$ puede ya tener un suffix link: en ese caso no subimos en el árbol, y usamos ese suffix link para después bajar. En cualquier caso, el algoritmo nunca sube más de un arco en el árbol antes de encontrar un suffix link.

Luego, para $j > 2$:

Algoritmo 2: Algoritmo para una Extensión

Input: Un suffix tree implícito I_i, luego de realizarse la extensión $j-1$ de la fase $i+1$
Output: El suffix tree implícito, luego de realizarse la extensión j de la fase $i+1$.

`// Este paso requiere subir a lo más un arco desde el final de` $S[j-1...i]$
$v \leftarrow$ el primer nodo en o sobre el final de $S[j-1...i]$ que tenga un suffix link o sea la raíz;
$\gamma \leftarrow$ el string entre v y el final de $S[j-1...i]$;

if *v no es la raíz* **then**
  Se sigue el suffix link de v a $s(v)$, y se baja desde $s(v)$ siguiendo el camino correspondiente a γ.
else
  `// Si` v `es la raíz, seguimos nuestra idea inicial`
  Se sigue el camino $S[j...i]$ desde la raíz.
`// En este punto ya llegamos a` $S[j...i]$`.`

Se agrega $S[j...i+1]$ al árbol, siguiendo las Reglas 1-3;

`// Si se siguió la regla 2 en la extensión anterior`
if *se creó un nodo interno nuevo w en la extensión $j-1$* **then**
  `// El camino` $S[j...i]$ `debe terminar en` $s(w)$
  Se crea el suffix link de w hacia el nodo interno al final del camino que se recorrió.

Además de esto, tenemos que mantener un puntero al nodo correspondiente a $S[1...i]$ durante la fase i, lo que, como ya se explicó, permite que la primera extensión de la siguiente fase no necesite hacer un descenso en el árbol.

6 Trucos adicionales

Sólo agregar y usar los suffix links no es suficiente para volver eficiente esta construcción del suffix tree. Los siguientes trucos completan

6.1 Compresión de etiquetas

En primer lugar, conviene usar índices en vez de strings para las etiquetas de los arcos, para que descender por un arco tome tiempo constante. Se usa un par de índices: la posicion inicial y final del caracter correspondiente en S. Así, para ver si se puede descender por un arco con la etiqueta

(p, q), se realizan las comparaciones correspondientes con los caracteres $S[p]$ y $S[q]$. En general, la explicación del resto del algoritmo sigue la intuición de arcos etiquetados con strings, pero sólo por claridad.

6.2 Skip/Count

Supongamos que usamos un suffix link para saltar de v a $s(v)$, y que el string que resta por buscar es γ, de largo g. Sabemos que existe el camino a seguir, pues corresponde a un sufijo que ya está en el árbol.

Como no pueden existir dos arcos que bajen de $s(v)$ y que comiencen con el mismo caracter, el primer caracter de γ debe aparecer como el primer caracter de sólo uno de estos arcos. Sea g' el número de caracteres de ese arco. Si $g' < g$, no es necesario inspeccionar el arco: se puede saltar directamente al siguiente nodo, setear $g \leftarrow g - g'$, $h \leftarrow g' + 1$, y ver qué arco corresponde al caracter h de γ.

En general, cuando el algoritmo identifica el siguiente arco a seguir, compara el valor de g con el número de caracteres g' en el arco. Si $g \geq g'$, se salta al siguiente nodo, se setea $g \leftarrow g - g'$, $h \leftarrow h + g'$, encuentra el arco cuyo primer caracter corresponda al caracter h de γ y repite.

6.3 La Regla 3 termina la fase

El segundo truco es el siguiente: si se aplica la Regla 3 en la extensión j_R, se va a aplicar en todas las siguientes extensiones de la fase: cuando se aplica la Regla 3, es porque el camino correspondiente a $S[j_R...i]$ debe continuar con $S[i+1]$. Por definición del árbol, todos los sufijos de este camino ya deben estar en el árbol, así que los caminos $S[j_R + 1...i]$, $S[j_R + 2...i]$, etc. también deben seguir con $S[i + 1]$.

Luego, si en una extensión j_R se aplica la regla 3, la fase completa termina enseguida. Es decir, sólo es necesario realizar las extensiones $j \leq j_R$: las demás son gratis.

6.4 Las hojas nunca dejan de ser hojas

Una vez que se crea una hoja (mediante la Regla 2), siempre será una hoja, pues no hay forma de colgar nodos de una hoja: sólo se reescribe el arco mediante la Regla 1.

Primero, sabemos que la hoja correspondiente a $S[1]$ se va a crear en la fase 1. Luego, dado que una hoja siempre es una hoja, existe una secuencia inicial de j_L extensiones consecutivas (comenzando con la 1) donde se aplica la Regla 1 o 2. Dado que la Regla 2 crea una hoja, en la siguiente fase cada aplicación de esta regla se vuelve una aplicación de la Regla 1.

Luego j_L sólo puede crecer. La primera parte del truco es mantener el valor j_L actualizado y en memoria, registrando cuál fue la última extensión que correspondió a la Regla 1 o 2. Esto nos permite saber con certeza que en la siguiente fase las primeras j_L extensiones corresponderán a aplicar la Regla 1.

La segunda parte del truco corresponde a hacer que estas aplicaciones de la Regla 1 sean gratis. Cada vez que creamos una hoja, en vez de usar el label (p, q) para el arco que da a ésta, usamos el label (p, e), con e un símbolo global que significa "el final del string hasta ahora". Es decir, e se *interpreta* como $i + 1$ en cada fase. Esto significa que todas las extensiones que apliquen la Regla 1 no requieren hacer nada, pues la reinterpretación de e hace todo el trabajo.

Es decir, sólo es necesario realizar las extensiones $j > j_L$: las demás son gratis.

Qué pasa con las extensiones $j_L < j \leq j_R$? Usaremos los *suffix links* para hacerlas más rápidas.

7 ¡Finalmente!

El algoritmo para la fase $i + 1$ queda de la siguiente forma:

Algoritmo 3: Algoritmo para una Fase

Input: Un suffix tree implícito I_i, el valor de j_L.
Output: El suffix tree implícito I_{i+1}, el nuevo valor de j_L.

```
// Esto ejecuta las extensiones 1...j_L implícitamente
Se setea la interpretación de e como i + 1;
for j = j_L, ..., i + 1 do
    Se computa explícitamente la extensión j, utilizando el Algoritmo 2;
    if se utiliza la Regla 3 then
        /* Si hemos llegado hasta aqui, todas las extensiones anteriores
           usaban las Reglas 1 o 2                                        */
        j_L ← j − 1;
        /* Como usamos la Regla 3, j = j_R, así que las extensiones que vienen
           están listas                                                   */
        break
    else
        /* Si ésta también usaba la Regla 1 o la 2.  Puede que nunca entremos
           al otro caso del if, así que hay que cuidar que j_L quede bien
           definido.                                                      */
        j_L ← j;
```

Adicionalmente, cuando comenzamos desde j_L, ya sabemos dónde termina el sufijo correspondiente, ¡porque es el que acabamos de recorrer en la fase anterior! De esta forma, el primer descenso de cada fase es gratis.

Todo esto nos da el suffix tree implícito para S. Para convertirlo en un suffix tree verdadero, se realizan los siguientes pasos:

- Se realiza una última fase del algoritmo de Ukkonen para agregar el caracter terminal \$ a S. Luego de este paso, se tiene un suffix tree implícito en el que ningún sufijo es prefijo de otro (dado que \$ no aparece en ningún otro lugar), por lo que cada sufijo de S aparece en una hoja de forma explícita.
- Se reemplazan los valores de e en los arcos por n (lo que requiere recorrer el árbol en su totalidad).

8 Datos y Experimentos

Utilice, al menos, textos de lenguaje natural. Preprocese los textos que usará: elimine saltos de línea, puntuación, lleve todo a minúsculas y otras operaciones que estime convenientes. Luego de este preprocesamiento, asegúrese de que sus textos tengan largos (al menos aproximados) de $N = 2^i$, con $i \in \{15, 16, ..., 25\}$ [1]

Para cada texto, construya el suffix tree, midiendo tanto el tiempo de construcción como el costo (en número de operaciones) por cada fase del algoritmo de construcción: en particular, grafique el costo de cada fase, de modo de observar cómo se amortiza.

Escoja $N/10$ palabras de forma aleatoria del texto y encuentre todas las ocurrencias de éstas, registrando los tiempos de búsqueda en función del largo m del patrón. Repita estos procesos (construcción y búsqueda) para obtener promedios confiables para los tiempos registrados.

Note que tiene libertad para escoger los textos que utilizará: de esta forma, puede escogerlos para ayudarse a poner a prueba su hipótesis, si fuera necesario. Otro grado de libertad es el tamaño del alfabeto con el que trabajará (por ejemplo, puede comparar lo que sucede al utilizar lenguaje natural versus un alfabeto binario).

[1] Puede usar `http://pizzachili.dcc.uchile.cl/texts/nlang/` como una fuente de datos, además de `http://www.gutenberg.org`

9 Entrega de la Tarea

- La tarea puede realizarse en grupos de a lo más 3 personas.
- Para la implementación puede utilizar `C`, `C++` o `Java`. Para el informe se recomienda utilizar LaTeX.
- Escriba un informe claro y conciso. Las ponderaciones del informe y la implementación en su nota final son las mismas.
- La entrega será a través de U-Cursos y deberá incluir el informe junto con el código fuente de la implementación (y todas las indicaciones necesarias para su ejecución).

10 Links (i.e. ¡Dibujos!)

Varias explicaciones de la construcción:

- `https://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf`
- `http://www.cbcb.umd.edu/confcour/Spring2010/CMSC858W-materials/lecture5.pdf`
- `https://www.biostat.wisc.edu/bmi776/spring-09/lectures/suffix-trees.pdf`
- `http://users-birc.au.dk/mailund/slides/SA2007/Ukkonen-2007.pdf`
- `http://web.stanford.edu/~mjkay/gusfield.pdf` (tengan cuidado, noté algunos errores con la notación en éste).

11 Apéndice

Demostración de la propiedad 1

Demostración. Se crea un nodo nuevo sólo cuando se aplica la Regla 2. Esto significa que el camino $x\alpha$ ya continuaba con un caracter $c \neq S[i+1]$. Luego, en la extensión $j+1$ habrá un camino correspondiente a α que continúa con el caracter c (existente desde la fase anterior, ya que todos los sufijos de $S[1...i]$ ya deben estar).

Hay dos casos a considerar. Si el camino correspondiente a α sigue sólo con el caracter c, la Regla 2 creará un nodo $s(v)$ al final del camino α, durante la extensión $j+1$ (de forma análoga a lo que pasó en la extensión j), lo que se corresponde con el segundo punto. Si α continúa con dos caracteres diferentes, ya debe existir un nodo al final de α (!el nodo en que los caminos se bifurcan!), con lo que se cumple el primer punto. En ambos casos, se tiene la propiedad. □

Demostración del Corolario 1

Demostración. La demostración es inductiva. El caso inicial, I_1 se cumple por vacuidad, pues no existen nodos internos. Supongamos que la propiedad se cumple al final de la fase i, y consideremos la $i+1$. Por la propiedad anterior, cuando se cree un nuevo nodo v en la extensión j, el nodo $s(v)$ hacia el cual apuntar el suffix link desde v va a ser encontrado en la extensión $j+1$. En la última extensión de una fase, no se crean nuevos nodos internos (ya que es simplemente agregar el caracter $S[i+1]$), con lo que todos los suffix links de nodos internos creados en la fase $i+1$ se han creado al final de ésta. □