Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Arreglos de sufijos para alineamiento de secuencias de ADN con memoria acotada

Tesis presentada para optar al título de Licenciado de la Universidad de Buenos Aires
en el área de Ciencias de la Computación

Alejandro Deymonnaz

Directora de Tesis: *Dra. Verónica Becher*, FCEyN, Universidad de Buenos Aires.

Jurados:
Dr. Javier Marenco, FCEyN, Universidad de Buenos Aires.
Lic. Martín Urtasún, FCEyN, Universidad de Buenos Aires.

Buenos Aires, Marzo de 2012

Arreglos de sufijos para alineamiento de secuencias de ADN con memoria acotada

Resumen

El secuenciamiento de ADN sufrió una reciente revolución a raíz de las llamadas tecnologías de Secuenciamiento de Nueva Generación o *Next Generation Sequencing* en inglés, que permiten secuenciar en la actualidad más de mil millones de nucleótidos por día en una sola máquina con un costo comparativamente bajo. La clave de esta eficiencia en el secuenciamiento radica en el alto paralelismo, secuenciando millones de cadenas cortas de ADN en una sola pasada.

El problema del mapeo de reads consiste en ubicar o *alinear* dentro de un genoma de referencia previamente secuenciado las millones de cadenas cortas de ADN (*reads*) para luego ensamblarlas en un nuevo genoma. Por el tamaño y cantidad de estos datos, que se espera siga creciendo con los avances de esta tecnología, el problema del mapeo de reads ofrece un desafío computacional. Entre las numerosas herramientas de software que surgieron para resolver este problema, las más eficientes se basan en una estructura de indexación completa del texto.

La indexación de texto por medio del arreglo de sufijos tiene ya 20 años y está tendiendo a ser reemplaza por los índices basados en arreglos de sufijos comprimidos o sobre la transformación de Burrows y Wheeler, que requieren menos espacio en memoria. No obstante, estos nuevos índices son más lentos.

En este trabajo proponemos un método para realizar un compromiso entre espacio y tiempo de ejecución a fin de poder utilizar un arreglo de sufijos en un contexto de memoria acotada aprovechando su eficiencia en la búsqueda aplicado al problema de mapeo de reads. Realizamos un estudio comparativo aplicando este método tanto a herramientas desarrolladas por nosotros como a herramientas existentes y concluimos las condiciones sobre las cuales es conveniente aplicarlo.

Abstract

DNA sequencing suffered a recent revolution as a result of the so-called Next Generation Sequencing technologies, now allowing to sequence more than a billion nucleotides per day in a single machine with considerable low cost. The key to this efficiency relies in the high sequencing parallelism, yielding more than a million short DNA reads in a single run.

The computational problem known as *the mapping problem* requires to locate within a previously sequenced reference genome millions of short DNA reads in order to assemble them as a new genome. Due to the size and number of the input data, and the high speed at which it is generated, the mapping problem is indeed a computational challenge. Among the many software tools have emerged to solve it, the most efficient in the trade off of memory and time consumption are based on a full-text index structure.

Text indexing using suffix arrays has already 20 years old and it is tending to be replaced by indices based on compressed suffix arrays or the Burrows-Wheeler transform, which require less memory space. However, these newer indexing structures are slower to operate.

In this work we propose a method to maximize the efficiency in the compromise between space and time when using a suffix array structure to solve the mapping problem in a context of limited memory. Using this method we performed a comparative study of how different memory sizes and data contexts impact on the mostly used existing tools as well as on tools developed by us based on known algorithms. As a conclusion we reveal the conditions that maximize efficiency in each case.

Agradecimientos

A los hijos de hermanos que me acompañaron en la carrera: Eze, Fede, Jota, Lata, Topa, PabloR, Toto, Mati, Piter, Luigi y Pancho. A los que además fueron mis compañeros del grupo KAPOW: PabloB, PabloH y Meli por las incontables líneas de código compartidas.

A mi directora de tesis, Vero, por bancarme en todas las retorcidas vueltas de este y otros trabajos que realizamos en KAPOW.

A Mathieu Raffinot por la conjetura inicial que dio lugar al estudio realizado en este trabajo.

A los investigadores que desinteresadamente colaboraron y atendieron a mis consultas: Eugene "Gene" Myers y Udi Manber por proporcionarme el código fuente de su implementación del algoritmo de búsquda, Stefan Kurtz por la licencia de uso de su programa Vmatch y comentarios de su implementación y Oguzhan Kulekci por sus comentarios sobre PSI-RA.

Finalmente, a mi familia, a mis viejos, por aguantarme mientras cursaba la carrera.

Índice general

Capítulo 1

Introducción

Presentación del problema El presente trabajo se realiza en torno al problema computacional de la *búsqueda de palabras en un texto fijo*. El problema inicial es encontrar la ubicación de una o más palabras determinadas dentro del texto, aunque existen algunas variantes. Indistintamente en vez de 'palabra' se puede utilizar el término cadena, secuencia (finita de símbolos), patrón, o en inglés *string*, *pattern*, o *query*. El problema presenta algunas variantes

- si se busca una sola palabra, un conjunto finito de palabras, o un conjunto infinito de ellas.
- si la búsqueda es con apareo exacto o inexacto.
- si se desea conocer las posiciones, o sólo la cantidad de ocurrencias.

Búsqueda de cadenas como algoritmo Los algoritmos de búsqueda de palabras, también llamados algoritmos de apareo de palabras, en inglés *string searching/matching algorithms*, tienen larga data y han sido bien estudiados. Una presentación excelente se puede leer en el Capítulo 32 [CLRS09].

Desde el punto de vista teórico, puede decirse que el problema de búsqueda de palabras tiene solución algorítmica óptima en tiempo lineal. Sin embargo, cada variante del problema de búsqueda tiene una gama de soluciones algorítmicas, que tienen un mejor o peor desempeño en un contexto real con memoria acotada. Su eficiencia en la práctica queda determinada por las constantes ocultas en las medidas de complejidad asintótica de tiempo y espacio.

Para buscar una sola palabra se destacan algoritmos óptimos que realizan primero un preprocesamiento lineal de la palabra a buscar y luego realizan la búsqueda en tiempo lineal en el tamaño del texto, entre los que se encuentran el algoritmo de KMP (Knuth-Morris-Pratt) [KMJP77] y el de Boyer-Moore [BM77].

Para buscar un conjunto finito de palabras dentro de un texto, existe también un algoritmo óptimo desde el punto de vista de la complejidad temporal, que realiza un preprocesamiento lineal sobre el conjunto de palabras a buscar y luego la búsqueda en simultáneo de todas las palabras linealmente en el tamaño del texto. El primero de estas características fue el algoritmo de Aho and Corasick [AC75] que generaliza el de KMP usando una máquina de estados finita.

Búsqueda de cadenas como estructura de datos Si bien el problema de buscar una o más palabras dentro de un texto pareciera estar computacionalmente resuelto de manera óptima, el panorama cambia cuando el texto sobre el que se busca es dado de antemano y permanece fijo. En este contexto, el algoritmo que resuelva el problema tendrá dos complejidades temporales asociadas: el tiempo de *inicialización* o *preprocesamiento* del texto y el tiempo de *consulta* en el que se proporciona además la palabra a buscar. Una solución al problema de la búsqueda de palabras pasa a verse como una estructura de datos si es posible reutilizar los datos de inicialización o preprocesamiento del texto, para más de una consulta de búsqueda.

De esta forma, los algoritmos "óptimos" recién mencionados tienen un tiempo de preprocesamiento nulo y un tiempo de consulta lineal en el tamaño de la palabra a buscar *más* el tamaño del texto, dado que al momento de inicialización sólo se dispone del *texto* y estos algoritmos comienzan procesando la *palabra* a buscar. Otros algoritmos podrían tener un desempeño mejor, o simplemende diferente, con un mayor tiempo de inicialización y un menor tiempo de consulta.

Desde este punto de vista, cobra interés una característica más de una solución al problema: la cantidad de memoria utilizada. En especial, en el contexto de un texto fijo proporcionado de antemano, cobra especial

interés la cantidad de memoria que requiere la estructura una vez finalizada la inicialización, dado que esta es la cantidad de memoria que se requiere para buscar. Dicho de otro modo, debido a que la inicialización se realiza sólo una vez pero las consultas son muchas, es especialmente importante la memoria necesaria para poder realizar la búsqueda, es decir, el tamaño de la estructura de datos.

Comparación En este trabajo nos dedicamos a estudiar el problema de búsqueda con apareo exacto de un conjunto finito de palabras, en un texto que permanece fijo y está dado de antemano, en un contexto de memoria acotada. Las soluciones eficientes a este problema preprocesan el texto donde se realiza la búsqueda, y generan una estructura de indexación que debe ser mantenida en memoria, de modo de aprovechar el hecho de que el texto es dado de antemano. De este modo, las distintas soluciones están caracterizadas no sólo por las complejidades temporales de la inicialización y consulta, sino también por el uso de memoria.

Si bien el tiempo de este preprocesamiento queda amortizado cuando el conjunto palabras a buscar es grande, la dificultad aquí es el espacio necesario para la estructura de índices. Surgen así distintas estructuras de datos para indexar el texto donde se realizan las búsquedas, con sus correspondientes algoritmos [ABM08]. Como es de esperar, hay una tensión entre el uso de memoria en la estructura de índices, y el tiempo de acceso a dicha estructura.

Si bien en el contexto de memoria acotada algunas estructuras de datos podrían ser prohibitivas por su uso excesivo de memoria, el problema de búsqueda de palabras admite un esquema de solución que consiste en partir el texto en bloques más chicos, y realizar la búsqueda en cada bloque por separado, reduciendo el uso de memoria por reducir el tamaño del texto de entrada. En un contexto de memoria acotada, esto implica que es posible usar una estructura de datos que tendría mayor tamaño que la memoria disponible si se la construyera para el texto entero, pero construyéndola sólo sobre partes más pequeñas del texto. Así, es posible buscar en cada una de estas partes manteniendo en memoria en todo momento la estructura de datos sólo para una de ellas y obtener el mismo resultado.

De este modo, en este trabajo nos dedicamos a comparar los algoritmos existentes para esta variante del problema de búsqueda centrando la comparación en el tiempo de consulta, en un ambiente de memoria acotada. Este problema de comparación fue planteado por Mathieu Raffinot (Laboratoire dÍnformatique Algorithmique: Fondements et Applications CNRS y Université Paris Diderot) en 2010[1] Así, esta comparación pretende discernir la estrategia a utilizar en un ambiente de memoria acotada optando entre una estructura de datos que utilice menos memoria, pero sea más lenta o una más rápida pero que por su excesivo uso de memoria deba usarse varias veces. Como resultado de este estudio damos un análisis paramétrico de la eficiencia de los algoritmos considerados. Entre los parámetros se incluyen principalmente el tamaño del texto, las características del conjunto de palabras a consultar y la cantidad de memoria disponible. En términos generales los resultados experimentales que obtuvimos confirman la relación entre los distintos parámetros que conjeturó Mathieu Raffinot. Del análisis de los resultados desprendemos una prescripción de qué solución algorítmica y qué implementación conviene usar para distintas combinaciones de estos parámetros.

1.1. Cómo leer este documento

El presente documento está dividido en varios capítulos que cada uno encierra un objetivo. Al comienzo de cada uno de ellos podrá encontrar una introdución a modo de resumen del mismo, con referencias a las distintas partes del capítulo, a fin de facilitar la consulta y proveer una visión general de la línea que sigue el documento.

En el capítulo 2 mostramos el contexto actual de las tecnologías de secuenciamiento de cadenas de ADN y su tendencia de cara al futuro, lo cual motiva y le da importancia a este trabajo en un contexto real. En el capítulo 3 presentamos los algoritmos y estructuras de datos diseñados durante aproximadamente los últimos 20 años desde un punto de vista teórico, dejando de lado nuestra aplicación y motivación principal.

A continuación de esto, estamos en condiciones de presentar en el capítulo 4 formalmente el problema a estudiar y nuestra estrategia para aprovechar la eficiencia de una estructura de datos con una elevada utilización de memoria en un contexto que no lo permite directamente. En el capítulo 5 definimos los detalles de las implementaciones y datos utilizados para realizar los experimentos, los cuales se ajustan al contexto de la búsqueda de cadenas de ADN mencionado anteriormente. Posteriormente presentamos en el capítulo 6 los resultados de las pruebas realizadas dando además una lectura de los mismos para luego resumir las conclusiones en el capítulo 7.

[1] comunicacin personal.

Finalmente, en el capítulo 8 explicamos temas estrechamente relacionados con este trabajo, que no fueron incluidos por las cuestiones allí explicadas. Incluimos además el apéndice A con una descripción más técnica y detallada de *software* desarrollado en el contexto de esta tesis.

1.2. Notación

A lo largo del documento usamos

- Σ para nombrar al alfabeto;
- *palabra, cadena* o *secuencia* para referirnos a una secuencia finita de símbolos de Σ;
- Σ^* para el conjunto de todas las cadenas finitas sobre Σ;
- n para indicar la longitud del texto en donde se realiza la búsqueda;
- m para indicar la longitud de una palabra a ser buscada;
- k para la cantidad de palabras;
- M para el tamaño de la memoria de la máquina;
- Letras del principio del alfabeto, a, b, c para denotar símbolos;
- Letras del medio, i, j, k para enteros no negativos, que usualmente usaremos como índices;
- Letras del final, s, t, u, v, w para palabras del alfabeto Σ;
- Letras mayúsculas para arreglos, en particular, T para el texto de entrada, R para el arreglo de sufijos ordenado, P para la permutación inversa de R;
- Para denotar conjuntos usamos letras caligráficas mayúsculas, y escribimos $|\mathcal{A}|$ para indicar la cardinalidad del conjunto $\mathcal{A}$.
- Para referirnos a funciones escribiremos su nombre en itálica, por ejemplo, *rank*, *select*.

La longitud de una palabra w, a la que denotamos $|w|$ es la cantidad de símbolos que la componen. Numeramos las posiciones de cada símbolo en una palabra w de 1 a $|w|$, denotando $w[i]$ el símbolo en la posición i-ésima de w.

Usamos el orden lexicográfico entre palabras: w *precede lexicográficamente* a u si en la primera posición en la que difieren, el símbolo en w precede lexicográficamente al correspondiente en u. En muchos casos hacemos uso de un marcador de fin de palabra con un símbolo adicional no incluído en el alfabeto Σ, que denotamos con \$. Como convención consideramos al símbolo \$ lexicográficamente inferior a todos los demás símbolos del alfabeto.

Si $w = w[1]w[2]...w[|w|]$, entonces $u = w[i]w[i+1]...w[j]$ es la *subcadena* de w que comienza en la posición i y termina en la posición j de w, con $i \geq 1$ y $j \leq |w|$. Se dice que u *ocurre* en la posición i de w y se abrevia como $u = w[i...j]$. Si u es una subcadena de w, w es una *extensión* de u. Si u es una subcadena de w y u es distinto de w, se dice que u es una *subcadena propia*, y que w es una *extensión propia* de u.

Cuando $j = |w|$ decimos que la subcadena $w[i..j]$ es *sufijo* de la cadena w, nombrándolo tanto explícitamente como $w[i..|w|]$ o simplemente como *el sufijo* i de la palabra w. Una *rotación* de w es una cadena de la forma $w[i]w[i+1]...w[|w|]w[1]w[2]...w[i-1]$ para algún entero i. Nombramos a las rotaciones de w tanto explícitamente como $w[i..|w|] + w[1..i-1]$ o simplemente como *la rotación* i de la palabra w.

Como es usual, usamos la notación asimptótica de Landau para describir el crecimiento de las funciones en términos de funciones más simples. Tomamos la notación estandard en unidades en potencias de 2 bytes, $1\ Kb = 2^{10}$, $1\ Mb = 2^{20}$, $1\ Gb = 2^{30}$. Usamos la nomeclatura de *bp* (base pair) para referirnos a una letra del alfabeto genómico $ACGT$, y abreviamos la palabra nucleótido como *nt*. Escribimos Kbp, Mbp, Gbp para denotar respectivamente, mil, un millon, y mil millones de estas letras.

Capítulo 2

Motivación

2.1. Del secuenciamiento al problema de mapeo

En los últimos diez años el secuenciamiento de ADN avanzó sustancialmente, en base a nuevos métodos y nuevos dispositivos que llevan el nombre de Secuenciamiento de Nueva Generación, en inglés *Next Generation Sequencing* [SJ08]. En esta tecnología el material de ADN a secuenciar es partido en pequeños pedazos aleatoriamente que luego se secuenciarán independientemente. De este modo, los secuenciadores producen secuencias correspondientes a la lectura de estos pedacitos, pero no proveen su posición o su hebra en un cromosoma. Por otro lado, estas secuencias cortas, que reciben el nombre de *reads*, se obtienen de a millones en cada proceso de secuenciamiento [Met10], gracias a que el mismo se realiza en paralelo para muchísimas de estas secuencias. De este modo, para ubicar la posición y hebra original de cada *read* es necesario realizar un tratamiento computacional posterior.

A partir de un conjunto de reads hay dos posibilidades para hallar la secuencia inicial, según se cuente con un genoma de referencia o no.

En caso de no disponer de un genoma de referencia, el problema se denomina *ensamblado de novo*, o simplemente *ensamblado* [MKS10]. Debido a la gran cantidad de lecturas que se realizan de las múltiples copias del material de ADN original, los reads se superponen unos con otros. El proceso consiste en ensamblarlos de manera de armar la secuencia inicial completa de ADN, proceso que está íntimamente emparentado con el problema matemático de encontrar una secuencia de longitud mínima que incluye a cada una de las secuencias de un conjunto, aunque en la práctica se debe tener una tolerancia a errores de lectura.

Por otro lado, cuando se cuenta con un genoma de referencia de la misma especie, el problema consiste en encontrar en él la ubicación de estos reads. Este problema se conoce como el *problema de mapeo*, que matemáticamente se corresponde con la búsqueda de un conjunto de palabras, los reads, en un texto, el genoma de referencia. No obstante, es interesante destacar aquí el uso de búsqueda inexacta no sólo por los posibles errores de lectura, sino también por las diferencias existentes entre el genoma de referencia y el que está siendo secuenciado. La búsqueda inexacta se basa en una medida adecuada de distancia (o similaridad) que captura distintos tipos de errores (SNPs – *Single-Nucleotide Polymorphism*, errores de lectura, etc.).

En este contexto, la dificultad radica en que los reads se generan a una velocidad y cantidad tal que excede la capacidad computacional, logrando que si la tecnología de secuenciamiento de nueva generación sige progresando como lo ha hecho hasta ahora, el *software* de indexación se convierta en el cuello de botella del análisis [RSK+09].

2.2. El problema de mapeo de reads

El problema de mapeo de reads tiene estas características particulares:

1. el alfabeto está fijo y tiene 4 letras, $\Sigma = \{A, C, G, T\}$. Estas representan los cuatro oligonucléotidos, aunque debido a la presentación de los datos, a veces es necesario utilizar la letra N para denotar el desconocimiento del oligonucleótido en una posición determinada.

2. el genoma de referencia está fijo de antemano, es grande (rondando unos pocos *Gbps*), y es razonable asumir que será usado numerosas veces. De esta característica surge naturalmente la idea de preprocesar el genoma de referencia una sola vez, y reutilizar numerosas veces la estructura de datos arrojada por el preprocesamiento.

3. La longitud de los reads en el conjunto está fija, para cada caso de búsqueda, variando en la actualidad entre $36bp$ y $500bp$. Por su parte, la cantidad de reads suele ser muy grande, rondando los millones de reads por cada pasada para las máquinas secuenciadoras actuales.

Como ya comentamos brevemente en el capítulo anterior, hay cuatro estrategias para resolver el problema de búsqueda de un conjunto de palabras en un texto: no preprocesar nada, preprocesar las palabras, el texto, o ambas. Para el problema de mapeo de reads, resulta más eficiente preprocesar el genoma de referencia, generando una estructura de *indexación*. Al paso de indexación le sigue un paso de *alineamiento* donde se buscan los reads dentro de los genomas.

2.3. Estructuras de indexación

Fuera del contexto del problema de mapeo de reads, numerosas estructuras de indexación se han diseñado para optimizar el tiempo de búsqueda de una palabra teniendo fijo y de antemano el texto. Entre estos ejemplos se encuentran los buscadores web, donde se han utilizado ampliamente índices invertidos. No obstante, estos resultan poco útiles para buscar una secuencia cualquiera dentro del texto, no predefinida ni delimitada por espacios.

Anteriormente, el árbol de sufijos, en inglés *suffix tree* [Wei73] era utilizado para indexar texto permitiendo buscar cualquier secuencia arbitraria dentro del mismo. Sin embargo, actualmente el *arreglo de sufijos* [MM90, MM93], en inglés *suffix array*, es la estructura de datos predilecta para aplicaciones genómicas, prefiriéndosela por sobre las otras estructuras clásicas de indexación, como tablas de *hash*, árboles de sufijos (*suffix trees*), y los grafos dirigidos acíclicos de palabras (*DAWG*) [Sto08]. Esta predilección se debe principalmente a que requiere menos memoria que el árbol de sufijos y el DAGW, mientras permite búsquedas rápidas para el apareo exacto, como veremos más adelante.

No obstante, los árboles y los arreglos de sufijos usan un espacio mucho mayor que los datos mismos. Dado que los datos genómicos son usualmente compresibles por un factor de 5 a 10, puede decirse que el tamaño de los árboles y de los arreglos de sufijos puede ser de 20 a 150 veces mayor que la longitud de los datos comprimidos. Si bien se creía que no había posibilidad de comprimir un arreglo de sufijos por ser este una permutación de los número de 1 a n, Grossi y Viter [GV00] mostraron que esto no es así. Esto marcó una nueva tendencia hacia el uso de *índices comprimidos*, que son índices que requieren mucho menos espacio en memoria que las estructuras recién mencionadas, permitiendo las mismas operaciones básicas que estas, con la contraparte de requerir un tiempo mucho mayor en algunos casos.

2.3.1. Estructuras de índices comprimidos

Los esquemas de indexación no comprimida requieren en el peor de los casos $\Omega(n \log(n))$ bits adicionales de espacio. Por ejemplo, los arreglos de sufijos requieren, en el costo estándar unitario de RAM, $\Omega(n)$ palabras de memoria cada una de tamaño $\Omega(\log(n))$ bits. Estos índices son más largos que el texto mismo por un factor multiplicativo de $\Omega(\log_{|\Sigma|}(n))$, que resulta significativo cuando Σ es un alfabeto fijo. Sin embargo, estos índices admiten una búsqueda rápida, ya sea en tiempo $\mathcal{O}(m + \log(n))$ o tiempo $\mathcal{O}(m)$, más un costo adicional bajo para listar la cantidad de ocurrencias de cada palabra encontrada.

Las estructuras de índices comprimidos tienen por objetivo indexar el texto completo de forma tal que convengan en cuanto al costo espacial, el tiempo de búsqueda de palabras y conteo de ocurrencias, y la practicidad de la implementación. Indexan el texto basándose en una representación comprimida del arreglo de sufijos. El tamaño final de esta construcción es muy cercana a la entropía del texto de entrada, y aún así es posible usar esta construcción para ubicar segmentos de la entrada. Por ejemplo, todo el genoma humano tiene 3 GB y puede ser indexado en poco menos que 3 GB. Esto permite mapear millones de secuencias en una computadora personal con una cantidad relativamente pequeña de memoria extra.

Entre las formas de construir y almacenar un índice comprimido, actualmente la principal se basa en una transformación específica del texto llamada Burrows-Wheeler (BWT) [BW94, ABM08] y funciones adicionales específicas de direccionamiento sobre la transformación de la BWT. Estas técnicas de fueron desarrolladas en la última década, ver por ejemplo [GV00, GV05, FM00], aunque existen otras alternativas.

Las estructuras de índices comprimidos han sido usados recientemente en biología computacional, y los principales paquetes de software que los han implementado son Bowtie [LTPS09] y BWA [LD09]. La ganancia en velocidad de estos paquetes de software se basa en:

a) avances algorítmicos relativamente recientes, en la estructura de índices comprimidos y la BWT.

b) un compromiso entre la velocidad de búsqueda y la calidad de los apareos: para el caso del apareo inexacto, es posible degradar la calidad del resultado pero obtenerlo más rápido.

A primera vista los índices comprimidos son muy promisorios. Sin embargo, al examinar sus propiedades, tienen dos inconvenientes:

En primer lugar, si bien la cantidad de memoria necesaria en el uso es pequeña, la memoria requerida para la construcción de los índices comprimidos es mucho mayor porque la BWT se construye a través de un suffix array no comprimido, volviendo al problema original de requerir $\Theta(n \log(n))$ bits de memoria. No obstante, existen algoritmos de construcción de la BWT como el de Kärkkäinen [KÖ7] que hace un compromiso entre los requerimientos de espacio y de tiempo para la construcción de la BWT, aunque el impacto en el tiempo necesario es muy marcado.

En segundo lugar pero no por eso menos importante se encuentra el inconveniente de que el tiempo necesario para realizar búsquedas y reportar sus ocurrencias en índices comprimidos es mucho mayor que en el caso no comprimido. Según [FGNV09], la referencia más citada para implementaciones de índices comprimidos, el tiempo necesario es entre 100 y 1000 veces más lenta que buscar en el suffix array.

2.4. ¿Los índices comprimidos escalan bien para el mapeo de reads?

El problema general que surge inmediatamente, y que este trabajo resuelve, es la pregunta de si las soluciones algorímicas para el mapeo actualmente implementadas en los paquetes de software escalan a un mayor número de reads, de mayor tamaño y sobre genomas de referencia más grandes. En los próximos años, en vez de mapear millones de reads, los paquetes de software deberán mapear miles de millones, y es de prever que los reads incrementen su tamaño debido a nuevas tecnologías de secuenciamiento. Recordemos entonces que con las soluciones algorítmicas del software actual, el tiempo de cómputo ya es el cuello de botella del proceso de Secuenciamineto de Nueva Generación.

Si bien en este estudio abordamos el análisis exclusivamente para el mapeo con apareo exacto, el problema es aún más crítico para apareo inexacto [Nav01]. Al lidiar con secuencias más largas también deben permitirse un mayor número de errores (en inglés *mismatches*), en el apareo inexacto. La búsqueda de secuencias cortas, digamos entre 28 y 35 nt, con a lo sumo uno o dos errores como lo hace el software Bowtie, permite trucos computacionales que no se pueden aplicar con un número mayor de errores. Por su parte, al aumentar el tamaño de los reads, el requerimiento de lidiar con intervalos (en inglés *gaps*) es inescapable al tratar secuencias largas, dado que en caso contrario, se pierden posibles alineamientos válidos.

Hasta ahora no se ha hecho un análisis riguroso de los requerimientos de tiempo y memoria de índices comprimidos en comparación con los índices clásicos, en relación a el tamaño de la entrada, la cantidad y tamaño de los reads a ser buscados.

2.5. Software existente para el problema de mapeo

Existen actualmente numerosos paquetes de software disponibles para resolver el problema del mapeo de reads, pero no todos están basados en *índices* de un texto de referencia, llamados índices *full-text*. Algunos programas como Maq [LRD08], SOAP [LLKW08], RMAP [SXZ08], ZOOM [LZZ$^+$08], PASS [CAB$^+$09], PatMaN [PSD$^+$08], RazerS [WER$^+$09] y SHRiMP [YRB08, RLD$^+$09], entre otros, resuelven el problema con una estrategia distinta a la indexación. Algunos construyen una tabla de hash de palabras presentes en los reads (Maq, RMAP, ZOOM y PASS) o en el genoma de referencia (SOAP). Con esto se ubican estas porciones de los reads en el genoma y luego se procede a un filtrado que varía según la implementación y que incluye por ejemplo una comparación exacta o inexacta a nivel de bits. SHRiMP por su parte emplea una combinación de *spaced seeds* y programación dinámica exacta con el algoritmo de Smith-Waterman [TM81]. MPscan [RSK$^+$09] por su parte realiza una búsqueda en paralelo de todos los reads recorriendo el texto sólo una vez y sin indexarlo.

Existen algunas implementaciones intermedias, que utilizan índices del genoma, pero realizan la búsqueda a través de heurísticas, como es el caso de BFAST [HMN09] que arma una lista de posiciones *candidatas* para cada read y luego realiza un apareo inexacto.

Respecto de las implementaciones que trabajan sobre el *espacio de color* generado por el secuenciamiento en la tecnología SOLiD, podemos nombrar a Mosaik [SL09] y SOCS [OVPB08]. En esta tecnología lo que se secuencia es el ligamento entre *dos* bases consecutivas, por lo que un error de lectura en un *ligamento* se traduce en que todas las *bases* siguientes son erradas si efectivamente se convierten los ligamentos en una secuencia de bases. Las implementaciones mencionadas tienen en cuenta esto para realizar el apareo inexacto directamente en el espacio de color.

Por otro lado, existen varias implementaciones que sí utilizan índices completos del genoma de referencia. Entre estas ya mencionamos a Bowtie [LTPS09] y BWA [LD09] pero también se encuentran otras también basadas en índices comprimidos como SOAP2 [LYL+09], Entre las implementaciones actuales basadas en índices clásicos, no comprimidos, podemos mencionar a Vmatch [Kur03] y segemehl [HOK+09] basadas en arreglos de sufijos aumentados.

En resumen, existe actualmente una diversidad de aplicaciones para resolver el problema de mapeo de reads, pero muchas de esas son heurísticas que funcionan bien en la práctica *hoy*. La reciente explosión en la cantidad de información genómica secuenciada por día nos plantea el interrogante de cómo escala esto a números mucho mayores de reads y genomas.

2.6. Contexto histórico y económico

La teconología convencional de secuenciamiento de ADN desarrollada en la década de 1970 permitió completar el secuenciamiento de cientos de proyectos genómicos, que van desde vertebrados (como el humano) hasta la bacteria. Es de conocimieno público que la cantidad de datos secuenciados disponibles en bases de datos públicas crece a ritmo exponencial y se predice que ésto se acentuará en los próximos años.

El Secuenciamiento de Nueva Generación también llamado secuenciamiento de *hight throughput* por el volumen de información que genera, ahora permite la adquisición más rápida y más económica de enormes cantidades de datos secuenciados, en un rendimiento que supera ampliamente los métodos de secuenciamento que eran convencionales [Mar08, PS08]. Por ejemplo, el pirosecuenciamiento disponible en la plataforma Roche/454 lanzado al mercado en 2004 puede analizar un millón de secuencias de, en promedio, 250 nucleótidos, con una exactitud del 99.5 % en menos de dos días. La tecnología SOLEXA/Ilumina, introducida en 2006 y basada en el concepto de secuenciamiento por síntesis, produce secuencias de entre 32 y 40 bp. Una corrida típica de esta tecnología produce entre 40 millones y 50 millones de reads, en un tiempo de 4 días. Finalmente, la tecnología SOLiD (Sequencing by Oligo Ligation and Detection) presentada en 2007 produce del orden de 10 millones de secuencias de entre 25 y 35 nucleótidos en un tiempo de 5 días de proceso.

Estas tecnologías se aplican a genómica, con el secuenciamiento de ADN y a transcriptómica, con el secuenciamiento de ARN que permite el análisis de expresión de genes. Prometen abordar un rango amplio de aplicaciones de análisis genético, que incluyen: genómica comparativa, detección de SNP en grandes volumenes, secuenciamiento para el análisis de pequeñas ARNs, identificación de genes mutantes en enefermedades hereditarias, identificación de transcriptomas para organismos donde hay poca información, y la investigación de genes débilente expresados. Estos descubrimientos traerán nueva comprensión en los mecanismos de salud y las enfermedades humanas. Más aún, estas nuevas técnicas tienen por objetivo circunscribir el uso de librerías de clonación para realizar cultivo de células en laboratorio. También volvieron posible, por primera vez, el secuenciamiento de muestras tomadas directamente del medio ambiente. Esto abre una nueva perspectiva en la posibilidad de realizar un compendio casi exhaustivo de biodiversidad y dinámica de población en escalas temporales y espaciales seleccionadas. Esto es la llamada *metagenómica* [BMSdlV09]: secuenciamiento de todas las secuencias de ADN recolectadas de todas las especies presentes en un nido del medioambiete. Desde el punto de vista computacional, la metagenómica abre nuevos desafíos algorítmicos, ya que el volumen de un metagenoma no baja de 8 Gbp.

Las técnicas de ensamblado desarrolladas para el secuenciamiento convencional no pueden aplicarse para el Secuenciamiento de Nueva Generación, por dos razones. Por un lado los reads son significativamente más cortos (de 40 nt para Solexa contra 700 nt para Sanger). Por el otro, el volumen de datos secuenciados para procesar es mucho mayor, porque ahora se pueden generar millones de reads en simultáneo.

Tanto para genómica, proteómica o metagenómica la tecnología de Secuenciamniento de Nueva Generación conduce a nuevos desafíos computacionales originados en la enorme cantidad de datos a ser procesados.

Capítulo 3

Algoritmos y estructuras de datos

Tanto en el contexto presentado en el capítulo anterior como en un contexto más general de búsqueda de cadenas, diferentes trabajos se han presentado sobre la indexación de texto. En este capítulo presentamos las estructuras de datos y algoritmos asociados a la indexación de texto que utilizaremos como base de este trabajo. No obstante, sólo incluiremos la descripción de la idea de los algoritmos, sin detenernos en los detalles de borde ni en su demostración de correctitud. El motivo de esto es poder darle al lector una idea sobre las alternativas planteadas por los distintos autores para cada una de las estructuras, de modo de entender el origen de las diferencias en el tiempo de ejecución y espacio utilizado. Sumado a esto, los algoritmos se presentan teniendo en cuenta el contexto histórico y situándolos dentro de una cronología de las estructuras de indexación, de modo de comprender mejor cómo y por qué se llegó a las implementaciones actuales, y su importancia al momento de publicarse.

Estas estructuras de datos pueden agruparse en tres categorías, siendo la primera de estas la indexación por arreglo de sufijos, sin compresión alguna, la cual describimos en la sección 3.1 y los algoritmos de búsqueda basados en esta estructura en la sección 3.2.

Por otro lado, en busca de disminuir el uso de memoria se encuentra la segunda categoría de algoritmos que presentaremos: los llamados índices *comprimidos*, descriptos en su versión fundacional en la sección 3.3. En la sección 3.4 encontrará otro enfoque de índice comprimido que permite una búsqueda más eficiente. A modo optimización en la búsqueda sobre las estructuras de las dos primeras categorías describimos la estrategia de la *lookup table* en la sección 3.5.

Finalmente, la tercera de estas estructuras es la llamada indexación *dispersa* o *sparse* en inglés, que si bien persigue el mismo objetivo de disminuir el uso de memoria lo hace con una estrategia completamente distinta, la cual describimos en la sección 3.6.

3.1. Arreglo de sufijos (*suffix array*)

En [MM93] se introduce la estructura de datos *suffix array* o arreglo de sufijos como una variante del *suffix tree* que resulta mucho más eficiente en cuanto al uso de la memoria. Esta estructura mantiene las propiedades de un *full-text index*, y con un poco de información extra permite las mismas operaciones que el suffix tree. Presentamos aquí un resumen completo de su estructura.

Esencialmente, el arreglo de sufijos es una permutación particular de todos los sufijos de una palabra w. Si representamos cada sufijo de una palabra w con el número de la posición en la palabra w en la que empieza el sufijo, entonces podemos ver al arreglo de sufijos como una permutación de estos números de 1 a n. Finalmente, la permutación particular que el arreglo de sufijos representa es la única permutación que ordena lexicográficamente los sufijos.

Definición 1 (Arreglo de Sufijos R_w [MM90, MM93])**.** *El suffix array R_w de una cadena w es un arreglo de permutación de largo $|w|$ tal que para cada i entre 1 y $|w|-1$, $w[R_w[i]..|w|]$ es lexicográficamente menor que $w[R_w[i+1]..|w|]$.*

Como mencionamos antes, consideramos que una palabra w termina con un símbolo terminador \$ lexicográficamente menor que todo otro símbolo del alfabeto. Esto sumado al hecho de que todos los sufijos tienen una longitud distinta nos deja en claro que esta permutación es única, ya que todos los elementos del ordenamiento son distintos.

Ejemplo 2 (Arreglo de Sufijos)**.** *Los sufijos de la palabra* yarara\$ *se ordenan lexicográficamente de la siguiente forma. En este caso, a modo de ejemplo, hacemos explícita mención del símbolo terminador.*

i	1	2	3	4	5	6	7
$w[i]$	y	a	r	a	r	a	\$

R_w	$w[i..\|w\|]$
7	\$
6	a\$
4	ara\$
2	arara\$
5	ra\$
3	rara\$
1	yarara\$

Notemos además que todos los sufijos que comienzan con la misma palabra v se encontrarán en un intervalo de R_w. Además, todas las ocurrencias de una palabra v son el prefijo de algún sufijo de w, por lo que basta nombrar el intervalo de R_w para indicar todas sus ocurrencias, siendo la longitud de este intervalo la cantidad de ocurrencias.

3.1.1. Construcción del arreglo de sufijos – Manber y Myers (1993)

Ya en la presentación del arreglo de sufijos hecha por Manber y Myers en [MM93] describieron un método de contrucción del mismo con una complejidad temporal de $\mathcal{O}(nlog(n))$. No obstante, dado que el arreglo de sufijos no es más que la lista lexicográficamente ordenada de los sufijos de una palabra, es posible construirlo en tiempo lineal realizando un árbol de sufijos de esta palabra y recorriendo sus hojas. Desgraciadamente, esta técnica tiene un costo muy elevado en cuanto al uso de memoria debido a la construcción del árbol de sufijos, que aunque aún sea lineal precisamente se intentaba reducir con la inclusión del arreglo de sufijos.

El algoritmo consiste en realizar una cantidad logarítmica de pasadas de ordenamiento sobre un arreglo, que al finalizar cada pasada contiene una permutación de los números de 1 a n. De este modo, en casa pasada se refina el ordenamiento lexicográfico del arreglo, donde cada número del arreglo representa un sufijo de w.

Este refinamiento del ordenamiento consiste en considerar inicialmente sólo el primer símbolo de cada sufijo, para luego considerar en pasadas posteriores una cantidad mayor de símbolos del comienzo de cada sufijo hasta terminar por considerar el total de símbolos. Esta estrategia de refinamiento motiva entonces la siguiente definición,

Definición 3 (h-orden)**.** *Dado un entero* h*, definimos el* h*-orden de los sufijos de una palabra w como el orden lexicográfico de los sufijos de w considerando sólo los primeros* h *símbolos de cada sufijo. Nótese que este ordenamiento podría no ser único cuando h es menor que $|w|$.*

En la primer pasada se ordenan los sufijos de acuerdo al primer símbolo, lo cual puede realizarse en tiempo lineal realizando *bucket sort* sobre el alfabeto. Esto deja un arreglo que satisface estar 1-ordenado. De allí en más, la estrategia consiste en tomar un arreglo t-ordendo y transformarlo luego de una pasada en un arreglo $2t$-ordenado. Esta idea de doblar el número de símbolos considerados es clave para lograr un ordenamiento en una cantidad logarítmica de pasadas y se origina en las ideas de Karp, Miller y Rosenberg presentadas en [KMR72].

Para entender cómo es posible pasar de un t-ordenamiento a un $2t$-ordenamiento es importante remarcar algunos hechos. En primer lugar, si dos sufijos son distintos bajo el t-ordenamiento, entonces también lo serán bajo el $2t$-ordenamiento, dado que difieren en algún símbolo de entre los primeros t símbolos. Esto implica que en una pasada se deberán ordenar según el $2t$-ordenamiento todos los conjuntos de sufijos que son *iguales* o *indistinguibles* según el t-ordenamiento. Más aún, todos los elementos iguales según un ordenamiento estarán en posiciones consecutivas en el arreglo ordenado según dicho ordenamiento. Si llamamos *h-bloque* a cada uno de estos conjuntos de sufijos indistinguibles según un h-ordenamiento vemos que el orden relativo de estos t-bloques se mantiene en el arreglo ordenado según el $2t$-ordenamiento, cambiando sólo el orden de los sufijos *dentro* de cada t-bloque.

Teniendo todo esto en cuenta, vemos que en cada pasada se debe tomar cada t-bloque y ordenar esos sufijos según el $2t$-ordenamiento para obtener al final de la pasada un arreglo $2t$-ordenado. No obstante, falta describir cómo es posible ordenar cada uno de estos bloques de manera eficiente. Si bien los primeros t símbolos de los

sufijos dentro de un mismo bloque son iguales, para comparar dos sufijos según el ordenamiento $2t$ es necesario comparar a lo sumo los *siguientes* t símbolos. La clave aquí es que todo sufijo propio de un sufijo de w es también sufijo de w. Así, si se desean comparar según los primeros $2t$ símbolos los sufijos $R_w[i]$ y $R_w[j]$ y se sabe que comparten los primeros t símbolos, entonces alcanza con comparar los sufijos $R_w[i]+t$ y $R_w[j]+t$ según los primeros t símbolos. Pero al comienzo de la pasada ya se cuenta con un ordenamiento completo según los primeros t símbolos. Luego, para comparar estos dos sufijos se debe simplemente saber en qué posición dentro del arreglo de sufijos t-ordenado se encuentran, y si se encuentran dentro del mismo bloque o no, es decir, si son distinguibles o no. Para ello se mantiene además de la permutación que se está refinando, otro arreglo de enteros con la permutación inversa, donde para cada sufijo indica la posición en el arreglo de sufijos. En este hecho radica la clave del pasaje de un t-ordenamiento a un $2t$-ordenamiento, y usa fuertemente el hecho de que lo que se está ordenando son sufijos de una misma palabra.

No obstante, si bien en este primer algoritmo de construcción del arreglo de sufijos se presentó de manera completa la forma de realizar cada pasada en $\mathcal{O}(n)$, se lo hizo a un costo de almacenar en memoria simultáneamente el texto T, R, la permutación inversa de R y un arreglo extra para mantener los contadores para realizar el ordenamiento basado en *bucket sort* dentro de cada pasada. Esta desventaja da lugar a la siguiente mejora realizada años después.

3.1.2. Faster suffix sorting – Larsson y Sadakane (1999)

Si bien la solución al problema de la construcción del arreglo de sufijos propuesta por Manber y Myers ya había alcanzado una complejidad $\mathcal{O}(n\log(n))$, las constantes involucradas tanto en el tiempo de ejecución como en el espacio utilizado dejaban margen para una mejora. Es entonces que Larsson y Sadakane publican en [LS99, LS07] un nuevo algoritmo para generar el arreglo de sufijos muy similar en estructura al descripto anteriormente. De hecho, una versión preliminar de este algoritmo había sido presentada por Sadakane un año antes en [Sad98]. Esencialmente la diferencia radica en la forma en la que se ordena cada bloque en una pasada. En este nuevo trabajo el ordenamiento se realiza a través del conocido algoritmo de Quick-sort [Hoa62] con la variante del algoritmo que divide el conjunto a ordenar en *tres* partes en cada llamada: los menores, los mayores y los iguales al *pivote*. Esta diferencia es de radical importancia teórica a la hora de analizar la complejidad del algoritmo. Al utilizar un algoritmo como Quick-sort en cada bloque dentro de una pasada, se pierde la propiedad de que cada pasada tenga una complejidad temporal $\mathcal{O}(n)$. No obstante, basándose en el esquema del algoritmo de ordenamiento de cadenas arbitrarias propuesto por Bentley y Sedgewick en [BS97], los autores demuestran que el proceso completo de construcción del arreglo de sufijos puede realizarse con una complejidad temporal de $\mathcal{O}(n\log(n))$.

De esta forma, sólo es necesario mantener en memoria dos arreglos de enteros: el arreglo de sufijos que se está construyendo y su inversa, reduciendo así el uso de memoria y mejorando en la práctica el tiempo de ejecución.

3.1.3. Construcción lineal – de Itoh y Tanaka (1999) a la actualidad

Paralelamente al trabajo de Larsson y Sadakane, aunque posterior a las ideas presentadas por Sadakane en 1998, Itoh y Tanaka presentan un nuevo algoritmo de construcción del arrego de sufijos [IT99] con un enfoque distinto que llamaron *two-stage*.

Este algoritmo fue el primero en utilizar la idea de muestreo, que retomaremos más adelante. Dividen los sufijos en dos conjuntos U_{IT} y V_{IT} definidos de la siguiente manera:

$$U_{IT} = \{w[i..n] : w[i] < w[i+1]\}$$
$$V_{IT} = \{w[i..n] : w[i] \geq w[i+1]\}$$

Notemos que esta descripción es una partición de los sufijos en dos conjuntos y que consideramos una notación relajada de $w[i]$ en la que $w[n+1]$ es el símbolo terminador \$.

Ejemplo 4. *Consideremos la cadena w = edabdccdeedab, vemos debajo anotada la partición con U o V.*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
w	e	d	a	b	d	c	c	d	e	e	d	a	b	\$
conj.	V	V	U	U	V	V	U	U	V	V	V	U	V	–

Con esta división, alcanza con ordenar sólo los sufijos en V_{IT}, dado que los sufijos en U_{IT} pueden ser ordenados e incluídos luego en tiempo proporcional a la cantidad de estos, de modo de obtener finalmente el arreglo de sufijos ordenados con todos los sufijos de w.

Esta idea es la que se denomina *muestreo*, dado que se elije una muestra adecuada de sufijos para ordenar y luego se hace un proceso adicional para incluir los restantes. Por esto es que los autores llamaron a este algoritmo de dos etapas, o *two-stage*. Esta idea de muestreo fue después retomada y refinada por Ko y Aluru en [KA03] y por Mori en [Mor05] aunque sólo en forma de implementación. Por ejemplo, Ko y Aluru definen la partición de la siguiente forma:

$$U_{KA} = \{w[i..n] : w[i..n] < w[i+1..n]\}$$
$$V_{KA} = \{w[i..n] : w[i..n] > w[i+1..n]\}$$

y ordenan el conjunto de menor tamaño entre esos dos, lo cual implica que en el peor caso ordenan $n/2$ sufijos. Esto no es otra cosa que llevar la idea de Itoh y Tanaka del nivel de símbolo al nivel de sufijo, dado que la partición en este caso se realiza comparando sufijos consecutivos en vez de sólo el primer símbolo. En efecto, si se lo analiza un poco más en detalle, cuando los símbolos $w[i]$ y $w[i+1]$ son distintos, ambas particiones coinciden para esa pareja de sufijos, pero cuando estos símbolos son iguales entonces la comparación para Ko y Aluru pasa al siguiente símbolo de cada sufijo, es decir, $w[i+1]$ y $w[i+2]$. De aquí es fácil ver que realizando un recorrido lineal desde el final de la cadena hasta el comienzo se puede etiquetar cada sufijo de manera sencilla en tiempo lineal.

Sin embargo, Yuta Mori lleva esta idea un poco más allá mostrando que sólo es necesario ordenar un subconjunto S_M de los sufijos. Si asumimos que el más pequeño de los dos conjuntos definidos por Ko y Aluru es U_{KA}, entonces definimos S_M como:

$$S_M = \{w[i..n] : w[i..n] \in U_{KA} \wedge w[i+1..n] \in V_{KA}\}$$

Pasemos ahora a la segunda etapa del algoritmo, en la que dado el ordenamiento de los sufijos en el conjunto U_{KA} debemos construir el arreglo de todos los sufijos ordenados. Veamos en el ejemplo de Ko y Aluru cómo se agregan los sufijos pertenecientes a V_{KA}. Para esto, destaquemos que si dos sufijos u y v comienzan con el mismo símbolo, con $u \in U_{KA}$ y $v \in V_{KA}$, entonces u es lexicográficamente mayor que v. La demostración sale rápidamente de la definición y puede verse en detalle en la publicación original. El corolario de este hecho es que en el arreglo de sufijos completamente ordenado, entre todos los sufijos que comienzan con el mismo símbolo, primero se encuentran los pertenecientes a V_{KA} y luego los pertenecientes a U_{KA}.

Entonces, para construir el arreglo de sufijos R de una palabra w, comenzamos de igual manera que en Larsson y Sadakane, construyendo un 1-ordenamiento basado en bucket sort sobre el primer símbolo de cada sufijo. Luego, dentro de cada bucket, debemos colocar todos los sufijos pertenecientes a U_{KA} al final del bucket. De hecho, dado que tenemos los sufijos pertenecientes a U_{KA} ordenados, podemos ubicar estos sufijos en el orden correcto y definitivo en R, al final del bucket correspondiente al primer símbolo de cada sufijo, simplemente recorriendo una vez el arreglo ordenado de los sufijos pertenecientes a U_{KA} desde el final hacia el comienzo. Finalizado esto, tenemos en R todos los sufijos de w, donde los pertenecientes a U_{KA} están en su posición correcta, mientras que los pertenecientes a V_{KA} están dentro del 1-bloque correcto, pero no necesariamente en el orden correcto. El paso final para terminar de ubicar los sufijos es simple, aunque su demostración un poco más compleja. Al final de cada bucket hay un conjunto de sufijos, los de tipo U_{KA}, que ya están ordenados y en su posición definitiva. En este paso ese conjunto de sufijos definitivamente ubicados al final de cada bucket va a ir creciendo. Este paso final consiste entonces en recorrer completamente el arreglo R desde el comienzo hasta el final de modo que para cada i, si $R[i]-1$ es un sufijo de tipo V_{KA} entonces se mueve el sufijo $R[i]-1$, dentro de su bucket, al final del grupo de elementos aún no ordenados y se lo marca en su posición definitiva haciendo crecer en 1 este conjunto. Es decir, se lo mueve justo al comienzo del mencionado conjunto de elementos definitivamente ubicados, en su bucket correspondiente. La demostración de correctitud puede realizar por inducción sobre i, probando que cuando le toca el turno de analizar el sufijo en la posición i de R, este ya se encuentra en su ubicación definitiva, y entonces, todos los anteriores también.

Para concluir este algoritmo, los autores muestran un algoritmo recursivo con una complejidad temporal lineal para ordenar los sufijos del conjunto U_{KA} en lo que llamaron la primer etapa. Con esta idea más algunos cuantos trucos de implementación, los autores describen entonces lo que ellos mismos denominan el primer algoritmo de complejidad temporal $\mathcal{O}(n)$ que construye *directamente* el arreglo de sufijos con un costo en memoria de $8n$ bytes más $1{,}25n$ bits.

Paralelamente al trabajo de Ko y Aluru otros dos autores, Kärkkäinen y Sanders, presentaron también el mismo año en [KS03] otro algoritmo de construcción *directa* del arreglo de sufijos, de complejidad temporal también lineal siguiendo una idea similar.

Desde ese entonces hasta la fecha, otros algoritmos de construcción directa y lineal han surgido, algunos de los cuales pueden verse por ejemplo en el *survey* publicado por Puglisi et al. en [PST07]. Posteriormente a este

estudio, podemos destacar por su reciente publicación y su eficiencia tanto en tiempo de ejecución como en memoria extra necesaria, el trabajo de Nong, Zhang y Chan [NZC09, NZC11] con el algoritmo que denominaron *SA-IS*. Este algoritmo continúa la idea descripta en esta sección de Ko y Aluru, pero utilizando un muestreo diferente y una estrategia de ordenamiento de cadenas demoninado *induced sorting*.

3.2. Búsqueda en el arreglo de sufijos

Si bien el *suffix array* fue propuesto entre otras cosas como un reemplazo más eficiente en el uso de la memoria del *suffix tree*, para cumplir con este objetivo es necesario cuanto menos poder realizar las mismas operaciones que aquella estructura de manera eficiente. En particular, la operación que nos interesa es la de poder determinar si una cadena v de largo m es subcadena de w.

En el *suffix tree*, eso es equivalente a encontrar un sufijo de w que *comience* con v. De su característica de ser un árbol Patricia surge que esta operación puede realizarse en $\mathcal{O}(m)$.

Sin embargo, para el *suffix array* las cosas son distintas. En primer lugar, este arreglo de sufijos es un arreglo de enteros o punteros, que de por sí no es un *self-index*. Dicho de otro modo, en el arreglo de sufijos sólo está la información del orden lexicográfico de los sufijos, pero no hay información acerca del texto. Por lo que además del mencionado arreglo, es necesario contar con el texto de alguna u otra forma. Luego, suponiendo que se cuenta con el texto w original, ya sea con el arreglo T de caracteres explícitamente o de otra forma, es posible determinar si v es subcadena de w realizando una búsqueda binaria sobre el arreglo de sufijos. Consideremos por un momento, para cada posición i de la cadena w, la subcadena de largo m que comienza en i (o más corta si $i + m > 1 + n$). Si tomamos el orden del arreglo de sufijos, vemos que ese mismo es un ordenamiento lexicográfico de estas subcadenas de longitud m, por lo que realizando una búsqueda binaria es posible encontrar el intervalo de R_w tal que todos los sufijos en ese intervalo comienzan con v. No obstante, cada comparación entre cadenas toma un tiempo $\mathcal{O}(m)$, por lo que la complejidad temporal de una consulta asciende a $\mathcal{O}(m \log(n))$.

Para mejorar esta complejidad es necesario contar con cierta información extra, lo que nos conduce a la definición de la siguiente función.

3.2.1. LCP – Longest Common Preffix

La función LCP recibe su nombre del inglés *Longest Common Preffix* o prefijo más largo común.

Definición 5 (Función LCP)**.** *Se define al* Longest Common Preffix *entre dos cadenas u y v, o $LCP(u, v)$, como el mayor entero no negativo l tal que $u[1..l] = v[1..l]$.*

Basándose en esta función, definimos ahora el arreglo LCP_w asociado a un arreglo de sufijos R_w, como un arreglo de n posiciones donde se almacena la longitud del prefijo común más largo entre cada par de sufijos de w *consecutivos* en el orden lexicográfico, de la siguiente manera:

$$LCP_w[i] := LCP(w[R_w[i]..n], w[R_w[i+1]..n])$$

Para completar la definición, tomamos además $LCP_w[n] = 0$.

Ejemplo 6. Ejemplo de un arreglo de sufijos R y el arreglo LCP correspondiente para la palabra $w =$ *mississippi*.

$LCP_w[i]$	$R_w[i]$	$w[R_w[i]..n]$
	11	i
1	8	ippi
1	5	issippi
4	2	ississippi
0	1	mississippi
0	10	pi
1	9	ppi
0	7	sippi
2	4	sissippi
1	6	ssippi
3	3	ssissippi

Como mencionamos antes, todas las ocurrencias de una subcadena s se encuentran en el arreglo de sufijos como el prefijo de un conjunto de sufijos consecutivos en este arreglo. Esto implica que si dos sufijos de w, $u = w[R_w[i]..n]$ y $v = w[R_w[j]..n]$, comparten un mismo prefijo s, entonces todos los sufijos entre las posiciones i y j del arreglo de sufijos comparten el mismo prefijo s.

Este hecho nos conduce rápidamente a la siguiente propiedad para todo par de índices $i < j$:

$$LCP(w[R_w[i]..n], w[R_w[j]..n]) = \min_{i \leq k < j} LCP_w[k]$$

Esto muestra la utilidad del arreglo de LCP_w, dado que incluso almacenando sólo n valores en este arreglo es posible computar el prefijo común más largo para todo *par* de sufijos de w.

Respecto a la construcción del arreglo LCP_w, Kasai et al. mostraron en [KLA+01] un algoritmo muy simple que lo computa a partir del arreglo de sufijos con una complejidad $\mathcal{O}(n)$. Sin embargo, en las dos siguientes secciones veremos que esto no es suficiente aún para resolver eficientemente el problema que nos interesa: la búsqueda de cadenas.

3.2.2. Búsqueda de cadenas con LCP

Ya en la presentación de los arreglos de sufijos que hicieran en [MM93] mostraron un algoritmo para resolver de manera más eficiente que $\mathcal{O}(m \log(n))$ la búsqueda de una cadena de largo m dentro de la estructura. Para entenderlo en detalle retomemos la idea de la búsqueda binaria descripta anteriormente. Dada una palabra v de largo m queremos encontrar el intervalo $[l_v, r_v)$ de sufijos del arreglo R_w tal que los sufijos $w[R[i]..n]$ para $i \in [l_v, r_v)$ tengan a v como prefijo y sean los únicos sufijos con esta propiedad.

En efecto, este algoritmo basado en búsqueda binaria realiza dos búsquedas independientes: una para hallar l_v y otra para r_v. Tomemos como ejemplo el caso de l_v, que se define como el menor entero positivo tal que $v \leqslant w[R_w[l_v]..n]$, o como $l_v = n + 1$ si todos los sufijos son lexicográficamente menores que v.

En cada iteración de la búsqueda binaria, tenemos un intervalo $[l, r]$ del arreglo R_w dentro del cual se encuentra la solución. La iteración comienza tomando un pivote $c = \lfloor (l + r)/2 \rfloor$ y de acuerdo a un criterio determinado se continuará en la siguiente iteración con $[l', r'] = [l, c]$ o $[l', r'] = [c, r]$. El invariante para el caso de l_v en cada iteración dice que $w[R_w[l]..n] < v \leqslant w[R_w[r]..n]$, por lo que para determinar qué rama tomar en la siguiente iteración, debemos comparar v con $w[R_w[c]..n]$, lo cual puede tomar en peor caso m comparaciones de símbolos. No obstante, si tomamos $h = LCP(w[R_w[l]..n], w[R_w[r]..n])$, sabemos que v y $w[R_w[c]..n]$ comparten al menos los primeros h símbolos, dado que todos los sufijos del intervalo $[l, r]$ comparten el prefijo de largo h.

Este valor h que sólo depende de l y r se puede mantener y actualizar en cada iteración, lo cual reduce la cantidad de comparaciones de símbolos a $m - h$ a lo sumo, en cada iteración, pero esto no alcanza para reducir la complejidad.

Consideremos ahora los cuatro valores en una iteración $[l, r]$ dada, con $c = \lfloor (l + r)/2 \rfloor$:

- $h_{cl} = LCP(w[R_w[l]..n], w[R_w[c]..n])$
- $h_{cr} = LCP(w[R_w[r]..n], w[R_w[c]..n])$
- $h_l = LCP(w[R_w[l]..n], v)$
- $h_r = LCP(w[R_w[r]..n], v)$

En efecto, vemos que si bien surge directamente de la definición que $h = \min(h_{cl}, h_{cr})$, podemos asegurar que v y $w[R_w[c]..n]$ comparten al menos $\max(\min(h_l, h_{cl}), \min(h_r, h_{cr}))$, por lo que la comparación de símbolos puede comenzar a realizarse a partir del siguiente símbolo y terminar cuando se encuentre el primer símbolo distinto entre v y $w[R_w[c]..n]$. Notemos que entre una iteración y la siguiente los siguientes valores h_l y h_r se pueden calcular en $\mathcal{O}(1)$ con la información disponible al final de la iteración, es decir, con el resultado $h_c = LCP(w[R_w[c]..n], v)$ que se obtiene luego de esta comparación. Es en este contexto donde los autores mostraron una real reducción de la complejidad, dado que el total de comparaciones de símbolos que realizada por toda la búsqueda binaria es a lo sumo $2m$, reduciendo la complejidad total de la búsqueda a $\mathcal{O}(m + \log(n))$.

No obstante, queda pendiente determinar cómo se calculan los valores h_{cl} y h_{cr} en cada iteración $[l, r]$. La respuesta que dan los autores es que simplemente se precalculan todos esos valores y se almacenan junto con el arreglo de sufijos en sendos arreglos que llamaremos *Llcp* y *Rlcp*. Lo destacable de esta solución que proponen es cómo se almacenan estos valores. Dejando de lado el criterio que se use, en una búsqueda binaria como la planteada, donde se detiene la búsqueda cuando el intervalo $[l, r]$ tiene tamaño 2, ocurre una propiedad muy útil: si la búsqueda comienza siempre por el intervalo $[1, n]$, entonces todos los posibles intervalos distintos por

los que el algoritmo puede iterar son exactamente $n-2$. Es decir, sólo hay $n-2$ posibles parejas de valores l, r por los que la búsqueda binaria puede *ejecutar una iteración* (es decir, donde $r - l > 1$). Más aún, si tomamos c como función de l y r definido como antes, $c = \lfloor (l+r)/2 \rfloor$, las $n-2$ triplas diferentes l, c, r que atraviesa el algoritmo tienen un valor distinto de c, que van entre 2 y $n-1$. Dicho esto, dentro del algoritmo se puede pensar a l y r como función de c, dado que para cada c hay una única pareja l, r por la que algoritmo puede iterar que tenga a c como pivote de la iteración, lo que permite almacenar los valores h_{cl} y h_{cr} con el siguiente esquema:

- $Llcp[c] := LCP(w[R_w[l(c)]..n], w[R_w[c]..n])$
- $Rlcp[c] := LCP(w[R_w[r(c)]..n], w[R_w[c]..n])$

Por otro lado, si bien $\mathcal{O}(m+\log(n))$ representa una complejidad mucho menor que la alterantiva $\mathcal{O}(m \log(n))$ sin información adicional, no alcanza el límite óptimo de $\mathcal{O}(m)$ que ya se lograba con un árbol de sufijos. Por este motivo, los trabajos que presentaremos en la sección siguiente se dirigen en esa dirección mostrando finalmente un algoritmo con complejidad $\mathcal{O}(m)$ para la búsqueda.

3.2.3. Arreglo de sufijos aumentado (*Enhanced Suffix Array*)

La intención de reemplazar los ya antiguos árboles de sufijos por arreglos de sufijos no puede verse mejor plasmada que en el título del trabajo [AKO04] presentado por Abouelhoda, Kurtz y Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.*

El arreglo de sufijos aumentado, en inglés *enhanced suffix array*, fue presentado con ese nombre por los mismos autores dos años antes en [AKO02] como el arreglo de sufijos R_w *aumentado* con el arreglo LCP_w. En este trabajo muestran cómo es posible reemplazar cualquier solución que realizaba un recorrido *bottom-up* en un árbol de sufijos, con otra solución que utilice un arreglo de sufijos aumentado, con sus consecuentes mejores en tiempo y espacio.

Pero, volviendo al problema que nos interesa aquí, la búsqueda de cadenas en un árbol de sufijos se realiza con un recorrido *top-down*. Así es que los mismos autores publicaron en [AOK02] un algoritmo de búsqueda con complejidad temporal $\mathcal{O}(m)$ en peor caso que utiliza como información adicional al arreglo de sufijos el arreglo LCP_w y un arreglo CLD o *child table*. Este complejo algoritmo con la información adicional mencionada simula el mismo recorrido que realiza el árbol de sufijos.

Si bien no describiremos este algoritmo aquí, por los motivos que mencionaremos en la sección 5.3.1, vale la pena destacar algunos detalles del mismo. En primer lugar, el arreglo CLD mencionado, es en realidad un arreglo donde cada posición tiene tres valores asociados: *up*, *down* y $next\ell$. Sin embargo, debido a que para muchas posiciones esos valores están indefinidos, encuentran la forma de reubicar los datos de modo que los tres arreglos ocupen sólo un único arreglo de enteros.

Por otro lado, los autores presentan este álgoritmo como óptimo alegando su complejidad $\mathcal{O}(m)$ para encontrar el intervalo de ocurrencias de una palabra v, *dado un alfabeto de tamaño constante*. En efecto, la complejidad temporal considerando también el tamaño del alfabeto como variable asciende a $\mathcal{O}(m \cdot |\Sigma|)$. Si bien desde el punto de vista teórico y en un contexto con un alfabeto fijo, este algoritmo es óptimo, el hecho de tener como factor multiplicativo al tamaño del alfabeto tiene un impacto directo en el tiempo de ejecución en la práctica.

Estos trabajos, y en especial [AKO04], mostraron cómo reemplazar finalmente el árbol de sufijos por un arreglo de sufijos con una reducción considerable del espacio utilizado e incluso una mejora en la práctica en el tiempo de ejecución debido en parte a la menor cantidad de accesos a memoria. No obstante, el espacio necesario para estas estructuras seguía siendo varias veces mayor que el tamaño del texto w, o incluso más, que la cantidad de información de este texto. De allí surge toda otra rama de algoritmos llamados *comprimidos.*

3.3. Arreglo de sufijos comprimido (*Compressed Suffix Array*)

El arreglo de sufijos comprimido o *compressed suffix array* es una estructura de datos presentada por Grossi y Vitter en un abstract extendido [GV00] que terminaron de completar 5 años más tarde al presentar [GV05]. El impacto de este trabajo fue tal, que solamente en el interín desde la publicación del abstract extendido hasta el artículo definitivo el trabajo recibió al menos 60 citas, creando una nueva rama en la idexación de texto.

Esta estructura surge a raíz del siguiente hecho: Los algoritmos más eficientes de búsqueda de palabras basados en índices, descriptos hasta el momento utilizaban $\Omega(n)$ *palabras* de una máquina RAM para almanecar el índice, es decir $\Omega(n \log(n))$ *bits*. Sin embargo, el texto sobre un alfabeto Σ ocupa en cambio $n \cdot \log(|\Sigma|)$ bits,

es decir, $\Theta(n)$ bits dado que el alfabeto es de tamaño fijo. Esto implica que, teóricamente hablando, el costo adicional en términos de espacio utilizado por el índice es mucho mayor que el texto en sí, lo que deja la puerta abierta a realizar cierta mejora.

Si tomamos como ejemplo el caso de los índices basados sobre el arreglo de sufijos, este arreglo contiene una permutación de los números de 1 a n. Por tal motivo, debido a que hay $n!$ diferentes permutaciones de los números de 1 a n, no es posible representar todas ellas en una estructura de datos equivalente a este arreglo que ocupe un espacio en peor caso menor que $\mathcal{O}(\log_2(n!)) = \mathcal{O}(n \log_2(n))$ bits por una simple cuestión de cardinalidad. Esto no significa que no se pueda realizar una estructura que ocupe en peor caso menos que $\mathcal{O}(n \log(n))$ bits basada en suffix array, sino que para realizarlo hay que hacer uso de un invariante de representación más fuerte sobre el arreglo de sufijos observando este hecho: no necesariamente todas las permutaciones de los números de 1 a n son el arreglo de sufijos R_w de alguna palabra w sobre el alfabeto Σ. En efecto, hay sólo $|\Sigma|^n$ posibles palabras de largo n, que para valores de n suficientemente grandes, son muchas menos que $n!$. En este sentido, los autores logran describir una estructura que permite calcular el valor de $R_w[i]$ dado i utilizando $\mathcal{O}(n)$ bits adicionales y, desde luego, sin almacenar explícitamente el arreglo R_w. Esta operación es llamada *lookup*(i).

En el mencionado trabajo se explica cómo representar un suffix array sobre un alfabeto de sólo *dos* símbolos, utilizando $\mathcal{O}(n)$ bits extra. En caso de contar con un texto sobre un alfabeto Σ de mayor tamaño, es posible reducir el problema utilizando $\lceil \log_2(|\Sigma|) \rceil$ bits para representar cada símbolo del texto.

La estructura consiste en una descomposición recursiva del arreglo de sufijos. En cada iteración k de esta recursión se parte de un arreglo R^k y se lo descompone para formar un arreglo R^{k+1} de la mitad de tamaño. La estrategia es entonces almacenar información extra suficiente para poder reconstruir el arreglo R^k a partir del R^{k+1} más esta información extra.

El arreglo R^k tiene largo n_k y es una permutación de los números de 1 a n_k, que representa el orden lexicográfico de los sufijos que comienzan en las posiciones múltiplo de 2^k del texto binario inicial w, pero representando estos sufijos por su posición divida por 2^k, para obtener así un número entre 1 y n_k. En particular, para $k = 0$ tenemos que este arreglo R^0 es precisamente el arreglo de sufijos R_w. La idea de esta descomposición es poder reconstruir el arreglo R_w a partir de un R^{k^*} almacenado explícitamente para un k^* dado usando la información adicional de cada uno de los pasos intermedios entre 0 y k^*.

Dado entoces el arreglo R^k en la iteración k, se procede de la siguiente manera:

1. Se construye un arreglo de bits B_k de largo n_k tal que $B_k[i] = \mathbf{1}$ si $R^k[i]$ es par y $B_k[i] = \mathbf{0}$ si $R^k[i]$ impar.

2. Se construye la función Ψ_k de los números de 1 a n_k a los números de 1 a n_k de modo que emparente cada índice con un $\mathbf{0}$ en B_k en su índice *compañero* con un $\mathbf{1}$ en B_k. Es decir, si $B_k[i]$ es $\mathbf{0}$, se define $\Psi_k(i)$ como el número j tal que $R^k[j] = R^k[i] + 1$. Para extender la función a una función total se define además $\Psi_k(i) = i$ cuando $B_k[i] = \mathbf{1}$.

3. Se construye la función $rank_k(i)$ como la cantidad de $\mathbf{1}$s que hay en las primeras i posiciones de B_k.

4. Finalmente, se construye el arreglo R^{k+1} tomando todos los valores *pares* de R^k en el orden dado y dividiéndolos por 2.

Con esta descomposición, es posible reconstruir R^k con la siguiente fórmula:

$$R^k[i] = 2 \cdot R^{k+1}[rank_k(\Psi_k(i))] + (B_k[i] - 1).$$

Analicemos brevemente la expresión. Las posiciones en $\mathbf{1}$ en el arreglo B_k representan los valores pares de R^k, que son precisamente los que luego pasan (divididos por 2) al arreglo R^{k+1} en el mismo orden relativo. De este modo, si $B_k[i]$ es $\mathbf{1}$, $rank_k(i)$ marca la cantidad de valores de R^k entre las posiciones 1 e i pasaron al arreglo R^{k+1}, por lo que el valor $R^k[i]$ fue a parar a la posición $rank_k(i)$ de R^{k+1}, dividido por 2. En efecto, en este caso donde $B_k[i] = \mathbf{1}$ tenemos que $R^k[i] = 2 \cdot R^{k+1}[rank_k(i)]$ puesto que $\Psi_k(i) = i$.

Por el otro lado, cuando $B_k[i] = \mathbf{0}$ es cuando usamos la información proporcionada por $\Psi_k(i)$ que indica el *compañero* de i. En este caso tenemos, por definición, que $R^k[\Psi_k(i)] = R^k[i] + 1$, por lo que obtenemos $R^k[\Psi_k(i)]$ de la misma forma que en el caso anterior, dado que $R^k[\Psi_k(i)]$ es impar. Esto muestra cómo con la información adicional de la descomposición, $rank_k$, Ψ_k y B_k es posible recuperar R^k a partir de R^{k+1}.

Resta ver cómo es posible almacenar esta información adicional, lo cual abordaremos en las siguientes dos secciones.

3.3.1. Representación de *rank* y *select*

Para el caso del arreglo de bits B_k y la función $rank_k$ sobre este arreglo de bits, descriptos anteriormente en la descomposición del arreglo de sufijos comprimido; ya existía en la bibliografía desde hacía tiempo una representación eficiente para esta función. Entre estos trabajos, Grossi y Vitter mencionan el de Jacobson [Jac89], en donde se explica cómo representar dos funciones muy relacionadas sobre un conjunto A de enteros entre 1 y n:

- $rank_A(x)$: indica la cantidad de elementos menores o iguales a x en el conjunto A.
- $select_A(i)$: indica el i-ésimo menor elemento del conjunto A.

Notemos que una función es la inversa de la otra dado que $rank_A(select_A(i)) = i$, pero $rank_A$ no es necesariamente inyectiva. Si se representa el conjunto A como un mapa de bits se pueden responder ambas consultas en $\mathcal{O}(n)$, lo cual es demasiado costoso en tiempo. Para disminuir este tiempo se construye una estructura adicional sobre el mapa de bits, llamada *directorio*.

La explicación *detallada* del directorio fue publicada por Jacobson el año anterior a ese trabajo en su tesis de Ph.D. [Jac88], pero es de una complejidad técnica elevada. Como contrapartida de esta complejidad, tiene unas propiedades teóricas importantes: utiliza un espacio en bits sublineal en n para almacenar el directorio (más precisamente $\Theta(n \log(\log(n))/\log(n))$ y permite una consulta de $rank_A$ y $select_A$ en $\mathcal{O}(\log(n))$. Sin embargo, en la práctica, si bien la complejidad espacial sublineal parece tentadora, nuestras pruebas y cálculos mostraron que para valores de n del orden de 10^{12} o menos, la constante es tan alta que es preferible utilizar otras alternativas.

Entre las alternativas, podemos destacar la estructura de Raman et al. descripta en [RRR02] por proveer un tiempo de consulta en $\mathcal{O}(1)$ en peor caso con una complejidad espacial $\Theta(n \log(\log(n))/\log(n))$ que probó ser óptima para una estructura de datos que provea esa complejidad temporal en la consulta.

Por otro lado, con un enfonque mucho más cercano a la implementación y por ende a la realidad de la ejecución en una computadora moderna se encuentra el reciente trabajo de Vigna [Vig08] en el que muestra cómo construir una representación para *rank* y *select* para una máquina de registros de 64 bits, teniendo en cuenta cuestiones técnicas como los accesos alineados a memoria. La estrategia se basa en una técnica llamada *broadword programming* introducida con ese nombre por Knuth en [Knu07]. Esta técnica de manejo de bits consiste en realizar operaciones de bits que simulan el manejo de operaciones de tipo SIMD sobre un registro de tamaño grande, considerando varios datos dentro del mismo registro. Con esta técnica se puede realizar la misma operación sobre varios datos en simultáneo con un costo *real* equivalente a si se hiciera la misma operación pero sobre pocos bits. La estructura en sí es simple, lo que en términos teóricos se traduce a una complejidad espacial subóptima de $\Theta(n)$, siendo exactamente de un 25 % del tamaño del arreglo A para almacenar el directorio de $rank_A$ y proveyendo una complejidad temporal $\mathcal{O}(1)$.

3.3.2. Representación de Ψ_k

Representar *eficientemente* Ψ_k significa almacenar información suficiente para obtener el valor de $\Psi_k(i)$ para cada i de manera eficiente. Si bien se podría almacenar Ψ_k explícitamente como un arreglo de enteros pudiendo calcular el resultado en $\mathcal{O}(1)$ con una simple consulta en el arreglo, el objetivo es hacerlo en un tamaño menor.

Los autores muestran un método que les permite almacenar Ψ_k utilizando solamente $n(1/2 + 3/2^{k+1}) + \mathcal{O}(n/(2^k \log(\log(n))))$ bits. Para entender cómo se logra esto, analizaremos brevemente este método.

En primer lugar, para los valores de i tales que $B_k[i] = \mathbf{1}$, el valor de Ψ_k es trivialmente i, por lo que no es necesario almacenarlo. Para los demás casos, donde $B_k[i] = \mathbf{0}$, surge la siguiente observación. Dado un símbolo $c \in \Sigma$, si consideramos en orden los índices i que satisfacen simultáneamente que $B_k[i] = \mathbf{0}$ y $w[R^k[i]] = c$, entonces la secuencia de valores $\Psi_k(i)$ es una secuencia creciente.

Esta observación surge de una propiedad básica del arreglo de sufijos. Supongamos que tenemos un conjunto $\mathcal{A}$ de sufijos de w tales que todos ellos comienzan con el mismo símbolo c, y consideremos como a_i la posición donde comienza el i-ésimo sufijo en el orden lexicográfico de los sufijos de $\mathcal{A}$. La secuencia de sufijos dada por $w[a_1..n], w[a_2..n], \ldots$ está ordenada lexicográficamente, por definición, y todos ellos comienzan además por el mismo símbolo. Entonces, la secuencia de sufijos $w[a_1+1..n], w[a_2+1..n], \ldots$ también es una secuencia ordenada lexicográficamente. Por este motivo, las posiciones en el arreglo de sufijos para los valores $a_1 + 1, a_2 + 1, \ldots$ estarán tambíen en orden creciente.

Si recordamos la definición de $\Psi_k(i)$ como el j tal que $R^k[j] = R^k[i] + 1$, el ejemplo recién mencionado se traduce en que si se toma un conjunto $\mathcal{I}$ en el dominio de Ψ_k tal que para cada i del conjunto satisface $w[R^k[i]] = c$ entonces los valores j de la imagen de Ψ_k reducida al dominio $\mathcal{I}$ son crecientes.

De aquí en más, la estrategia para almacenar Ψ_k versa sobre cómo representar de manera comprimida una secuencia creciente de s elementos de q bits cada uno, lo cual resuelven en $s(2+q-\lfloor\log(s)\rfloor)+\mathcal{O}(s/\log(log(s)))$ bits con un tiempo de acceso constante. Para esto emplean una representación unaria de la diferencia entre los números con las operaciones *rank* y *select* sobre esta representación para obtener el resultado en tiempo constante.

Almacenando estas secuencias para *cada* símbolo del alfabeto binario logran almacenar todos los valores no triviales de Ψ_k, que son en total $n_k/2$. No obstante, esto así expresado sólo es válido para $k = 0$, dado que en el siguiente nivel, el arreglo R^k se corresponde sólo con los sufijos que comienzan en las posiciones múltiplos de 2^k. Para subsanar esto, consideramos que en el nivel k, el alfabeto en vez de ser binario, tiene como símbolos las 2^{k+1} cadenas binarias de largo $k+1$. En ese contexto, el arreglo R^k es esencialmente el arreglo de sufijos del texto que resulta de agrupar los símbolos binarios en grupos de $k+1$ símbolos consecutivos. Así, en el nivel k de la recursión se almacenan las listas correspondientes a las 2^{k+1} secuencias de símbolos binarios.

3.4. Búsqueda en arreglos de sufijos comprimidos

Si bien utilizando le representación del arreglo de sufijos R_w descripta en la sección anterior es posible buscar una palabra dentro del texto, con los algoritmos presentados anteriormente y un costo temporal mayor, a continuación presentamos otras estructuras de datos asociadas que permiten una búsqueda más eficiente.

3.4.1. FM-index – Ferragina y Manzini (2000)

A lo largo de la descripción de la estructura de datos denominada arreglo de sufijos comprimido, vimos cómo se exprimía la información hasta el último bit para lograr una representación eficiente en el uso de la memoria de este, permitiendo consultar el valor de $R_w[i]$ en un tiempo razonable. No obstante, nada se dijo de la representación del texto, necesario para realizar la búsqueda. Esto se debe a que la representación explícita del arreglo T, utilizando $\lceil\log(\Sigma)\rceil$ bits por cada símbolo ya es una representación óptima del mismo en peor caso.

No obstante, esta última afirmación deja de ser cierta cuando en vez de encontrarnos en un caso genérico, estamos frente a una cadena w con una baja entropía. En este contexto, cobra sentido pensar en una representación *comprimida* del texto, o más aún, una representación comprimida del texto más el índice sobre el texto, que aún permita realizar búsquedas sobre el texto sin descromprimirlo por completo.

Así es como Ferragina y Manzini publican en [FM00] una estructura de datos llamada *FM-index*[1] que combina en la misma estructura el *texto* y el *índice* sobre el texto de una manera comprimida, en medio del auge que produjo el entonces recientemente publicado trabajo de Grossi y Vitter [GV00]. Un diferencia importante de este *índice comprimido* con el anteriormente mencionado es que en esta propuesta el espacio utilizado por la estructura es una función de la entropía del texto, en particular, $\mathcal{O}(H_k(T)) + o(1)$ bits por cada símbolo de entrada, donde H_k es la entropía de orden k, para un valor de $k \geqslant 0$ fijo. Es en este sentido que describen la estructura de datos como *oportunista* en la acepción de la palabra utilizada por [FT98], dado que se aprovecha de la entropía de la palabra para reducir el espacio utilizado, en este caso.

El índice descripto se basa en una transformación íntimamente relacionada con el arreglo de sufijos, llamada la transformación de Burrows y Wheeler o BWT [BW94]. Esta transformación consiste en ordenar lexicográficamente no los sufijos $w[i..n]$ de una palabra w sino las rotaciones $w[i..n]+w[1..i-1]$ de la misma. No obstante, ordenar las rotaciones de una palabra $w\$$ es equivalente a ordenar los sufijos de w, dado que ninguna comparación de cadenas mirará más allá del símbolo terminador \$ al ordenar las rotaciones.

La transformación en sí convierte el texto $w\$$ en otro texto u de igual longitud donde $u[i] = w[R_{w\$}[i] - 1]$ para cada i, o $u[i] = \$$ cuando $R_{w\$}[i] = 0$. Esta transformación fue bien estudiada desde su aparición y tiene propiedades interesantes para compresión de datos, entre las que se destaca que es una transformación reversible y que el resultado tiene el mismo tamaño que los datos originales aunque en una forma más fácil de comprimir con otros métodos de compresión local. Un análisis de esta transformación puede verse en [Man01].

Para realizar la búsqueda de una cadena dentro del texto vamos a enunciar una importante propiedad de la BWT. Para ello, llamaremos T al texto $w\$$, U a la transformada de T definida en el párrafo anterior, R_T al arreglo de índices de las rotaciones en el orden lexicográfico, que en este caso es igual a $R_{w\$}$ y finalmente

[1] El nombre "FM" proviene del hecho de que es un índice ***f**ull-text*, que requiere sólo un ***m**inute-space* y no del nombre de los autores.

llamaremos P al arreglo de símbolos que contiene los mismos símbolos que T pero ordenados de menor a mayor, es decir, $P[i] = T[R_T[i]]$.

Proposición 7 (*LF-mapping*)**.** *Dado un índice i, llamamos c al símbolo $U[i]$ y llamamos* rango *de i, o simplemente $occ(c, i)$, a la cantidad de ocurrencias del símbolo c en $U[1..i]$. Tomamos j como la cantidad de símbolos en T (o en U)* menores *que c más el rango de i. Entonces la rotación $R_T[i] - 1$ ocupa la j-ésima posición en el orden lexicográfico de las rotaciones de T.*

Se dice que *LF-mapping(i) = j*, debido a que establece la relación entre el último (***L**ast*) símbolo cada rotación $R_T[i]$ con el primer (***F**irst*) símbolo de la *rotación* $R_T[j]$. En efecto, por definición de rotación, el último símbolo de la rotación l es el primer símbolo de la rotación $l - 1$.

Si consideramos todos los sufijos de w (o equivalentemente las rotaciones de T) que comienzan con el símbolo c, el orden lexicográfico entre ellos está enteramente determinado por los símbolos que están *a continuación* de c. Luego, el orden relativo de estos en el arreglo de sufijos, es decir el orden entre ellos, será el mismo que el orden relativo de todos los sufijos de w que comienzan *un símbolo después* que el símbolo c en w. Para resumirlo, borrarle el primer símbolo a un conjunto de sufijos que comienzan todos con el mismo símbolo, mantiene el orden entre ellos. Teniendo en cuenta esto, vemos que el rango de i es la posición relativa del sufijo $R_T[i]$ entre todos los sufijos que comienza un símbolo después que c en w. Dicho de otra forma, que el rango i es la posición relativa del sufijo $R_T[i] - 1$ entre todos los sufijos que empiezan con el símbolo c.

Este *mapping* es la clave de los índices basados en la BWT, debido a que sólo es necesario conocer el arreglo U (la BWT), la cantidad de ocurrencias totales en T (o en U) para cada uno de los símbolos de Σ, y la función occ definida sobre el arreglo U que nos permite calcular el rango de i.

Con esta información es fácil determinar recursivamente el rango de ocurrencias de una cadena v de largo m si se conoce el rango de ocurrencias de la cadena $v[2..m]$ usando el *LF-mapping*. Esta búsqueda se conoce como *búsqueda hacia atrás* o *back search* debido a que se comienza analizando la cadena a buscar desde el final hacia el comienzo.

Para almacenar esta información debemos ocuparnos de tres elementos: el arreglo U con la BWT, el arreglo C de tamaño Σ con los contadores de ocurrencias de los símbolos del alfabeto, y la función $occ(c, i)$. Para el arreglo U, ya desde su concepción existen métodos que aprovechan su estructura para almacenarlo de forma comprimida, habiendo incluso cotas de tamaño en función de la entropía del texto [Man01]. El arreglo de contadores C, dado que el alfabeto es constante y suele ser de tamaño mucho menor que el texto, no presenta un importante punto de ahorro en el espacio. Respecto de la función occ, para el caso de un alfabeto binario estamos frente una función similar a la función $rank_A$ definida en la sección 3.3.1 para la cual hay numerosas alternativas de representación. No obstante, los autores dan una solución para el caso general de alfabetos más grandes basada en una complicada estructura que aprovecha propiedades de la BWT para responder la consulta de occ en tiempo $\mathcal{O}(1)$ ocupando un espacio extra de $\mathcal{O}(n \log(\log(n))/\log(n))$ bits. Otras soluciones fueron presentadas más tarde por otros autores que difieren básicamente en la forma de representar esta función occ (vea [NM07] para un resumen de estas).

Esto resume la búsqueda de una palabra v a una complejidad óptima en tiempo $\mathcal{O}(m)$. No obstante, para saber efectivamente dónde se encuentran en w esas ocurrencias es necesario evaluar $R_{w\$}[i]$ para cada i dentro del rango de ocurrencias de v. Para evaluar esto, los autores proponen un método que utiliza algunos trucos para llegar a una complejidad temporal $\mathcal{O}(\log(n)^2)$.

3.5. Optimizando la búsqueda en un arreglo de sufijos

Una técnica popular entre los trabajos de la última década, presente en la mayoría de los programas de alineamiento basado en arreglo de sufijos es la de usar una tabla de prefijos, también llamada *lookup table*. Si bien esta idea fue presentada por Manber y Myers junto con la presentación de los arreglos de sufijos como un método para reducir el costo temporal de una búsqueda de una palabra de largo m a $\mathcal{O}(m)$ esperado manteniendo el tamaño asintótico del índice, posteriormente se ha utilizado como una optimización que en la práctica funciona muy bien.

La técnica consiste en precalcular para cada una de las cadenas posibles v sobre el alfabeto Σ de un largo ℓ dado, el intervalo del arreglo de sufijos donde se encuentran todas las ocurrencias de esta cadena. Esto permite reducir el espacio de búsqueda de una consulta buscando el prefijo de la consulta en la tabla y luego continuando con la búsqueda binaria a partir de ese rango. De este modo, si se desea encontrar todas las ocurrencias de la palabra v de largo $m \geqslant \ell$ se busca primero en la *lookup table* el rango correspondiente a $v[1..\ell]$, en $\mathcal{O}(\ell)$. Luego, a partir de este rango se continúa con un algoritmo de búsqueda binaria que comienza en este intervalo. Para el

caso de $m < \ell$, se puede considerar directamente el intervalo que abarca desde la menor hasta la mayor cadena u de largo ℓ que tenga a v como prefijo.

El costo de almacenamiento de esta tabla es entonces $2*|\Sigma|^\ell$ palabras, aunque también es posible almacenarla sólo con $|\Sigma|^\ell$ palabras teniendo especial cuidado con los sufijos más cortos que ℓ. Si las subcadenas de largo ℓ están bien distribuidas el tamaño esperado del intervalo luego de consultar la tabla es $n/|\Sigma|^\ell$. Con esta mejora, la complejidad esperada baja a $\mathcal{O}(m + \log(n/|\Sigma|^\ell))$.

En su publicación, Manber y Myers propusieron tomar $\ell = \lfloor \log_{|\Sigma|}(n/4) \rfloor$ de modo que la tabla ocupara a lo sumo n bytes, asumiendo enteros de 4 bytes, y bajando la complejidad esperada a $\mathcal{O}(m)$. No obstante eso, valores menores de ℓ aún reducen considerablemente el tiempo de ejecución, por lo que se ha incluído en varias de las implementaciones.

3.6. Arreglo de sufijos disperso (*Sparse Suffix Array*)

Como una alternativa a los arreglos de sufijos comprimidos aunque con el mismo propósito de indexar el texto utilizando menor espacio en memoria, se encuentra la estructura de datos llamada arreglo de sufijos disperso, o *sparse suffix array* en inglés.

Previo a la definición de esta estructura nos retrotraemos a 1996, incluso antes de la aparición del arreglo de sufijos comprimido, para mencionar el trabajo de Kärkkäinen y Ukkonen, llamado *Sparse Suffix Trees* [KU96]. Ya sea aplicado a los árboles de sufijos o a los arreglos de sufijos, el concepto introducido es el mismo: se toma un *sub*conjunto de los sufijos de la palabra w, apropiadamente elegido y se ordenan lexicográficamente sólo estos sufijos. Como contrapartida, esto implica un mayor tiempo de búsqueda dentro del texto.

El problema de la búsqueda de cadenas radica en que, si bien el todo substring v de w es el prefijo de algún sufijo de w, existe la posibilidad de que ese sufijo no esté en el conjunto elegido a la hora de construir el arreglo de sufijos disperso. Si bien esto no prohibe la búsqueda de una cadena v arbitraria, implica un rediseño de la estrategia de búsqueda. En ese trabajo, aunque explicado sobre árboles de sufijos, muestran un método para encontrar una cadena con caso promedio en tiempo polinomial, el peor caso resulta exponencial.

Recientemente, Chien et al. [CHSV08] retomaron esta idea pero ya como una alternativa a la compresión del arreglo de sufijos, mostrando unos resultados teóricos satisfactorios, en los cuales se basa el siguiente trabajo que presentamos a continuación.

3.6.1. Ψ-RA

Concentrados en el problema de indexar secuencias de ADN utilizando un arreglo de sufijos disperso, Külekci et al. presentaron en [KHS+10] un algoritmo paralelo para alinear *reads* dentro de un genoma de referencia.

Una de las claves del diseño de este algoritmo es la selección de un conjunto de sufijos de w tal que las posiciones de inicio de estos dentro de la palabra w están uniformemente distribuidos y a una distancia d paramétrica. A esta distancia se la llama *factor de dispersión*.

Así, dado un parámetro entero d, eligen como subconjunto de los sufijos de w aquellos que comiencen en una posición i tal que $i \equiv 0 \mod d$ para $1 \leqslant i \leqslant n$, totalizando n/d sufijos. Luego, si reinterpretamos el texto sobre el alfabeto Σ agrupando los caracteres de a d símbolos consecutivos obtenemos un texto w' de n/d símbolos sobre un alfabeto Σ^d. De esta forma, el arreglo de sufijos disperso utilizado en este trabajo no es más que el arreglo de sufijos de la cadena w': $R_{w'}$, el cual se puede almacenar directamente en $\mathcal{O}(n/d \log(n/d))$ bits.

Para realizar una búsqueda de la cadena v dentro de esta estructura, la estrategia es realizar d búsquedas diferentes. Si consideramos todas las ocurrencias de la cadena v en w, podemos particionarlas en d conjuntos clasificándolas según el valor de $i \mod d$ donde i es la posición de w donde comienza una ocurrencia de v. Para cada uno de los conjuntos de esta partición se realiza una búsqueda como se explica a continuación:

Comencemos por el caso simple, tomando el subconjunto de las ocurrencias de v que comienzan en una posición $i \equiv 0 \mod d$. En este caso, todas estas ocurrencias son prefijo de sufijos que están en el arreglo de sufijos disperso de w, por definición. Entonces una búsqueda de v sobre este arreglo de sufijos usando alguno de los métodos de búsqueda descriptos anteriormente (ver sec. 3.1) resuelve el problema para este caso.

Para los casos restantes, consideremos el subconjunto de las ocurrencias de v que comienzan en una posición $i \equiv k \mod d$ para cada k entre 1 y $d-1$. Como v ocurre en una posición i tal que $i \equiv k \mod d$, entonces podemos afirmar que la cadena $v[1+d-k..m]$ ocurre en la posición $i+d-k$ que satisface $i+d-k \equiv i-k \equiv 0 \mod d$. Esto nos lleva de nuevo al primer caso recién explicado. De esta forma, para buscar las ocurrencias de v pertenecientes a este subconjunto podemos recorrer todas las ocurrencias de la cadena $v[1+d-k..m]$ en posiciones múltiplos de d de la misma forma que en el caso anterior, y luego verificar para cada una de ellas si está efectivamente precedida por la cadena $v[1..d-k]$.

Si la búsqueda de cadenas en el arreglo de sufijos $R_{w'i}$ tiene una complejidad $\mathcal{O}(m \log(n/d))$, entonces la búsqueda en el arreglo de sufijos disperso tiene una complejidad $\mathcal{O}(d \cdot m \log(n/d) + verificación)$. El costo de verificación puede ser muy alto si el valor de m es cercano a d. Más aún, este algoritmo resulta inapropiado cuando la palabra a buscar es más corta que d, lo cual requiere un método más sofisticado y poco elegante [CHSV08]. De todos modos, dado que el parámetro d es arbitrario y que en la aplicación planteada las palabras a buscar tienen un tamaño mínimo, que se espera que siga creciendo en los próximos años, esto no es un problema grave.

La ventaja de estos índices es la existencia del factor de dispersión d, que permite elegir el compromiso que se hace entre tiempo de ejecución y memoria utilizada de acuerdo a los recursos disponibles.

Capítulo 4

Problema

Una vez que ya definimos el marco teórico de trabajo en el capítulo 1, la aplicación y motivación fundamental que dan un marco de trabajo en la práctica según describimos en el capítulo 2 y las soluciones algorítmicas que aparecieran durante los últimos años descriptas en el capítulo 3; estamos en condiciones de definir formalmente los problemas a analizar y nuestra estrategia para la utlización de estos algoritmos.

En la sección 4.1 definimos los dos problemas que analizaremos empíricamente: *count* y *locate*. Luego en la sección 4.2 mostramos una forma de reducir el problema a tamaños del texto de referencia arbitrariamente más pequeños, lo que representa la estrategia central de este trabajo para aprovechar la eficiencia en tiempo de las estructuras que utilizan una cantidad de memoria mucho mayor. Finalmente, en la sección 4.3 mostramos cómo es posible utilizar el apareo exacto de cadenas más cortas para realizar un apareo inexacto.

4.1. Definiciones

En el contexto de nuestro trabajo, emplearemos los siguientes dos problemas esenciales: *count* y *locate*.

Definición 8 (*count*)**.** *Dado un texto de referencia w de largo n y una palabra a buscar v de largo m, definimos como* count(w, v) *a la cantidad de valores i entre 1 y $n-m+1$ tales que la subcadena $w[i..i+m-1]$ es igual a v.*

Definición 9 (*locate*)**.** *Dado un texto de referencia w de largo n y una palabra a buscar v de largo m, definimos como* locate(w, v) *al conjunto de todos los valores i entre 1 y $n-m+1$ tales que la subcadena $w[i..i+m-1]$ es igual a v.*

Dado que de la definición se desprende directamente que $count(w, v) = |locate(w, v)|$, podríamos argumentar que son problemas muy similares. Sin embargo, si tenemos en cuenta las estructuras de datos vistas en el capítulo anterior, es interesante analizar los dos problemas por separado debido a que para el problema *locate* es necesario poder recorrer los valores de las posiciones donde ocurre la cadena buscada, lo cual en las distintas estructuras tiene una complejidad distinta. Esencialmente, en las estructuras de indexación basadas en arreglos de sufijos, ya sea comprimidos o no, esta diferencia radica en la forma de calcular el valor $R_w[i]$ al momento de la consulta, se tenga o no el arreglo R_w explícitamente.

4.2. Nuestra propuesta

Como ya adelantamos en la introducción de este trabajo, los dos problemas definidos anteriormente admiten partir el texto en porciones más pequeñas procesables individualmente, reduciendo así el tamaño de la entrada en los algoritmos antes mencionados.

Proposición 10 (Partición de *locate*)**.** *Dado un texto de referencia w de largo n, una palabra a buscar v de largo m y una secuencia ordenada de enteros de largo $p+1$ tal que $0 = c_1 < c_2 < \ldots < c_p < c_{p+1} = n$, entonces*

$$locate(w, v) = \bigcup_{i=1}^{p} \{a_i + j : j \in locate(w[a_i + 1..c_{i+1}], v)\}$$

donde $a_i = \text{máx}(0, c_i - m + 1)$.

Demostración. Sea $\mathcal{L}_i = locate(w[a_i + 1..c_{i+1}], v)$. Para probar la igualdad planteada procedemos a probar la doble inclusión.

- $\subseteq$ Dado $j \in locate(w, v)$ una posición del texto w en la que ocurre v, notemos que $j + m - 1 \leqslant n$ dado que la cadena v ocurre completamente dentro del texto. Para este j, consideramos el único valor i^* tal que $c_{i^*} < j + m - 1 \leqslant c_{i^*+1}$. Vemos que este valor i^* existe dado que $c_1 = 0$ y $j + m - 1 \leqslant c_{p+1} = n$. Por su parte, este valor es único debido a que la secuencia c_i es estrictamente creciente.

  Vamos a probar ahora que la ocurrencia correspondiente a j se encuentra en el conjunto $\mathcal{L}_{i^*}$. Llamemos $u_i = w[a_i + 1..c_{i+1}]$, el texto de referencia sobre el que busca *locate* para el conjunto $\mathcal{L}_i$. Deducimos de $c_{i^*} < j + m - 1$ que $j > c_{i^*} - m + 1$, del hecho de que j es la posición de una ocurrencia de v en w, que $j > 0$. Por lo tanto j es mayor que el máximo entre estos dos valores, que podemos escribir como $j > a_{i^*}$, o bien $a_{i^*} + 1 \leqslant j$. Por otro lado, por la definición de i^*, tenemos que $j + m - 1 \leqslant c_{i^*} + 1$. De estos dos últimas dos desigualdades vemos que las posiciones de los símbolos de la subcadena $w[j..j + m - 1] = v$ están dentro del intervalo de la subcadena u_{i^*}. Por este motivo, esta ocurrencia será reportada en $\mathcal{L}_{i^*}$ en la posición $j - a_{i^*}$, dado que la cadena u_i comienza a_i posiciones después que el comienzo de w.

- $\supseteq$ Dado un valor j en el conjunto de la derecha, el mismo se encuentra en al menos uno de los conjuntos que participan de la unión. Sea entonces i^* alguno de estos conjuntos tales que $j = a_{i^*} + j'$ y $j' \in \mathcal{L}_{i^*}$. Esto significa que v ocurre en u_{i^*} en la posición j', pero dado que $u_i = w[a_i + 1..c_{i+1}]$, vemos que v ocurre en w en la posición $j' + a_{i^*} = j$.

□

Observación 11. *De la unicidad del valor i^* en la parte $\subseteq$ de la demostración anterior, se deduce que en la proposición 10 la unión es además una unión disjunta.*

Corolario 12 (Partición de *count*)**.** *Dado un texto de referencia w de largo n, una palabra a buscar v de largo m y una secuencia ordenada de enteros de largo $p + 1$ tal que $0 = c_1 < c_2 < \ldots < c_p < c_{p+1} = n$, entonces*

$$count(w, v) = \sum_{i=1}^{p} count(w[a_i + 1..c_{i+1}], v)$$

donde $a_i = \max(0, c_i - m + 1)$.

Demostración. Surge de inmediato de la proposición 10 y la observación 11. □

Si bien este esquema permite efectivamente calcular *count* y *locate* sobre un texto de referencia resolviendo estos mismos problemas pero sobre varios textos de tamaño menor, existe una desventaja en el esquema así planteado. El problema radica en que el conjunto de subtextos de referencia u_i depende no sólo de w, sino del valor de m, el largo de la consulta. Esto significaría que no se podrían reutilizar los índices generados para los textos u_i con una secuencia c_i y un valor de m dado, si se desea buscar una cadena de largo m'. Esto no representa *per se* un problema serio en el contexto real planteado en el capítulo 2 dado que en varias tecnologías de máquinas de secuenciamiento el largo del *read* es fijo. No obstante, esto no ocurre para todas ellas por lo que amerita la siguiente modificación:

Proposición 13 (Partición de *locate*)**.** *Dado un texto de referencia w de largo n, una palabra a buscar v de largo m, una secuencia ordenada de enteros de largo $p + 1$ tal que $0 = c_1 < c_2 < \ldots < c_p < c_{p+1} = n$, y un parámetro entero $\ell \geqslant m$, entonces*

$$locate(w, v) = \biguplus_{i=1}^{p} \{a'_i + j : j + a'_i > a_i \wedge j \in locate(w[a'_i + 1..c_{i+1}], v)\}$$

donde $a'_i = \max(0, c_i - \ell + 1)$ y $a_i = \max(0, c_i - m + 1)$.

Este reemplazo de m por un parámetro ℓ mayor o igual que m, modifica la definición de u_i que dimos en la demostración de la proposición 10 haciéndola depender de ℓ en vez de m, al reemplazar a_i por a'_i en la llamada a *locate*. Esto hace que el mismo índice del texto u_i sirva para cualquier consulta de largo menor o igual que ℓ, requiriendo una pequeña comprobación adicional para evitar las ocurrencias repedidas manteniendo así la unión disjunta. Este es un esquema mucho más compatible con la realidad, dado que el largo de *read* está acotado en todas las tecnologías actuales por un valor relativamente chico, menor que 1000.

Si bien esta misma modificación no se puede aplicar directamente al problema *count* dado que no se cuenta con las ocurrencias de toda la cadena, es posible calcular rápidamente cuántas veces ocurre v en posiciones entre $a'_i + 1$ y a_i y descontarlas del total, dado que el tamaño de ese intervalo es $\mathcal{O}(\ell)$ lo cual es usualmente pequeño y acotado sólo por la tecnología de la máquina usada.

4.3. Apareo inexacto

Dentro de las variantes que mencionamos sobre el problema de la búsqueda de palabras, por el contexto planteado donde pueden ocurrir diferencias entre la palabra a buscar y el texto de referencia ya sea por errores de lectura como por diferencias biológicas efectivas, se destaca la variante de la búsqueda *inexacta*. Si bien dejaremos fuera del análisis empírico esta variante, principalmente por los motivos que expresaremos en la sección 8.1, queremos mencionar cómo desde el punto de vista teórico es posible reducir el problema de búsqueda inexacta, al menos en alguna de sus formas, al problema de la búsqueda exacta.

Esta reducción, sumado al hecho de que las implementaciones de las distintas variantes de búsqueda inexacta sobre las estructuras planteadas son esencialmente una modificación del algoritmo de búsqueda exacta, muestra la importancia radical que tiene el estudio del problema de la búsqueda exacta aún para este otro problema.

Definición 14 (*approxm*)**.** *Dado un texto de referencia w de largo n, una palabra a buscar v de largo m y un parámetro entero no negativo κ definimos como $approxm_\kappa(w, v)$ al conjunto de todos los valores i entre 1 y n tales que la distancia de Hamming entre $w[i..i + m - 1]$ y v es menor a igual a κ.*

Nótese que por definición, $locate = approxm_0$. La estrategia para reducir este problema a *locate* se basa en que una subcadena de v entre dos errores consecutivos tiene un apareo exacto con el texto de referencia. El problema con esta idea enunciada así es que por un lado se desconoce la ubicación de los errores en v, y por otro lado, la ubicación de estos errores depende de la posición i dentro del texto de referencia donde se realiza el apareo. Sin embargo, aún considerando estos inconvenientes se puede tomar una postura más conservadora que da lugar a la siguiente proposición.

Proposición 15 ($approxm_\kappa$ como *locate*)**.** *Dado un texto de referencia w, una palabra a buscar v de largo m, un parámetro entero positivo κ y una secuencia ordenada de enteros de largo $\kappa + 2$ tal que $0 = c_1 < c_2 < \ldots < c_{\kappa+1} < c_{\kappa+2} = m$,*

$$approxm_\kappa(w, v) = \bigcup_{i=1}^{\kappa+1} \{j - c_i : j \in locate(w, v[c_i + 1..c_{i+1}]) \wedge Hamming(w[j - c_i..j - c_i + m - 1], v) \leqslant \kappa\}$$

donde $Hamming(a, b)$ es la distancia de Hamming entre las cadenas a y b.

Notemos que esta proposición es trivialmente cierta por definición cuando $\kappa = 0$, dado que $c_1 = 0$ y la condición de la distancia de Hamming es equivalente a la igualdad en ese caso. Pero pasemos ahora a la demostración en el caso general.

Demostración. Nuevamente, procedemos a probar la doble inclusión:

- $\subseteq$ Dada una posición $j' \in approxm_\kappa(w, v)$ vemos que, por definición, $Hamming(w[j'..j' + m - 1], v) \leqslant \kappa$. Debido a esto, sabemos que $w[j'..j' + m - 1]$ y v difieren en a lo sumo κ símbolos, ubicados en a lo sumo κ posiciones distintas dentro de v. Por su parte, la secuencia c_i arbitrariamente elegida particiona la cadena v en $\kappa+1$ secuencias de la forma $v[c_i+1..c_{i+1}]$. Como para la posición j' considerada, la cadena v tiene a lo sumo κ símbolos diferentes con el texto de referencia en la posición j', por el principio de Dirichlet, podemos asegurar que existe al menos una subcadena i^* de la forma $v[c_{i^*} + 1..c_{i^*+1}]$ que no contiene ninguna diferencia con el texto de referencia en la posición correspondiente $j' + c_{i^*}$. Entonces, esa subcadena ocurre de manera exacta en el texto de referencia en la posición $j' + c_{i^*}$, por lo que será reportada por el correspondiente $locate(w, v[c_{i^*} + 1..c_{i^*+1}])$.

- $\supseteq$ Para probar la inclusin opuesta, como la parte derecha de la ecuación es una unión de conjuntos donde todos sus elementos j' satisfacen la conjunción entre $Hamming(w[j'..j'+m-1], v) \leqslant \kappa$ y otro predicado, vemos que todas estas posiciones j' serán reportadas en $approxm_\kappa(w, v)$ directamente de su definición.

□

Esta reducción del problema implicaría en principio realizar solamente $\kappa + 1$ ejecuciones del algoritmo de *locate* sobre una cadena a buscar más pequeña. Si tomamos una secuencia c_i que particione la consulta en partes iguales, salvo cuestiones de redondeo, vemos que el tamaño de la palabra a buscar se reduce a $m/(\kappa+1)$ para *locate*.

Si bien un menor tamaño en la consulta se traduce en general en los algoritmos vistos, en una consulta más rápida, desde el punto de vista teórico del problema *locate*, la cantidad de ocurrencias reportadas es mayor o

igual. En efecto, para un texto de largo n generado de una fuente sin memoria y equiprobable sobre el alfabeto Σ, la cantidad de ocurrencias esperada de una cadena de largo m es $(n-m)/|\Sigma|^m$. Esto significaría que además del costo que representa realizar $\kappa+1$ evaluaciones del problema *locate*, el tiempo que lleva realizar el filtrado de la salida de este problema, necesario para asegurar que $Hamming(w[j-c_i..j-c_i+m-1], v) \leqslant \kappa$ aumenta exponencialmente en κ, en el caso planteado.

Capítulo 5

Experimentación

Para responder el interrogante central planteado por este trabajo, debemos efectivamente ejecutar en la práctica los algoritmos de las estructuras de datos descriptas en el capítulo 3, para los problemas formalizados en el capítulo 4 con las estrategias de reducción allí expresadas y en el contexto de memoria acotada mencionado. No obstante, las diferentes estructuras descriptas tiene lugar a varias decisiones de implementación que pueden afectar el desempeño, aunque más no sea en una constante temporal, que pueden cambiar completamente los resultados en la práctica.

Por estos motivos, para realizar una comparación empírica que tenga validez para el problema *real* del mapeo de reads planteado en el capítulo 2, debemos comparar tanto las implementaciones puras de las estructuras descriptas en los trabajos, como así tambien el software disponible en la comunidad para resolver el problema del mapeo de reads.

En este capítulo definimos el marco de experimentación donde encajan todas las pruebas. En la sección 5.1 comenzaremos con la definición de los elementos involucrados en la comparación, abarcando generalidades como los parámetros que dejaremos fijos (vea sección 5.1.1), los que variaremos (vea sección 5.1.2) y los resultados que mediremos (vea sección 5.1.3).

Respecto del conjunto de pruebas a realizar en concreto, tomamos la postura de variar todos los parámetros seleccionados de manera independiente, por lo que en la sección 5.2 describiremos las pruebas realizadas indicando sólo sus límites independientes, y no la combinación de estos.

En la sección 5.3 listamos las *implementaciones* seleccionadas haciendo referencia a los *algoritmos y estructuras de datos* previamente descriptos en el capítulo 3. Hacemos mención aquí de todos los detalles de *implementación* que impactan sobre los resultados, pero que no son propios del *algoritmo* en sí.

Finalmente, en la sección 5.4 damos una breve explicación del método utilizado para llevar a cabo los experimentos, dejando los detalles del mismo para el apéndice A.

5.1. Definición de la comparación

Comparamos principalmente la eficiencia en cuanto al tiempo de ejecución para el problema *locate* y utilizamos además los tiempos provenientes de las ejecuciones del problema *count* para comprender mejor el origen de las diferencias.

El principal objetivo es determinar bajo qué condiciones es conveniente utilizar una búsqueda basada en arreglos de sufijos en vez de una basada en la versión comprimida o dispersa de estos. Para esto debemos controlar una serie de parámetros que nos permitan establecer las correlaciones entre estos y el tiempo de consulta en cada caso.

5.1.1. Datos

Siguiendo la línea de la motivación definida en el capítulo 2, tomaremos como textos de referencia genomas de mamíferos descargados del Proyecto Ensembl [FAB+12]. En este contexto vamos a asumir que los datos se encuentran descriptos sobre el alfabeto $\Sigma = \{A, C, G, T\}$ que representa los cuatro posibles nucleótidos, aunque en algunos casos los genomas cuentan además con regiones con la letra N. Este símbolo representa el desconocimiento de la base que se encuentra realmente en el genoma, normalmente asociado a regiones difíciles de secuenciar y no se corresponde con un nucleótido en particular.

Para utilizar estos datos reales, de alguna u otra forma, todas las implementaciones retiran los símbolos que no estén en Σ y realizan la estructura sobre un alfabeto de tamaño 4.

Por otro lado, para poder controlar los distintos parámtros de la consulta, los reads a buscar son generados artificialmente a partir de los genomas de referencia, eligiendo así el largo y cantidad de ocurrencias en el genoma.

5.1.2. Parámetros

Los parámetros considerados que variamos en las distintas pruebas y para los cuales analizaremos su impacto son principalmente:

M: cantidad de memoria RAM de la máquina

m: longitud de los reads

n: longitud del genoma de referencia

occ: cantidad de ocurrencias de los reads dentro del genoma de referencia.

Es destacable mencionar además que usaremos varios genomas de referencia para evitar un posible sesgo existente en algún genoma en particular que altere nuestras pruebas, pero consideraremos los resultados independientes del genoma en sí, aunque desde luego, no de su longitud, alfabeto, etc.

Además, todas las implementaciones permiten buscar *varios* reads en la misma ejecución del programa, dado que usualmente se desean alinear todos los reads de un secuenciamiento. Esto permite prorratear el costo de la carga del índice a memoria entre todos los reads de la misma ejecución, haciéndolo despreciable frente al tiempo de ejecución del algoritmo de búsqueda. No obstante esto, mediremos por separado cuando sea posible, el tiempo de carga del índice y el tiempo de búsqueda de los reads. Por otra parte, este tiempo de búsqueda suele ser muy pequeño para poder medirlo con suficiente precisión para la búsqueda de un solo read evitando artefactos indeseables producidos por el sistema. Por estos motivos, decidimos realizar la búsqueda utilizando lotes de reads compuestos de un millón de consultas de las mismas características: largo de los reads y aproximadamente la misma cantidad de ocurrencias dentro del genoma.

Nótese además que no consideramos como parámetro de entrada de nuestras pruebas la cantidad de bloques en los que partiremos el texto de referencia para buscar por separado en cada uno de estos bloques. Esto se debe a que esta cantidad de bloques depende de cada estructura e implementación en particular de acuerdo a la memoria disponible y la memoria que ocupe cada una de estas. Para controlar indirectamente este parámetro permitiendo una comparación justa, tomamos como parámetro de una prueba la cantidad de memoria RAM de la máquina en la que simulamos ejecutar la búsqueda. De esta forma, podemos comparar los resultados de varias implementaciones en el mismo contexto real. Los bloques luego son elegidos de forma de tener aproximadamente el mismo tamaño.

5.1.3. Mediciones

Para cada una de las pruebas definidas en base a los parámetros recién explicados mediremos los resultados de la misma. Entonces, para cada implementación y para cada uno de los problemas discutidos (*count* y *locate*) anotaremos los siguientes valores como resultados de la prueba:

- el tiempo de construcción del índice
- el tiempo de carga del índice desde disco
- el tiempo de búsqueda de los reads
- la cantidad de bloques en la que fue necesario partir el texto

Además de esto, anotaremos también la cantidad de occurrencias encontradas en total, a modo de verificador de la coherencia de los resultados entre todas las implementaciones.

5.2. Variación de los parámetros

Respecto de la longitud del texto de referencia, que llamamos n, variaremos el valor entre 100MB y el tamaño de la región secuenciada del genoma humano (aprox. 2,8GB). Desde luego es imposible conseguir genomas reales de tamaños variados entre estos dos valores, por lo que los construimos artificialmente recortando otros más grandes.

En lo que a la longitud del patrón a buscar respecta, llamada m, el rango sobre el que variamos este parámetro recorre desde subcadenas pequeñas de largo 15, hasta largo 250. Si bien es posible que las máquinas actuales produzcan reads más largos que ese valor, pierde sentido no considerar los errores en un read como ese. Si recordamos la reducción del problema $approx_\kappa$ a *locate* explicado en la sección 4.3, vemos que el tamaño de los reads disminuye como consecuencia de dicha reducción, mientras que por otro lado aumenta el número de consultas.

Finalmente, la cantidad de ocurrencias de los reads no es un parámetro que se pueda elegir directamente sobre los datos, ya que depende tanto de la consulta en cuestión como del genoma dónde se lo busca. Para construir lotes de reads con aproximadamente la misma cantidad de ocurrencias procedimos de la siguiente manera: dado un genoma de referencia y un largo de patrón fijos, seleccionamos al azar suficientes fragmentos del genoma del largo requerido. Para cada uno de ellos *medimos* la cantidad de ocurrencias del mismo y luego los *clasificamos* según este valor en distintos lotes de prueba. Esto lo hicimos de modo de que en cada lote haya exactamente la misma cantidad de reads, un millón, independientemente de la distribución inherente de esta variable al descartar los excedentes. De este modo, lo que elegimos son las distintas *clases*, variando entre una ocurrencia y hasta 100 ocurrencias del patrón.

5.3. Implementaciones seleccionadas

Muchas veces resultados satisfactorios en el terreno teórico en términos de espacio utilizado pueden esconder una constante elevada detrás de una inocente complejidad espacial sublineal haciendo su uso deficiente para valores de n razonables en la práctica, en favor de otras soluciones con complejidades espaciales *peores* en la teoría, pero mejores en la práctica, incluso para valores de n mucho más grandes que los que se utilizan en la actualidad. Es por esto que en la práctica existe una diversidad de opciones para resolver el problema del mapeo de reads.

Listamos a continuación las implementaciones seleccionadas para incluir en esta comparación. Entre las alternativas elegidas, se encuentran tanto versiones implementadas *ex profeso* en el contexto de este trabajo, como versiones implementadas por terceros, destinadas a resolver al menos el problema del mapeo de reads en un contexto real. Debido a que esta es un area de constante innovación, todas las implementaciones seleccionadas cuentan además con una o más publicaciones científicas donde se explica el algoritmo utilizado y sus propiedades, de modo de poder verificar su comportamiento.

Para cada una de estas implementaciones, detallamos brevemente las estrategias y estructuras que utilizan, en base a las descripciones dadas en el capítulo 3, de modo de poder catalogarlas según utilicen un arreglo de sufijos, un arreglo de sufijos comprimido o un arreglo de sufijos disperso.

Como denominador común de todas las implementaciones, podemos mencionar que todas ellas se encuentran programadas en C/C^{++} y sobre índices que soportan texto de longitud hasta 2^{32} símbolos. En algunos casos, como en Vmatch, existen versiones de 64-bits capaces de indexar textos más grandes, pero son pocos. Por tal motivo, salvo que se haga explícita mención, haremos referencia a enteros de 32-bits, es decir, de 4 bytes.

5.3.1. Vmatch – *The large scale sequence analysis software*

El año siguiente a la publicación de los trabajos donde se presentaban los arreglos de sufijos aumentados [AKO02] y el método de búsqueda en estos con complejidad óptima [AOK02], uno de sus autores, Stefan Kurtz, publica en [Kur03] una herramienta llamada Vmatch no casualmente basada en los arreglos de sufijos aumentados.

Esta herramienta se presenta como un analizador de secuencias de símbolos de propósito general. En efecto, el alfabeto sobre el que puede ejecutar no está limitado a ninguno en particular sino que puede especificarse de forma paramétrica. Este software no es de código abierto, aunque puede utilizarse sin fines comerciales bajo licencia gratuita. Esto implica entre otras cosas que no es posible analizar los detalles de la implementación a través del código fuente y que el programa no puede ser modificado para adecuarse a la comparación, aunque eso último no representa un problema importante dada la amplia variedad de opciones del programa.

El programa genera un índice basado en arreglos de sufijos, almacenándolo explícitamente en $4n$ bytes. Además *aumenta* este arreglo con el arreglo LCP_w explicado en la sección 3.2.1, pero almacenándolo en la práctica en poco más de n bytes. Para lograr esto, almacenan LCP_w en un arreglo de enteros de 8-bits colocando en la posición i el valor $LCP_w[i]$ si este es menor que 255, o 255 en otro caso. Para estos casos donde $LCP_w[i] \geqslant 255$, almacenan además una lista de las parejas $(i, LCP_w[i])$ donde esto ocurre, utilizando dos enteros de 4 bytes por cada pareja y ordenándola por el valor de i. Como los valores de LCP_w suelen ser en su mayoría pequeños para muchos casos, esta optimización espera tener que almacenar muy pocas parejas. Si se desea recorrer en orden los valores de LCP_w es posible recorrer el arreglo de enteros de 8-bits en orden, y cada vez que se encuentra un valor 255 consultar la lista ordenada de parejas con el valor real en 4 bytes. Como se están recorriendo los valores en orden, la pareja necesaria se encuentra a continuación de la última pareja consultada, por lo que se puede encontrar en $\mathcal{O}(1)$. Por otro lado, para una consulta aleatoria del valor de $LCP_w[i]$ en caso de ser este mayor o igual a 255 simplemente se realiza una búsqueda binaria sobre esta lista, que se espera sea corta. Finalmente, el programa almacena además el texto original utilzando 1 byte por símbolo, sin realizar ninguna mejora basada en un alfabeto particular debido a que no es un programa específico para ADN, como otros analizados a continuación.

Además, este programa permite utilizar la tabla de prefijos explicada en la sección 3.5, almacenando para cada cadena de largo ℓ sobre el alfabeto dado, el intervalo correspondiente en el arreglo de sufijos con dos enteros de 32-bits. Esto adiciona un espacio de memoria no dependiente del tamaño del texto, siendo en total $8 \cdot 4^\ell$.

En resumen, esta implementación utiliza en total $6n + 8 \cdot 4^\ell$ bytes. Una importante omisión en este análisis es la del arreglo CLD, necesario para realizar la búsqueda en un arreglo de sufijos en $\mathcal{O}(m)$ según el algoritmo descripto por estos mismos autores. La razón de esto, según nos mencionara el autor de este programa en una comunicación privada, es que este algoritmo era más lento en la práctica que realizar una simple búsqueda binaria.

Para los problemas *count* y *locate* se utilizaron diferentes opciones de línea de comando que permiten para el caso de *count* reportar sólo la cantidad total de ocurrencias sin necesidad de recorrerlas, mientras que en el otro caso se reporta solamente la posición de cada una de las ocurrencias encontradas.

5.3.2. Suffix Array – Implementación de KAPOW

Como alternativa a la implementción comercial recién explicada, implementamos un índice basado en el arreglo de sufijos no comprimidos con el algoritmo de búsqueda propuesto por Manber y Myers [MM90, MM93]. Entre estas ideas de búsqueda en el arreglo de sufijos basada en búsqueda binaria, que explicamos a lo largo de la sección 3.2, implementamos la versión que utiliza los arreglos *Llcp* y *Rlcp* que detallamos en especial en la sección 3.2.2, logrando una complejidad en peor caso de $\mathcal{O}(m + log(n))$.

Por otro lado, para almacenar estos dos arreglos adicionales, debido a que cada elemento es el mínimo de un intervalo dado del arreglo LCP_w, aplicamos las mismas ideas que en Vmatch descriptas en la sección anterior por tratarse en la práctica de números pequeños, que nos permiten almacenar estos arreglos en la práctica en poco más de n bytes cada uno.

Siguiendo además las ideas propuestas en el trabajo fundacional de los arreglos de sufijos para reducir el tiempo esperado de búsqueda, implementamos también la tabla de prefijos de largo ℓ que permite acelerar el inicio de la búsqueda. Esto reduce la complejidad para este caso a $\mathcal{O}(m+log_2(n/4^\ell))$ disminuyendo notablemente el impacto del factor n en la complejidad temporal de la búsqueda. Para esta tabla utilizamos un entero de 4 bytes por cada entrada, de modo que el intervalo deseado del arreglo de sufijos correspondiente a un prefijo dado de largo ℓ se encuentra entre dos entradas consecutivas de la tabla.

Finalmente, el texto de referencia es almacenado utilizando 2 bits por símbolo. De este modo, tanto para el problema *count* como para *locate* nuestra implementación utiliza en la práctica $6,25n + 4 \cdot 4^\ell$ bytes para almacenar el índice.

5.3.3. Bowtie – *Ultrafast short-read aligner*

En 2009, Langmead et al. presentaron una *herramienta* de software concreta diseñada para ser *ultra*-rápido en la búsqueda de pequeñas cadenas de ADN dentro de un genoma de referencia de un mamífero [LTPS09]. Este programa fue realizado sobre un índice basado en la BWT, utilizando la estrategia del FM-index descripta en la sección 3.4.1 utilizándolo tanto la búsqueda de una cadena con el *LF-mapping* como para recuperar las ocurrencias en el arreglo de sufijos comprimido. A diferencia de otros programas existentes en ese momento como SOAP [LLKW08] (no confundir con SOAP2) y Maq [LRD08], Bowtie fue innovador al utilizar un índice *full-text* basado en la BWT para alineamiento de ADN.

Gracias a que esta herramienta es de código abierto, la versión utilizada en la comparación fue modificada ligeramente. En primer lugar, bowtie permite realizar alguna variante de la búsqueda aproximada reemplazando el algoritmo de búsqueda exacta en el FM-index por uno muy similar pero que usa además la técnica de *back tracking*, pero, para esto utiliza algunas estructuras adicionales innecesarias para la búsqueda con apareo exacto. Por tal motivo, fueron removidas tanto en el algoritmo de construcción como en la consulta. Además, como se trata de una herramienta destinada a la búsqueda en secuencias de ADN, se duplicaba el trabajo al buscar cada cadena tanto en el texto de entrada como en su reverso complementario, que vendría a ser el texto de la hebra opuesta de la molécula de ADN. Esta funcionalidad también fue deshabilitada, para mantener la comparación justa. Finalmente, para poder medir el tiempo de ejecución para el problema *count*, implementamos una opción que no recorre ni reporta las ocurrencias de los patrones buscados, sino sólo su cantidad.

5.3.4. SOAP2 – *Improved ultrafast tool for short read alignment*

En 2009, los mismos autores de SOAP, publicaron en [LYL+09] una *mejora* de su versión anterior que llamaron SOAP2. En realidad, esta nueva mejora poco tiene que ver con la anterior desde el punto de vista del índice utilizado debajo. En esta ocasión los autores plantean un índice basado en la BWT que mejoró la velocidad y el uso de la memoria respecto de la versión anterior. Este programa está diseñado para resolver el problema del mapeo de reads en cadenas de ADN y es de código abierto, al menos en sus últimas versiones.

En esta implementación, los autores almacenan explícitamente el resultado de la transformación de la BWT utilizando 2 bits por símbolo. Aprovechan además las ventajas de la *lookup table* o tabla de prefijos para acelerar la búsqueda, pero fijan en el código el largo del prefijo en 13, lo cual no puede ser modificado. Para esta tabla utilizan un entero de 4 bytes por entrada, logrando así un tamaño fijo de 256MB para esta tabla.

Además de corregir algunos problemas de implementación para la utilización de archivos de más de 2GB, se modificó el programa para que reporte las ocurrencias sólo en una de las hebras de la cadena de ADN y para que permita reportar sólo la cantidad de ocurrencias totales en vez de reportar cada una de ellas.

5.3.5. Compressed Suffix Array – Implementación de KAPOW

Realizamos una implementación propia de un índice basado en el arreglo de sufijos comprimido. Para ello utilizamos por un lado el algoritmo de construcción lineal del arreglo de sufijos, *no comprimido*, basado en la estrategia de ordenamiento de cadenas por *induced sorting* que mencionamos en la sección 3.1.3. Sin embargo, la implementación en sí de este algorimo fue hecha por Yuta Mori [Mor10], sobre la cual luego nosotros construímos el arreglo de sufijos comprimido.

Para resolver el problema *count* implementamos un índice basado en FM-index, que requiere almacenar el LF-mapping, pero no requiere en ningún momento el valor de una posición arbitraria de R_w. Para este índice, a fin de almacenar la función $occ(c, i)$ descripta en la sección 3.4.1, usamos una estructura de $rank_A(i)$ sobre un conjunto binario, para *cada* símbolo del alfabeto. Es decir, como el alfabeto Σ en cuestión tiene 4 símbolos, tendríamos 4 estructuras de para responder $rank_A(i)$, una para cada símbolo $c \in \Sigma$, sobre un arreglo de n bits A tal que $A[i] = 1$ si y sólo si el símbolo i de la BWT es igual al símbolo c, en cada caso. No obstante, nuestra implementación indexa sobre un alfabeto de 5 símbolos que incluye además la letra N.

Entre las estructuras discutidas en la sección 3.3.1 implementamos la propuesta por Vigna en [Vig08], que ocupa un 25 % más por sobre el arreglo de bits. De este modo, para resolver *count* utilizamos $5 \cdot 1,25 \cdot n$ bits, es decir $6,25n$ bits, aunque podría implementarse utilizando $5n$ bits si se restringe el alfabeto a 4 símbolos. Esta implementación permite buscar una palabra de largo m en $\mathcal{O}(m)$, realizando sólo dos consultas de *rank* en cada iteración.

Por su parte, para el problema *locate*, necesitamos además reportar las posiciones que se encuentran en intervalos dados del arreglo R_w. Para esto implementamos un arreglo de sufijos comprimido según las ideas planteadas por Grossi y Vitter en [GV00] que explicamos en la sección 3.3. Si bien es posible obtener esta información usando la idea del FM-index basado en la BWT, decidimos usar el arreglo de sufijos comprimido directamente por su mejor complejidad. Nuevamente, para este caso usamos la estructura de Vigna para resolver *rank* y *select*.

5.3.6. FM-index – Implementación de KAPOW

El FM-index fue propuesto paralelamente al arreglo de sufijos comprimido de Grossi y Vitter, como un *self-index*, es decir un índice que además contiene el texto implícitamente. Sin embargo, la idea detrás de él, la función LF-mapping es la clave para lograr una búsqueda eficiente en términos de la complejidad temporal y del tiempo de ejecución en la práctica.

Por este motivo, planteamos una implementación que al igual que la recién descripta en la sección anterior, utiliza un FM-index para buscar las cadenas. Sin embargo, para el problema de *locate* utilizamos además el arreglo de sufijos almacenado de manera explícita, al igual que lo hicimos en la implementación descripta en la sección 5.3.2.

Dado que no es necesario almacenar por separado el texto para realizar la búsqueda, nos queda de esta forma un índice que utiliza aproximadamente $4{,}78n$ bytes, que permite una búsqueda en $\mathcal{O}(m)$ y reportar todas las ocurrencias en tiempo lineal esa cantidad, teniendo además un patrón de acceso a memoria eficiente, dado que las ocurrencias a reportar se encuentran consecutivas en la memoria.

5.3.7. Ψ-RA – *a parallel sparse index for genomic read alignment*

La implementación de los autores de Ψ-RA que describimos en la sección 3.6.1 está fuertemente limitada al alfabeto de 4 símbolos $\{A, C, G, T\}$ y a un factor de dispersión d múltiplo de 4. Esta limitación se debe en primer lugar a que el texto es codificado utilizando 2 bits por símbolo, haciendo que en un byte haya 4 símbolos. Luego, al construir el arreglo de sufijos de esta entrada codificada de esta forma pero tomando el alfabeto como si fuera de 256 símbolos, se obtiene un factor de dispersión de 4. Así, eligiendo un subconjunto uniforme de los sufijos de este nuevo arreglo codificado sólo se pueden tomar múltiplos de 4.

Por otro lado, debido al proceso de verificación necesario para cada patrón encontrado en las d búsquedas, no hay mejora alguna al requerir sólo la *cantidad* de ocurrencias del patrón como en el problema *count*. Por este motivo, sólo compararemos los resultados para el problema *locate*.

Como comentario al margen, además de modificar la implementación para omitir el costo de escribir efectivamente los resultados en disco, podemos destacar que se corrigieron algunos errores técnicos de implementación que además fueron reportados al desarrollador.

5.4. Plataforma de comparación

Para realizar exhaustivamente la comparación planteada, construimos una plataforma de trabajo que permite definir independientemente los textos de referencia, las consultas a realizar, las implementaciones y la memoria máxima permitida para la máquina. Con esta plataforma es posible simular entonces la ejecución de una serie de consultas para las distintas implementaciones en una máquina con una determinada memoria acotada. Si bien los detalles técnicos de la misma pueden encontrarse en el apéndice A, podemos mencionar algunas características de diseño a modo de resumen.

Todas las implementaciones son tratadas de la misma manera dado que todas resuelven los mismos problemas *count* y *locate*, no habiendo diferencia entre aquellas que utilicen mucha memoria como las basadas en arreglos de sufijos de las que utilizan menos memoria como las basadas en un arreglo de sufijos comprimido. La diferencia se dará naturalmente con la limitación de la memoria disponible a utilizar por el índice. Esta plataforma de comparación es la que se encarga de calcular el tamaño del índice para cada caso y determinar la cantidad de bloques en los que partirá la entrada para luego realizar la búsqueda en cada uno de estos.

Además de ejecutar los problemas, empleando la partición en bloques de ser necesario, también registra los resultados, junto con el tiempo de ejecución y de construcción de cada índice en una base de datos. Desde esta base de datos es posible extraer los distintos gráficos de comparación, *filtrando* los datos necesarios. Esto es necesario debido a la postura tomada de realizar los experimentos variando los parámetros de forma independiente.

Capítulo 6

Resultados y análisis

Luego de ejecutar las pruebas presentamos en este capítulo los resultados obtenidos junto con un breve análisis de estos. Debido a la política mencionada respecto de realizar pruebas independientes y luego *seleccionar* las pruebas ejectuadas en los distintos lotes para extraer la información, presentaremos antes de cada resultado una breve explicación del experimento mostrado. A continuación de esto damos una lectura de los resultados limitándonos a una explicación somera de los motivos que podrían estar detrás de cada resultado.

Comenzamos entonces el capítulo con las consideraciones generales sobre la presentación de los resultados en la sección 6.1 que aplican a todo el capítulo. A continuación podemos partir el mismo en tres enfoques distintos. En primer lugar, mostramos los factores que dependen sólo del texto de referencia: El tiempo de construcción del índice (sección 6.3) y el espacio utilizado (sección 6.2). Luego, en segundo lugar, mostramos en la sección 6.4 los distintos aspectos que afectan al tiempo de búsqueda *sin* tener en cuenta los límites de memoria. Finalmente, en la sección 6.5 agregamos al análisis el aspecto fundamental de la memoria acotada y damos algunas métricas para medir los resultados en dicho contexto.

6.1. Consideraciones generales

Presentaremos los resultados obtenidos en forma de gráficos y tablas, según sea conveniente. Para el caso de los gráficos, decidimos utilizar para cada magnitud la misma unidad en todos los gráficos. Así es como por ejemplo el *tiempo* es reportado en *segundos* en todos los gráficos, incluso cuando su valor amerite utilizar otra unidad como minuto u hora. Esta conciente decisión facilita la comparación de resultados entre distintos gráficos. Asimismo, en pos de este objetivo, en todos los gráficos utilizamos el mismo tipo de marca para la misma implementación, la cual se puede apreciar en la esquina superior derecha de cada gráfico.

A lo largo de este capítulo, presentaremos los resultados haciendo referencia siempre a la implementación correspondiente. Si bien estas implementaciones ya fueron especificadas desde el punto de vista de sus algoritmos en el capítulo 3 y desde el punto de vista de la implementación en sí en el capítulo 5, en este último dejamos algunos parámetros sin definir. En la Tabla 6.1 se puede apreciar un resumen de las implementaciones ya definidas y con los parámetros restantes fijados, dando lugar a varias versiones de la misma implementación, que podrían tener un comportamiento distinto.

Finalmente, las pruebas fueron realizadas utilizando un solo núcleo de un procesador Intel® Core™ i5 760, de 2.80GHz con una memoria RAM DDR3-1333 de 8GB funcionando en *dual channel*.

6.2. Espacio utilizado

Una de las primeras características a analizar de los resultados es el espacio utilizado por cada índice en función del tamaño del texto de referencia. Si bien esto es algo que se puede calcular exactamente analizando el algoritmo e implementación para algunos casos, en otros casos como el de los arreglos de sufijos comprimidos el tamaño real en la práctica depende de tantos factores que resulta más representativa una medición para el caso analizado.

En el gráfico de la Figura 6.1 vemos la relación entre el tamaño del índice y el tamaño del texto de referencia en el caso del problema *locate*, para distintas subcadenas del genoma humano. Recordemos que si nos restringimos al problema *count* algunas implementaciones requieren menos espacio para albergar el índice. Sin embargo, no reflejamos esto en un gráfico por tratarse sólo de pocos casos.

Impl.	Estructura	Algoritmo de búsqueda	Opciones	Sección
`vmatch`	ESA + *lookup table*	Búsqueda binaria $\mathcal{O}(m \cdot \log(n/4^\ell))$	$\ell = 1$	5.3.1
`vmatch11`	”	”	$\ell = 11$	”
`ksamm0`	SA + Llcp/Rlcp	Búsqueda binaria $\mathcal{O}(m + \log(n))$		5.3.2
`ksamm11`	SA + Llcp/Rlcp + *lookup table*	Búsqueda binaria $\mathcal{O}(m + \log(n/4^\ell))$	$\ell = 11$	5.3.2
`bowtie`	FM-index	*Backward search* $\mathcal{O}(m)$		5.3.3
`soap`	FM-index + *lookup table*	*Backward search* $\mathcal{O}(m)$		5.3.4
`kcsa`	CSA + FM-index	*Backward search* $\mathcal{O}(m)$		5.3.5
`ksabs`	SA + FM-index	*Backward search* $\mathcal{O}(m)$		5.3.6
`psira4`	SSA	Búsqueda binaria $\mathcal{O}(d \cdot m \cdot \log(n/d))$	$d = 4$	5.3.7
`psira8`	”	”	$d = 8$	”
`psira12`	”	”	$d = 12$	”

Tabla 6.1: Resumen de las implementaciones y estructuras subyacentes.

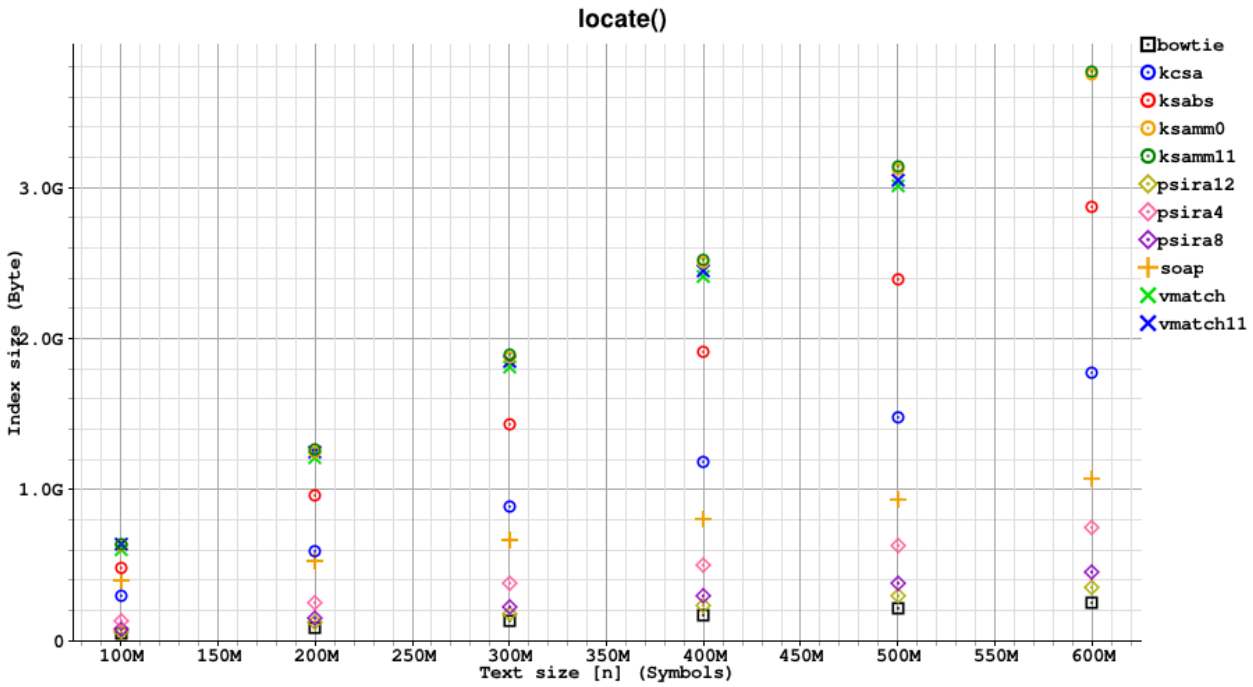


Figura 6.1: Espacio utilizado por cada índice en función del tamaño del texto.

La evidente correspondencia lineal entre las dos variables analizadas en la Figura 6.1 nos invita a formalizar el estudio a través de un ajuste lineal de estos mismos datos por medio del método de cuadrados mínimos. En la Tabla 6.2 puede verse para cada implementación la aproximación lineal del uso de memoria medido para almacenar el índice.

Es destacable que el valor del coeficiente de correlación R^2 es muy cercano a 1 en todos los casos, lo que implica que podemos decir que en la práctica la utilización del espacio *es* lineal respecto del tamaño del texto. Como segunda observación, vemos efectivamente en la columna a la constante asociada a cada implementación en cuanto a la complejidad espacial. Para las implementaciones basadas en el arreglo de sufijos almacenado de manera explícita, más la información adicional para realizar la búsqueda, este coeficiente ronda valores de aproximadamente 6 bytes por símbolo. Mientras tanto, las implementaciones basadas en el arreglo de sufijos comprimido o el FM-index, como `bowtie`, `soap` o `kcsa`, tienes un coeficiente mucho menor, que puede llegar hasta 0,41 como ocurre en el caso de `bowtie`.

Finalmente, de la constante b de la tabla, vemos directamente el impacto de la utilización de una *lookup table*, la cual como dijimos, no depende del tamaño del índice sino del parámetro ℓ.

Implementación	a	b	R^2
`vmatch`	6.03	1.8MB	0.999994868682
`vmatch11`	6.03	35.3MB	0.999994868682
`ksamm0`	6.25	–	1.000000000000
`ksamm11`	6.25	16.8MB	1.000000000000
`bowtie`	0.41	4.2MB	1.000000000000
`soap`	1.35	260.8MB	0.999992030619
`kcsa`	2.96	-0.3MB	0.999999766480
`ksabs`	4.78	–	1.000000000000
`psira4`	1.25	–	1.000000000000
`psira8`	0.75	–	1.000000000000
`psira12`	0.58	–	1.000000000000

Tabla 6.2: Ajuste lineal del tamaño del índice a la función $a \cdot n + b$

6.3. Construcción del índice

El siguiente aspecto general analizado es el tiempo de construcción del índice que nuevamente no depende de la consulta a realizar. Si bien en el contexto de utilización de un índice sobre un texto estático que se construye una vez y se utiliza muchas veces, el tiempo que demora la construcción no es crucial, es al menos importante analizar su factibilidad para no encontrarse con factor que prohíba el uso del índice. En la Figura 6.2 vemos una comparación del tiempo de búsqueda respecto del tamaño del texto de entrada. Además, en la tabla 6.3 vemos nuevamente un ajuste lineal del tiempo de ejecución respecto del tamaño del texto, junto con una reseña de los algoritmos de construcción involucrados.

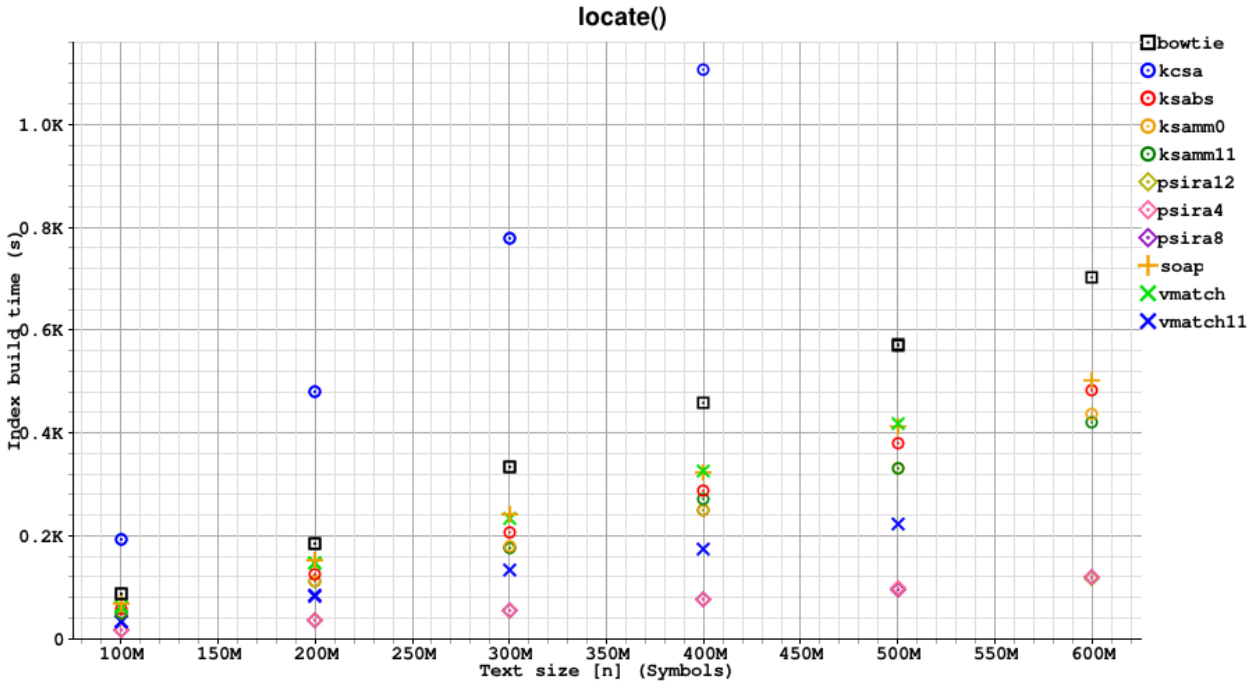


Figura 6.2: Tiempo de construcción en función del tamaño del texto.

Del bajo valor del coeficiente de correlación R^2, el cual en la mayoría de los casos dista de 1 en más de 10^{-2} y del valor negativo del término constante de la función lineal, podemos deducir que el ajuste no es bueno. Si miramos por ejemplo el tiempo para Ψ-RA, debemos recordar que el arreglo de sufijos se construye sobre una entrada cuyo tamaño es 4 veces menor por encontrarse codificada en 2 bits por símbolo. Con esto en cuenta, los tiempos por símbolo son comparables a los de los de más algoritmos.

Por su parte, el algoritmo utilizado en `kcsa` tiene una complejidad temporal $\mathcal{O}(n \log(n))$ frente a los utilizados en `ksabs`, `vmatch` y `psira` que tienen complejidad $\mathcal{O}(n)$ y también realizan operaciones sobre alfabetos

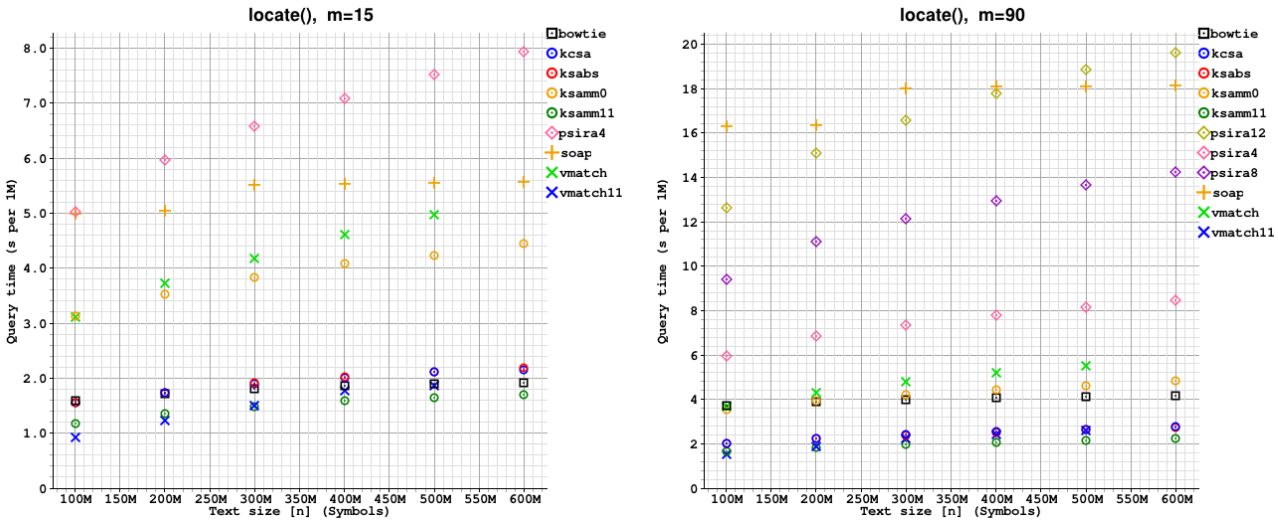
Implementación	a	b	R^2	Algoritmo de construcción
`vmatch`	$0{,}89\mu s/B$	$-30s$	0.9996073	`mkvtree`
`vmatch11`	$0{,}47\mu s/B$	$-12s$	0.9995341	Ídem
`ksamm0`	$0{,}76\mu s/B$	$-41s$	0.9900765	Induced sorting (Yuta Mori)
`ksamm11`	$0{,}74\mu s/B$	$-37s$	0.9933049	Ídem
`bowtie`	$1{,}24\mu s/B$	$-47s$	0.9977779	Kärkkäinen, algoritmo *blockwise*
`soap`	$0{,}86\mu s/B$	$-20s$	0.9997969	Block sorting sobre $n/4$ / BWT
`kcsa`	$3{,}18\mu s/B$	$-149s$	0.9984004	Larsson y Sadakane + derivación del CSA
`ksabs`	$0{,}85\mu s/B$	$-42s$	0.9955020	Induced sorting (Yuta Mori) + FM-index
`psira4`	$0{,}20\mu s/B$	$-5s$	0.9989514	Induced sorting sobre $n/4$
`psira8`	$0{,}20\mu s/B$	$-5s$	0.9988359	Ídem
`psira12`	$0{,}20\mu s/B$	$-5s$	0.9989179	Ídem

Tabla 6.3: Ajuste lineal del tiempo de construcción en función de n bajo la fórmula $a \cdot n + b$

generales.

6.4. Búsqueda de cadenas

Pasando ahora a la medición del tiempo de búsqueda de cadenas, presentamos a continuación los resultados obtenidos. En primer lugar, analizamos la dependencia de este tiempo respecto de n para un caso donde las demás variables no influyan. En la Figura 6.3 vemos una comparación entre el tiempo de ejecución para la búsqueda de un millón de reads de largo 15 (a) y largo 90 (b), en ambos casos correspondientes a patrones que *no* se encuentran en el genoma de referencia. Debido a esto, el tiempo adjudicable a reportar las ocurrencias debería ser nulo.



(a) con $m = 15$, `psira8` y `psira12` no se incluyen

(b) con $m = 90$

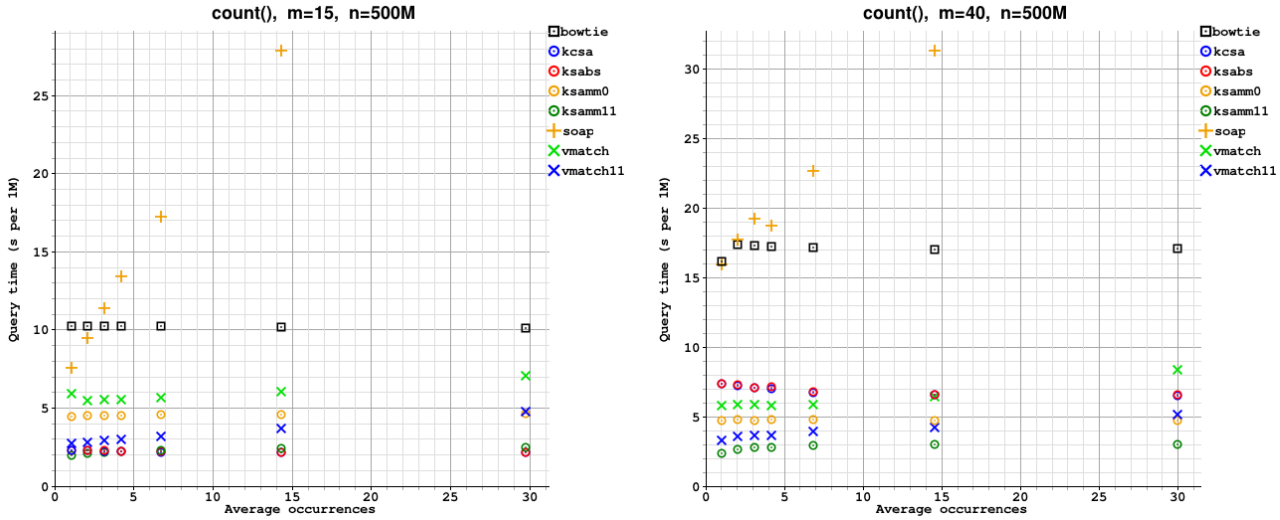
Figura 6.3: Tiempo de búsqueda en función del tamaño del índice, para reads *no presentes*, del mismo largo m

Vemos en estos dos gráficos que para la implementaciones basadas en búsqueda binaria sobre el arreglo de sufijos (`ksamm*`, `vmatch*` y `psira*`) existe una marcada dependencia respecto del tamaño del índice, atribuíble al término $log(n)$ de la complejidad temporal de la búsqueda. Esto es así en todos los casos, dado que aún con cadenas inexistentes, no hay forma de reducir el íntervalo sobre el cual actúa la búsqueda binaria a un ritmo mayor que diviendo por dos en cada caso. Sobre esto, vemos además que la utilización de la tabla de prefijos de largo 11 en `vmatch11` y `ksamm11` reduce considerablemente el tiempo de búsqueda y la dependencia con n al reducir este término $log(n)$, dado que la búsqueda se realiza finalmente sobre un intervalo de tamaño esperado

pequeño.

Por su parte, las implementaciones basadas en un arreglo de sufijos comprimido (`bowtie`, `soap` y `kcsa/kcsabs`) no tienen una dependencia tan marcada respecto del tamaño del índice. Más aún, en el caso de `soap` no parece tener relación alguna.

Pasando ahora a los resultados al variar la cantidad de ocurrencias de los reads buscados, analizamos en primer lugar el problema *count*, en donde en teoría este parámetro no debe afectar significativamente dado que no es necesario *recorrer* todas las ocurrencias encontradas. En la Figura 6.4 vemos dos gráficos del tiempo de búsqueda en función de la cantidad de ocurrencias en promedio de los reads dentro del lote de un millón de reads buscados. Los demás factores como el tamaño del texto de referencia y el largo de los reads se mantuvieron constantes en cada gráfico.



(a) con $m = 15$ (b) con $m = 40$

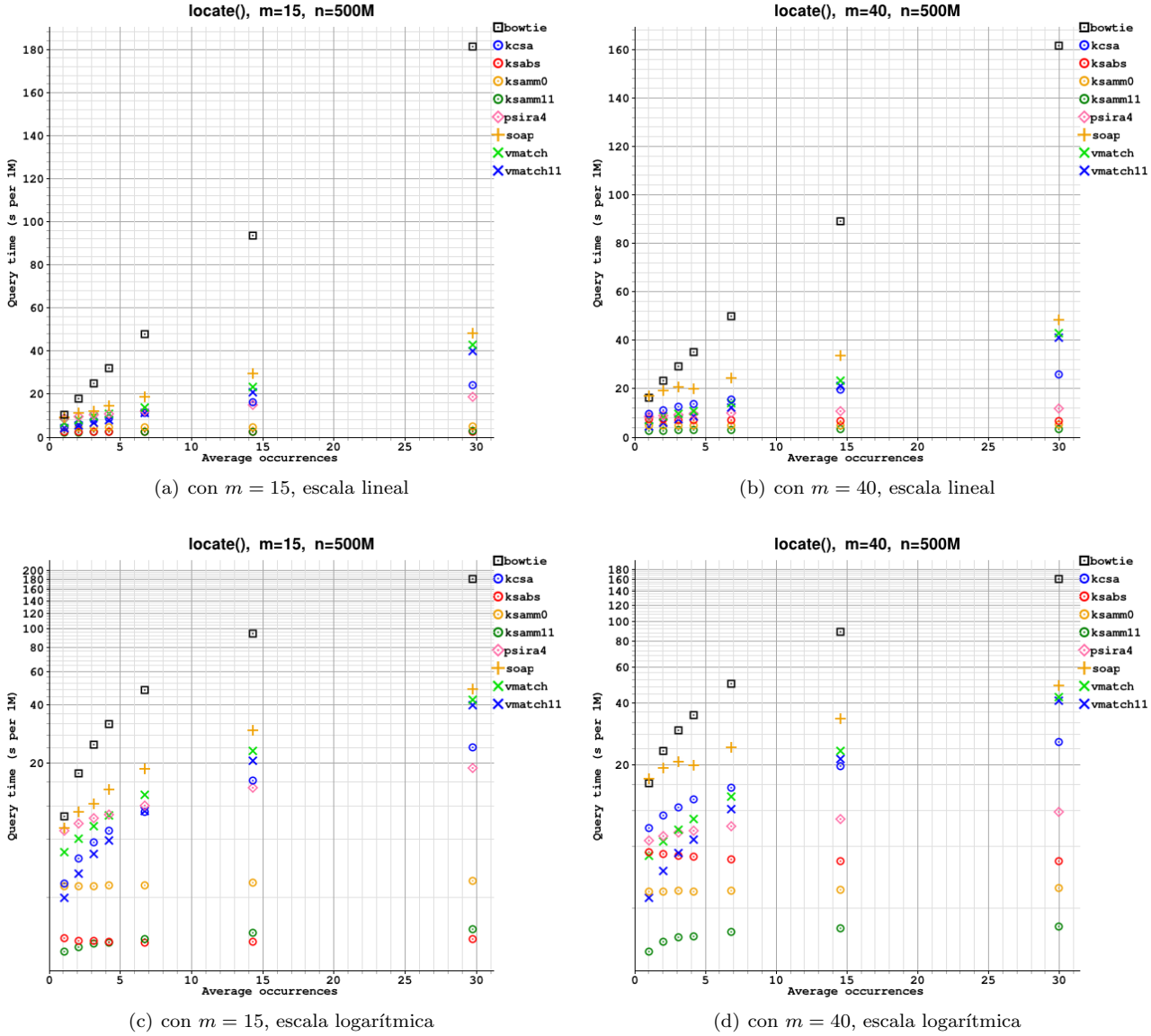
Figura 6.4: Tiempo de búsqueda para *count* en función de la cantidad de ocurrencias promedio del lote de reads del mismo largo m

De los resultados podemos ver que en todas las implementaciones el tiempo de búsqueda para el problema *count* es constante, excepto para `soap`, que tiene una marcada dependencia en este factor. No hay ninguna explicación desde el punto de vista teórico para este hecho más que una implementación descuidada. Para el caso de `vmatch` y `vmatch11`, también observamos una tendencia, aunque más sutil.

Luego de esto, si analizamos los resultados para el problema *locate*, esperaremos tener una diferencia en el tiempo de búsqueda en función de la cantidad de ocurrencias producto de tener que recorrer todas estas ocurrencias. Esta diferencia se debería dar tanto por una variación en la cantidad de ocurrencias como así también en la implementación. En la Figura 6.5 podemos encontrar graficado el tiempo de búsqueda para el problema *locate* en función de la cantidad de ocurrencias, en cuatro versiones. Además de mostrar los resultados para dos valores de m distintos, a fin de independizar los resultados de un contexto particular de m, mostramos también el mismo gráfico en escala logarítmica para apreciar las diferencias de valores en la zona cercana al 0.

Del análisis de esta figura podemos rescatar que varias de las implementaciones basadas en el arreglo de sufijos para reportar las ocurrencias (`ksabs`, `ksamm*` y `psira*`) no se ven prácticamente afectadas por la cantidad de ocurrencias de los patrones buscado, dado que la constante asociada a reportar los números consecutivos del arreglo de sufijos es pequeña. Para el caso particular de `vmatch*`, existe una constante elevada asociada a repotar las ocurrencias que puede deberse a la forma en la que genera la salida este programa, la cual no puede deshabilitarse por línea de comando. En cambio, las implementaciones basadas en un índice comprimido para reportar las ocurrencias (`kcsa`, `soap` y `bowtie`), donde para recuperar la posición de una ocurrencia es necesario realizar una operación compleja, tienen una clara dependencia con la cantidad de ocurrencias.

Volviendo hacia atrás hasta los gráficos de la Figura 6.4, ya se puede apreciar algo que a continuación veremos en más detalle. Si se compara el gráfico de la izquierda (a) con el de la derecha (b) se aprecia que el tiempo, en términos absolutos, para algunas implementaciones cambia bastante. Si bien esto es esperable, porque el parámetro m aparece en todas las expresiones de la complejidad temporal de búsqueda, el orden *relativo* de la



(a) con $m = 15$, escala lineal

(b) con $m = 40$, escala lineal

(c) con $m = 15$, escala logarítmica

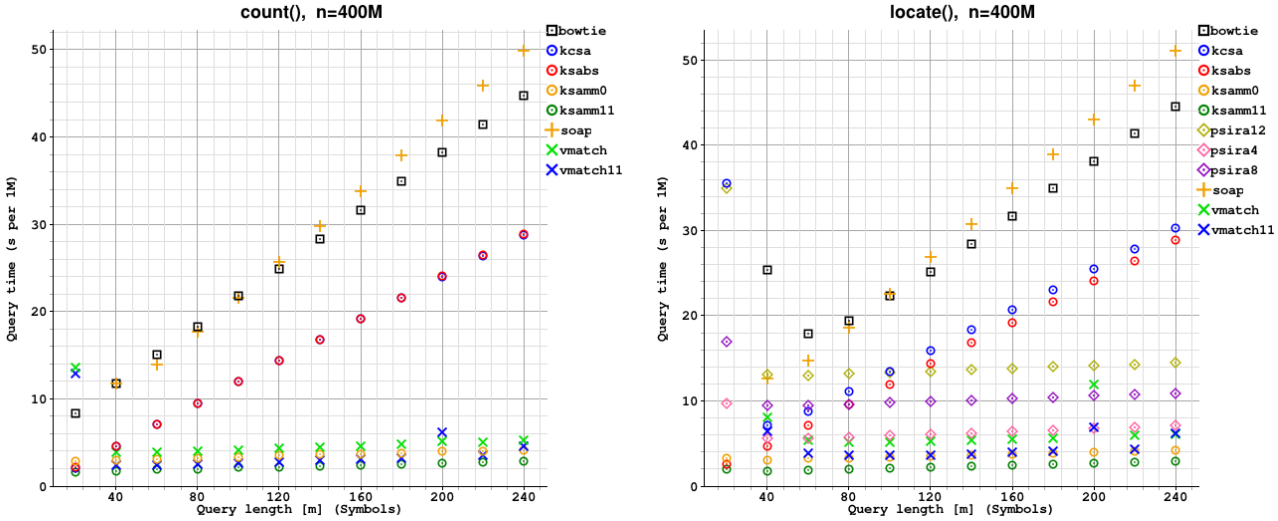
(d) con $m = 40$, escala logarítmica

Figura 6.5: Tiempo de búsqueda para *locate* en función de la cantidad de ocurrencias promedio del lote de reads del mismo largo m

curva casi constante (en la mayoría de los casos) de cada implementación en uno y otro gráfico es distinto. Esto quiere decir que el impacto del valor m no es el mismo en cada implementación, incluso para el contexto del problema *count* donde no es necesario reportar las ocurrencias. Habiendo analizado ya la dependencia respecto de n y de la cantidad de ocurrencias, damos paso al análisis respecto de m.

La Figura 6.6 muestra el tiempo de búsqueda en función de la longitud del read para sendos problemas. En este caso, en vez de tomar una cantidad de ocurrencias fija, mostramos los resultados de tomar una muestra de reads de longitud m fija de posiciones aleatorias dentro del texto de referencia. Esto sería más parecido a la situación real en la que los reads son tomados al azar de una molécula de ADN para luego ser buscados en un genoma de referencia. Esto también implica que cada read efectivamente ocurre al menos una vez dentro del texto, pero que la probabilidad de ocurrir más veces decrece en la práctica de manera exponencial por lo argumentado en la sección 3.5. En la práctica se traduce en que a partir de un cierto valor de m, la cantidad de ocurrencias en promedio de los reads del lote es aproximadamente 1.

Del gráfico mencionado vemos una similitud entre las respectivas curvas mostradas para el problema *count* y *locate* de la misma implementación, a excepción de los valores más chicos de m, donde el reporte de las potencialmente numerosas occurencias influye en el tiempo total de búsqueda. En ambos gráficos se aprecia



(a) Problema *count*

(b) Problema *locate*

Figura 6.6: Tiempo de búsqueda en función de la longitud del read, tomando una muestra aleatoria de subcadenas del genoma de referencia.

además la clara separación entre las implementaciones basadas en arreglos de sufijos y las basadas en alguna versión comprimida de estos.

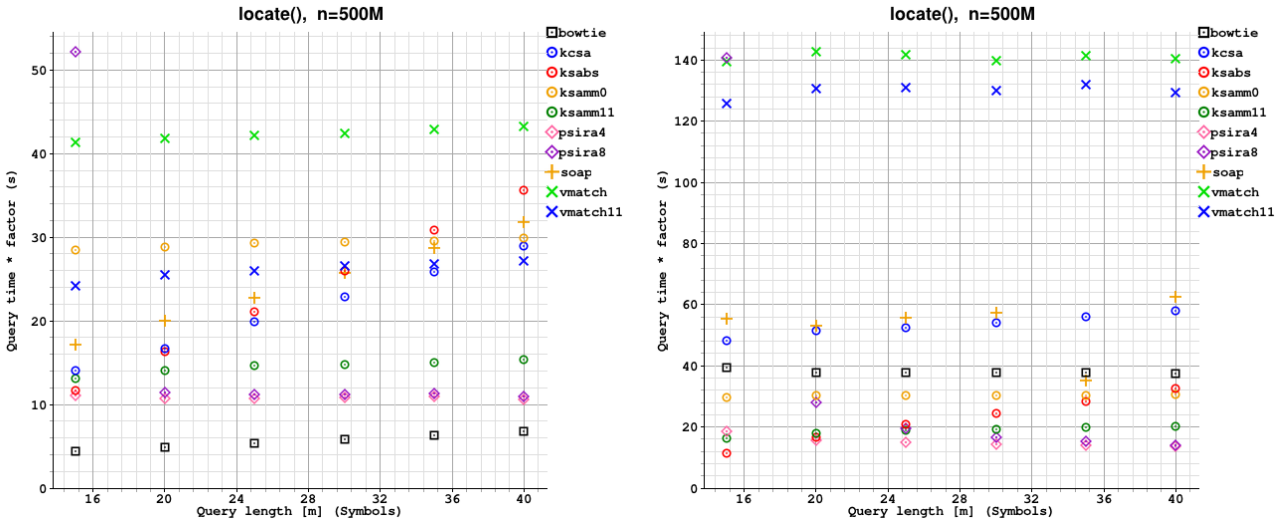
Si bien en la teoría los algoritmos de búsqueda vistos tienen un término al menos lineal en m en su complejidad temporal, vemos que en la práctica la constante asociada a ese término es importante, separando claramente unos de otros. Como lectura secundaria de estos gráficos, en un contexto de reads tomados con una distribución que asumimos es similar a la que nos encontraríamos en una muestra real, vemos que en ambos extremos del rango de m tomado ocurre lo mismo: para el problema *locate* los algoritmos de búsqueda basados en arreglos de sufijos comprimidos son más lentos para un valor de m alto debido a la constante asociada a este término, pero también son más lentos para un valor de m bajo debido a que la cantidad de ocurrencias aumenta.

6.5. Búsqueda de cadenas con memoria acotada

En los gráficos mostrados hasta aquí, en todo momento estuvimos trabajando con índices que entraban completamente en memoria principal. De aquí en más mostraremos los resultados teniendo en cuenta el tamaño del índice y la memoria principal.

Analizaremos primero los resultados del tiempo de búsqueda a través de una métrica distinta. Anteriormente mostramos directamente el tiempo de búsqueda en segundos por millón de consultas. Teniendo en cuenta los resultados de la Figura 6.3 donde se ve que existe una dependencia entre el tiempo de búsqueda y el tamaño del texto, más marcada en los arreglos de sufijos, la misma es *leve* dado que se debe a un término logarítmico en n. En cualquiera de los casos, la relación es al menos constante o mayor. Por otro lado, si por cuestiones de memoria acotada debemos partir el texto de entrada en p partes, el tamaño del índice disminuirá linealmente como vimos en la Tabla 6.2. Uniendo estas dos cuestiones, vamos a mostrar a continuación los gráficos en términos del tiempo de ejecución necesario para la búsqueda multiplicado por un factor de corrección que es el cociente entre el tamaño del índice y el tamaño del texto. De este modo, si tenemos dos implementaciones distintas que requieren el mismo tiempo para buscar una cadena para la misma situación, pero una de ellas utiliza la mitad del espacio, querríamos ver esto reflejado en un gráfico. Al multiplicar el valor por el factor propuesto, la implementación que utiliza la mitad del espacio se verá en esta métrica que utiliza *la mitad* de "tiempo" que la otra. Esta métrica tiene sentido en un contexto de memoria acotada porque si la implementación que utiliza más memoria no cabe en la memoria disponible pero la otra sí, entonces para poder utilizar una de las implementaciones deberemos dividir el texto en 2 partes de aproximadamente el mismo tamaño donde cada una de estas partes requerirá un tiempo *similiar* al que demandaría la búsqueda sobre el texto completo. Aquí es donde estamos asumiendo que el tiempo de búsqueda en un texto de tamaño n/p, al menos para valores

de p chicos, es similar al tiempo de la misma búsqueda para un texto de tamaño n. Desde luego que en la práctica esto es mejor que similar, dado que las ocurrencias a reportar son menos en un texto de tamaño n/p, lo cual reduce el tiempo para el caso del problema *locate*. Para resumir, esta métrica es un poco menos ventajosa para las estructuras basadas en arreglos de sufijos, que son las que ocupan más.



(a) Cantidad de ocurrencias promedio = 1.5

(b) Cantidad de ocurrencias promedio = 14.5

Figura 6.7: Tiempo de búsqueda *corregido* en función de la longitud del read.

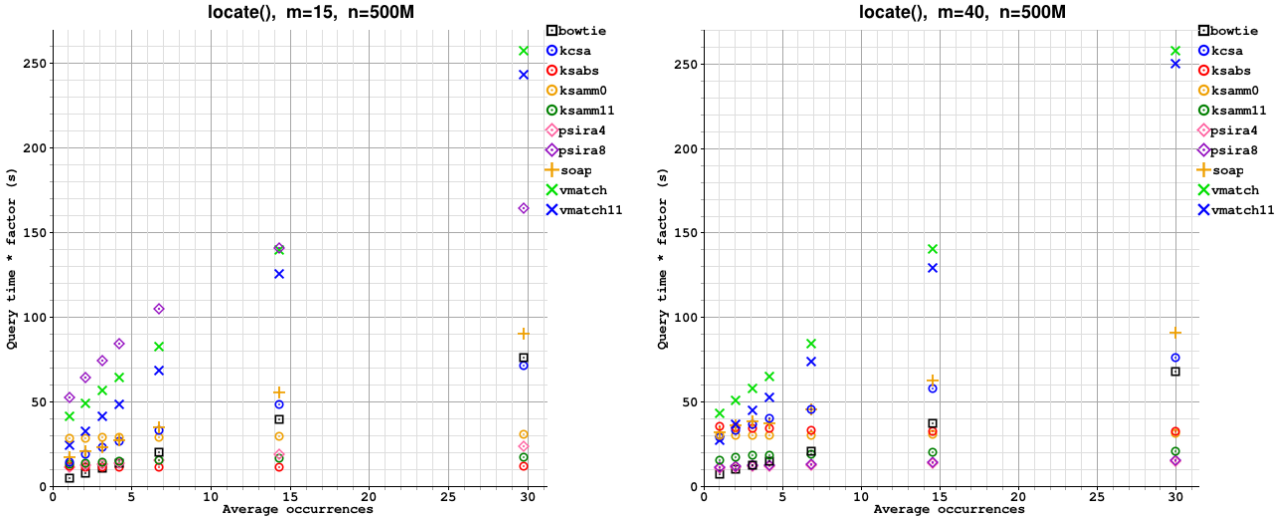
Pasemos entonces a ver los resultados bajo esta métrica. En la Figura 6.7 se observan dos gráficos del tiempo de búsqueda *corregido* multiplicándolo por el tamaño del índice sobre el tamaño del texto indexado. Vemos cómo bajo esta métrica en el gráfico (a), donde la cantidad de ocurrencias promedio es baja (aproximadamente 1.5), `bowtie` tiene un claro desempeño más eficiente que las demás implementaciones gracias a su extremadamente reducido espacio requerido para almacenar el índice. Sin embargo, si miramos el gráfico (b), donde la cantidad de ocurrencias promedio es mayor, la situación se invierte. Es destacable mencionar que las implementaciones basadas en el arreglo de sufijos como `ksamm*` y `psira*` tienen un comportamiento estable entre estos dos gráficos frente a la fuerte dependencia de `bowtie` y `kcsa`.

En efecto, si tomamos un tamaño de read fijo y variamos nuevamente la cantidad de ocurrencias, vemos en la Figura 6.8 que a partir de entre 3 y 4 ocurrencias por patrón, incluso bajo esta métrica que desfavorece a las implementaciones basadas en búsqueda binaria a raíz de lo discutido al comienzo de esta sección, es conveniente utilizar una estructura basada en arreglos de sufijos y búsqueda binaria como lo es `psira*` o `ksamm11`.

Este mismo análisis de las figuras 6.7 y 6.8 lo podemos hacer sobre el problem *count*, considerando el espacio necesario para responder sólo esta consulta, que en muchos casos es el mismo, aunque para el `kcsa` implementado sólo es necesario el FM-index. En efecto, este análisis lo hicimos, y lo presentamos resumido en el gráfico (a) de la Figura 6.9. Por su parte, en el gráfico (b) presentamos el mismo resumen, pero para el problema *locate*.

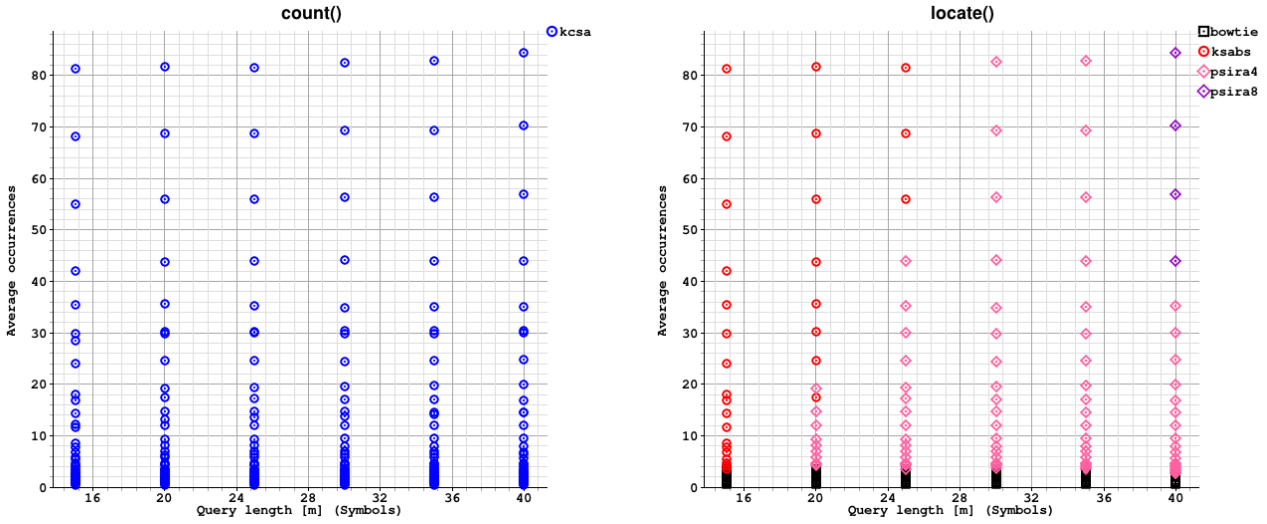
Este resumen que vemos en la figura Figura 6.9 consiste en lo siguiente. Tomamos tamaños de entrada diversos, tamaños de reads también diversos y seleccionamos distintos lotes de prueba con distinta cantidad de ocurrencias por read en cada uno de ellos. Para cada una de estas pruebas, realizamos la búsqueda con cada una de las implementaciones y nos quedamos con la que *mejor* tiempo tenga bajo la métrica de tiempo corregido con la que venimos trabajando.

Los resultados obtenidos para el caso del problema *count* son contundentes. En todos los casos analizados, el gráfico muestra que teniendo en cuenta el factor de uso de memoria respecto del tamaño del texto, la implementación `kcsa` es la más rápida si sólo se necesita resolver el problema *count*. Por su parte, para el problema *locate*, los resultados se encuentran divididos. Si la cantidad de ocurrencias es poca, de hasta 3 o 4 ocurrencias por patrón, un índice comprimido como `bowtie` hace mejor el trabajo. Sin embargo, cuando la cantidad de ocurrencias sube por encima de este umbral, la cuestión se divide entre dos estructuras que utilizan el arreglo de sufijos de manera explícita para reportar las ocurrencias, aunque utilizan un método completamente distinto para la búsqueda. Para el caso de `ksabs`, la búsqueda se realiza a través del FM-index,



(a) $m = 15$ (b) $m = 40$

Figura 6.8: Tiempo de búsqueda *corregido* en función de la cantidad de ocurrencias.

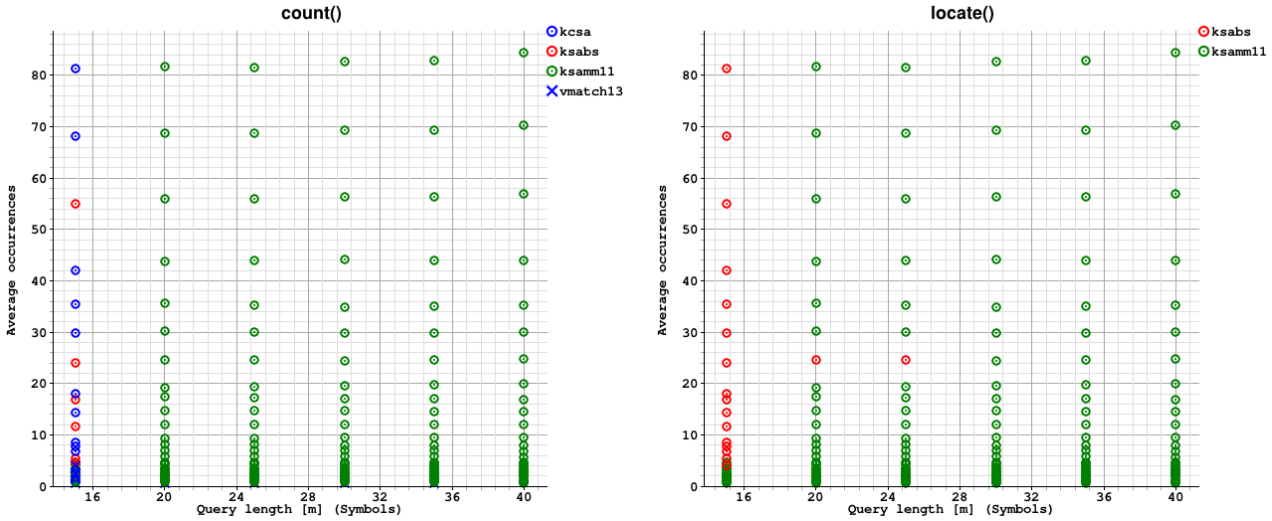


(a) Problema *count* (b) Problema *locate*

Figura 6.9: Para cada longitud de patrón y cantidad de ocurrencias, marcamos en el gráfico la implementación que menor tiempo requiera para cada caso, bajo la métrica de tiempo *corregido*.

el cual es muy sensible a m, mientras que las ocurrencias se reportan directamente desde un arreglo de sufijos.

Finalmente, en la Figura 6.10 podemos apreciar el mismo análisis resumido pero bajo la métrica de tiempo real ignorando el factor de corrección por uso de la memoria que empleamos al comienzo de este capítulo.



(a) Problema *count*

(b) Problema *locate*

Figura 6.10: Para cada longitud de patrón y cantidad de ocurrencias, marcamos en el gráfico la implementación que menor tiempo requiera para cada caso.

Capítulo 7

Conclusiones

Luego de analizar los resultados de las pruebas realizadas en el capítulo anterior, podemos deducir algunas conclusiones que enumeramos aquí. Debemos recordar que estas conclusiones son válidas al menos en el marco de trabajo utilizado aquí: el problema del mapeo de reads a un genoma, con un alfabeto de 4 símbolos.

En primer lugar, podemos decir que las premisas en relación a las diferencias entre los arreglos de sufijos comprimidos y no comprimidos enunciadas al comienzo de este trabajo como un resumen de los argumentos en los trabajos citados, son ciertas. De la Figura 6.1 podemos ver que efectivamente los arreglos de sufijos comprimidos son estructuras que requieren mucho menos espacio de almacenamiento y aún mantienen un tiempo de consulta *razonable*. Por otro lado, los gráficos presentados en la sección 6.4 muestran que, salvo para reads muy pequeños que ocurren muy pocas veces, incluso la búsqueda binaria sobre arreglos de sufijos, con la correspondiente tabla de prefijos para acelerar la búsqueda, es el método más rápido. En este sentido podemos decir que los arreglos de sufijos no comprimidos son más rápidos manteniendo un uso de memoria *razonable*, frente a alternativas anteriores como el árbol de sufijos.

Gracias a los argumentos esgrimidos al comienzo de la sección 6.5 que utiliza el método que propusimos en la sección 4.2 para partir el problema realizando un compromiso entre memoria disponible y velocidad de búsqueda, nos encontramos ante la posibilidad de comparar bajo una misma métrica dos grupos de implementaciones de características distintas: unas utilizan menos memoria pero buscan más lento mientras que otras buscan más rápido, pero utilizan más memoria.

Bajo esta métrica, aplicable al contexto de memoria acotada, los resultados son contundentes. Si sólo se desea contar las ocurrencias de un patrón en un texto, pero no reportar las mismas, lo mejor es utilizar un índice comprimido, dado que la memoria necesaria para indexar un genoma con un índice basado en la BWT como el FM-index es extremadamente pequeña. Por otro lado, si además se desea reportar las ocurrencias, a partir de aproximadamente 4 ocurrencias por patrón conviene utilizar una estructura basada en arreglos de sufijos no comprimidos, dado que el costo de obtener las posiciones de cada una de las ocurrencias comienza a dominar el tiempo total de la búsqueda en las estructuras de índices comprimidas. Entre estas estructuras, el tradicional arreglo de sufijos con la búsqueda binaria propuesta por Manber y Myers mostró ser la implementación más estable ante todas las situaciones. Si tomamos la implementación del arreglo de sufijos disperso de Ψ-RA, tenemos buenos resultados para tamaños de reads medianos y grandes, pero para tamaños pequeños más cercanos al coeficiente de dispersión el tiempo se dispara. Por otro lado, un índice basado en la BWT y el concepto del FM-index para la búsqueda, aún con implementaciones modernas y eficientes de *rrequirank*, y con un arreglo de sufijos debajo para reportar las ocurrencias eficientemente como la implementación de `ksabs`, el tiempo de búsqueda crece considerablemte con el tamaño de los reads.

En resumen, de los resultados de nuestro trabajo, se puede concluir que:

- Si se dispone de memoria suficiente, siempre es preferible utilizar un arreglo de sufijos junto con una búsqueda binaria y una tabla de prefijos de tamaño adecuado.
- Si no se dispone de memoria suficiente, entonces la estrategia depende del tipo de búsquedas que se desee realizar. Si normalmente se realizan búsquedas con pocos resultados (hasta 4 ocurrencias), o bien, sólo es necesario reportar pocos resultados aunque haya muchos más, entonces lo mejor es utilizar un índice comprimido, basado en la BWT y la búsqueda en un FM-index.
- Si no se dispone de memoria suficiente para un arreglo de sufijos del texto completo y se desea recorrer todas las ocurrencias de un patrón que pueden ser potencialmente muchas, entonces lo mejor es utilizar un índice basado nuevamente en arreglos de sufijos almacenado explícitamente para reportar las ocurrencias

eficientemente, pudiendo realizar la búsqueda por diversos métodos. Para realizar esto, es necesario partir el texto de entrada en bloques más pequeños para que el índice de cada bloque entre en memoria.

Mathieu Raffinot al mismo tiempo que planteó el problema de la comparación había conjeturado que existía esta relación entre los parámetros de cantidad de memoria y cantidad de ocurrencias. Nuestro estudio, precisa además los parámetros para minimizar el tiempo dada la situación de memoria acotada.

Notemos además que una de las aplicaciones directas de la tercer situación es la búsqueda inexacta discutida en la sección 4.3 donde es necesario recorrer *todas* las ocurrencias de patrones más pequeños, evaluando una función de distancia en cada paso.

Finalmente, en relación a nuestra propuesta de realizar un compromiso entre la memoria utilizada y el tiempo consumido para resolver el problema partiendo el texto en bloques; podemos ver la propuesta del arreglo de sufijos disperso como una alternativa equivalente. Esto es debido a que el arreglo de sufijos disperso maneja un parámetro d que permite reducir el tamaño de la estructura en un factor de $1/d$, aumentando el tiempo de búsqueda en un factor d. Sin embargo, nuestra propuesta mostró tener una eficiencia similar en el caso general y además ser más estable y eficiente para reads de poca longitud.

Capítulo 8

Trabajo futuro

Finalmente, en este capítulo mencionamos los temas intimamente relacionados con el presente trabajo, que por algunas diferencias propias de cada caso implicaban realizar un análisis completamente distinto al planteado. Por este las incluímos en esta sección, a modo de propuesta de trabajo para el futuro.

8.1. Apareo inexacto

Uno de los notables puntos que dejamos de lado en este trabajo es el análisis de los algoritmos en el contexto del apareo inexacto, es decir, donde se acepta alguna pequeña diferencia entre la palabra que se está buscando y la subcadena del texto donde esta ocurre, pero de manera inexacta.

Esta simple diferencia acarrea dos problemas ya desde su definición. Para decir que una palabra v aparece de *manera inexacta* en la posición i del texto w debemos definir al menos dos cosas:

- La métrica de distancia utilzada para determinar qué tan parecidas son las cadenas v y una subcadena de w que comienza en i.
- Un umbral en esta medida de distancia.

Para la primer cuestión, existen varias métricas clásicas como la distancia de Hamming o la distancia de edición que permite además inserciones y borrados. Además de estas, en el contexto del alineamiento de secuencias de ADN existen otras métricas que ponderan distintos los errores dependiendo de la posición dentro de la palabra v, de modo de reflejar fielmente la real fuente de errores de secuenciamento. A modo de ejemplo, en las máquinas de secuenciamiento de la tecnología Roche/454, debido a su método de secuenciamiento las secuencias repetitivas de un mismo nucleótido son más propensas a tener errores de inserción o borrado de ese mismo nucleótido dentro de la repetición. Por otra parte, las máquinas de secuenciamiento de nueva tecnología tienen como denominador común que la calidad de las cadenas leídas es mayor al principio de la cadena leída. Esto hace que los errores hacia el final de la secuencia leída sean más propensos que al principio.

Respecto de la segunda cuestión, el umbral en la medida de distancia, adems de depender del largo de la cadena, es un parámetro que debe poder ajustarse para manipular la cantidad de resultados obtenidos.

Finalmente, por sobre estas diferencias ya desde la definición del problema de apareo inexacto, existen diferencias en la implementación de estos algoritmos, que en casos como el de Bowtie [LTPS09] relajan la definición estricta del problema en favor del tiempo de ejecución.

Todas estas diferencias dificultaban una comparación *justa* de los algoritmos en el contexto planteado utilizando implementaciones reales. No obstante, un estudio con un enfoque distinto sin estar atado a la aplicación inmediata de los resultados, utilizando implementaciones nuevas, bajo la misma definición del problema, daría la posibilidad de realizar esta misma comparación en el contexto del apareo inexacto.

8.2. Cómputo en la GPU

La misma comparación que realizamos en este trabajo se puede aplicar también a la búsqueda de cadenas utilizando la unidad de procesamiento gráfico (GPU). Muy recientemente comenzaron a realizarse implementaciones de estos algoritmos de indexación que aprovechan el paralelismo de la GPU, entre los que podeos citar a SOAP3 [LWW+12] y BarraCUDA [KLL+12] basados en un FM-index, y a MUMmerGPU 2.0 [TS09] basado en el árbol de sufijos.

No obstante, debido al modelo de programción paralela, diferencias entre el patrón de uso de la memoria, la cantidad de memoria usada y los saltos condicionales pueden ser mucho más importantes en este contexto que las complejidades involucradas. A modo de ejemplo, la implementación de *rank* propuesta por Vigna [Vig08] utiliza más operaciones en paralelo sobre los bits de un registro en vez de llamadas recursivas como lo haría la implemetación propuesta por Grossi y Vitter [GV05].

Apéndice A

Detalles del Software utilizado

Cómo explicamos a lo largo del capítulo 5, el *software* es en realidad un conjunto de diversos programas, algunos diseñados exclusivamente para este trabajo y otros de terceros.

Para utilizar toda esta variedad de software sobre una definición común y así comparar los resultados es necesario algo que los aglutine. Para ello implementamos una plataforma de pruebas que unifica la interfaz y la definición de las pruebas. A continuación encontrará una explicación de la estructura de la misma así como también información esencial para su uso.

A.1. Descripción general

La plataforma de pruebas está escrita en el lenguaje de programación Python, por su versatilidad y facilidad de uso. El punto de entrada de esta es el programa `runtest.py` que toma como único argumento un archivo de configuración.

En este archivo de configuración se pueden definir todas las pruebas a realizar sobre las distintas entradas. La sintaxis de este archivo permite escribir de forma reducida y simple varias pruebas. Para una descripción y ejemplos de uso, vea el archivo `tests.conf` en el directorio raíz.

Una vez cargados todos los casos de prueba y los archivos correspondientes con los genomas de referencia y reads a buscar, se pasa a ejecutar cada caso de prueba. En la ejecución de un caso de prueba, para una implementación dada, con un límite de memoria específico, se calcula la cantidad de bloques en los que es necesario partir el texto de entrada. De este modo, el texto de entrada se parte en uno o más y se llama a las funciones de construcción y búsqueda de la implementación correspondiente. Las implementaciones disponibles no pueden ser llamadas directamente dado que cada una tiene su propia interfaz. Para subsanar esto, definimos una interfaz común en términos de una clase Python que encapsula todas las cuestiones particulares de la implementación que se quiera usar. Para una descripción detallada de esto vea la sección A.4.

Los resultados de cada ejecución, tiempos de ejecución, tamaño del índice, bloques en los que fue partida la entrada, etc, junto con la información relativa a los datos de entrada que fueron utilizados en la prueba son almacenados en una base de datos SQLite para su posterior examinación.

Sobre esta base de datos corre una aplicación web hecha también en Python que permite visualizar los datos a través de gráficos y tablas. Estos gráficos se generan a partir de consultas sobre la base de datos, donde se puede elegir fácilmente qué variables graficar en función de qué otras variables, pudiendo además definirse un gráfico directamente en lenguaje SQL. Finalmente, para los gráficos seleccionados, se confeccionan automáticamente todos las imágenes que se incluyeron en este informe. De esta forma, al agregar nuevas ejecuciones para completar *huecos* en los gráficos, estos son actualizados automáticamente.

A.2. Compilación e instalación

El programa que realiza las pruebas en sí no se compila dado que está escrito en un lenguaje interpretado. No obstante, varios módulos fueron desarrollados en C para ser utilizados desde este programa en Python. Estos módulos requieren claramente ser compilados, para lo cual se provee un archivo `Makefile`.

Es necesario por tanto, contar con los encabezados de compilación de Python, además del programa intérprete en sí y el compilador de C/C++.

Respecto de la implementación Vmatch [Kur03], la misma no se incluye junto con el software por cuestiones de licencia. Para obtener el programa y una licencia de uso no comercial puede contactar al desarrollador del mismo.

A.3. Estructura de directorios

- `compare/`: Contiene el código Python de toda la plataforma de comparación, a excepción del programa `runtests.py` que se encuentra en la raíz. Dentro de este directorio se encuentran todos los módulos que componen las interfaces con las distintas implementaciones, según explicaremos en la sección A.4.
- `bowtie/`: Contiene la implementación de Bowtie con las opciones necesarias para la medición agregadas al código fuente.
- `kapow-pyapi/`: La implementación de los índices de KAPOW. En este directorio se encuentra la implementación en C de la interfaz Python a todas las estructuras de este paquete. Dentro de él encontrará además un directorio `ksrc` con las implementaciones en C de los algoritmos propiamente dichos junto con algunas herramientas en Python como `scatter.py` que permite generar archivos de prueba con las características necesarias.
- `psira/`: Contiene el código de Ψ-RA, tanto del programa para construir el índice (subdirectorio `PSI-RA`) como del programa para realizar las búsquedas (subdirectorio `PSIRA-index`). Junto a esto encontrará además un programa para convertir la entrada FASTA al formato requerido por Ψ-RA.
- `soap/`: En este directorio se encuentra la implementación en C de la interfaz Python que realizamos del índice de SOAP2. Dentro de él se encuentra además el código fuente de la versión 2.20 del alineador SOAP2 (subdirectorio `soap2.20` Contiene la implementación
- `vmatch/`: Este directorio se encuentra disponible para albergar los archivos binarios de Vmatch según el `README` que puede leerse allí dentro. No obstante, estos archivos no se incluyen por cuestiones de licencia.
- `web/`: Aquí se encuentran las herramientas de análisis de los resultados. Principalmente, se cuenta con un servidor web basado en Python-bottle que permite seleccionar de varias maneras configurables los gráficos a generar.

A.4. Módulos de implementación

Para agregar soporte para las distintas implementaciones utilizadas dentro de esta plataforma es necesario definir un módulo de Python con una clase para cada implementación. Posteriormente esta clase podrá ser cargada desde el archivo de configuración a ejecutar.

Esta clase definida debe tener para cada problema "*prob*" que implemente (`count` y `locate`) las siguientes tres funciones:

- *prob*`_build(src)` que dado el texto a indexar construye un índice y lo almacena en uno o más archivos cuyo prefijo comienza con un nombre dado. De esta forma, esta función resuelve todas las conversiones necesarias y construye efectivamente el índice. No está permitido almacenar ninguna información fuera de los archivos mencionados. Como resultado, esta función devuelve el tiempo requerido para construir el índice
- *prob*`_mem(n)` Esta función devuelve una estimación del tamaño que requerirá un índice sobre un texto de largo n. Esta estimación es utilizada por la plataforma de pruebas para determinar en cuántos bloques requerirá ser partido un texto de referencia.
- *prob*`(patt)` Dada una lista de patrones en formato FASTA realiza la búsqueda de todos ellos sobre el índice almacenado en disco bajo el mismo nombre utilizado en la función *prob*`_build(src)`. Esta función debe devolver tres valores: la cantidad de ocurrencias encontradas, el tiempo requerido para cargar el índice a memoria y el tiempo requerido para buscar todas las ocurrencias. Independientemente del problema, el primer valor que devuelve esta función es la cantidad de ocurrencias, la cuál será utilizada posteriormente como un verificador más de que las ejecuciones fueron correctas y no hay ningún error con el índice en cuestión.

Bibliografía

[ABM08] Donald Adjeroh, Tim Bell, and Amar Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching.* Springer, 2008.

[AC75] A. V. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[AKO02] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proceedings of the Second International Workshop on Algorithms in Bioinformatics*, WABI '02, pages 449–463, London, UK, UK, 2002. Springer-Verlag.

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[AOK02] Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz. Optimal exact strring matching based on suffix arrays. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, SPIRE 2002, pages 31–43, London, UK, 2002. Springer-Verlag.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of ACM*, 20:762–772, 1977.

[BMSdlV09] Michael Brudno, Paul Medvedev, Jens Stoye, and Francisco M. de la Vega. A report on the 2009 sig on short read sequencing and algorithms (short-sig). *Bioinformatics*, 25(21):2863–2864, 2009.

[BS97] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[BW94] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center, Palo Alto California, May 1994.

[CAB+09] Davide Campagna, Alessandro Albiero, Alessandra Bilardi, Elisa Caniato, Claudio Forcato, Svetlin Manavski, Nicola Vitulo, and Giorgio Valle. PASS: a program to align short sequences. *Bioinformatics*, 25(7):967–968, 2009.

[CHSV08] Yu-Feng Chien, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Geometric burrows-wheeler transform: Linking range searching and text indexing. In *Proceedings of the Data Compression Conference*, pages 252–261, Washington, DC, USA, 2008. IEEE Computer Society.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3rd ed.)*. MIT Press, 2009.

[FAB+12] Paul Flicek, M. Ridwan Amode, Daniel Barrell, Kathryn Beal, Simon Brent, Denise Carvalho-Silva, Peter Clapham, Guy Coates, Susan Fairley, Stephen Fitzgerald, Laurent Gil, Leo Gordon, Maurice Hendrix, Thibaut Hourlier, Nathan Johnson, Andreas K. Khri, Damian Keefe, Stephen Keenan, Rhoda Kinsella, Monika Komorowska, Gautier Koscielny, Eugene Kulesha, Pontus Larsson, Ian Longden, William McLaren, Matthieu Muffato, Bert Overduin, Miguel Pignatelli, Bethan Pritchard, Harpreet Singh Riat, Graham R. S. Ritchie, Magali Ruffier, Michael Schuster, Daniel Sobral, Y. Amy Tang, Kieron Taylor, Stephen Trevanion, Jana Vandrovcova, Simon White, Mark Wilson, Steven P. Wilder, Bronwen L. Aken, Ewan Birney, Fiona Cunningham, Ian Dunham, Richard Durbin, Xos M. Fernndez-Suarez, Jennifer Harrow, Javier Herrero, Tim J. P. Hubbard, Anne Parker, Glenn Proctor, Giulietta Spudich, Jan Vogel, Andy Yates, Amonida Zadissa, and Stephen M. J. Searle. Ensembl 2012. *Nucleic Acids Research*, 40(D1):D84–D90, 2012.

[FGNV09] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *J. Exp. Algorithmics*, 13:12:1.12–12:1.31, 2009.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society.

[FT98] M. Farach and M. Thorup. String matching in lempel–ziv compressed strings. *Algorithmica*, 20:388–404, 1998. 10.1007/PL00009202.

[GV00] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 397–406, New York, NY, USA, 2000. ACM.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[HMN09] Nils Homer, Barry Merriman, and Stanley F. Nelson. BFAST: an alignment tool for large scale genome resequencing. *PLoS ONE*, 4(11):e7767, 11 2009.

[Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[HOK+09] Steve Hoffmann, Christian Otto, Stefan Kurtz, Cynthia M. Sharma, Philipp Khaitovich, Jörg Vogel, Peter F. Stadler, and Jörg Hackermuller. Fast mapping of short sequences with mismatches, insertions and deletions using index structures. *PLoS Comp. Biol*, 5(9):e1000502, 2009.

[IT99] Hideo Itoh and Hozumi Tanaka. An Efficient Method for in Memory Construction of Suffix Arrays. In *Proceedings of the IEEE String Processing and Information Retrieval Symposium*, pages 81–88, 1999.

[Jac88] Guy Joseph Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1988. AAI8918056.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 549–554, Washington, DC, USA, 1989. IEEE Computer Society.

[KÖ7] Juha Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249 – 257, 2007. The Burrows-Wheeler Transform.

[KA03] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Computer Engineering*, 2676:200–210, 2003.

[KHS+10] M. Oguzhan Külekci, Wing-Kai Hon, Rahul Shah, Jeffrey Scott Vitter, and Bojian Xu. Psi-ra: A parallel sparse index for read alignment on genomes. In *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine*, pages 663–338, 2010.

[KLA+01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc.12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192, London, UK, 2001. Springer-Verlag.

[KLL+12] Petr Klus, Simon Lam, Dag Lyberg, Ming Cheung, Graham Pullan, Ian McFarlane, Giles Yeo, and Brian Lam. Barracuda - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5(1):27, 2012.

[KMJP77] Donald Knuth, James H. Morris Jr., and Vaughan Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[KMR72] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, STOC '72, pages 125–136, New York, NY, USA, 1972. ACM.

[Knu07] Donald E. Knuth. *The Art of Computer Programming.* Pre-Fascicle 1A. Draft of Section 7.1.3: Bitwise Tricks and Techniques, 2007.

[KS03] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th international conference on Automata, languages and programming*, ICALP'03, pages 943–955, Berlin, Heidelberg, 2003. Springer-Verlag.

[KU96] Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In *Proceedings of the Second Annual International Conference on Computing and Combinatorics*, COCOON '96, pages 219–230, London, UK, 1996. Springer-Verlag.

[Kur03] Stefan Kurtz. The vmatch large scale sequence analysis software. http://www.vmatch.de, 2003.

[LD09] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[LLKW08] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[LRD08] Heng Li, Jue Ruan, and Richard Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18:1851–1858, 2008.

[LS99] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. Lu-cs-tr:99–214, Dept. of Computer Science, Lund University, 1999.

[LS07] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.

[LTPS09] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25+, 2009.

[LWW+12] Chi-Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu-Ming Yiu, Yingrui Li, B. Wang, C. Yu, X. Chu, K. Zhao, Ruiqiang Li, and Tak-Wah Lam. SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics*, 2012.

[LYL+09] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[LZZ+08] Hao Lin, Zefeng Zhang, Michael Q. Zhang, Bin Ma, and Ming Li. ZOOM! zillions of oligos mapped. *Bioinformatics*, 24(21):2431–2437, 2008.

[Man01] Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48:407–430, May 2001.

[Mar08] Elaine R. Mardis. The impact of next-generation sequencing technology on genetics. *Trends in Genetics*, 24(3):133 – 141, 2008.

[Met10] Michael L. Metzker. Sequencing technologies [mdash] the next generation. *Nat Rev Genet*, 11(1):31–46, Jan 2010.

[MKS10] Jason R. Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315 – 327, 2010.

[MM90] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *Soda '90: proceedings of the first annual ACM-SIAM symposium on discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. SODA '90: Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 319–327, San Francisco, 1990.

[Mor05] Yuta Mori. Short description of improved two-stage suffix sorting algorithm. http://homepage3.nifty.com/wpage/software/itssort.txt, 2005.

[Mor10] Yuta Mori. An implementation of the induced sorting algorithm. http://sites.google.com/site/yuta256/sais, 2010.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[NM07] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39, April 2007.

[NZC09] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Proceedings of the 2009 Data Compression Conference*, pages 193–202, Washington, DC, USA, 2009. IEEE Computer Society.

[NZC11] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Comput.*, 60:1471–1484, 2011.

[OVPB08] Brian D. Ondov, Anjana Varadarajan, Karla D. Passalacqua, and Nicholas H. Bergman. Efficient mapping of Applied Biosystems SOLiD sequence data to a reference genome for functional genomic applications. *Bioinformatics*, 24(23):2776–2777, 2008.

[PS08] Mihai Pop and Steven L. Salzberg. Bioinformatics challenges of new sequencing technology. *Trends in Genetics*, 24(3):142–149, 2008.

[PSD+08] Kay Prüfer, Udo Stenzel, Michael Dannemann, Richard E. Green, Michael Lachmann, and Janet Kelso. PatMaN: rapid alignment of short sequences to large databases. *Bioinformatics*, 24(13):1530–1531, 2008.

[PST07] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39, July 2007.

[RLD+09] Stephen M. Rumble, Phil Lacroute, Adrian V. Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. SHRiMP: Accurate mapping of short color-space reads. *PLoS Comp. Biol*, 5(5), 2009.

[RRR02] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '02, pages 233–242, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[RSK+09] Eric Rivals, Leena Salmela, Petteri Kiiskinen, Petri Kalsi, and Jorma Tarhio. Mpscan: fast localisation of multiple reads in genomes. In *Proceedings of the 9th international conference on Algorithms in bioinformatics*, WABI'09, pages 246–260, Berlin, Heidelberg, 2009. Springer-Verlag.

[Sad98] K. Sadakane. A fast algorithms for making suffix arrays and for burrows-wheeler transformation. In *Proceedings of the Conference on Data Compression*, pages 129–, Washington, DC, USA, 1998. IEEE Computer Society.

[SJ08] Jay Shendure and Hanlee Ji. Next-generation dna sequencing. *Nat Biotech*, 26(10):1135–1145, Oct 2008.

[SL09] Michael Strömberg and Wan-Ping Lee. MOSAIK read alignment and assembly program. http://bioinformatics.bc.edu/marthlab/Mosaik, 2009.

[Sto08] Jens Stoye. Suffix tree construction in ram. In *Encyclopedia of Algorithms*. Springer, 2008.

[SXZ08] Andrew D. Smith, Zhenyu Xuan, and Michael Q. Zhang. Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC Bioinformatics*, 9(1):128–135, 2008.

[TM81] Smith TF and Waterman MS. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[TS09] Cole Trapnell and Michael C. Schatz. Optimizing data intensive gpgpu computations for dna sequence alignment. *Parallel Comput.*, 35:429–440, August 2009.

[Vig08] Sebastiano Vigna. Broadword implementation of rank/select queries. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 154–168, Berlin, Heidelberg, 2008. Springer-Verlag.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

[WER+09] David Weese, Anne-Katrin Emde, Tobias Rausch, Andreas Döring, and Knut Reinert. RazerS–fast read mapping with sensitivity control. *Genome Research*, 19(9):1646–1654, 2009.

[YRB08] Vladimir Yanovsky, Stephen M. Rumble, and Michael Brudno. Read mapping algorithms for single molecule sequencing data. In *Proceedings of the 8th international Workshop on Algorithms in Bioinformatics (WABI)*, volume 5251 of *LNCS*, pages 38–49. Springer, 2008.