Árboles de Sufijos: fundamentos, búsqueda y construcción lineal en el reconocimiento de genomas únicos

(SUS/dSUS sobre Árbol de Sufijos Generalizado para genómica)

Jose Huaman, Paulo Miranda, Elmer Villegas
Ciencias de la Computación, Universidad de Ingeniería y Tecnología (UTEC), Lima, Perú
{jose.huaman, paulo.miranda, elmer.villegas}@utec.edu.pe

***Resumen*—Estudiamos el problema de *Subcadenas Únicas Más Cortas* (SUS) y su variante *discriminativa* (dSUS) sobre múltiples genomas. Dado un objetivo G_i y un conjunto de fondo, buscamos la subcadena de menor longitud que identifica de forma inequívoca a G_i (unicidad exacta). Proponemos un enfoque basado en el Árbol de Sufijos Generalizado (GST): (i) construcción lineal del índice mediante Ukkonen; (ii) cómputo en postorden de la *profundidad de cadena* $d(v)$ y del *conjunto de colores* $C(v)$ por nodo; (iii) selección de dSUS global por genoma a partir del primer cambio de color en aristas $u \to v$ (de $|C(u)| > 1$ a $C(v) = \{i\}$). La búsqueda exacta de un patrón P cuesta $O(|P|)$ y, tras construir el GST en tiempo/espacio $O(N)$, la selección de dSUS global es $O(N)$. Incluimos un ejemplo didáctico y notas prácticas para implementación.**

***Index Terms*—árbol de sufijos; árbol de sufijos generalizado; SUS; dSUS; búsqueda de patrones; bioinformática; complejidad algorítmica**

I. Introducción

Los índices de cadenas permiten consultas a gran escala con baja latencia. En genómica, identificar *fragmentos mínimos* que distingan un genoma objetivo del resto habilita tareas como diseño de *primers/probes*, clasificación de lecturas y extracción de firmas.

Problema. Dada una colección $\mathcal{G} = \{G_1, \ldots, G_k\}$, buscamos para cada objetivo G_i una subcadena de *longitud mínima* que sea *única* respecto de la colección (aparece en G_i y no en G_j, $j \neq i$). Llamamos a esta subcadena *dSUS* (SUS si no hay fondo). En este trabajo adoptamos *unicidad exacta* (sin tolerancias).

Contribuciones. (i) Formalización de SUS/dSUS en colecciones y criterio de unicidad en el GST vía *conjuntos de colores* $C(v)$; (ii) diseño e implementación del GST explícito construido con Ukkonen (tiempo/espacio $O(N)$); (iii) recorrido en postorden para calcular $d(v)$ y $C(v)$ (bitsets) y *selección* de dSUS global por genoma mediante el primer cambio de color en aristas; (iv) análisis de complejidad y ejemplo ilustrativo ($G_1 = \texttt{ATC}$, $G_2 = \texttt{ATT}$).

Organización. La Sección II presenta el fundamento teórico. La Sección V detalla las estructuras y algoritmos (Ukkonen, coloreo y selección de dSUS). La sección final resume conclusiones y líneas futuras.

II. Fundamento teórico

A. Cadenas, sufijos y terminador

Sea $T = t_1 t_2 \ldots t_n$ una cadena sobre un alfabeto finito Σ. Un *sufijo* de T es $T[i..n]$. Para asegurar que ningún sufijo es prefijo de otro, se usa un símbolo terminador $\$ \notin \Sigma$ y se trabaja con $T\$$ [1], [2].

Por ejemplo, para $\Sigma = \{a, b, c\}$ una cadena de longitud 3 puede ser `abc`. Los sufijos de la cadena son $\{\texttt{abc}, \texttt{bc}, \texttt{c}\}$; mientras que con terminador son $\{\texttt{abc\$}, \texttt{bc\$}, \texttt{c\$}, \texttt{\$}\}$.

B. Trie de sufijos y árbol de sufijos

El *trie* de sufijos contiene todos los sufijos de $T\$$ como caminos desde la raíz a hojas. El *árbol de sufijos* (suffix tree, ST) es el trie comprimido: se contraen caminos de nodos de grado 1. En la implementación estándar, las aristas se etiquetan con intervalos de T y cada hoja representa un sufijo distinto [1], [8], [9], [10], [11].

C. Explícito vs. implícito

Sin terminador, el ST es *implícito*: algunos sufijos terminan en medio de aristas y pueden confundirse con prefijos de otros. Con terminador, el ST es *explícito*: cada sufijo finaliza en una hoja distinta [1], [2].

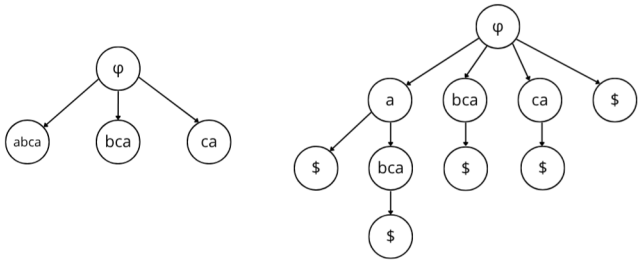


Figura 1. Suffix Tree para `abca` sin (izquierda) y con (derecha) terminador. Sin terminador, el sufijo `a` termina en mitad de una arista del camino `abca` (árbol implícito). Con terminador, el árbol es explícito, es decir, diferenciamos cada sufijo con una hoja. Elaboración propia.

D. Árbol de sufijos generalizado (GST)

Para una colección de genomas $\mathcal{G} = \{G_1, \ldots, G_k\}$, el *GST* se define sobre el texto concatenado $G_1\$_1 G_2\$_2 \cdots G_k\$_k$, con sentinelas distintos $\$_i$. Cada hoja almacena el par `(docID, pos)`. Para cada nodo v se define:

- *string-depth* $d(v)$: longitud del string en el edge raíz$\rightarrow v$,
- el conjunto de colores $C(v) \subseteq \{1, \ldots, k\}$: los genomas presentes entre sus hojas.

El tamaño $|C(v)|$ es el *color set size* (CSS), útil para decidir en qué genomas aparece un substring [12], [1].

E. Unicidad respecto de una colección: SUS y dSUS

Sea S un substring de G_i. Diremos que S es *único respecto de la colección* si no aparece en ningún G_j con $j \neq i$. La **dSUS global de** G_i es cualquier substring de G_i que sea único respecto de la colección y tenga longitud mínima entre todos los substrings de G_i con esa propiedad. Opcionalmente, la *dSUS por posición* en G_i es la subcadena más corta que cubre una posición dada y es única respecto de la colección.

Observación. Si G_i es idéntico a algún G_j o está completamente contenido en él, entonces $S = \varnothing$ y no existe dSUS para G_i. En la literatura se introduce una *tolerancia de repetición* $\tau \geq 0$ para permitir hasta τ ocurrencias fuera de G_i [3], [4], [5]. En este trabajo fijamos $\tau = 0$ (unicidad exacta) [1].

F. Caracterización en el GST

Sea v un nodo del GST y sea $S(v)$ el string del camino hasta v. Se tiene:

1. $S(v)$ es único de G_i si y solo si $C(v) = \{i\}$.
2. Si $u = \text{parent}(v)$, $|C(u)| > 1$ y $C(v) = \{i\}$, entonces cualquier string de longitud $d(u) + 1$ cuyo locus caiga bajo v es ya único de G_i.

Por tanto, la dSUS global de G_i se obtiene tomando, entre todas las aristas $u \rightarrow v$ con $|C(u)| > 1$ y $C(v) = \{i\}$, la mínima longitud $d(u)+1$ y el correspondiente substring en G_i (si hay empates, se puede romper por orden lexicográfico o por posición). Esta regla se implementa con un recorrido postorden que calcula $C(v)$ y $d(v)$ para todos los nodos [1], [12].

G. Ejemplo sencillo

Considérese G_1 = `ATC` y G_2 = `ATT`, con sentinelas distintos sólo para construir el GST (la unicidad se evalúa sobre $\{A, C, G, T\}$, ignorando sentinelas). En el GST, los caminos `A` y `AT` tienen $C = \{1, 2\}$; las ramas `ATC` y `ATT` son exclusivas con $C = \{1\}$ y $C = \{2\}$, respectivamente. De aquí se obtiene de forma directa:

- $\mathbf{dSUS_{global}}(G_1)$ = `C` (longitud 1), y $\mathbf{dSUS_{global}}(G_2)$ = `TT` (longitud 2).
- Si pedimos la dSUS que *cubre la posición 1*, el primer cambio de color $\{1, 2\} \rightarrow \{1\}$ (o $\{2\}$) ocurre en la arista `AT`$\rightarrow$`ATC` (o `ATT`), por lo que d(`AT`)+1 = 3: `ATC` para G_1 y `ATT` para G_2.

(Análogo por posición: en G_1, p=2$\rightarrow$`TC`, p=3$\rightarrow$`C`; en G_2, p=2 y p=3 quedan cubiertas por `TT`.)

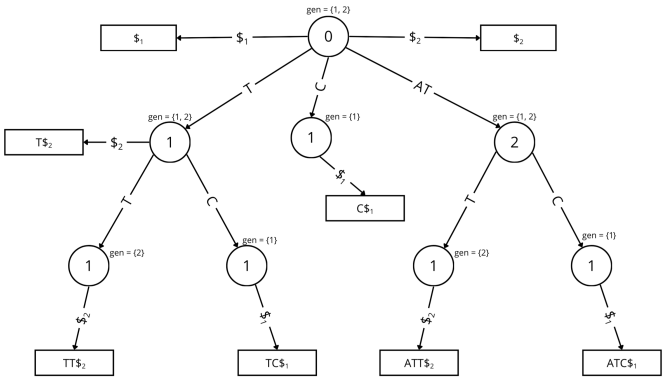


Figura 2. Árbol de sufijos generalizado (GST) para G_1 = `ATC` y G_2 = `ATT`. *Aclaración.* En la figura, la etiqueta `gen` denota el conjunto de colores $C(v)$. Elaboración propia.

H. Cálculo de colores y complejidad

Para cada nodo v del GST, el conjunto de colores $C(v)$ se obtiene en postorden uniendo los colores de sus hijos (p. ej., con bitsets por genoma). Con N la longitud total concatenada y alfabeto acotado, la construcción del GST y el recorrido para $C(v)$ y $d(v)$ son $O(N)$ en el modelo estándar. La selección de dSUS consiste en examinar cada arista $u \rightarrow v$ y, cuando $|C(u)| > 1$ y $C(v) = \{i\}$, actualizar la mejor longitud para G_i con $d(u)+1$ (lineal en el número de aristas).

I. Convenciones de reporte y empates

Usamos sentinelas sólo para construir el GST; las dSUS se reportan sobre Σ (sin sentinelas). Si existen varias dSUS de igual longitud para un mismo G_i, por defecto reportamos la lexicográficamente menor; en aplicaciones donde importa la localización, se puede romper el empate por la posición más a la izquierda en G_i. Cuando indiquemos dSUS por posición, "cubre la posición p" significa que el intervalo del substring incluye p (no exigimos que comience en p).

J. Notas genómicas mínimas

Para ADN, Σ = {`A`, `C`, `G`, `T`}. Los símbolos ambiguos como `N` deben tratarse de forma consistente. Si se requiere unicidad con respecto a ambas hebras, se indexa también la reverso–complementaria de cada G_i con sentinelas propios. En colecciones con genomas muy similares, la longitud de la dSUS puede crecer de forma notable por la profundidad del primer cambio de color en el GST [13], [14], [15], [16].

III. RMQ + LCS vs GST

Además del GST, existen alternativas como *SA+LCP+RMQ*, índices comprimidos y el *FM-index*. A continuación compararemos en busca de ver el rendimiento.

Cuadro I
RMQ+LCS vs. GST (Generalized Suffix Tree)

Aspecto	RMQ + LCS	GST
Estructura	SA de cadenas concatenadas con sentinelas, LCP y RMQ sobre LCP; estructuras planas basadas en arreglos.	Árbol de sufijos multi-cadena; aristas como rangos en el texto; sentinela por cadena y marcas por documento.
Preprocesamiento	SA en $O(N)$–$O(N \log N)$, LCP $O(N)$, RMQ $O(N)$ con consultas $O(1)$.	Construcción $O(N)$ (Ukkonen, alfabeto acotado); otras variantes $O(N \log \sigma)$ o más si son ingenuas.
Consulta LCS	Filtra sufijos de distintas cadenas y aplica RMQ en LCP; casi $O(1)$ por consulta tras el preprocesado; el LCS global con un barrido lineal del LCP.	Nodo más profundo con ambas marcas (1&2); tras construir el GST, se halla en $O(N)$ con un DFS.
Memoria	$O(N)$ con constantes bajas y buena caché; existen variantes comprimidas (CSA/FM-index).	$O(N)$ con constantes altas (muchos nodos/punteros); evitar guardar cadenas, usar índices (inicio/fin).
Implementación	Más directa y estandarizada (SA+LCP+RMQ) con bibliotecas optimizadas.	Más delicada: Ukkonen, enlaces de sufijo y particiones para mantener $O(N)$.
Dinámico / en línea	Poco flexible ante inserciones: suele requerir reconstrucción (ideal para lotes estáticos).	Naturalmente *online*: Ukkonen añade caracteres incrementalmente.
Caché	Muy buena (arreglos contiguos).	Peor (múltiples punteros); puede rendir menos en grandes escalas.
Multi-documento	Concatenación con sentinelas distintos y etiqueta por sufijo.	Marcado explícito de nodos/hojas por documento para substrings comunes.

IV. Aplicación y comparación de los usos del Suffix Tree

A. Usos del suffix tree

- **Búsqueda exacta de patrones:** Tras preprocesar el texto T en tiempo lineal con un árbol de sufijos, un patrón p se localiza en $O(|p| + k)$ (siendo k el número de ocurrencias) recorriendo un único camino y enumerando hojas en el subárbol alcanzado [1], [2].
- **Conteo y listado de posiciones:** El nodo alcanzado al seguir p determina directamente el número de apariciones (hojas) y sus índices en T, útil en análisis de logs, documentos o secuencias biológicas [1].
- **LCS y comparación de cadenas (GST):** Con un árbol de sufijos generalizado (GST) para S_1 y S_2, marcar hojas por cadena permite hallar substrings comunes; el más largo se obtiene en tiempo lineal tras construir el árbol [1], [12].
- **Índices comprimidos y búsqueda a gran escala:** En corpora grandes, estructuras como el FM-index y arreglos de sufijos mejorados (ESA/SA+LCP) permiten búsquedas eficientes con menor huella de memoria que los árboles explícitos [7], [6], [18].
- **Casos prácticos:**
  - Listas/logs y textos largos → presencia y conteo de términos [1].
  - Corpus académicos → LCS/similitud con GST [1], [12].
  - Bioinformática y búsqueda masiva → ESA/FM-index para consultas a gran escala [18], [6], [7].

B. Limitaciones

- **Memoria:** Aunque el espacio es $O(N)$, las constantes son altas por nodos y punteros; se recomienda etiquetar aristas con rangos (índices) en T y usar un centinela único [1], [6].
- **Construcción:** Existen algoritmos lineales clásicos (Weiner, McCreight, Ukkonen; y variantes para alfabetos grandes) pero su implementación es compleja; enfoques ingenuos pueden degradarse a superlineales [8], [9], [2], [10], [11].
- **Eficiencia práctica:** La menor localidad de caché suele perjudicar a los árboles frente a ESA/SA y a índices comprimidos en corpora grandes [18], [6], [7].
- **Coste de consulta y reporte:** Comprobar un patrón de longitud M cuesta $O(M)$; listar todas las apariciones añade $+k$, quedando $O(M + k)$ [1].

C. Comparativa

Según los experimentos de [18], la Figura 3 compara *enumeration queries* (1M patrones, long. mín. 30–40) entre un enfoque ESA (SA+LCP+RMQ) y un GST en varios corpus. Se observa que ESA logra sistemáticamente menores tiempos que GST en alfabetos pequeños (E. coli, Yeast, Hs21, Swissprot), mientras que en *Shaks* (alfabeto grande) los arreglos simples dominan. La Figura 4 muestra *matching statistics* en pares de genomas: GST tiende a ser más rápido en tiempo, mientras que ESA usa menos memoria, ilustrando el compromiso tiempo–espacio según la tarea [18].

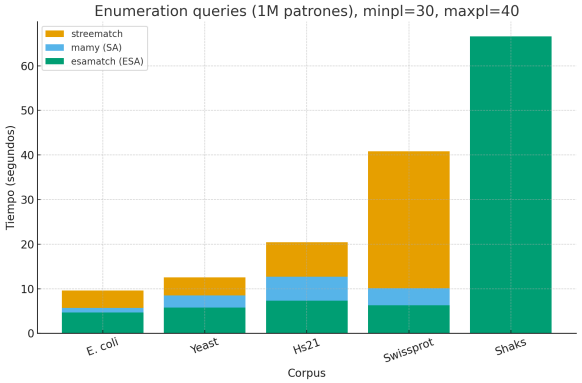


Figura 3. *Enumeration queries* (1M patrones, minpl=30–40) en cinco corpus; menores tiempos para ESA (RMQ+LCS) en alfabetos pequeños; excepción en *Shaks* (alfabeto grande), según [18].

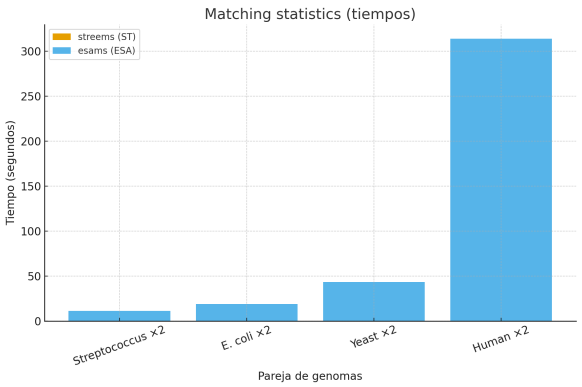


Figura 4. *Matching statistics* en pares de genomas: GST es más rápido en tiempo, mientras que ESA reduce memoria; compromiso tiempo–memoria, según [18].

V. Estructura de datos y algoritmos involucrados

A. Entrada y notación

Sean $G_1, \ldots, G_k$ genomas sobre $\Sigma = \{A, C, G, T\}$, con longitudes $n_1, \ldots, n_k$ y $N = \sum_{i=1}^{k} n_i$. Construimos el texto concatenado

$$S = G_1\$_1\, G_2\$_2 \cdots G_k\$_k$$

donde cada $\$_i \notin \Sigma$ y $\$_i \neq \$_j$ si $i \neq j$. Objetivo: para cada i, la **dSUS global** de G_i (se reporta sobre Σ, ignorando sentinelas).

B. Estructuras de datos (GST explícito)

Aristas etiquetadas por intervalos $[\ell..r]$ de S.

- `Node`: `children` (mapa char→hijo), `suffixLink`, `parent`, `edgeL`, `edgeR`, `depth` $d(v)$, `color` $C(v)$ (bitset $\{1, \ldots, k\}$), `repUnique` (si $|C(v)| = 1$).
- `Leaf`: además `start` (inicio del sufijo en S), `docID` y `posInDoc`.

Arreglos auxiliares: `docOfPos`, `posInDoc`.

C. Construcción del GST (Ukkonen, integrado)

Usamos Ukkonen en línea sobre S [2], [1]. Dividimos en: *estado e invariantes*, *primitivas*, *bucle principal*.

C1. Estado e invariantes: Mantenemos el punto activo, el contador de sufijos pendientes y un extremo global para hojas; tras cada fase, el árbol representa todos los sufijos que terminan en $S[i]$.

- **Punto activo**: `activeNode`, `activeEdge` (índice en S), `activeLen`.
- **Remainder**: número de sufijos pendientes por insertar en la fase i.
- **end** global: extremo derecho compartido por hojas (hojas extensibles).
- **lastNewInternal**: último nodo interno creado en la fase (para encadenar `suffixLink`).

C2. Primitivas (pseudocódigo breve): WalkDown: consume aristas completas para avanzar el punto activo en $O(1)$ amortizado.

Algorithm 1 WalkDown

```
function WalkDown(next)
    edgeLen ← next.edgeR − next.edgeL + 1
    if activeLen ≥ edgeLen then
        activeEdge ← activeEdge + edgeLen
        activeLen ← activeLen − edgeLen
        activeNode ← next
        return true
    else
        return false
```

NewLeaf: crea y retorna una hoja $[\,i\,..\,\texttt{end}\,]$ colgándola desde `from` (Regla 2).

Algorithm 2 NewLeaf

```
function NewLeaf(from, i)
    return CreateLeaf(from, [ i .. end ])
```

SplitEdge: divide la arista hacia `next` tras `activeLen` caracteres y retorna el nodo interno (Regla 2).

Algorithm 3 SplitEdge

```
function SplitEdge(next)
    split ← CreateInternalAfter(next, activeLen)
    Relabel(next, [next.edgeL + activeLen .. next.edgeR])
    return split
```

C3. Extender una fase: Inserta $S[i]$ aplicando Reglas 2/3; actualiza enlaces de sufijo y el punto activo (incluye *root-trick*).

Algorithm 4 Extend(i) *// añade $S[i]$*

```
end← i; remainder++; lastNewInternal← null
while remainder > 0 do
    if activeLen = 0 then activeEdge← i
    if no existe hijo next de activeNode con letra S[activeEdge] then   ▷ Regla 2 (nueva hoja)
        NewLeaf(activeNode, i)
        if lastNewInternal ≠ null then lastNewInternal.suffixLink←activeNode
        lastNewInternal← null
    else
        if WalkDown(next) then continue
        if S[next.edgeL + activeLen] = S[i] then   ▷ Regla 3 (punto final)
            activeLen++
            if lastNewInternal ≠ null then lastNewInternal.suffixLink←activeNode
            break
        split ← SplitEdge(next) ▷ Regla 2 (división)
        NewLeaf(split, i)
        if lastNewInternal ≠ null then lastNewInternal.suffixLink← split
        lastNewInternal← split
    remainder–
    if activeNode = root and activeLen > 0 then ▷ root-trick
        activeLen–; activeEdge← i − remainder + 1
    else
        activeNode← (activeNode.suffixLink si existe, si no root)
```

C4. Bucle principal de Ukkonen y etiquetado de hojas: Aplica EXTEND a cada posición de S y, al final, anota metadatos (docID, posInDoc) en hojas.

Algorithm 5 Construir-GST-Ukkonen($G_1, \ldots, G_k$)

```
construir S ← G1$1 ··· Gk$k; poblar docOfPos, posInDoc
init: root, activeNode←root, activeEdge← −1, activeLen← 0, remainder← 0, end← 0
for i ← 1 to |S| do
    Extend(i)
for all hoja x do
    x.start ← inicio de su sufijo; x.docID ← docOfPos[x.start]; x.posInDoc ← posInDoc[x.start]
return root
```

Complejidad. Construcción $O(N)$ tiempo y espacio con alfabeto acotado u operaciones de hijo amort. $O(1)$; $O(N \log \sigma)$ con mapas balanceados y alfabeto de tamaño σ [2], [1].

D. Coloración y profundidades (postorden)

Postorden que computa $d(v)$ y $C(v)$ como OR de colores de hijos; si $|C(v)| = 1$ guardamos una hoja representativa (repUnique).

Algorithm 6 Colorear-y-Profundidades(root)

```
function DFS(v, dParent)
    v.depth ← dParent + (v.edgeR − v.edgeL + 1)
    if v es hoja then v.color ← Bit(v.docID)
    elsev.color ← 0;
        for all u hijo do DFS(u, v.depth); v.color ← v.color OR u.color
    v.repUnique ← (∃ hoja si Popcount(v.color) = 1, si no null)
DFS(root, 0)
```

Complejidad. $O(N + N \cdot k/w)$ con bitsets (lineal cuando k es pequeño).

E. Selección de dSUS global

Recorrido de aristas; un candidato aparece al cambiar de un padre con varios colores a un hijo con un único color.

Algorithm 7 Seleccionar-dSUS-Global(root, k)

```
bestLen[1..k]← ∞, bestLeaf[1..k]← null, bestParentDepth[1..k]← −1
for all arista u→v do
    if Popcount(v.color) = 1 and Popcount(u.color) > 1 then
        i ← IdxSingleton(v.color); ℓ ← u.depth + 1
        if ℓ < bestLen[i] then
            bestLen[i]← ℓ; bestLeaf[i]← v.repUnique; bestParentDepth[i]← u.depth
return (bestLen, bestLeaf, bestParentDepth)
```

Complejidad. Recorrido $O(N)$; operaciones de bitset $O(1)$ con conteos cacheados.

F. Resumen de complejidades

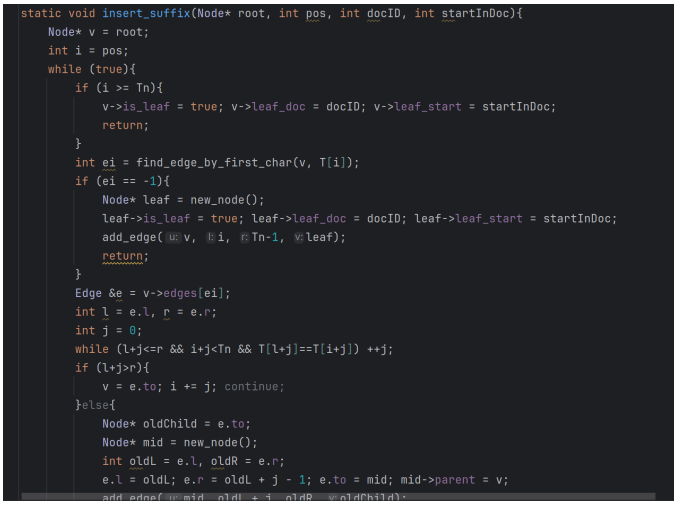
Construcción GST (Ukkonen): $O(N)$; coloreo: $O(N + N \cdot k/w)$; selección dSUS: $O(N)$; memoria total $O(N + N \cdot k/w)$.

VI. Ejemplos de código

En esta sección se ilustran fragmentos de implementación del *Árbol de Sufijos Generalizado* (GST) usado en el trabajo. Mostramos *imágenes* de los listados y un breve contexto técnico de cada uno.

A. Construcción del GST (Ukkonen, trie comprimido/Patricia)

- **Entrada:** texto concatenado $S = G_1\$_1\, G_2\$_2\, \ldots\, G_k\$_k$ con sentinelas únicos $\$_i$.
- **Idea:** Ukkonen en línea con *punto activo* (activeNode, activeEdge, activeLen), *remainder* y *end* global para



```
static void insert_suffix(Node* root, int pos, int docID, int startInDoc){
    Node* v = root;
    int i = pos;
    while (true){
        if (i >= Tn){
            v->is_leaf = true; v->leaf_doc = docID; v->leaf_start = startInDoc;
            return;
        }
        int ei = find_edge_by_first_char(v, T[i]);
        if (ei == -1){
            Node* leaf = new_node();
            leaf->is_leaf = true; leaf->leaf_doc = docID; leaf->leaf_start = startInDoc;
            add_edge( u: v,  l: i,  r: Tn-1,  v: leaf);
            return;
        }
        Edge &e = v->edges[ei];
        int l = e.l, r = e.r;
        int j = 0;
        while (l+j<=r && i+j<Tn && T[l+j]==T[i+j]) ++j;
        if (l+j>r){
            v = e.to; i += j; continue;
        }else{
            Node* oldChild = e.to;
            Node* mid = new_node();
            int oldL = e.l, oldR = e.r;
            e.l = oldL; e.r = oldL + j - 1; e.to = mid; mid->parent = v;
            add_edge( u: mid,  oldL + j,  oldR,  v: oldChild);
```

Figura 5. Rutinas centrales de Ukkonen (`Extend`, `SplitEdge`, `NewLeaf`) con aristas etiquetadas por intervalos de S.

```
static void dfs_accumulate(Node* u){
    if (u->occ.empty()) u->occ.assign(numDocs,  val: 0);
    for (auto &e :Edge & : u->edges){
        dfs_accumulate(e.to);
        for (int d=0; d<numDocs; ++d) u->occ[d] += e.to->occ[d];
    }
    if (u->is_leaf && u->leaf_doc>=0) u->occ[u->leaf_doc] += 1;
}
```

Figura 6. DFS postorden para calcular $C(v)$ por OR de bitsets y $d(v)$; se marca `repUnique` cuando $|C(v)| = 1$.

hojas; creación de hojas y *SplitEdge* cuando hay *mismatch*. Los sentinelas aseguran árbol *explícito*.

- **Detalles:** cada arista guarda índices $[\ell, r]$ sobre S (no se copian cadenas); cada hoja guarda `docID` y `posInDoc`; se mantienen *enlaces de sufijo*.

B. Coloración por documento: conjuntos $C(v)$ (bitsets)

- **Objetivo:** para cada nodo v, almacenar $C(v) \subseteq \{1, \ldots, k\}$, el conjunto de genomas presentes bajo v (un bit por `docID`).
- **Método:** DFS en postorden: en hojas, fijar el bit de su `docID`; en internos, $C(v) = \bigcup C(\text{hijo})$ (OR de bitsets). En el mismo recorrido se calcula la *profundidad de cadena* $d(v)$. Si Popcount($C(v)$)=1, se guarda una hoja representativa (`repUnique`).
- **Uso posterior:** *unicidad exacta* (nuestro marco): $S(v)$ es exclusivo de $G_i \Leftrightarrow C(v) = \{i\}$. No se usan vectores de conteos por documento.

C. Selección de dSUS (global y por posición)

- **Global por genoma G_i:** recorrer todas las aristas $u \to v$; si $|C(u)| > 1$ y $C(v) = \{i\}$, hay un candidato de longitud $d(u) + 1$. Quedarse con la mínima longitud; reconstruir la coordenada desde `repUnique` bajo v.
- **(Opcional) Por posición p en G_t:** seguir el sufijo $G_t[p..]$; al llegar a una arista $u \to v$ con $|C(u)| > 1$ y

```
Ingrese N (numero de genomas):
3
Genoma[0]:
act
Genoma[1]:
abt
Genoma[2]:
ayt
Genoma target (debe coincidir con uno de los anteriores):
abt
b
```

Figura 7. Selección de dSUS: (izq.) global escaneando aristas; (der.) por posición siguiendo el sufijo y tomando el primer cambio de color.

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie( tiestr nullptr);

    cout << "Ingrese N (numero de genomas): " << endl;
    int N;
    if (!(cin >> N)) return 1;
    if (N <= 0) return 1;

    vector<string> genomes(N);
    for (int i = 0; i < N; ++i) {
        cout << "Genoma[" << i << "]: " << endl;
        if (!(cin >> genomes[i])) return 1;
    }

    cout << "Genoma target (debe coincidir con uno de los anteriores): " << endl;
    string target;
    if (!(cin >> target)) return 1;

    int targetID = -1;
    for (int i = 0; i < N; ++i) {
        if (genomes[i] == target) { targetID = i; break; }
    }
    if (targetID < 0) {
        cout << "ERROR: target no coincide con los genomas dados." << endl;
ain
```

Figura 8. `main` minimalista: construcción del GST, postorden de colores y selección de dSUS.

$C(v) = \{t\}$, la dSUS que *cubre* p tiene longitud $d(u) + 1$ (primer cambio de color). Reportar sobre Σ (ignorando sentinelas).

- **Salida:** string mínimo y su intervalo en G_i; en la variante por posición, el string que cubre p.

D. Interfaz principal (I/O mínima)

- **Flujo:** leer k y los genomas $G_1, \ldots, G_k$; construir una vez el GST sobre S; postorden para $C(v)$ y $d(v)$; ejecutar selección de dSUS global para cada G_i (y, si se desea, consultas por posición para un G_t); imprimir la cadena mínima identificadora y coordenadas.
- **Complejidad:** construcción $O(N)$, coloreo $O(N+N \cdot k/w)$ (bitsets), selección $O(N)$.

VII. Conclusiones y trabajo futuro

A. Conclusiones

- Se construyó un **árbol de sufijos generalizado (GST)** sobre $S = G_1\$_1 \cdots G_k\$_k$, con aristas etiquetadas por intervalos $[\ell..r]$ de S y hojas anotadas con `(docID, posInDoc)`.

- Un postorden computa para cada nodo v la **profundidad de cadena** $d(v)$ y el **conjunto de colores** $C(v)$ (bitset en $\{1, \ldots, k\}$). Se usa el criterio: $S(v)$ es exclusivo de G_i si y sólo si $C(v) = \{i\}$.
- La **dSUS global de** G_i se obtiene escaneando aristas $u \to v$ con $|C(u)| > 1$ y $C(v) = \{i\}$, tomando la mínima longitud $d(u) + 1$ y reconstruyendo la coordenada desde una hoja representativa bajo v. Tras la construcción, el proceso completo es $O(N)$.
- Búsqueda de pertenencia de un patrón P en el GST toma $O(|P|)$ siguiendo aristas por etiquetas en S (propiedad útil para validación).
- (Opcional) La *dSUS por posición* se obtiene restringiendo la selección a la ruta del sufijo que *cubre* la posición en G_i; no es el foco principal de este trabajo.

B. Limitaciones

- El GST explícito requiere memoria $O(N)$ para nodos/aristas más $O(N \cdot k/w)$ bits para colores (w: tamaño de palabra). No se abordan estructuras comprimidas ni técnicas de muestreo.
- El criterio de unicidad es **exacto** sobre Σ (sin tolerancias ni desajustes). El manejo de símbolos ambiguos (`N`) y reverso–complementarias queda fuera del alcance principal.
- Si no existe substring exclusivo de G_i (por ejemplo, G_i es idéntico a algún G_j), entonces **no existe dSUS global** para G_i.

C. Trabajo futuro

- Optimizar el cálculo de $C(v)$ mediante bitsets de palabra y POPCOUNT vectorizado; paralelizar el postorden por subárboles.
- Implementar consultas de dSUS por posición/ventana apoyadas en enlaces de sufijo para compartir trabajo entre loci contiguos.
- Extender el índice para considerar reverso–complementarias y/o un tratamiento sistemático de símbolos ambiguos (`N`), manteniendo la semántica de dSUS.
- Añadir utilidades de depuración/visualización del GST (enlaces de sufijo, $d(v)$, $C(v)$) para colecciones grandes.

D. Trabajo futuro

- **Optimización:** acelerar el postorden de $C(v)$ con bitsets de palabra y POPCOUNT vectorizado; paralelizar por subárboles.
- **Consultas por posición/ventana:** extender la selección de dSUS restringida a rutas de sufijos y ventanas deslizantes, reutilizando enlaces de sufijo.
- **Muestreo/ruido:** explorar una *tolerancia de repetición* $\tau > 0$ para datos con ruido o muestreo, lo que implicaría pasar de sólo $C(v)$ a contadores por genoma y reglas basadas en frecuencias (fuera del alcance del marco actual).
- **Aspectos biológicos:** incorporar reverso–complementarias y políticas para símbolos ambiguos (`N`) manteniendo el mismo pipeline de GST+dSUS.

Referencias

[1] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*. Cambridge Univ. Press, 1997.

[2] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, 14, pp. 249–260, 1995.

[3] J. Pei, W. C.-H. Wu, and M.-Y. Yeh, "On Shortest Unique Substring Queries," in *Proc. IEEE ICDE*, pp. 937–948, 2013.

[4] S. V. Thankachan et al., "Range Shortest Unique Substring Queries," CWI Tech. Rep., 2019.

[5] P. Abedin et al., "Efficient Data Structures for Range SUS Queries," *Algorithms*, 13(11):276, 2020.

[6] G. Navarro, *Compact Data Structures*. Cambridge Univ. Press, 2016.

[7] P. Ferragina and G. Manzini, "Indexing compressed text," *Journal of the ACM*, 52(4), pp. 552–581, 2005.

[8] P. Weiner, "Linear pattern matching algorithms," in *Proc. 14th IEEE Symp. on Switching and Automata Theory (SWAT)*, pp. 1–11, 1973.

[9] E. M. McCreight, "A space-economical suffix tree construction algorithm," *J. ACM*, 23(2), pp. 262–272, 1976.

[10] M. Farach, "Optimal suffix tree construction with large alphabets," in *Proc. 38th IEEE FOCS*, pp. 137–143, 1997.

[11] R. Giegerich and S. Kurtz, "From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction," *Algorithmica*, 19(3), pp. 331–353, 1997.

[12] L. C. K. Hui, "Color set size problem with applications to string matching," in *Proc. CPM*, LNCS 644, pp. 230–243, 1992.

[13] B. Haubold, N. Pierstorff, F. Möller, and T. Wiehe, "Genome comparison without alignment using shortest unique substrings," *BMC Bioinformatics*, 6:123, 2005.

[14] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, "Alignment of whole genomes," *Nucleic Acids Research*, 27(11), pp. 2369–2376, 1999.

[15] A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg, "Fast algorithms for large-scale genome alignment and comparison," *Nucleic Acids Research*, 30(11), pp. 2478–2483, 2002.

[16] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg, "Versatile and open software for comparing large genomes," *Genome Biology*, 5(2):R12, 2004.

[17] S. Kurtz, J. V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich, "REPuter: the manifold applications of repeat analysis on a genomic scale," *Nucleic Acids Research*, 29(22), pp. 4633–4642, 2001.

[18] Abouelhoda, M. I., Kurtz, S., & Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms, 2*(1), 53–86.