UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MAPEO DE SECUENCIAS DE ADN A ÁRBOLES FILOGENÉTICOS USANDO COMPRESIÓN DE GRAMÁTICAS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

VICTOR ANTONIO ALFARO PINO

PROFESOR GUÍA:
GONZALO NAVARRO

MIEMBROS DE LA COMISIÓN:
PATRICIO POBLETE
ARTURO MERINO F.

SANTIAGO DE CHILE
2026

RESUMEN DE LA MEMORIA PARA OPTAR AL
TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: VICTOR ANTONIO ALFARO PINO
FECHA: 2026
PROF. GUIA: GONZALO NAVARRO

MAPEO DE SECUENCIAS DE ADN A ÁRBOLES FILOGENÉTICOS USANDO COMPRESIÓN DE GRAMÁTICAS

En el contexto de la clasificación taxonómica, uno de los problemas centrales es determinar de qué organismo proviene un fragmento de ADN (*read*). Para ello, se cuenta con una colección de genomas organizados en un árbol filogenético, donde las hojas representan genomas y los nodos internos representan ancestros comunes. Los genomas se concatenan en el orden de las hojas del árbol, y cuando un patrón (o MEM, una coincidencia exacta maximal entre el *read* y la colección) aparece en varios genomas, es necesario identificar el primero y el último genoma donde ocurre, para luego calcular el ancestro común más bajo (LCA) en el árbol y así clasificar el *read*. Este problema se traduce, a nivel de estructuras de datos, en encontrar el mínimo y el máximo del arreglo de sufijos (SA) dentro de un intervalo: dado un intervalo [sp, ep] que representa todas las ocurrencias de un MEM, min(SA[sp..ep]) indica la posición más a la izquierda en el texto (primer genoma) y max(SA[sp..ep]) indica la posición más a la derecha (último genoma). El enfoque directo, utilizado por índices como el sr-index, consiste en recorrer, una por una, las $ep - sp + 1$ posiciones para encontrar el mínimo y el máximo, con un costo proporcional al número de ocurrencias. Para MEMs con miles o millones de ocurrencias, este costo se vuelve significativo. Este trabajo propone un enfoque alternativo: construir un índice basado en la compresión gramatical del arreglo diferencial de sufijos (DSA) mediante el algoritmo Repair, almacenando en cada nodo de la gramática información agregada (metadata) que permite responder las consultas de mínimo y máximo descendiendo por la estructura de la gramática, sin enumerar las ocurrencias. El costo de cada consulta depende de la altura de la gramática y no del número de ocurrencias, lo que representa una ventaja sustancial en colecciones repetitivas. La viabilidad de este enfoque se sustenta en que el DSA de textos repetitivos, como colecciones de genomas, tiene una estructura altamente repetitiva inducida por las runs de la BWT, lo que permite que Repair lo comprima eficientemente. Las ideas, lemas y algoritmos que fundamentan esta construcción fueron adaptados a partir de los resultados expuestos en el paper *Fully-Functional Suffix Trees and Optimal Text Searching in BWT-runs Bounded Space*, el cual fue escrito por Gonzalo Navarro, Travis Gagie y Nicola Prezza. El índice propuesto se compara experimentalmente contra el sr--index en términos de tiempo de consulta y espacio ocupado. Los resultados se presentan mediante gráficos y tablas comparativas que evidencian el *trade-off* entre ambos enfoques.

Declaración de Uso de Inteligencia Artificial

En la elaboración de esta memoria se utilizaron herramientas de inteligencia artificial (IA), específicamente Claude Opus y Claude Sonnet. Estas fueron utilizadas como apoyo para la redacción del informe y para obtener borradores de código e ideas para la implementación, proceso que se describe en detalle en el Capítulo 6: Uso de inteligencia artificial. Toda la ayuda proporcionada por la IA fue revisada y validada por el autor.

Para mis padres.

Agradecimientos

Primero quiero agradecer a mis padres, quienes siempre estuvieron para mí cuando más lo necesité. Nunca me faltaron en toda mi vida, y me apoyaron en cada uno de mis proyectos personales y mis decisiones, siempre aconsejándome. También quiero agradecer a mi familia, mis primos, mi abuela, mis tíos, todos. Forman parte de mi vida y siempre desearon lo mejor para mí.

Quiero también agradecer al profesor Gonzalo por darme la oportunidad de trabajar en esta memoria, por ayudarme cuando tuve dudas y por tener siempre la disposición para ayudarme cada vez que lo necesité.

También quiero agradecer a mis amigos de la universidad, en particular al gran grupo que tuve en el DCC (panas). Cada uno me aportó algo en mi vida. Extrañaré por siempre los momentos juntos en la facultad. Especial mención al champi, fue mi mejor amigo en la u y siempre me apoyó en todo.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Contexto y problemática

El estudio de la evolución de las especies es uno de los objetivos principales de la bioinformática, una disciplina que combina biología e informática para almacenar y analizar grandes volúmenes de datos biológicos como secuencias de ADN, aminoácidos y demás moléculas biológicas. Un gen es un segmento de ADN, y un genoma es el conjunto de la información genética de un organismo. Teniendo en cuenta la gran cantidad de especies existentes y la gran cantidad de colecciones de genomas disponibles, resulta desafiante responder preguntas como: ¿Dónde aparece una secuencia particular dentro de un genoma? ¿Cuántas veces aparece una secuencia en un conjunto de genomas?

Un árbol filogenético es una estructura que permite representar las relaciones de parentesco entre especies y que muestra los cambios en los genes con el paso del tiempo. Una colección de genomas se organiza según esta estructura y la problemática principal surge a partir de esto: se busca una secuencia de ADN o un gen en esta colección, y se quiere identificar todas sus ocurrencias, mapearlas a los genomas que las contienen y calcular el ancestro común más bajo (Lowest Common Ancestor, LCA) de esos nodos, el cual es un candidato a la especie donde se originó este gen. Este procedimiento —identificar el genoma o los genomas de origen de una secuencia dada— es lo que se conoce como clasificación taxonómica, y la secuencia que se busca clasificar suele denominarse *read*. Lo difícil radica en que las secuencias no suelen encontrarse exactamente debido a mutaciones, inserciones o recombinaciones. Por lo tanto, la búsqueda se basa en encontrar las coincidencias exactas maximales, que se conocen como Maximal Exact Matches (MEMs). Calcular MEMs requiere estructuras de datos como el Suffix Array (SA) y el FM-index, entre otras que se definirán más adelante.

Los genomas de la colección se concatenan en un único texto, siguiendo el orden de las hojas del árbol filogenético de izquierda a derecha. Cuando un MEM tiene múltiples ocurrencias en la colección, estas se distribuyen a lo largo de esta concatenación, y sus posiciones en el texto reflejan la posición de los genomas correspondientes en el árbol. La ocurrencia más a la izquierda en el texto (el mínimo del arreglo de sufijos en el intervalo del MEM) corresponde al primer genoma en el orden del árbol donde aparece el MEM, y la ocurrencia más a la derecha (el máximo) corresponde al último genoma. El ancestro común más bajo (LCA) de estas dos hojas extremas necesariamente contiene a todas las hojas intermedias, porque las hojas de cualquier subárbol forman un rango contiguo en el orden de izquierda a derecha. Por lo tanto, no es necesario examinar todas las ocurrencias para determinar el LCA: basta con encontrar la primera y la última. Formalmente, dado el intervalo [sp, ep] del arreglo de sufijos que contiene las posiciones de todas las ocurrencias de un MEM, el problema se reduce a calcular la posición en el texto de la ocurrencia más a la izquierda $\min(\text{SA}[\text{sp..ep}])$ y más a la derecha $\max(\text{SA}[\text{sp..ep}])$.

Estas dos consultas corresponden a lo que se conoce como Range Minimum Query (RMQ) y Range Maximum Query (RMaxQ) sobre el arreglo de sufijos. Una RMQ sobre un arreglo $A[1..n]$ consiste en, dado un rango $[i, j]$, encontrar el valor mínimo de A en ese rango: $\text{RMQ}(i, j) = \min(A[i], A[i + 1], ..., A[j])$. La RMaxQ es análoga pero busca el máximo.

El sr-index (subsampled r-index), presentado por Dustin Cobas, Travis Gagie y Gonzalo Navarro [1], es un índice comprimido diseñado para colecciones de texto altamente repetitivas. Su espacio es proporcional a r, el número de runs de la Transformada de Burrows-Wheeler (BWT) —la BWT es una permutación de los caracteres del texto que tiende a agrupar caracteres iguales en bloques consecutivos, obtenida al ordenar lexicográficamente todas las rotaciones del texto y tomar su última columna; una run es uno de esos bloques maximales de caracteres iguales (se profundiza en la sección 2.4)—, lo que lo hace muy compacto cuando la colección tiene alta repetitividad. El sr-index soporta la operación de localización (*locate*): dado un intervalo [sp, ep] producido por el *backward search* —el procedimiento que busca un patrón de derecha a izquierda sobre la BWT y calcula el rango [sp, ep] del arreglo de sufijos donde ocurren sus apariciones—, puede recuperar las posiciones SA[sp], SA[sp+1], …, SA[ep] en el texto. Sin embargo, para obtener el mínimo y el máximo, el sr-index debe enumerar todas las $\text{ep} - \text{sp} + 1$ posiciones una por una (mediante aplicaciones sucesivas de la función Φ, que a partir de una posición conocida SA[p] permite obtener SA[p-1] o SA[p+1]) y luego recorrerlas linealmente para encontrar los extremos. El costo de este procedimiento es $O(s \cdot \text{occ})$, donde $\text{occ} = \text{ep} - \text{sp} + 1$ es el número de ocurrencias y s es un parámetro de submuestreo. Para MEMs con miles o millones de ocurrencias, este costo lineal se vuelve un cuello de botella.

Una gramática libre de contexto es un conjunto de reglas de reemplazo. Tiene dos tipos de símbolos: los terminales (valores finales) y los no-terminales (nombres que representan patrones). Cada regla indica los símbolos que reemplazan un no-terminal dado.

Partiendo de un símbolo inicial, se aplican las reglas hasta que no queden no-terminales, y lo que queda es la cadena que la gramática genera. La compresión por gramáticas es un caso particular donde la gramática genera exactamente una cadena (la secuencia original). Dado el siguiente ejemplo:

$$S \to \text{AB}, \quad A \to \text{hola}, \quad B \to \text{mundo}$$

Partiendo de S, se reemplazan A y B: $S \to$ hola mundo

Esta restricción —que cada no-terminal tenga exactamente una regla y que no haya ciclos— garantiza que la gramática genere una única cadena. Cada no-terminal captura un patrón repetido de la secuencia: en vez de almacenar todas sus ocurrencias, se almacena una sola regla que lo define, y se referencia por su nombre. El tamaño de la gramática (la suma de símbolos en los lados derechos de todas las reglas) es la medida de compresión, y para secuencias altamente repetitivas puede ser dramáticamente menor que la longitud original.

El arreglo diferencial de sufijos (DSA) almacena las diferencias entre valores consecutivos del arreglo de sufijos:

$$\text{DSA}[1] = \text{SA}[1]$$

$$\text{DSA}[p] = \text{SA}[p] - \text{SA}[p-1] \text{ para todo } p \geq 2$$

Los valores originales del SA se pueden reconstruir a partir del DSA mediante sumas parciales: SA[p] = SA[q] + Σ desde j=q+1 hasta p de DSA[j], para cualquier posición de referencia $q < p$. Esta propiedad permite expresar las consultas de mínimo y máximo como:

$$\min(\text{SA}[\text{sp..ep}]) = \text{SA}[\text{sp} - 1] + \text{mín de las sumas parciales de DSA[sp..ep]}$$

$$\max(\text{SA}[\text{sp..ep}]) = \text{SA}[\text{sp} - 1] + \text{máx de las sumas parciales de DSA[sp..ep]}$$

El DSA es útil porque, para colecciones de texto altamente repetitivas, hereda la estructura repetitiva de la BWT. Travis Gagie et al. [2] demostraron que si dos posiciones consecutivas del arreglo de sufijos están dentro de una misma run de la BWT (es decir, BWT[p−1] = BWT[p]), entonces DSA[LF(p)] = DSA[p], donde LF(p) es el mapeo Last-to-First: la función que, dada una posición p en la BWT, entrega la posición que ese mismo carácter ocupa en la primera columna de la matriz de rotaciones ordenadas del texto; es decir, el mismo valor del DSA aparece copiado en otra posición. Como consecuencia, el DSA admite un esquema de macros bidireccional de tamaño $O(r)$ —una representación en que el arreglo queda cubierto por bloques que son copias exactas de otros bloques, más un conjunto de solo $O(r)$ bloques «primarios» no repetidos—, lo que implica que puede representarse mediante una gramática libre de contexto de tamaño $O\big(r \log\big(\frac{n}{r}\big)\big)$, siendo n la longitud de la concatenación de genomas (se profundiza en la sección 2.1). Para colecciones genómicas donde r es mucho menor que n, esto permite una compresión dramática del DSA, y al almacenar metadatos de sumas parciales (mínimo y máximo) en cada no-terminal de la gramática, se pueden responder las consultas de RMQ y RMaxQ sobre el SA sin enumerar las ocurrencias, en tiempo proporcional a la altura de la gramática.

Se ha demostrado que la compresión por gramáticas es una herramienta muy efectiva para representar secuencias diferenciales altamente repetitivas, como el arreglo diferencial de LCP (Longest Common Prefix, que almacena, para cada par de sufijos consecutivos en el orden lexicográfico del arreglo de sufijos, cuántos caracteres comparten al inicio), permitiendo responder consultas de rango y de prefijos a partir de metadatos almacenados en cada no-terminal de la gramática. Este enfoque sugiere que, si en lugar de comprimir LCP se comprime directamente el DSA, y si a cada regla se le asocia la suma de su expansión junto con el mínimo y máximo de sumas parciales, es posible reconstruir implícitamente la información relevante del SA y responder consultas de tipo RMQ sobre SA sin necesidad de mantenerlo explícito. De esta forma, el costo de calcular min(SA[sp..ep]) y max(SA[sp..ep]) se traslada a operaciones sobre la gramática, en tiempo proporcional a la altura de la gramática —típicamente sublineal o logarítmico respecto al tamaño del texto—, mientras que el espacio ocupado depende del tamaño de la gramática y no del número total de sufijos. Esta combinación de compresión y soporte de consultas de rango es precisamente lo que hace atractivo el uso de gramáticas sobre el DSA como base para el índice propuesto en esta memoria.

1.2. Baseline

El baseline con el cual se compara la propuesta es el sr-index, cuya implementación —hecha por Dustin Cobas— se puede encontrar en https://github.com/duscob/sr-index. Este índice resuelve el mismo problema, pero enumerando todas las ocurrencias una por una en vez de usar una gramática comprimida.

1.3. Implementaciones

1. **Comprimir el DSA con Repair**, el algoritmo de compresión gramatical que reemplaza iterativamente los pares de símbolos más frecuentes de una secuencia por un nuevo no-terminal (se profundiza en la sección 2.7): almacenar en cada no-terminal seis campos: longitud $l(X)$, suma total $d(X)$, mínimo de sumas parciales $m_{\min(X)}$, posición del mínimo $p_{\min(X)}$, máximo de sumas parciales $m_{\max(X)}$ y posición del máximo $p_{\max(X)}$.
2. **Responder min(SA[sp..ep]) y max(SA[sp..ep]) en tiempo $O(h)$**: siendo h la altura de la gramática, se desciende recursivamente combinando los metadatos, sin necesidad de enumerar las ocurrencias.
3. **Integrar esta estructura en el pipeline de clasificación taxonómica**: los intervalos [sp, ep] producidos por el FM-index para cada MEM se consultan sobre la gramática del DSA para obtener la primera y última posición en el texto.

Capítulo 2

Conceptos Básicos

2.1. Árbol filogenético

Un árbol filogenético es un árbol con raíz donde las hojas representan genomas y los nodos internos representan ancestros comunes hipotéticos. Las hojas tienen un orden de izquierda a derecha el cual cumple la siguiente propiedad: si dos hojas h_1 y h_2 están en el mismo subárbol de un nodo interno v, entonces todas las hojas entre h_1 y h_2 en el orden de izquierda a derecha también están en el subárbol de v. Esta propiedad forma la base de lo que se quiere lograr: si un patrón P aparece en los genomas de las hojas h_1 y h_2, y el LCA de h_1 y h_2 es v, entonces sabemos que P podría aparecer en cualquier genoma del subárbol de v. Si se toma la primera posición en el texto donde ocurre P (que cae en algún genoma, hoja i) y la última posición (hoja j, con i<j en orden de izquierda a derecha) , entonces el LCA(hoja i,hoja j) es el nodo interno que representa el ancestro común más antiguo donde P apareció por primera vez, es decir, P no necesariamente está presente en todas las hojas de ese subárbol, pero sí se puede afirmar que el LCA cubre todas las ocurrencias de P. Cualquier subárbol más pequeño dejaría fuera al menos una ocurrencia de P, es por esto que basta la primera y la ultima.

Dados h_1, h_2,...,h_d las hojas del árbol en orden (de aquí en adelante se simplemente "en orden", asumiendo que es de izquierda a derecha), con $T[1...n]= G_1\ G_2...G_d$, donde \$ es un carácter separador, lexicográficamente menor que cualquier carácter del alfabeto definido por los genomas del árbol.

2.2. Bitvector B

Se define el bitvector $B[1..n]$ donde $B[j] = 1$ si y solo si T[j]=\$. A partir de este arreglo se puede definir la operación $B.\text{rank}(j)$, la cual cuenta el número de 1s en $B[1..j]$. Esto dice a qué genoma pertenece la posición j.

2.3. Arreglo de sufijos SA

El arreglo de sufijos SA[1..n] es una permutación de {1,2,...,n} tal que T[SA[1]] < T[SA[2]] < ...< T[SA[n]] en orden lexicográfico. Es decir, SA[i] es la posición de inicio del i-ésimo

sufijo en orden lexicográfico. $T[j..]$ denota el sufijo de T que empieza en la posición j, o sea, $T[j]$,$T[j+1]$,...,$T[n]$. En palabras simples, es un arreglo que ordena todos los sufijos de la concatenación en orden lexicográfico.

2.4. Transformada de Burrows-Wheeler BWT

La transformada de Burrows-Wheeler BWT de T es una cadena BWT[1..n] definida como:

$$\text{BWT}[i] = T[\text{SA}[i] - 1] \text{ si } \text{SA}[i] > 1$$

$$\text{BWT}[i] = T[n] \text{ si } \text{SA}[i] = 1$$

BWT[i] es el carácter que precede inmediatamente al i-ésimo sufijo en orden lexicográfico. La BWT es una permutación de los caracteres de T y la transformación es reversible, es decir, se puede reconstruir T a partir de BWT. Su propiedad clave es que tiende a agrupar caracteres iguales en posiciones consecutivas, lo que facilita la compresión.

Una run de la BWT es una subsecuencia maximal de posiciones consecutivas BWT[i..j] donde todos los caracteres son iguales. El número de runs de la BWT se denota por r. Es una medida de la repetitividad del texto: textos altamente repetitivos tienen un valor de r mucho menor que n (la longitud del texto). El número r juega un rol fundamental como se demostró [2], las runs de la BWT inducen repeticiones en el arreglo diferencial de sufijos (DSA), lo que permite comprimir el DSA con una gramática de tamaño proporcional a r.

2.5. Run-Length FM-index

El FM-index es un índice comprimido que permite buscar patrones en T sin almacenar T explícitamente, y se basa en la BWT. El Run-Length FM-index es una variante del FM--index, diseñada para textos repetitivos. Usa el hecho de que la BWT de un texto repetitivo tiene pocas runs: almacena la BWT de forma compacta representando cada run por su carácter y su longitud.

2.6. Arreglo diferencial de sufijos DSA

El arreglo diferencial de sufijos se define de la siguiente manera:

$$\text{DSA}[1] = \text{SA}[1]$$

$$\text{DSA}[p] = \text{SA}[p] - \text{SA}[p-1] \text{ para todo } p \geq 2$$

Es decir, DSA almacena las diferencias entre valores consecutivos del arreglo de sufijos.

El arreglo de sufijos original se puede reconstruir a partir del DSA mediante sumas parciales: para cualquier posición de referencia $q < p$:

Propiedad 2.6.1: $$\text{SA}[p] = \text{SA}[q] + \sum_{j=q+1}^{p} \text{DSA}[j]$$

Esta propiedad es bastante directa y se verá más adelante.

2.7. Compresión gramatical Repair

Repair es un algoritmo de compresión gramatical que transforma una secuencia de símbolos en una gramática libre de contexto que genera únicamente esa secuencia. Funciona iterativamente: en cada paso identifica el par de símbolos adyacentes que aparece con mayor frecuencia en la secuencia actual, crea un nuevo no-terminal que lo representa, reemplaza todas las ocurrencias de ese par por el nuevo no-terminal, y repite hasta que no quede ningún par con frecuencia mayor que uno. El resultado es un conjunto de reglas donde cada regla tiene exactamente dos símbolos en el lado derecho ($X \to Y_1 Y_2$), más una secuencia comprimida (la regla del símbolo inicial) que contiene los símbolos que no fueron reemplazados.

En el contexto de este trabajo, Repair se aplica sobre el arreglo diferencial de sufijos DSA. La repetitividad del DSA hace que muchos pares de valores adyacentes se repitan a lo largo de la secuencia, lo que permite a Repair capturarlos jerárquicamente: primero los pares más frecuentes de valores originales, luego pares de no-terminales que representan patrones cada vez más largos. La gramática resultante tiene un tamaño dramáticamente menor que la secuencia original para colecciones genómicas repetitivas, y su estructura de árbol binario (cada no-terminal tiene exactamente dos hijos) permite almacenar metadatos agregados en cada nodo y responder consultas por descenso recursivo sin descomprimir.

2.8. RMQ, PSV y NSV

- **RMQ (Range Minimum Query)**: Dado un arreglo $A[1..n]$ y un rango $[p, q]$, RMQ(p,q) devuelve la posición del valor mínimo en $A[p..q]$. Es decir, RMQ(p,q) = $\text{argmin}_{\{p \leq k \leq q\}} A[k]$. Dado un preprocesamiento en $O(n)$, se puede responder cualquier consulta RMQ en $O(1)$ usando estructuras sucintas.

- **PSV (Previous Smaller Value)**: Dado un arreglo $A[1..n]$ y una posición p, PSV(p) devuelve la mayor posición $q < p$ tal que $A[q] < A[p]$. Es decir, busca hacia la izquierda la posición más cercana donde el valor del arreglo es estrictamente menor que el valor actual.

- **NSV (Next Smaller Value)**: Dado un arreglo $A[1..n]$ y una posición p, NSV(p) devuelve la menor posición $q > p$ tal que $A[p] > A[q]$. Es decir, busca hacia la derecha la posición más cercana donde el valor del arreglo es estrictamente menor que el valor actual.

Capítulo 3

Estado del arte

3.1. Fully-Functional Suffix Trees and Optimal Text Searching in BWT-runs Bounded Space

Este paper (que a partir de ahora llamaremos Jacm19 [2]) resuelve cómo indexar textos altamente repetitivos en espacio proporcional a r, manteniendo las operaciones eficientes. Antes de este paper, existía el Run-Length FM-index que podía contar ocurrencias de un patrón en $O(r)$ espacio, pero no podía localizar las posiciones de esas ocurrencias eficientemente sin agregar $O(\frac{n}{s})$ espacio extra para muestreo del arreglo de sufijos. Es decir, o usabas poco espacio pero localizabas lento, o localizabas rápido pero el espacio ya no dependía solo de r. El paper presenta el r-index junto con los siguientes resultados, cada uno usando más espacio a cambio de más funcionalidad:

1. **Locate en $O(r)$ de espacio**: Muestran cómo localizar las occ ocurrencias en tiempo $O(n \log\log n)$ usando solo $O(r)$ palabras. La clave está en la función φ: si se conoce SA[p], se puede obtener SA[p-1] y SA[p+1] en $O(\log\log n)$, usando solo $O(r)$ datos almacenados en las fronteras de las runs de la BWT. Entonces, durante el backward search siempre se mantiene al menos un valor de SA[p] conocido dentro del rango actual, y se recorre desde ahí para encontrar los demás.

2. **Count óptimo en $O(r \log\log n)$ de espacio**: Subiendo el espacio, se logra contar en tiempo óptimo $O(m)$ y localizar en tiempo óptimo $O(m + \text{occ})$.

3. **Acceso al SA en $O\big(r \log(\frac{n}{r})\big)$ de espacio**: Se muestra cómo acceder a cualquier celda SA[p] en tiempo $O\big(\log(\frac{n}{r})\big)$ usando $O\big(r \log(\frac{n}{r})\big)$ de espacio. Aquí es donde aparece el DSA y la demostración de que las runs de la BWT inducen repeticiones en él. Se construye un Block Tree sobre el DSA que explota estas repeticiones.

4. **Árbol de sufijos comprimido completo en $O\big(r \log(\frac{n}{r})\big)$ de espacio**: Usando el mismo espacio $O\big(r \log(\frac{n}{r})\big)$, se implementa un árbol de sufijos que contiene todas las operaciones necesarias de navegación en tiempo $O\big(\log(\frac{n}{r})\big)$. Para esto, se requiere

soportar RMQ, PSV y NSV sobre el LCP, y se hace construyendo una gramática sobre el DLCP con metadatos en cada no-terminal.

El paper presenta algunos puntos que son parte esencial de la propuesta de esta memoria:

1. **Repetitividad del DSA**: El paper demuestra que si BWT[p-1] = BWT[p] (misma run), entonces DSA[LF(p)] = DSA[p]. Esto significa que el DSA hereda la estructura repetitiva de la BWT. Cualquier bloque del DSA dentro de una run tiene una copia en otro lugar, y hay solo $O(r)$ "puntos de frontera" necesarios para cubrir todas las copias. Esta es la justificación teórica de que comprimir el DSA con Repair produciría una gramática pequeña. Sin este resultado, no se tendría garantía de que la gramática del DSA fuera compacta.

2. **Gramática sobre un arreglo diferencial con metadatos (sección 6.3)**: Para soportar RMQ sobre el LCP, el paper construye una gramática sobre el DLCP y almacena en cada no-terminal X los valores $l(X)$, $d(X)$, $m(X)$, $p(X)$: longitud, suma total, mínimo de sumas parciales y posición del mínimo. Luego muestra cómo descender por la gramática para responder RMQ en $O\big(\log\big(\frac{n}{r}\big)\big)$. Esto es exactamente lo que se hace en esta memoria, pero aplicado al DSA y extendiendo los metadatos.

3. **Algoritmo de descenso recursivo**: El paper describe que para una consulta RMQ(p,q) sobre el LCP, se descompone el rango en 3 partes: izquierda, símbolos centrales complejos y derecha. Se resuelve la parte central en $O(1)$ con una estructura RMQ sucinta sobre una estructura auxiliar M, y se resuelven las partes laterales descendiendo recursivamente por la gramática.

4. **Estructuras auxiliares**: Los arreglos L (longitudes acumuladas), A (diferencias acumuladas), M (mínimos por símbolo) y la estructura RMQ sucinta sobre M.

3.2. Taxonomic classification with maximal exact matches in KATKA kernels and minimizers digests

Este trabajo (que llamaremos Sea24 [3]) , aborda cómo clasificar un read de ADN (determinar de qué organismo proviene) usando una colección de genomas organizados en un árbol filogenético. Aquí es donde se propone usar MEMs en vez de k-mers (estructura que funciona pero no en todos los casos, por ejemplo, bases de datos de distinto tamaño, árboles desbalanceados). Lo más valioso de este paper en el contexto de esta memoria es lo siguiente:

Pipeline MEM para clasificación taxonómica

1. Concatenar los genomas en el orden de las hojas del árbol filogenético, separados por $

2. Construir un FM-index aumentado sobre la concatenación (con soporte para access, rank y select sobre la BWT; range-minimum y range-maximum sobre el SA; RMQ, PSV y NSV sobre el LCP; y rank sobre un bitvector B de separadores)

3. Para cada MEM de un read, encontrar su intervalo [sp,ep]

4. Obtener min(SA[sp..ep]) y max(SA[sp..ep]) para encontrar la primera y última posición en el texto

5. Usar B.rank para determinar en qué genomas están esas posiciones

6. Calcular el LCA de esos genomas en el árbol filogenético

7. Clasificar el read basándose en el MEM más largo.

Es decir, Sea24 define exactamente el problema a resolver: dado un intervalo [sp, ep] del SA (producido por el backward search para un MEM), calcular min(SA[sp..ep]) y max(SA[sp..ep]) para determinar el primer y último genoma. Se deja claro también que lo único que se necesita del arreglo de sufijos son dos valores por MEM: el mínimo y el máximo. No se necesitan todas las ocurrencias, no se necesita el SA completo, no se necesita navegar el árbol de sufijos. Solo min y max.

3.3. Fast and small subsampled r-indexes

Este paper (Talg25 [1]) presenta el baseline con el que se compararía el índice propuesto en esta memoria. La versión original del r-index requería estructuras adicionales para la recuperación eficiente de posiciones en el arreglo de sufijos, así como mecanismos complejos para extender los valores muestreados a posiciones vecinas. Además, aunque su espacio teórico era $O(r)$, las implementaciones reales arrastraban constantes y sobrecargas que reducían su competitividad práctica frente a índices no repetitivos como el FM-index altamente optimizado. El sr-index es una variante práctica del r-index que introduce mejoras concretas tanto en espacio como en tiempo. El objetivo principal del sr-index es optimizar el diseño del muestreo del arreglo de sufijos (SA-sampling), reduciendo la distancia entre muestras y simplificando la navegación necesaria para recuperar SA[i], todo ello manteniendo un espacio proporcional a r. El sr-index también introduce mejoras en la representación de las estructuras de búsqueda de predecesor utilizadas para soportar la variante run-aware del backward search. El diseño liviano de estas estructuras, junto con el muestreo adicional, genera un índice que, en la práctica, es más pequeño que el r-index original y considerablemente más rápido en la operación de localización. Los experimentos presentados en el paper muestran que el sr-index mantiene la compactividad del r-index pero supera significativamente su desempeño en términos de tiempo de construcción, búsqueda y recuperación de posiciones.

Al igual que el r-index, el sr-index puede recuperar el intervalo SA[sp..ep] donde se encuentran las ocurrencias de un patrón, pero no permite obtener directamente valores como min(SA[sp..ep]) o max(SA[sp..ep]), esenciales para la detección de maximal exact matches (MEMs) y el cálculo de ancestros comunes más bajos (LCA) en colecciones filogenéticas. Incluso con el submuestreo, sigue siendo necesario extender los valores muestreados del SA a posiciones vecinas mediante iteraciones repetidas de φ, lo que introduce un costo adicional y carece de soporte nativo para RMQ sobre SA.

El sr-index representa así una evolución importante del r-index, orientada a su uso práctico en entornos donde se requiere un equilibrio entre compresión extrema y eficiencia operacional. Sin embargo, tanto el r-index como su variante sr-index siguen basándose exclusivamente en la repetición observada en la BWT y en técnicas de muestreo, sin explotar otras formas de estructura repetitiva más adecuadas para soportar consultas de rango sobre el arreglo de sufijos.

Capítulo 4

Compresión del arreglo diferencial de sufijos

Dada la definición de arreglo diferencial de sufijos, se puede hacer la siguiente deducción:

$$\mathrm{SA}[p] = \mathrm{SA}[p-1] + \mathrm{DSA}[p]$$

Aplicando una sumatoria a ambos lados de la igualdad para $q < p$:

$$\sum_{j=q+1}^{p} \mathrm{SA}[j] - \mathrm{SA}[j-1] = \sum_{j=q+1}^{p} \mathrm{DSA}[j]$$

El lado izquierdo corresponde a una sumatoria telescópica, por lo tanto:

$$\mathrm{SA}[p] = \mathrm{SA}[q] + \sum_{j=q+1}^{p} \mathrm{DSA}[j]$$

Esta igualdad corresponde a la propiedad 2.6.1 mostrada en el capítulo 2. Ahora, dado un rango [sp..ep], queremos calcular min(SA[sp],SA[sp+1],...,SA[ep]). Definimos la suma parcial k-ésima como:

Definición 4.1: $\mathrm{PS}(k) = \sum_{j=\mathrm{sp}}^{\mathrm{sp}+k-1} \mathrm{DSA}[j]$ para $1 \leq k < \mathrm{ep} - \mathrm{sp} + 1$

Por lo tanto:

$$\text{SA}[sp + k - 1] = \text{SA}[sp - 1] + \text{PS}(k)$$

Teniendo en cuenta que SA[sp-1] es una constante (no depende de k), el valor de k que minimiza SA[sp+k-1] es el mismo que minimiza PS(k):

$$\min(\text{SA}[\text{sp..ep}]) = \text{SA}[\text{sp} - 1] + \min_{1 \leq k \leq \text{ep}-\text{sp}+1} \text{PS}(k)$$

$$\max(\text{SA}[\text{sp..ep}]) = \text{SA}[\text{sp} - 1] + \max_{1 \leq k \leq \text{ep}-\text{sp}+1} \text{PS}(k)$$

Para calcular el mínimo/máximo de las sumas parciales, no se necesita descomprimir el DSA: Usaremos metadatos en la gramática libre de contexto del arreglo, para así obtener lo que se requiere.

La justificación y la idea de hacer esto nace del trabajo realizado por Travis Gagie, Gonzalo Navarro y Nicola Prezza el 2019 llamado "Fully-Functional Suffix Trees and Optimal Text Searching in BWT-runs Bounded Space". Por un lado, se muestra la repetitividad del DSA gracias a los lemas 5.2 y 5.3. El DSA puede representarse con una gramática de tamaño $O\big(r \log\big(\frac{n}{r}\big)\big)$, y esto es la justificación teórica de que comprimir el DSA con una gramática tiene sentido. Sin esta propiedad, no habría garantía de que la gramática fuera pequeña. Gracias a esta, confirmamos que para textos repetitivos ($r \ll n$), la gramática será mucho menor que el DSA original. Por otro lado, la idea de almacenar metadatos en no-terminales de una gramática también se obtiene de este trabajo. Jacm19 [2] construye una gramática libre de contexto sobre una estructura de datos llamada DLCP, y almacena en cada no-terminal X los valores $l(X)$, $d(X)$, $m(X)$, $p(X)$. Luego muestra cómo resolver RMQ, PSV, y NSV sobre el LCP descendiendo por la gramática en tiempo $O\big(\log\big(\frac{n}{r}\big)\big)$. Esto se usará pero aplicado en el DSA para responder RangeMin y RangeMax sobre el SA. Es decir, se usarán metadatos y también el mismo tipo de algoritmo de descenso recursivo por la gramática, pero las diferencias son: Se usa DSA, RangeMin y RangeMax sobre SA (para encontrar primera y última ocurrencia de MEMs) y también se requiere tanto mínimo como máximo. También se adaptará el algoritmo de descenso presentado en la sección 6.3, que es un procedimiento para responder RMQ(p,q) sobre el LCP usando la gramática del DLCP. Los pasos son esencialmente los mismos:

1. Localizar qué símbolos de la secuencia inicial de la gramática cubren el rango [sp..ep].

2. Descomponer en parte izquierda (parcial), parte central(símbolos completos), parte derecha (parcial)

3. Resolver la parte central en $O(1)$ con una estructura RMQ sucinta sobre un arreglo auxiliar M.

4. Resolver las partes laterales descendiendo recursivamente por la gramática, usando los metadatos.

5. Combinar los tres resultados.

4.1. Definición de los metadatos

Teniendo en mente que el objetivo primordial es tener min(SA[ep..sp]) y max(SA[ep…sp]), y que se demostró previamente que esto se reduce a encontrar el mínimo y máximo de las sumas parciales del DSA en el rango [sp..ep], ahora cada no-terminal X de la gramática expande a un bloque contiguo del DSA que diremos DSA[p..q], gracias a la idea propuesta en Jacm19, podemos definir metadatos dentro de ese bloque, precalculando mínimo y máxi mo de sumas parciales, combinando así la información de dos bloques adyacentes pudiendo responder sobre cualquier rango descendiendo por la gramática.

Sea X un no-terminal (o terminal) de la gramática, y sea D=DSA[p..q] la secuencia de valores a la que X expande, se define:

Metadata 4.1.1. (l(X)-Longitud)

$$\mathrm{l}(x) = |\mathrm{D}| = \mathrm{q} - \mathrm{p} + 1$$

Corresponde a cuantos valores del DSA abarca la expansión de X. Sirve para saber donde termina la expansión y donde empieza la del siguiente símbolo. Cuando se desciende por la gramática buscando una posición especifica, se necesita saber si esa posición cae en el hijo izquierdo o derecho de una regla $X \to Y_1 Y_2$, y para eso comparamos la posición con $\mathrm{l}(\mathrm{Y}_1)$

Metadata 4.1.2. (d(X)-Suma total)

$$\mathrm{d}(x) = \mathrm{D}[1] + \mathrm{D}[2] + ... + \mathrm{D}[l(X)] = \mathrm{DSA}[p] + \mathrm{DSA}[p+1] + ... + \mathrm{DSA}[q]$$

Corresponde a la suma de todos los valores del DSA en la expansión de X.

Como DSA[j]=SA[j]-SA[j-1], por la suma telescópica, esto implica:

$$\mathrm{d(X)} = \mathrm{SA}[q] - \mathrm{SA}[\mathrm{p} - 1]$$

Es decir d(X) es la diferencia entre el último y el primer valor del arreglo de sufijos cubierto por X. Esto sirve para cuando estamos procesando la gramática y queremos saltar la expansión de X, por lo que solo basta sumar d(X) al acumulador.

Metadata 4.1.3. ($m_{\min}$ – Mínimo de sumas parciales)

$$\mathrm{m}_{\min}(X) = \min_{1 \le k \le l(\mathrm{X})} \mathrm{PS}(k)$$

Recordando que SA[p+k-1] = SA[p-1] + PS(k), entonces $m_{\min}$ se puede escribir como:

$$\mathrm{m}_{\min}(X) = \min_{1 \le j \le l(\mathrm{X})} \mathrm{SA}[j] - \mathrm{SA}[p-1]$$

Esto quiere decir que $m_{\min(X)}$ es el valor mínimo del SA dentro de la expansión de X, relativo al valor de SA justo antes de la expansión. Cuando se hace el descenso recursivo y se sabe que el acumulador actual es f, el mínimo absoluto del SA dentro de este bloque es f+$m_{\min(X)}$. Esto permite comparar con mínimos de otros bloques sin descender más.

Metadata 4.1.4. ($p_{\min}$ – Posición del mínimo)

$$\mathrm{p}_{\min}(X) = \mathrm{argmin}_{1 \le k \le l(\mathrm{X})} \mathrm{PS}(k)$$

Es la posición relativa, dentro de la expansión de X contando desde 1, donde se alcanza el mínimo de las sumas parciales. Si hay empate se elige la posición más a la izquierda. Si $m_{\min(X)}$ se alcanza en PS(k*), entonces la posición absoluta en el arreglo de sufijos donde SA es mínimo dentro de la expansión de X es p+k*−1 (donde p es la posición del DSA donde empieza la expansión de X) sirve para reportar donde está el mínimo, no solo su valor.

Metadata 4.1.5. ($m_{\max}$ – Máximo de sumas parciales)

$$m_{\max}(\mathrm{X}) = \max_{1 \le k \le l(\mathrm{X})} \mathrm{PS}(k)$$

Análogo a $m_{\min}$, corresponde a valor máximo del SA dentro de la expansión de X, relativo al valor de SA justo antes.

Metadata 4.1.6. ($p_{\max}$ – Posición del máximo)

$$\mathrm{p_{max}}(X) = \mathrm{argmax}_{1 \leq k \leq l(\mathrm{X})} \mathrm{PS}(k)$$

Posición relativa donde se alcanza el máximo. En caso de empate se elige la posición más a la izquierda

Por lo tanto, cada no-terminal almacena 6 valores enteros.

4.2. Cálculo de los metadatos

Para terminales, el cálculo es trivial: dado v=DSA[p] para alguna posición p, las sumas parciales de una secuencia de un solo elemento corresponden a PS(1)=v. Los metadatos entonces son:

- $\mathrm{l(v)} = 1$
- $\mathrm{d(v)} = v$
- $\mathrm{m_{min(v)}} = v$
- $\mathrm{p_{min(v)}} = 1$
- $\mathrm{m_{max(v)}} = v$
- $\mathrm{p_{max(v)}} = 1$

Para el caso de no-terminales, se tiene una regla $X \to Y_1 Y_2$, se requiere calcular los metadatos de X a partir de los metadatos de Y_1 e Y_2. Primero se calculan los metadatos de los terminales, luego los de los no-terminales que solo tienen terminales como hijos, luego de los no-terminales cuyos hijos ya tienen metadatos, y así sucesivamente.

Sea $X \to Y_1 Y_2$. La expansión de X es la concatenación de la expansión de Y_1 seguida de la expansión de Y_2. Sea D_1=la expansión de Y_1 de longitud $l(Y_1)$ y sea D_2=la expansión de Y_2 de longitud $l(Y_2)$. La expansión de X es:

$$\mathrm{D} = \mathrm{D_1 D_2} = \mathrm{D_1}[1], \mathrm{D_1}[2], ..., \mathrm{D_1}[l(Y_1)], \mathrm{D_2}[1], \mathrm{D_2}[2], ..., \mathrm{D_2}[\mathrm{l(Y_2)}]$$

Longitud: Directo, la concatenación tiene la longitud de ambas partes sumadas:

$$\mathrm{l(X)} = \mathrm{l(Y_1)} + \mathrm{l(Y_2)}$$

Suma total: También directo, la suma de todos los valores es la suma de la primera parte más la suma de la segunda parte:

$$\mathrm{d(X)} = \mathrm{d(Y_1)} + \mathrm{d(Y_2)}$$

Mínimo de sumas parciales: Aquí no es tan directo, pero se puede apreciar que las sumas parciales se dividen en dos grupos:

$$m_{\min}(X) = \min_{1 \leq k \leq l(X)} \text{PS}(k)$$

$$\text{PS}(k) = \sum_{j=\text{sp}}^{\text{sp}+k-1} \text{DSA}[j], \text{ para } \quad 1 \leq k \leq \text{ep} - \text{sp} + 1$$

Tomando sp=1 y ep=l(X), entonces la sumatoria queda:

$$\text{PS}(k) = \sum_{j=1}^{k} \text{DSA}[j], \text{ para } \quad 1 \leq k \leq \text{l(X)}$$

La longitud ya se calculó previamente, por lo tanto la sumatoria se puede separar:

$$\text{PS}(k) = \sum_{j=1}^{l(Y_1)} \text{DSA}[j] + \sum_{j=1}^{l(Y_2)} \text{DSA}[j]$$

Por las definiciones hechas, el mínimo del primer grupo es $m_{\min(Y_1)}$. Para el segundo grupo, corresponde a las últimas $l(Y_2)$ sumas parciales. Dado que $k > l(Y_1)$:

$$\text{PS}(l(Y_1) + j) = d(Y_1) + \text{PS}_{Y_2}(j), \text{ para } \ 1 \leq j \leq l(Y_2)$$

Donde $\text{PS}_{Y_2}(j)$ denota las sumas parciales de la expansión de Y_2. Esto se cumple porque la suma parcial en la posición $l(Y_1) + j$ incluye toda la expansión de Y_1 (cuya suma es $d(Y_1)$) más las primeras j posiciones de la expansión de Y_2 (cuya suma parcial es $PS_{Y_2}(j)$). El mínimo de este grupo por lo tanto es $d(Y_1) + m_{\min}(Y_2)$. El mínimo global es el menor de los dos:

$$m_{\min}(X) = \min(m_{\min}(Y_1), d(Y_1) + m_{\min}(Y_2))$$

Posición del mínimo: Si $m_{\min}(Y_1) \leq d(Y_1) + m_{\min}(Y_2)$, entonces el mínimo global está en el grupo 1, y su posición relativa dentro de X es la misma que dentro de Y_1:

$$p_{\min}(X) = p_{\min}(Y_1)$$

En caso contrario, entonces el mínimo global está en el segundo grupo, y su posición relativa dentro de X es la posición de Y_2 más la longitud de Y_1:

$$p_{\min}(X) = l(Y_1) + p_{\min}(Y_2)$$

En caso de igualdad, se elige la posición más a la izquierda que está en el primer grupo.

Máximo de sumas parciales: El razonamiento es totalmente simétrico al mínimo:

$$m_{\max}(X) = \max(m_{\max}(Y_1), d(Y_1) + m_{\max}(Y_2))$$

Posición del máximo: Si $m_{\max}(Y_1) \geq d(Y_1) + m_{\max}(Y_2)$, el máximo global está en el primer grupo, y su posición relativa en X es la misma que dentro de Y_1:

$$p_{\max}(X) = p_{\max}(Y_1)$$

En caso contrario (análogo al mínimo):

$$p_{\max}(X) = l(Y_1) + p_{\max}(Y_2)$$

4.3. Complejidad del cálculo de los metadatos

Para cada regla $X \to Y_1 Y_2$, el cálculo de los 6 campos de metadatos toma $O(1)$ en tiempo dado que solo son comparaciones y sumas. Si la gramática tiene g reglas, el tiempo total es $O(g)$. En cuanto espacio, si la gramática tiene g no-terminales, el espacio total de los metadatos es $O(g)$. Los terminales también tienen metadatos, pero es trivial y se pueden calcular en el camino, así que no requieren almacenamiento adicional. Recordar que g$\ll$ n y g $\geq \log n$ siempre en gramáticas , por lo que si no hay un par repetido, Repair no encuentra ningún par adyacente con frecuencia mayor a 1, y entonces no crea ningún no--terminal, lo que significa que g=n, y por lo tanto la complejidad O(g) se sigue cumpliendo de igual forma.

4.4. Estructuras auxiliares

Lo siguiente es una adaptación de lo que se hace en Jacm19 en la sección 6.3 para el DLCP. La gramática del DLCP tiene una regla inicial:

$$\mathrm{S} \to \mathrm{X}_1\mathrm{X}_2...$$

Se construyen arreglos auxiliares sobre esta secuencia para poder localizar rápidamente en que símbolo X_x cae una posición dada, y para resolver la parte central de una consulta RMQ en $O(1)$. Específicamente, en Jacm19 se define:

- L[x]: longituds acumuladas (para localizar posiciones)
- A[x]: diferencias acumuladas (para convertir entre valores relativos y absolutos)
- M[x]: mínimos por símbolo (para resolver parte central de RMQ)
- RMQ_M: Estructura RMQ sucinta sobre M (para consultas $O(1)$)

Estas mismas ideas se aplican al DSA en vez del DLCP, y se duplican para manejar tanto mínimos como máximos.

La regla del símbolo inicial de la gramática Repair es de la forma:

$$\mathrm{S} \rightarrow \mathrm{s_1 s_2 s_3 ...}$$

Esta secuencia se denotará por $\mathrm{DSA_0}$. Cada símbolo s_x es un terminal o un no-terminal y expande a un bloque contiguo del DSA. Cuando llega una consulta RangeMin(SA[ep..sp]), lo primero que se necesita saber es en qué símbolos de DSA_0 caen las posiciones sp y ep. Y para la parte central de la consulta (los símbolos completos entre sp y ep), se necesita encontrar rápidamente cual tiene el menor / mayor valor del SA. Las estructuras auxiliares resuelven estos problemas. Las siguientes definiciones son sacadas directamente desde Jacm19, adaptadas para el arreglo DSA:

Estructura 4.4.1. (Longitudes acumuladas L[x]) Dado K=$|DSA_0|$:

$$\mathrm{L}[0] = 0$$

$$\mathrm{L}[x] = \mathrm{L}[x-1] + l(s_x), \text{ para } \ 0 \leq x \leq K$$

$L[x]$ es la posición del DSA hasta donde llega la expansión de s_x. La expansión de s_x cubre las posiciones DSA[L[x-1]+1..L[x]]. Para encontrar en qué símbolo s_x cae la posición DSA[p], hacemos una búsqueda binaria sobre L para encontrar el x tal que $L[x-1] < p \leq L[x]$, lo cual toma $O(\log(|\mathrm{DSA_0}|))$.

Estructura 4.4.2. (Diferencias acumuladas A[x]) Dado K=$|DSA_0|$:

$$A[0] = 0$$

$$A[x] = A[x-1] + d(s_x), \text{ para } \ 0 \leq x \leq K$$

Recordando que $d(s_x)$ es la suma de los valores del DSA en la expansión de s_x, como el DSA son diferencias del SA, la suma acumulada da las diferencias del SA entre posiciones más lejanas:

$$A[x] = d(s_1) + d(s_2) + ... + d(s_x) = \mathrm{DSA}[1] + \mathrm{DSA}[2] + ... + \mathrm{DSA}[L[x]] = \mathrm{SA}[L[x]] - \mathrm{SA}[0]$$

Si se define SA[0]=0, entonces $A[x]$=SA[L[x]]. Esto sirve cuando se requiera saber SA[L[x-1]](el valor del SA justo al final de la expansión del símbolo x-1, que es el punto de referencia para el símbolo x). Más precisamente, en el algoritmo de consulta que se verá más adelante, $A[x-1]$ es el valor base f a partir del cual los valores relativos $m_{\min}$ y $m_{\max}$ del símbolo s_x se convierten en valores absolutos del SA, es decir:

- El mínimo absoluto del SA dentro de la expansión de s_x es: A[x-1]+$m_{\min}(s_x)$
- El máximo absoluto del SA dentro de la expansión de s_x es: A[x-1]+$m_{\max(s_x)}$

Estructura 4.4.3. ($M_{\min}$) Dado K=$|DSA_0|$:

$$M_{\min}[x] = A[x-1] + m_{\min}(s_x), \text{para } \ 1 \leq x \leq K$$

En Jacm19 se define un solo arreglo M para los mínimos, en este caso se necesitan dos, uno para los mínimos y otro para los máximos. $M_{\min}$ es el valor mínimo absoluto del SA dentro de la expansión de s_x.

Estructura 4.4.4. ($M_{\max}$) Dado K=$|DSA_0|$:

$$M_{\max}[x] = A[x-1] + m_{\max}(s_x), \text{ para } \ 1 \leq x \leq K$$

$M_{\max}$ es el valor máximo absoluto del SA dentro de la expansión de s_x

Estas estructuras son necesarias pues queremos comparar mínimos y máximos entre distintos valores del DSA_0. Cada símbolo tiene un punto de referencia distinto ($A[x-1]$), y los metadatos $m_{\min}$ y $m_{\max}$ son relativos a ese punto de referencia. Para poder comparar es necesario convertir a valores absolutos.

Estructura 4.4.5. (R M $Q_{M_{\min}}$) Dado un rango [x..y], devuelve en $O(1)$ la posición z ∈ [x..y] tal que $M_{\min[z]}$ es mínimo.

Estructura 4.4.6. (R M $Q_{M_{\max}}$) Dado un rango [x..y], devuelve en $O(1)$ la posición z ∈ [x..y] tal que $M_{\max[z]}$ es máximo.

En Jacm19 [2], la estructura realizada por Fischer y Heun resuelve RMQ en $O(1)$ usando solo $O(r)$ bits, sin almacenar el arreglo M explícitamente. Aquí se toma esta idea pero

almacenando $M_{\min}$ y $M_{\max}$ (que son solo $O(|DSA_0|)$). Estas estructuras sirven para cuando la consulta RangeMin(SA[sp..ep]) tiene una parte central que abarca los símbolos DSA_0 [x+1..y-1], se necesita encontrar cual de estos símbolos tiene el menor $M_{\min}$. El espacio total de las estructuras auxiliares es $O(|DSA_0|)$, que es proporcional al número de símbolos en la secuencia comprimida. Como $|\mathrm{DSA}_0| \leq g$ (el tamaño de la gramática) $\leq n$, y para textos repetitivos $g \ll n$.

4.5. Algoritmo de consulta

El algoritmo siguiente es una adaptación del procedimiento de RMQ sobre la gramática del DLCP descrito en Jacm19. Jacm19 describe cómo resolver RMQ(p,q) sobre el LCP descomponiendo el rango en tres partes, y descendiendo por la gramática para las partes parciales. Se hará exactamente lo mismo pero sobre el DSA para obtener RangeMin y RangeMax sobre el SA. La estructura general es:

1. Descomponer el rango [sp,ep] en 3 partes respecto a la secuencia DSA_0.
2. Resolver la parte central en $O(1)$ con $\mathrm{RMQ}_{M_{\min}}$ y $\mathrm{RMQ}_{M_{\max}}$.
3. Resolver las partes laterales descendiendo por la gramática.
4. Combinar los 3 resultados.

El input del algoritmo es un intervalo [sp..ep] del SA, donde 1≤sp≤ep≤n. Ese intervalo viene de backward search de la BWT: cuando se busca un MEM en el FM-index, se obtiene el rango de posiciones del SA donde aparecen las ocurrencias de ese MEM. El output sería finalmente min(SA[sp],SA[sp+1],…,SA[ep]) y max(SA[sp],SA[sp+1],…,SA[ep]) junto con sus posiciones. Con estos dos valores y el bitvector B, se obtiene:

$$\text{first-genome} = \mathrm{B.rank}(\min(\mathrm{SA[sp]}, \mathrm{SA[sp+1]}, ..., \mathrm{SA[ep]}))$$

$$\text{last-genome} = \mathrm{B.rank}(\max(\mathrm{SA[sp]}, \mathrm{SA[sp+1]}, ..., \mathrm{SA[ep]}))$$

Luego, LCA(first-genome,last-genome) entrega la clasificación.

Algoritmo de consulta

- ***Paso 1: Localizar los símbolos extremos***
  Hacemos dos búsquedas binarias sobre el arreglo L para encontrar:

  - x: el índice del símbolo de DSA_0 que contiene la posición sp. Es decir, el x tal que $L[x-1] < \text{sp} \leq L[x]$.

  - y: el índice del símbolo de DSA_0 que contiene la posición ep. Es decir, el y tal que $L[y-1] < \text{ep} \leq L[y]$.

- ***Paso 2: Descomponer en 3 partes***

  1. Parte derecha del símbolo s_x: DSA[sp…L[x]]

  2. Parte central: DSA[L[x]+1…L[y-1]]

  3. Parte izquierda del símbolo s_y: DSA[L[y-1]+1…ep]

  Casos especiales
  - x=y: Las posiciones sp y ep caen dentro del mismo símbolo. No hay parte central, y las partes 1 y 3 se fusionan en una sola consulta parcial dentro de s_x
  - x+1=y: No hay símbolos completos entre s_x y s_y. La parte 2 está vacía. Solo hay partes 1 y 3
  - sp $= L[x-1]+1$: sp empieza justo al inicio de s_x, entonces la parte 1 cubre toda la expansión de s_x y no se necesita descender, se puede usar $M_{\min}$ y $M_{\max}$ directamente.
  - ep $= L[y]$: ep llega justo al final de s_y, entonces la parte 3 cubre toda la expansión de s_y y se puede usar $M_{\min}$ y $M_{\max}$

- ***Paso 3: Resolver la parte central***
  Dado un no-terminal X con expansión D=DSA[p..q] de longitud $l(X)$, y un subrango [a..b] con 1≤a≤b≤$l(X)$, se debe encontrar mínimo y máximo de las sumas parciales de $D[a..b]$

  Caso base(X es terminal): Si X es un terminal con valor v, entonces $l(X) = 1$ y por lo tanto, a=b=1: min=v, max=v, posición=1.

  Caso recursivo: $X \to Y_1 Y_2$: Hay 3 subcasos donde caen a y b con respecto a $l(Y_1)$:

  - **Subcaso A ($b \leq l(Y_1)$)**: Todo el rango está dentro de Y_1. Descendemos recursivamente en Y_1 con el mismo rango [a..b]

- **Subcaso B ($a > l(Y_1)$)**: Todo el rango está dentro de Y_2. Descendemos recursivamente en Y_2 con rango [a-l(Y_1),b-l(Y_1)]
- **Subcaso C ($a \leq l(Y_1) < b$)**: Este es el caso más interesante, se requiere:
  1. El min/max de sumas parciales de Y_1[a..l(Y_1)] (cola derecha de Y_1)
  2. El min/max de sumas parciales de $Y_2[1..b - l(Y_1)]$ (cola izquierda de Y_2)
  3. Combinar ambos resultados

Para el punto 1, se desciende recursivamente en Y_1 con rango [a,l(Y_1)]. Para el punto 2, se desciende recursivamente en Y_2 con rango [1,b-l(Y_1)] con acumulador f_2=f+(suma de D[a..l(Y_1)]).

Esa suma es el resultado del descenso en Y_1, específicamente, es d de la sub-consulta en Y_1 (la suma total de las sumas parciales en $Y_1[a..l(Y_1)]$). Cuando se desciende en Y_1, además de obtener el min y max, también se obtiene la suma total del sub-rango, que es el valor de la última suma parcial.

Formalizando:

$$\left(\min_L, \max_L, \text{sum}_L, \text{pos}_{\min_L}, \text{pos}_{\max_L}\right) = \text{descenso}(Y_1, a, l(Y_1), f)$$

Donde sum_L es la suma DSA[p+a-1] +…+DSA[p+l(Y_1)-1], que equivale a la suma parcial final del sub-rango en Y_1, luego:

$$f_2 = f + \text{sum}_L$$

$$\left(\min_R, \max_R, \text{sum}_R, \text{pos}_{(\min)_R}, \text{pos}_{(\max)_R}\right) = \text{descenso}(Y_2, 1, b - l(Y_1), f_2)$$

Combinando: $\text{result}_{\min} = \min(\min_L, \min_R)$

$$\text{result}_{\max} = \max\left(\max_L, \max_R\right)$$

La suma del subrango Y_1[a..l(Y_1)] se puede calcular durante el descenso. Cuando se llega a un nodo completo que está dentro del rango, su suma es $d(X)$. Cuando se parte un nodo, se acumulan las sumas de las partes que se cubren completamente. Si el subrango cubre toda la cola derecha de Y_1 desde la posición a hasta $l(Y_1)$, la suma es: $\text{sum}_L = d(Y_1) - \text{PS}_{Y_1}^{a-1}$, donde $\text{PS}_{Y_1}^{a-1}$ es la suma parcial de Y_1 hasta la posición a-1. En la práctica, se va acumulando las sumas de los nodos que se saltan, y al final, sum_L se obtiene como $d(Y_1)$ menos esa acumulación.

4.6. Ejemplo completo

A continuación se presentará un ejemplo de cómo funciona este índice aplicado al DSA:

Dado el siguiente texto: T= A T G C A $ A T G C G $ C T G C A $

Se tienen los siguientes genomas:

- Genoma 0: A T G C A
- Genoma 1: A T G C G
- Genoma 2: C T G C A

4.6.1. Arreglo de sufijos SA

Teniendo en cuenta que ordenados de forma lexicográfica se cumple que $ <A<C<G<T, dada la tabla Tabla 1, el arreglo de sufijos es SA = [18,6,12,17,5,1,7,16,4,10,13,11,15,3,9,14,2,8].

Sufijos del texto T			
i	**SA[i]**	**Sufijo**	**Grupo**
1	18	$	$
2	6	$ATGCG$CTGCA$	$
3	12	$CTGCA$	$
4	17	A$	A
5	5	A$ATGCG$CTGCA$	A
6	1	ATGCA$ATGCG$...	A
7	7	ATGCG$CTGCA$	A
8	16	CA$	C
9	4	CA$ATGCG$...	C
10	10	CG$CTGCA$	C
11	13	CTGCA$	C
12	11	G$CTGCA$	G
13	15	GCA$	G
14	3	GCA$ATGCG$...	G
15	9	GCG$CTGCA$	G
16	14	TGCA$	T
17	2	TGCA$ATGCG$...	T
18	8	TGCG$CTGCA$	T

Tabla 1: Sufijos del texto T

4.6.2. Arreglo diferencial de sufijos

Arreglo diferencial de sufijos DSA			
p	**SA[p]**	**SA[p-1]**	**DSA[p]**
1	18	-	18
2	6	18	−12
3	12	6	6
4	17	12	5
5	5	17	−12
6	1	5	−4
7	7	1	6
8	16	7	9
9	4	16	−12
10	10	4	6
11	13	10	3
12	11	13	−2
13	15	11	4
14	3	15	−12
15	9	3	6
16	14	9	5
17	2	14	−12
18	8	2	6

Tabla 2: Arreglo diferencial de sufijos DSA

Dado el SA, se calcula el DSA. De acuerdo a Tabla 2 se tiene que DSA= [18, −12, 6, 5, −12, −4, 6, 9, −12, 6, 3, −2, 4, −12, 6, 5, −12, 6]. Se pueden apreciar los siguientes pares repetidos:

- [−12,6] aparece cuatro veces
- [6,5] aparece dos veces
- [5,-12] aparece dos veces

4.6.3. Compresión Repair

- Iteración 1: El par más frecuente es [−12,6] con frecuencia 4:

$$\mathrm{A} \rightarrow (-12) \quad 6$$

Reemplazando las 4 ocurrencias:

$$\mathrm{DSA}_0 = [18, A, 5, -12, -4, 6, 9, A, 3, -2, 4, A, 5, A]$$

- Iteración 2: El par más frecuente es [A,5] con frecuencia 2:

$$B \to A \quad 5$$

Reemplazando:

$$\text{DSA}_0 = [18, B, -12, -4, 6, 9, A, 3, -2, 4, B, A]$$

- Iteración 3: Repair termina dado que no hay par con frecuencia mayor a 1.

La gramática final entonces:

$$\text{A} \to (-12) \quad 6$$

$$\text{B} \to A \quad 5$$

$$\text{S} \to [18, B, -12, -4, 6, 9, A, 3, -2, 4, B, A]$$

4.6.4. Metadatos

- Para cualquier terminal v: $l = 1$, $d = v$, $m_{\min} = v$, $p_{\min} = 1$, $m_{\max} = v$, $p_{\max} = 1$.
- **No-terminal A: $A \to$(-12) 6**: $l(A) = 1 + 1 = 2$

  $d(A) = -12 + 6 = -6$

  Sumas parciales= PS(1)= -12, PS(2)= -12 $+6 = -6$

  $m_{\min(A)} = \min(-12, -6) = -12$

  $p_{\min(A)} = 1$

  $m_{\max(A)} = \max(-12, -6) = -6$

  $p_{\max(A)} = 1 + 1 = 2$

- **No-terminal B: $B \to A \quad 5$**

  $l(B) = 2 + 1 = 3$

  $d(B) = -6 + 5 = -1$

  Sumas parciales= PS(1)= -12, PS(2)= -6, PS(3) = -6 + 5= -1

  $m_{\min(B)} = -12$

  $p_{\min(B)} = 1$

  $m_{\max(B)} = \max(-6, -1) = -1$

  $p_{\max(B)} = 3$

Tabla de metadatos							
	Regla	**l**	**d**	$m_{\min}$	$p_{\min}$	$m_{\max}$	$p_{\max}$
A	(-12)6	2	-6	-12	1	-6	2
B	A 5	3	-1	-12	1	-1	3

Tabla 3: Tabla de metadatos

4.6.5. Estructuras auxiliares

La secuencia DSA_0 tiene 12 símbolos: s_1=18, s_2=B, s_3=-12, s_4=-4, s_5=6, s_6=9, s_7=A, s_8=3, s_9=-2, s_{10}=4, s_{11}=B, s_{12}=A.

1. Arreglo L:

Arreglo L			
x	s_x	$l(s_x)$	**L[x]**
0	-	-	0
1	18	1	1
2	B	3	4
3	-12	1	5
4	-4	1	6
5	6	1	7
6	9	1	8
7	A	2	10
8	3	1	11
9	-2	1	12
10	4	1	13
11	B	3	16
12	A	2	18

Tabla 4: Tabla de cálculo de arreglo L

2. Arreglo A

Arreglo A		
x	$d(s_x)$	**A[x]**
0	-	0
1	18	18
2	-1	17
3	-12	5
4	-4	1
5	6	7
6	9	16
7	-6	10
8	3	13
9	-2	11
10	4	15
11	-1	14
12	-6	8

Tabla 5: Tabla de cálculo de arreglo A

3. Arreglos $M_{\min}$ y $M_{\max}$

Arreglos $M_{\min}$ y $M_{\max}$						
x	s_x	$A[x-1]$	$m_{\min}$	$m_{\max}$	$M_{\min}[x]$	$M_{\max}[x]$
1	18	0	18	18	18	18
2	B	18	−12	−1	6	17
3	−12	17	−12	−12	5	5
4	−4	5	−4	−4	1	1
5	6	1	6	6	7	7
6	9	7	9	9	16	16
7	A	16	−12	−6	4	10
8	3	10	3	3	13	13
9	−2	13	−2	−2	11	11
10	4	11	4	4	15	15
11	B	15	−12	−1	3	14
12	A	14	−12	−6	2	8

Tabla 6: Tabla de cálculo de arreglos $M_{\min}$ y $M_{\max}$

4.6.6. Read y MEMs

Sea el read ATGCG, que es exactamente el genoma 1, el FM-index encuentra dos MEMs:

- MEM 1: El read completo se encuentra en el intervalo [sp=7,ep=7]. Solo una ocurrencia SA[7]=7. min=max=7
- MEM 2: TGC. Intervalo= [sp=16,ep=18]. Esto arroja tres ocurrencias. SA[16]=14, SA[17]=2, SA[18]=8. Se requiere min y max. Es decir, RangeMin(SA[16..18]) y RangeMax(SA[16..18])

4.6.7. Consulta RangeMin(SA[16..18]) y RangeMax(SA[16..18])

Paso 1: Localizar los símbolos extremos

$$\text{sp} = 16 : L[10] = 13 < 16 \leq L[11] = 16 \rightarrow x = 11 (S_{11} = B \text{ cubre DSA}[14..16])$$

$$\text{ep} = 18 : L[11] = 16 < 18 \leq L[12] = 18 \rightarrow y = 12 (S_{12} = A \text{ cubre DSA}[17..18])$$

Paso 2: Descomponer en partes

- Parte (i): DSA[16..16] -Cola derecha de s_{11}=B (parcial: solo la última posición de B)
- Parte (ii): vacía - No hay símbolos completos entre x=11 e y=12
- Parte (iii): DSA[17..18] - s_{12} = A completo (ep=18=L[12])

Paso 3: Resolver parte (iii) Como A es un símbolo completo, se usa $M_{\min}$ y $M_{\max}$ directamente:

$$\min_{\text{der}} = M_{\min}[12] = 2, \text{posición} : L[11] + p_{\min}(A) = 16 + 1 = 17$$

$$\max_{\text{der}} = M_{\max}[12] = 8, \text{posición} : L[11] + p_{\max}(A) = 18$$

Esto quiere decir que A cubre SA[17..18]=[2,8]. min=2 en SA[17] y max=8 en SA[18] es consistente.

Paso 4: Resolver la parte (i)-Descenso en B: B cubre DSA[14..16], pero solo se necesita DSA[16..16]: la posición 3 de B.

$$\text{offset local} : a = 3, b = 3(\text{una sola posición})$$

$$\text{valor base} : f = A[10] = 15 \quad (\text{porque SA}[L[10]] = \text{SA}[13] = 15)$$

Se desciende en $B \to A$ 5, donde $l(A) = 2$ y $l(5) = 1$. El offset cumple con a=$3 > l(A) = 2$, así que la posición 3 está completamente en el hijo derecho. Este es el sub-caso B (en el procedimiento general expuesto en el algoritmo): todo el rango está en Y_2. Antes de entrar en Y_2, se debe saltar $Y_1 = A$. Actualizando el acumulador:

$$\text{f} \to \text{f} + d(A) = 15 + (-6) = 9$$

Esto es así pues $d(A) = -6$ significa que SA avanza -6 a lo largo de la expansión de A. SA[L[10]] era el punto de referencia. Después de pasar por los 2 valores de A, el nuevo punto de referencia es SA[L[10]+2]=SA[15]=15+(-6)=9, lo cual es consistente. Estando en el terminal 5:

$$\min_{\text{izq}} = f + 5 = 9 + 5 = 14, \text{ posición} : L[10] + 3 = 13 + 3 = 16$$

$$\max_{\text{izq}} = f + 5 = 9 + 5 = 14, \text{ posición} : 16$$

Lo cual es consistente: SA[16]=14

Paso 5: Combinar

Arreglos $M_{\min}$ y $M_{\max}$				
Parte	**Min**	**Pos min**	**Max**	**Pos Max**
Descenso en B	14	16	14	16
A completa	2	17	8	18

Tabla 7: Tabla resumen descenso en gramática

$$\text{result}_{\min} = \min(14, 2) = 2, \text{en posición } 17$$

$$\text{result}_{\max} = \max(14, 8) = 14, \text{en posición } 16$$

Verificando, se cumple justamente que: SA[16..18]=[14,2,8]. min=2 en SA[17] y max=14 en SA[16]

4.7. Integración con el pipeline

Lo que se ha hecho hasta ahora ha sido construir una forma eficiente de resolver el paso central del pipeline: dado un intervalo [sp..ep] del arreglo de sufijos, encontrar min(SA[sp..ep]) y max(SA[sp..ep]) sin enumerar todas las ocurrencias. A continuación se presentará cómo unir todo desde un read del ADN hasta una clasificación taxonómica. Dado un árbol filogenético T con genomas G_1 a G_D:

Fase de construcción

1. Construir el arreglo de sufijos SA[1..n] de T
2. Calcular el DSA
3. Comprimir el DSA con Repair: gramática g con reglas $X \to Y_1 Y_2$ y secuencia DSA_0
4. Calcular metadatos para cada no-terminal X: $l(X)$, $d(X)$, $m_{\min(X)}$, $p_{\min(X)}$, $m_{\max(X)}$, $p_{\max(X)}$
5. Construir estructuras auxiliares $L, A, M_{\min}, M_{\max}$ y estructuras RMQ sucintas sobre $M_{\min}$, $M_{\max}$
6. Construir bitvector B[1..n]
7. LCA
8. Construir el índice con ropebwt3 para encontrar MEMs

Fase de consulta

Dado un read R:

1. Encontrar MEMs de R respecto a la colección usando ropebwt3. Resultado: lista de MEMs, cada uno con intervalo $[\mathrm{sp}_i, \mathrm{ep}_i]$
2. Para cada MEM_i con intervalo $[\mathrm{sp}_i, \mathrm{ep}_i]$:
   - $\mathrm{first}_{\mathrm{pos}_i} = \mathrm{RangeMin}(\mathrm{SA}[\mathrm{sp}_i..\mathrm{ep}_i])$ usando el algoritmo de descenso por la gramática del DSA
   - $\mathrm{last}_{\mathrm{pos}_i} = \mathrm{RangeMax}(\mathrm{SA}[\mathrm{sp}_i..\mathrm{ep}_i])$ usando el mismo algoritmo con metadatos de máximo
   - $\mathrm{first}_{\mathrm{genome}_i} = B.\mathrm{rank}_1\left(\mathrm{first}_{\mathrm{pos}_i}\right)$
   - $\mathrm{last}_{\mathrm{genome}_i} = B.\mathrm{rank}_1\left(\mathrm{last}_{\mathrm{pos}_i}\right)$
   - $\mathrm{lca}_i = \mathrm{LCA}\left(\mathrm{hoja}\left[\mathrm{first}_{\mathrm{genome}_i}\right], \mathrm{hoja}\left[\mathrm{last}_{\mathrm{genome}_i}\right]\right)$
3. Clasificar el read basándose en el MEM más largo

Capítulo 5

Experimentación

5.1. Setup experimental

Los experimentos fueron ejecutados en el servidor Knuth del departamento de Ciencias de la Computación de la Universidad de Chile, el cual tiene las siguientes especificaciones:

- CPU: Procesador Intel Xeon de 8 cores, corriendo a 2.4 GHz, apoyado por 10 MB de memoria caché.
- RAM: 125 GB
- Disco: 2 TB
- Sistema Operativo: Linux

El código se puede encontrar en **https://github.com/gitonina/DSA-Grammar** . El lenguaje ocupado fue C++, y la única librería externa usada fue SDSL para la estructura RMQ sucinta. Los datasets fueron descargados de la página Pizza&Chili Corpus, con excepción del dataset llamado synth, que aparece en la Tabla 8 y fue construido con un archivo makeData.

Datasets

Dataset	**n (MB)**	σ	**r**	**n/r**
synth	100	5	908.820	110,03
influenza	154,8	16	3.022.822	51,21
ecoli	112,69	16	15.044.487	7,49
einstein	467,62	139	288.909	1.610,36
worldleaders	46,96	89	569.732	82,26

Tabla 8: Características de los datasets utilizados

1. **Synth**: Es un dataset generado con un archivo makeData.c: son 1000 genomas sintéticos de alrededor de 100.000 pb cada uno. Aplica mutaciones controladas para crear las 1000 variantes, por esto su repetitividad es intermedia y predecible.

2. **Escherichia coli (ecoli)**: Colecciones de múltiples genomas completos de la bacteria E. coli, concatenados. Aunque son genomas de la misma especie, cada cepa acumula mutaciones puntuales propias, por esto, es el dataset con r más alto de todos (menos compresible por BWT).

3. **Influenza**: Colección de genomas completos del virus de la influenza, concatenados. Al ser un virus de ARN con tasa de mutación muy alta, también divergen bastante entre sí punto a punto, aunque un poco menos que ecoli.

4. **Einstein**: Corresponde a todas las versiones/revisiones sucesivas del artículo de Wikipedia en inglés sobre Albert Einstein, concatenadas una tras otra (historial de ediciones). Es el dataset más repetitivo de todos.

5. **World Leaders (worldleaders)**: Colección de todos los archivos pdf de los líderes mundiales de la CIA desde 2003 a 2009. Son cientos de biografías breves con estructura repetitiva, aunque menos que Einstein porque el contenido entre sí varía.

5.2. Metodología

1. Tiempo: Se midió en microsegundos (μs) por consulta RMQ (min y max juntos).
   - **Baseline (sr-index)**: se mide solo Locate() - que enumera todas las ocurrencias del intervalo [sp,ep] y posteriormente se calcula min/max. El Count() previo (que obtiene el intervalo) no se cronometra.
   - **DSA-Index**: Se mide $range_{\min}$(sp,ep) y $range_{\max}$(sp,ep) juntos (el descenso sobre la gramática aumentada con metadatos, sin enumerar ocurrencias).
   - El número reportado por dataset es el promedio de todas las consultas de ese dataset.

2. Patrones:
   - Para los datasets de synth, influenza y ecoli, fueron 1000 patrones iniciales, 100 pb cada uno, extraídos aleatoriamente del texto (random.seed(42)). Las consultas efectivas fueron 2890/1000/1000 MEMs respectivamente. El intervalo [sp,ep] se generó a partir de ropebwt3 mem -l40, el cual busca MEMs del patrón contra toda la colección. Un patrón puede generar 0, 1 o varios MEMs. Para los datasets einstein y worldleaders, los patrones iniciales consistieron en 1000 substrings de 100 caracteres, con posiciones aleatorias sobre el texto linealizado (sin \n). No se usó ropebwt3 para generar el intervalo, se usó Count() del sr-index sobre el substring completo para obtener el [sp,ep] real. Todos los patrones/MEMs se comparten entre baseline y DSA-Index, mismos intervalos y mismo hardware.

3. Espacio: La unidad de medida principal es bps (bits per symbol). Es la métrica estándar para comparar índices de distinto tamaño de texto en la misma escala. La unidad secundaria es MB absolutos, es útil para dimensionar el costo real en memoria. Se mide y se comparan DSA-Index y sr-index, variando en este último su parámetro para el trade-off espacio-tiempo s ($s = 8$ y $s = 16$). Este parámetro es la tasa de muestreo del arreglo de sufijos del sr-index: cada cuántas posiciones se guarda un valor explícito de SA:

   - $s = 8$: un valor explícito cada 8 posiciones, lo que quiere decir que hay más muestras guardadas, y más espacio, pero Locate() es más rápido.

   - $s = 16$: la mitad de muestras, por lo que ocupa menos espacio pero Locate() es más lento.

Observaciones importantes:

1. Antes de hacer el calculo de tiempos y espacio, se procuró que sr-index y dsa index dieran exactamente los mismos resultados, es decir, min(SA[sp..ep]) y max(SA[sp..ep]) para todos los datasets bajo el mismo intervalor

2. Para los datasets einstein y worldleaders, los archivos originales contienen saltos de línea que el sr-index indexa como caracteres. Para garantizar que los patrones sean substrings válidos, se crearonvesiones linealizadas, eliminando todos los \n antes de construir el índice y extraer los patrones.

5.3. Resultados

5.3.1. Trade-off espacio-tiempo global

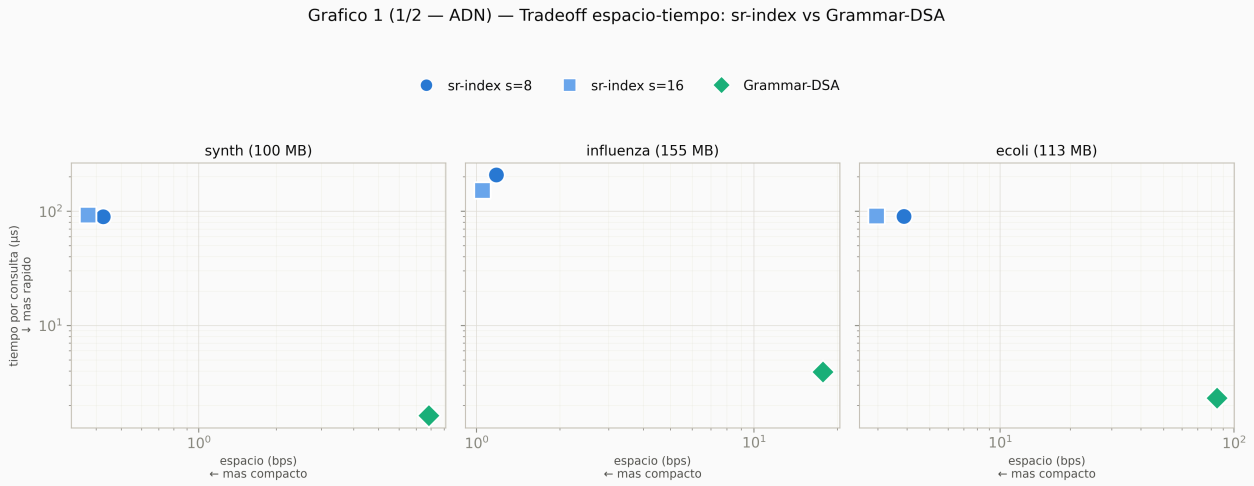

Figura 1: Trade-off espacio-tiempo: synth, influenza y ecoli

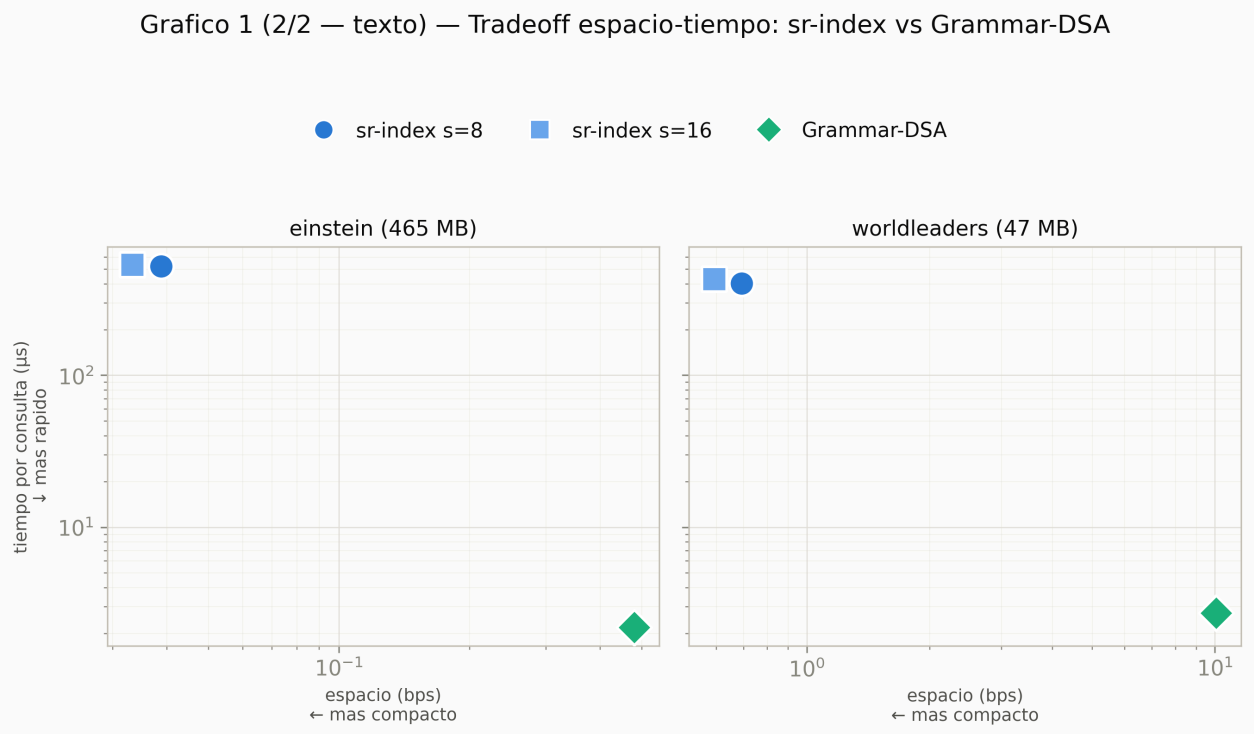

Figura 2: Trade-off espacio-tiempo: einstein y worldleaders

5.3.2. Tiempo vs número de ocurrencias

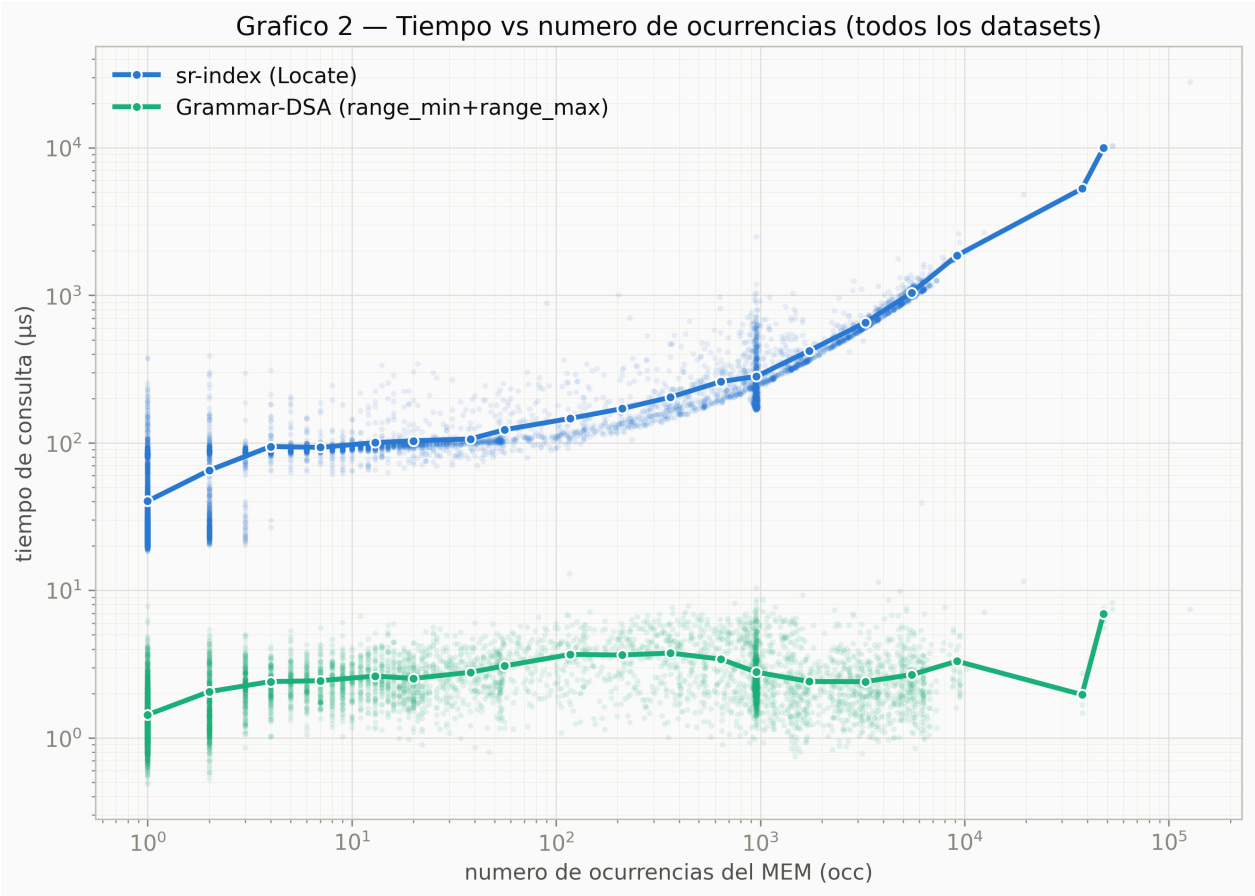

Figura 3: Tiempo vs número de ocurrencias para todos los datasets

5.3.3. Desglose de espacio

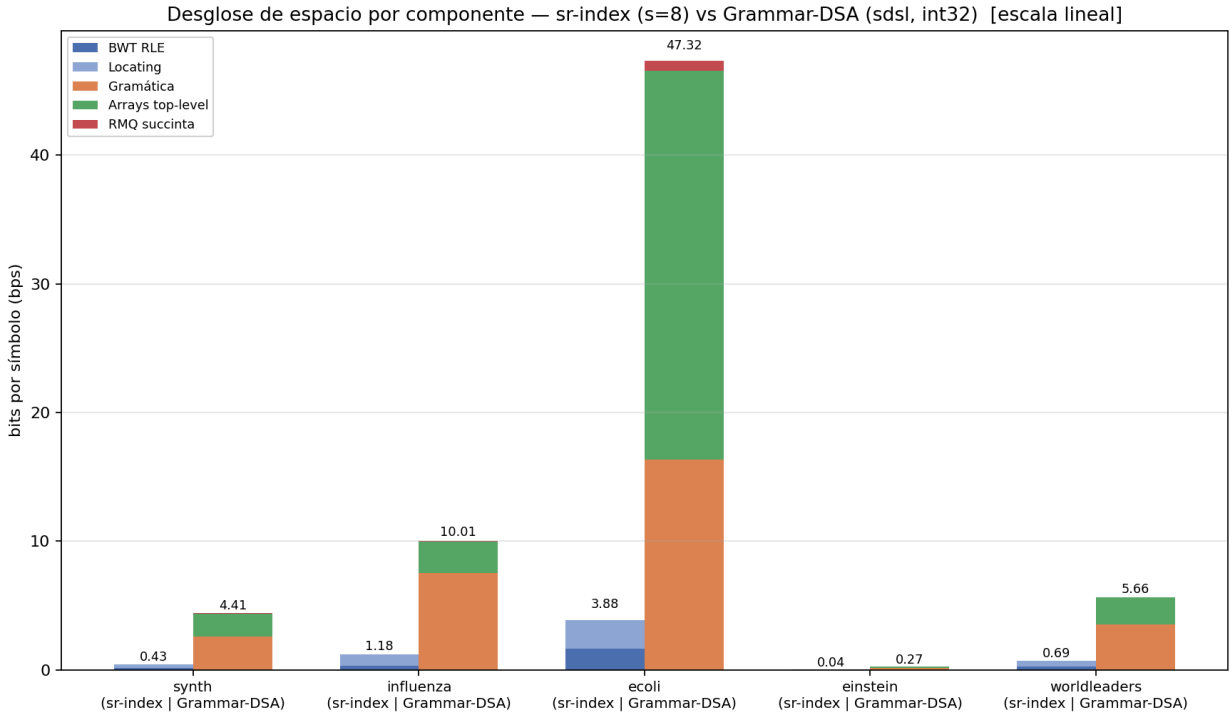

Figura 4: Desglose de espacio por dataset

Dataset	Tamaño	Queries	Baseline (µs)	DSA index (µs)	Speedup
`synth`	100 MB	2890	89.5	1.62	**55.2×**
`influenza`	148 MB	1000	207.7	3.92	**53.0×**
`ecoli`	108 MB	1000	89.9	2.31	**38.9×**
`einstein`	446 MB	999	521.3	2.19	**238.0×**
`worldleaders`	45 MB	1000	403.0	2.73	**147.6×**

Tabla 9: Tiempo promedio por query

Dataset	n	DSA bps	Sr-index s=8 bps	Sr-index s=16 bps
synth	100,000,000	4.40	0.43	0.37
influenza	154,808,555	10.01	1.18	1.05
ecoli	112,689,515	47.31	3.88	2.97
einstein	465,246,913	0.26	0.04	0.03
worldleaders	46,865,141	5.66	0.69	0.59

Tabla 10: Espacio DSA index y sr-index

Dataset	Gramática	Arrays top-level	RMQ sucinta	Total
synth	32.1 MB	22.5 MB	0.6 MB	55.1 MB
influenza	145.5 MB	47.1 MB	1.2 MB	193.8 MB
ecoli	230.2 MB	425.6 MB	10.6 MB	666.5 MB
einstein	9.7 MB	5.8 MB	0.1 MB	15.7 MB
worldleaders	20.6 MB	12.3 MB	0.3 MB	33.2 MB

Tabla 11: Desglose del espacio del DSA index

5.4. Intepretaciones

5.4.1. Tiempo

1. El grammar index es 39-261 × más rapido que sr-index con s=8, y se mantiene en un ran angosto de 1.6-3.9 μs sin importar el tamaño del dataset ni el número de ocurrencias por query

2. El speedup crece con la repetitividad del intervalo, no con el tamaño del dataset: ecoli (6.9 ocurrencias/query en promedio) tiene el speedup más bajo (39.7×) porque el baseline apenas tiene trabajo que hacer — el costo fijo del grammar domina la comparación. einstein (2,659 ocurrencias/query en promedio, máximo 37,582) tiene el speedup más alto (260.7×) porque el baseline enumera miles de ocurrencias con Locate() mientras el grammar responde en tiempo constante.

3. Esto confirma el comportamiento esperado: tiempo_grammar ≈ O(log n) independiente de k (ocurrencias), tiempo_sr-index ≈ O(k · s) lineal en k. El grammar index gana más cuanto más repetitivo es el intervalo consultado.

5.4.2. Espacio

1. El sr-index sigue siendo, por un margen amplio, la estructura más compacta: usa espacio O(r) (runs del BWT). El dsa index usa espacio proporcional al tamaño de la gramática RePair, que solo captura repetitividad estructural (substrings exactos largos) — por eso en ecoli (cepas genéticamente diversas, poca repetición exacta) el dsa index ocupa 47.3 bps contra apenas 3.9 bps del sr-index s=8 (12.2×).

2. Aumentar el sampling de s=8 a s=16 en el sr-index reduce su espacio un 11–24% adicional, lo que agranda la brecha con el dsa index a 8.1–16.0× (vs 6.9–12.2× contra s=8).

5.4.3. Tradeoff espacio-tiempo

El dsa index cambia espacio por tiempo de forma consistente en los 5 datasets: paga entre 6.9× y 16.0× más espacio que el sr-index (según dataset y sampling) para eliminar por completo la dependencia del tiempo de query con el número de ocurrencias, logrando speedups de 39.7×–260.7×. Ese tradeoff es más favorable al grammar cuanto más repetitivo

es el patrón de acceso (einstein, worldleaders) y menos favorable cuando las queries tienen pocas ocurrencias (ecoli), donde ni el tiempo ni el espacio lo favorecen tanto.

Capítulo 6

Uso de inteligencia artificial

El desarrollo de esta memoria tuvo uso de IA generativa. La asistencia con IA fue precisamente eso: asistencia para arreglar problemas de sintaxis con Typst (lenguaje de marcado que se usó para la confección de este informe), equivalencias entre Typst y LaTeX (en un principio este informe fue hecho con LaTeX, pero se decidió moverse a Typst por conveniencia y por problemas con el compilador de LaTeX), entendimiento de conceptos complejos en papers de la bibliografía, arreglo de bugs en C++, bosquejos y borradores de códigos para estructuras, y finalmente ayuda para gramática, ortografía y coherencia en la redacción de este informe. Las sugerencias de parte de la IA no fueron aceptadas inmediatamente, sino que pasaron por un proceso de testing y comprobación de que realmente servían y contribuían para esta memoria.

6.1. Modelos usados

- **Claude Opus 4.6**: Fue usado para entender conceptos complejos de los papers de la bibliografía. En un principio no se tenían los conocimientos en el ámbito de la bioinformática, y este modelo agilizó el aprendizaje de estos. En particular, ayudó a profundizar las secciones 5.2 y 6.3 de Jacm19, en donde se explica la gramática del DSA y la forma de calcular RMQs en una gramática similar a Repair, respectivamente. Los demás conceptos fueron revisados a la par con los papers de parte mía. Se usó también para hacer estructuras iniciales de la gramática y cómo combinarlo con los archivos existentes previos y necesarios (FM-index, sr-index, Repair y archivos para construir SA). Por último, se usó para entender código ajeno, esto porque daba una interpretación distinta a la mía y que podía ayudar a entender el modus operandi de la persona que programó cierto código.

- **Claude Sonnet 4.5**: Este modelo fue usado para arreglar bugs que tuve con el compilador de LaTeX. También, para traspasar lo que tenía en LaTeX, moverlo a Typst. Problemas de ortografía, redacción, gramática y coherencia también fueron vistos con este modelo, todo con su respectiva supervisión. Es aquí donde más se usó, dado que ayudó mucho a tener personalización con Typst. Un ejemplo de esto son los recuadros de definiciones, estructuras, algoritmo de consulta, entre otros que hay en este informe. La motivación de usarlo fue para que fuera todo más legible y se viera mucho mejor en mi opinión, y esta herramienta agilizaba esta tarea puesto que no manejo Typst al 100%.

La motivación de usar estas herramientas viene más que nada de acelerar los procesos mencionados anteriormente. Bajo ningún concepto fueron aceptadas las sugerencias de manera inmediata, sino más bien, se siguió el siguiente flujo:

1. Entender los conceptos básicos, con ayuda y asistencia de IA. Hacer ejemplos para practicar y revisar las respuestas de la IA junto con la lectura.

2. Habiendo entendido los conceptos, ya se tenía una idea mucho más clara de lo que se tenía que hacer, dado que justamente en Jacm19 se plasma muy claramente la idea.

3. Se instalaron las cosas necesarias para el baseline (ropebwt3 y sr-index), junto con archivos adjuntos dados por mi profesor guía que eran los siguientes: constructor de arreglos de sufijos (kkp.zip) y constructor de dataset sintético para pruebas (makeData.c, mtwister.c y mtwister.h).

4. Se usó IA para entender los repositorios y para entender la forma de instalar todo sin tener conflictos.

5. A partir de acá, el uso de la IA fue para construir bosquejos y borradores para códigos como para el DSA y la estructura de la gramática.

6.2. Ejemplos de prompts

El flujo de uso de la IA siempre fue primero revisar detenidamente la información que tenía (papers), y a partir de aquí, decirle cómo me gustaría que lo hiciera, que siempre me dé ejemplos y también que demuestre que lo que hizo está correcto (mediante tests). Ejemplos de esto se destacan a continuación:

- En base al paper Sea24, estoy leyendo el ejemplo de la figura 2 del paper, se busca el read ACATA, aquí es donde no me queda claro el procedimiento. Primero quiero que me expliques todos los sufijos que hay en el texto y cómo los obtienes paso a paso, detallando los conceptos que necesiten ser explicados (sobre todo los que no han sido definidos). Una vez hecho esto, ordénalos lexicográficamente y explícame por qué uno va primero que otro. Verifica que tus resultados sean idénticos a los del paper y valida tu procedimiento y razonamiento.

- En base a este texto, quiero que verifiques ortografía, gramática, coherencia y puntuación. Es muy importante tener también en cuenta el contexto, que a pesar de que no lo conozcas, considera siempre palabras fuera de lugar, y si este es el caso, recomiéndame siempre mejores palabras para usar. En este texto en particular no es necesario ser técnico, pero sí formal.

- Teniendo en cuenta la idea de crear metadatos para cada nodo de la gramática, quiero que me hagas un bosquejo de la estructura inicial de los metadatos, llamémosla NodeMeta, y a partir de esta, crea las funciones básicas, o sea, añadir regla, obtener metadatos, etc. Valida que tu recomendación sea útil y que sirva mediante tests.

- Dada la siguiente imagen (aquí adjunté una imagen de cómo me gustaría que se viera el formato de la definición de estructuras, ejemplos y algoritmos de este informe. Esto fue puramente estético, como se puede apreciar en el capítulo 4), quiero que me des un código en Typst para que yo pueda usarlo y agregar mis cambios. No dejar de lado que el fondo debe ser blanco y los bordes negros.

- Quiero escribir código en Typst, por ejemplo, código C++ y que tenga formato parecido a Markdown. Quiero que me des una estructura para que se me genere un recuadro con bordes y que dentro de este pueda escribir este código. Te doy un ejemplo: (aquí adjunté un ejemplo de C++ y alrededor un recuadro, donde los tipos, funciones y la sintaxis estaban explícitos mediante colores).

- En base a este repositorio, quiero que me expliques a grandes rasgos cómo funciona el sr-index, cómo se compila y ejecuta para así poder comparar resultados.

El último prompt fue uno de los que más se repitió, puesto que muchas herramientas no pude compilarlas bien, por lo que tuve que pedir ayuda incluso hasta enviando un correo en su momento a Dustin Cobas para que me explicara cómo usar su software de manera correcta. Los prompts que no funcionaron bien fueron aquellos donde no di contexto de nada y solo di una instrucción directa, por ejemplo, «dame los tests para verificar que esto está bien». En particular, cuando quise cambiarme de LaTeX a Typst, le pedí muchas veces: «a partir de este extracto hecho en LaTeX, escríbeme lo mismo pero en Typst», esto prácticamente siempre me salía mal, y tuve que cambiarlo manualmente revisando la documentación de Typst. Cuando noté este patrón, fue cuando decidí cambiar mis prompts para esto y pedirle explícitamente que siempre revisara la documentación (hubo veces en las que me dio código obsoleto).

Capítulo 7

Conclusión

En este apartado se revisa el cumplimiento de los objetivos planteados al iniciar este trabajo. También, se hace una reflexión general y las posibles mejoras futuras que se le pueden aplicar.

7.1. Objetivos

7.1.1. Objetivo general

El objetivo general fue diseñar, implementar y evaluar un índice comprimido basado en gramáticas sobre el arreglo diferencial de sufijos , capaz de responder consultas fundamentales de tipo RMQ de manera eficiente en colecciones genómicas altamente repetitivas, con el fin de mejorar la detección de ocurrencias relevantes de secuencias y apoyar la identificación del nodo ancestral de aparición de genes en árboles filogenético. Este objetivo se cumplió, dado que este índice responde eficientemente en tiempos mucho menores que su comparación que es el sr-index a costo de aumentar el espacio entre 7 y 16 veces dependiendo de la repetitividad del dataset.

7.1.2. Objetivos especificos

OE1 (Revisar el estado del arte en índices comprimidos para colecciones repetitivas): Antes de empezar con el trabajo propiamente tal, se revisó todo el material disponible para de esta manera entender mejor lo que se requería.

OE2 (Definir formalmente la estructura de datos propuesta sobre el DSA): Se definió los metadatos necesarios para el DSA .

OE3 (Implementar un prototipo funcional del índice DSA-gramática): Se implementó la estructura del DSA, su compresión (gracias a una implementación para enteros balanceados del profesor Gonzalo) y las operaciones de consulta requeridas.

OE4 (Integrar el índice en el pipeline de detección de MEMs): En efecto, teniendo instalado el baseline, el dsa index se integra al pipeline calculando min(SA[sp..ep]) y max(SA[sp..ep]).

OE5 (Evaluar experimentalmente el rendimiento del índice): Lo primero que se hizo fue comparar los resultados con el sr-index, en el sentido de correctitud. Habiendo pasado esta primera fase, se procedió a evaluar tiempo y espacio, y se llegó a las conclusiones de la sección de experimentación.

OE6 (Analizar los resultados): Se evaluó en qué grado se mejora la eficiencia espacio/ tiempo respecto al sr-index

7.2. Reflexión

El trabajo de esta memoria fue muy desafiante y de ensayo y error. En un principio se empezó por hacer funcionar el baseline, con el cual, tuve muchos problemas con que me funcionara, no por un problema del baseline en sí, sino que un tema mío de no poder entender como hacerlo. Gracias a la ayuda de Dustin, pude entender de mejor manera como hacer funcionar el sr-index, y de aquí en adelante, la verdad todo resultó mucho más sencillo no así sin complicaciones.

Habiendo terminado la primera versión de la implementación, gracias al *feedback* del profesor pude identicar mejoras para el espacio del índice: usar simplemente la implementación de la libreria de sdsl para la estructura RMQ sucinta, y cambiar los tipos de para los metadatos de 64 bits a 32 bits. Este cambio significó una mejora increible con respecto a como estaba en un principio.

7.3. Trabajo futuro

El profesor Gonzalo sugirió, para poder optimizar el espacio, lo siguiente: guardar los nodos de la gramática en un array nodes[] en orden DFS, haciendo que el hijo izquierdo de nodes[i] siempre esté en nodes[i+1], lo que significaría almacenar solo el hijo derecho. El problema con esto es que la implementación actual de para la compresión Repair (la que se usa en esta memoria) usa un DAG y no un árbol, haciendo que en realidad no se gane nada guardando los nodos en el array. Hay una forma de guardarlos en la mitad de espacio, y este podría ser una dirección para el trabajo futuro.

Bibliografía

[1] Dustin Cobas, Travis Gagie, y Gonzalo Navarro, «Fast and Small Subsampled R-indexes», *ACM Transactions on Algorithms*, vol. 21, pp. 1-32.

[2] Travis Gagie, Gonzalo Navarro, y Nicola Prezza, «Fully-Functional Suffix Trees and Optimal Text Searching in BWT-runs Bounded Space», *Journal of the ACM*, vol. 66, pp. 1-54.

[3] Dominika Draesslerová *et al.*, «Taxonomic Classification with Maximal Exact Matches in KATKA Kernels and Minimizer Digests», en *22nd International Symposium on Experimental Algorithms (SEA 2024)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, p. 10:1–10:13.

Anexo A

Gramática

```
struct NodeMeta {
    int32_t l;      // expansion length
    int32_t d;      // sum of all values (= last prefix sum)
    int32_t m_min;  // minimum prefix sum over positions 1..l
    int32_t p_min;  // position of minimum (1-indexed, leftmost on tie)
    int32_t m_max;  // maximum prefix sum over positions 1..l
    int32_t p_max;  // position of maximum (1-indexed, leftmost on tie)
};

NodeMeta terminal_meta(int32_t v);

NodeMeta combine(const NodeMeta& y1, const NodeMeta& y2);

struct Rule {
    int left, right;
    NodeMeta meta;
};

class Grammar {
public:
    std::vector<int32_t> terminals;
    std::vector<Rule>    rules;


    int add_terminal(int32_t v);

    int add_rule(int left, int right);
    NodeMeta get_meta(int sym) const;
    void expand(int sym, std::vector<int64_t>& out) const;
};
```

Anexo B

RMQ y algoritmo

```
  static int64_t descend_min(const Grammar& g, int sym, int64_t f, int64_t a,
int64_t b) {
    const NodeMeta m = g.get_meta(sym);
    if (a == 1 && b == m.l) return f + m.m_min;
    const Rule&    rule = g.rules[sym];
    const NodeMeta ml   = g.get_meta(rule.left);
    int64_t        l1   = ml.l;
    if (b <= l1) return descend_min(g, rule.left,  f,  a,  b);
    if (a >  l1) return descend_min(g, rule.right, f + ml.d, a - l1, b - l1);

    int64_t r1 = descend_min(g, rule.left,  f,  a, l1);
    int64_t r2 = descend_min(g, rule.right, f + ml.d, 1, b - l1);
    return std::min(r1, r2);
}

class RangeQuery {
public:
    RangeQuery(const Grammar& g, const std::vector<int>& top_level);

    int64_t range_min(int64_t sp, int64_t ep) const;
    int64_t range_max(int64_t sp, int64_t ep) const;

    int64_t total_length() const { return L.empty() ? 0 : L.back(); }

    size_t size_in_bytes() const {
        // rmq_succinct_sct almacena un bit_vector de 2K bits
        size_t rmq_bits = rmq_min.sct_bp.size() + rmq_max.sct_bp.size();
        return symbols.size() * sizeof(int)
             + L.size()      * sizeof(int64_t)
             + A.size()      * sizeof(int64_t)
             + M_min.size()  * sizeof(int64_t)
             + M_max.size()  * sizeof(int64_t)
             + (rmq_bits + 7) / 8;
    }
```

```
RangeQuery::RangeQuery(const Grammar& g_ref, const std::vector<int>&
top_level)
    : g(&g_ref), symbols(top_level) {
    int K = (int)top_level.size();
    L.resize(K + 1, 0);
    A.resize(K + 1, 0);
    M_min.resize(K);
    M_max.resize(K);
    for (int i = 0; i < K; i++) {
        NodeMeta m = g->get_meta(top_level[i]);
        L[i + 1] = L[i] + m.l;
        A[i + 1] = A[i] + m.d;
        M_min[i] = A[i] + m.m_min;
        M_max[i] = A[i] + m.m_max;
    }
    constexpr uint64_t SIGN = (uint64_t)1 << 63;
    sdsl::int_vector<64> iv(K);
    for (int i = 0; i < K; i++) iv[i] = (uint64_t)M_min[i] ^ SIGN;
    rmq_min = sdsl::rmq_succinct_sct<true>(&iv);
    for (int i = 0; i < K; i++) iv[i] = (uint64_t)M_max[i] ^ SIGN;
    rmq_max = sdsl::rmq_succinct_sct<false>(&iv);
}
```

Anexo C

DSA

```
void build_dsa(const std::string& text_path, const std::string& dsa_path) {
    FILE* tf = fopen(text_path.c_str(), "rb");
    if (!tf) throw std::runtime_error("build_dsa: cannot open " + text_path);

    fseek(tf, 0, SEEK_END);
    long n = ftell(tf);
    fseek(tf, 0, SEEK_SET);
    if (n <= 0) { fclose(tf); throw std::runtime_error("build_dsa: empty file
" + text_path); }

    unsigned char* text = new unsigned char[n];
    if ((long)fread(text, 1, n, tf) != n) {
        fclose(tf); delete[] text;
        throw std::runtime_error("build_dsa: read error on " + text_path);
    }
    fclose(tf);

    int* sa = new int[n];
    if (divsufsort(text, sa, (int)n) != 0) {
        delete[] text; delete[] sa;
        throw std::runtime_error("build_dsa: divsufsort failed");
    }
    delete[] text;


    unsigned* dsa = new unsigned[n];
    dsa[0] = 2u * (unsigned)sa[0];
    for (int i = 1; i < (int)n; i++) {
        int d = sa[i] - sa[i - 1];
        dsa[i] = (d >= 0) ? (unsigned)(2 * d) : (unsigned)(-2 * d - 1);
    }
    delete[] sa;

    FILE* of = fopen(dsa_path.c_str(), "wb");
    if (!of) {
        delete[] dsa;
        throw std::runtime_error("build_dsa: cannot create " + dsa_path);
    }
    if ((long)fwrite(dsa, sizeof(unsigned), n, of) != n) {
```

```
        fclose(of); delete[] dsa;
        throw std::runtime_error("build_dsa: write error on " + dsa_path);
    }
    fclose(of);
    delete[] dsa;
}
```