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

AGREGANDO BÚSQUEDAS SOBRE COLECCIONES GENÓMICAS EN FORMATO VCF

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERA CIVIL EN COMPUTACIÓN

FERNANDA ISIDORA SANCHIRICO BARRERA

PROFESOR GUÍA:
GONZALO NAVARRO
PROFESOR GUÍA 2:
DIEGO ARROYUELO B.

MIEMBROS DE LA COMISIÓN:
JOSÉ M. SAAVEDRA R.
FEDERICO OLMEDO

Este trabajo ha sido parcialmente financiado por el Centro de Biotecnología y Bioingeniería (CeBiB)

SANTIAGO DE CHILE
JULIO 2022

Resumen

Cuando se trata de hacer estudios de variantes en el genoma, el principal formato que se utiliza para representar la información de interés es el *Variant Calling Format* (VCF). La principal característica de VCF es que almacena el genoma de un conjunto de individuos en base a las variantes (edits) que tiene con respecto a un genoma de referencia. Si bien este formato hace buen uso de que el genoma de individuos tiene más semejanzas que diferencias, cuando se trata de realizar consultas sobre los genomas que este representa, VCF no da abasto de forma eficiente. Si quisiéramos buscar la ocurrencia de un patrón dentro de VCF, deberíamos recrear la cadena, es decir descomprimir VCF, y sobre ella buscar. Esto es impracticable sobre grandes colecciones genómicas.

Ahora, el problema descrito corresponde a una tarea bastante estudiada en computación, que es la buśqueda de patrones sobre textos de alta repetitividad. Como respuesta a este problema se han desarrollado los índices comprimidos para colecciones repetitivas, donde uno de sus exponentes posee un proceso de construcción análogo a la forma de trabajo de VCF. Este índice corresponde a *Relative Lempel-Ziv* (RLZ), el cual nos permite realizar búsquedas sobre el texto indexado sin nunca descomprimirlo.

En este Trabajo de Título se busca soportar búsquedas sobre colecciones genómicas en formato VCF sin descomprimir. Para hacer esto posible, se diseña e implementa un módulo de conversión de VCF a RLZ, el cual a partir de la construcción de este índice permite realizar búsquedas y entregar posiciones de ocurrencias en un formato concordante a la información que representa VCF. Esta conversión se hace sin descomprimir el VCF.

La solución implementada consta de tres etapas de procesamiento de VCF: caracterización de edits, agrupación de caracterizaciones e interpretación de caracterizaciones para construir RLZ. En paralelo, a partir de la interpretación generamos una estructura de datos compacta extra que nos permitirá soportar la transformación de posiciones de ocurrencia en RLZ a VCF. Se utilizó una implementación de RLZ proveída por los profesores guía, donde con el fin de no requerir la cadena original para su construcción, se editó el constructor del índice.

Este módulo fue validado sobre distintos datasets generados a partir de los VCF publicados por el proyecto *1000 Genomes*, con el fin de evaluar los tiempos de conversión y volumen de los productos. El módulo cumplió con el objetivo propuesto, procesando hasta 72 genomas humanos en 6 horas, y dejando la información lista para ser consumida por RLZ. Sin embargo, el ordenamiento de prefijos y sufijos requerida por el índice RLZ significó un cuello de botella significativo en los tiempos de construcción. Se consiguió indexar el cromosoma 21 hasta para 12 individuos en 11 horas y demostrar la funcionalidad de las búsquedas.

Porque sobrevivimos

Agradecimientos

En primer lugar, agradecer a Gonzalo, mi profe guía, por la paciencia y serenidad de explicarme todas aquellas cosas teóricas que me resultaron muy difíciles de entender, partiendo por RLZ. Maldigo a los papers de conferencia, que explican con 3 capas de procesamiento ocultas, y prendo una vela por cada pobre alma como yo que tuvo que aprender a partir de ellos. También agradecer a Diego, mi profe co-guía, quién fue mi salvador ante aquellos místicos errores de C++ que jamás hubiese resuelto sola sin pedir 3 prórrogas. Terminó de enseñarme las mañas de C++ que desconocía y darme alivio cuando ya daba todo por perdido.

Agradecer a Karen Oróstica, junto a su equipo de investigación, que me permitieron acotar y medio terminar de entender VCF. Y jamás olvidaré la paciencia de santo que me tuvo Sergio Aguilera cada vez que tenía problemas con Knuth, que por no saber cómo hacer un CMakeLists, me instaló todas las librerías que no sabía incluir en ese momento. Y gracias a ese pdf que ya no recuerdo que me enseñó a hacer CMakeLists, eso si que lo recordaré.

Además agradecer a mi panita Seba, que siempre estuvo dándome su apoyo y por carrearme en otro proyecto paralelo, lo que me permitió llegar viva a este punto. Porque sí, quienes me conozcan saben que soy mandada a hacer para irme en la *multitask* y hacer mil cosas a la vez, pero este año ese gusto por la mala vida me pasó la cuenta.

Agradecer a mi familia por hacer posible este momento, en particular al esfuerzo de mis papás y todo el mega carreo de mi mamá. Agracerte Neni por regalonearme cuando lo necesitaba, y la Mandy por entender que estudiar computación no significaba solo hacer mods de Minecraft (ojalá), y que para hacer eso primero me tenía que titular y necesitaba que me dejara tranquila. Las amo monas chicas. Con ello a mis hermanas del corazón, mis washas del colegio, que fueron pilar de lo que soy hoy.

Y finalmente agradecer al Rorro por escuchar mis explicaciones como buen patito de goma, escucharme recitar mi informe una y otra vez para verificar que no escribiera payasadas, y por sobre todo el cariño y contención que necesitaba en este proceso. Por cada vez que recibió una Feña hecha polvo y devolvía una con los puntos de vida llenos para seguir.

Ah y gracias a mis perros por ser tan gordos :) Qué cosas más bellas. Y al mojito por existir.

Puede que digan “Oh el agradecimiento meloso”, bueno, sí lo es. Pero estoy escribiendo esto a las 00:12 del 26 de Septiembre, que es mi cumpleaños. Así que déjenme piola.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Antecedentes

Dentro de la bioinformática, el secuenciamiento de ADN es un proceso cada vez más rápido y barato. Por ejemplo, al alero de Oxford, está disponible la línea de dispositivos Nanopore[1], la cual nos permite secuenciar ADN en tiempo real, cuyo dispositivo más económico es de 1000USD (Modelo MinION Mk1B). Mismo valor al que también se pueden adquirir kits de secuenciamiento Illumina[2], precios que se consideran económicos dentro del rubro. Notemos que MinION Mk1B es capaz de generar un output de 50GB de texto plano[3] a partir del proceso de secuenciamiento, donde el resto de los dispositivos pueden generar hasta 14TB.

Sin ir más lejos, lo anteriormente descrito ilustra un estado análogo al que se observa en múltiples áreas vinculadas al procesamiento de datos, la velocidad de crecimiento en tamaño de los repositorios de genomas, nuestros datos, es muy alta. El cual si bien es un panorama positivo, esta situación tensiona nuestras capacidades para almacenar y procesar los datos, donde eventualmente las técnicas clásicas no son una posibilidad óptima extensible en el tiempo. Sin embargo, esto supone una gran oportunidad de innovación y estudio para su solución.

Nuestro gran aliado para poder almacenar estos gigantescos volúmenes de información es la *repetitividad*. Se ha notado que los genomas de una especie difieren poco entre sí, lo suficiente como para que sea mucho más conveniente almacenar sus diferencias en base a lo que tienen en común. Es por ello que los archivos de secuenciamiento en texto plano de múltiples individuos se hacen pasar por diversos procesos, como el alineamiento de secuencias (*Alignment*) y la recuperación de variantes (*Variant Calling*), los cuales nos permiten capturar diferencias, los "edits", con respecto a un genoma de referencia. Se construye así un documento en formato VCF[4] (*Variant Call Format*), el cual codifica esos edits y en la práctica reduce el espacio necesario para almacenar el genoma de una polación en un 75 %

[1] https://nanoporetech.com/applications/dna-nanopore-sequencing
[2] https://www.illumina.com/science/technology/next-generation-sequencing/beginners/ngs-cost.html
[3] https://nanoporetech.com/products/comparison
[4] https://github.com/samtools/hts-specs/blob/master/VCFv4.3.pdf

aproximadamente.

De esta forma podemos decir que VCF consiste en el almacenamiento de una colección genómica en forma comprimida, cuya representación corresponde a un genoma de referencia más un conjunto de edits, donde al recrear las variaciones sobre esta referencia es que obtenemos el genoma de cada individuo declarado en el archivo.

Hacer búsquedas y análisis sobre las secuencias que representan a los individuos de una población es clave para toda clase de procesamiento bioinformático. Por ejemplo, si buscamos caracterizar mutaciones con fines médicos o bien identificar variantes genéticas a lo largo del tiempo para estudios evolutivos, tendremos que buscar patrones de interés dentro de los genomas. Actualmente, esto requiere descomprimir el formato VCF y recrear todas las cadenas a partir de los edits indicados, ya sea para procesarlas secuencialmente sin almacenarlas o para almacenarlas y construir estructuras de datos sobre ellas. El uso de VCF no apoya significativamente este proceso, sólo presta una ayuda temporal para el problema de archivar contenido.

1.2. Motivación

Hablando en términos computacionales, el problema del estudio del genoma responde a trabajar sobre secuencias dentro de un alfabeto sumamente acotado, pues dentro del ADN trabajamos 4 bases nitrogenadas, representadas con A, C, G y T, y usamos la letra N para cuando desconocemos qué base se encuentra en esa posición. Lo que distingue este escenario es principalmente que, al secuenciar muchos individuos de una misma especie, aparte de lo masivos que se vuelven los datos, se induce en un alto grado de repetitividad [23], situación sobre la que VCF sustenta su ventaja.

Lo que buscamos en esta memoria es potenciar VCF para que, mediante el uso de índices comprimidos para colecciones repetitivas [27], podamos realizar búsquedas en la colección sin necesidad de descomprimirla en ningún momento, en particular, acercar un formato alternativo, que responde favorablemente a los problemas de ejecución y espacio, permitiendo soportar consultas sobre el genoma en VCF sin nunca descomprimir. Para ello, se hará uso de una implementación de indexación de colecciones repetitivas con Relative Lempel-Ziv (RLZ) [28].

RLZ es una técnica reciente que se aprovecha de la repetitividad dentro de una colección de secuencias, cuya estrategia corresponde a tomar un substring de referencia, perteneciente a la cadena más grande, y almacenar las variaciones como un edit sobre el mismo, guardando un puntero a la sección de la referencia donde ocurre el cambio.

Este índice es un representante más de una familia de estructuras compactas llamadas *autoíndices comprimidos para texto*, que soportan accesos directos sobre la data comprimida, ahorrando memoria, y junto con ello ofrecen tiempos de búsqueda eficientes sobre la misma [28]. Vale destacar que las estructuras de datos compactas que soportan consultas sin descompresión corresponden a estructuras de datos sucintas. Estas estructuras pueden representar un objeto (como un bitvector o un árbol) en el espacio cerca del límite inferior de

la teoría de la información del objeto, soportando operaciones del objeto original de forma eficiente.

Como ilustran párrafos anteriores, VCF y RLZ comparten un mismo espíritu en su forma de trabajo, lo que facilita su combinación. Esta es una ventaja única de RLZ, sobre todas sus alternativas, para este trabajo. Sin embargo, esta afinidad presume un considerable desafío, pues en este caso de estudio, RLZ sería capaz de leer e indexar alelo por alelo, en cambio VCF observa edit por edit en cada cromosoma, donde cada individuo declara si cada uno de sus alelos codifica la variante o no. Se podría construir el índice de RLZ a partir de VCF trivialmente mediante descomprimir las secuencias codificadas en VCF y luego indexarlas con RLZ, pero esto implicaría un costo de espacio y tiempo de procesamiento impracticable. El desafío principal de esta memoria es realizar esta conversión sin descomprimir el formato VCF, construyendo un único índice RLZ para todos los alelos de todos los genomas de los individuos representados. Debe notarse que tampoco es útil construir índices individuales para cada alelo o individuo, lo que podría ser computacionalmente factible, pero se perdería la repetitividad y la capacidad de RLZ buscar en toda la colección de una sola vez.

Consideremos además que pese a que RLZ promete excelentes resultados, este no es conocido ni explotado aún. Con este trabajo podremos acercar e ilustrar su potencial, por medio de generar un software que soporte aquellas funcionalidades que, a día de hoy, no provee ningún otro formato de archivo como el acceso a los datos y búsquedas sin descompresión, y con ello aportar a la bioinformática desde la computación pura.

Por otro lado, vale destacar la relevancia de apuntar a pasar desde VCF a RLZ, y no otro formato de origen. Dentro de la bioinformática, VCF es altamente conocido, de hecho es el formato canon cuando se trata de estudiar variantes y mutaciones en genoma. Proponer un tipo de archivo que reemplace a VCF no tendría buena aceptación, pues es un área conservadora para sus herramientas. Es por ello que la combinación es particularmente afortunada, porque VCF provee directamente el genoma de referencia y los edits, que es la parte más costosa de construir cuando se crea el índice de RLZ. De esta forma, a los ojos de quienes utilicen la herramienta, nunca dejarán de usar VCF, simplemente el software desde adentro utilizará RLZ como una herramienta de optimización.

Desde una perspectiva en aporte social, recordemos que sobre el estudio de variaciones y mutaciones en el genoma es que podemos ser capaces de identificar, de forma temprana y no invasiva, diversas condiciones de salud. Por lo que acercar la utilidad de RLZ no solo significa un aporte a la forma en la que se investiga, sino incluso la velocidad a la que se pueden llevar éstas a cabo. De hecho, la reducción de costos tanto en tiempo como en equipos para procesamiento de los datos puede hacer la investigación en salud más accesible a sociedades con menores recursos.

Finalmente, incluso en el marco de la contingencia sanitaria, la caracterización de la nocividad de las variantes del SARS-CoV-2 (COVID-19) depende netamente de la identificación de mutaciones en el genoma de ciertas proteínas que componen al virus [18], la cantidad y tipo de mutaciones son información clave. Información que es fruto del proceso que se busca mejorar en esta memoria.

1.3. Descripción General de la Solución

Con el fin de introducir la forma en la que se trabajará la construcción de RLZ a partir de VCF, se hace necesario ilustrar con un poco de detalle la forma de trabajar de cada uno de éstos.

Primero, entendamos cómo funciona VCF. A nivel de formato de texto, contiene líneas de meta-información (antecedidas por un "##"), una línea de header (antecedida por "#"), y cierra con líneas que contienen los datos sobre la posición de la variación en un cromosoma (edits) e información del genotipo en muestras por cada posición (separadas por tabs), a estas líneas las llamaremos registros. Aquellos campos obligatorios que no contienen información deben ser rellenados con un punto (".").

Dado que VCF codifica las variantes en el genoma de un conjunto de individuos, solo para efectos ilustrativos los ejemplos simularán la muestra para un individuo, ejemplos más completos se verán en la sección 2.2. Entonces, observemos el siguiente ejemplo de formato VCF para un individuo `INDV_1`, donde su cromosoma número 1 corresponde a los alelos {`ATGGA`, `ATGA`}, con referencia en un único cromosoma `ATCGA`:

```
##fileformat=VCFv4.3
##filedate=20090805
##source=myImputationProgramV3.1
##reference=file:///seg/references/1000GenomesPilot-NCBI36.fasta
##contig=<ID=20,length=9,assembly=B36,md5=f124g6453a,species="Homo Sapiens">
#CHROM  POS ID  REF     ALT    QUAL  FILTER  INFO    FORMAT INDV_1
 1      2   .   TC      TG,T   .     PASS    DP=100     GT  2|1
```

La referencia se encuentra en el archivo indicado en `reference`, que contendrá la secuencia de cromosomas en formato FASTA, mayores detalles de este formato serán vistos en la sección 4.1. Para reconstruir la primera variante, `ATGGA`, nos interesan las columnas `POS`, `REF` y `ALT` del header, donde se debe interpretar lo siguiente: En la posición (`POS`) 2 con respecto a al cromosoma 1 (`CHROM`) de la referencia, tomamos la base `TC` (`REF`) y la alternamos por la base `TG` (`ALT`). Análogamente, para recuperar `ATGA` decimos lo mismo, pero alternamos `TC` por `T`, el segundo valor disponible en `ALT`. Luego, la columna `INDV_1` nos indica que este individuo, su primer alelo para el cromosoma 1 es la segunda variante declarada (el 2 declarado antes de "|"), y el segundo alelo es la primera variante (el 1 declarado después de "|"). Notemos que si los alelos de `INDV_1` no contienen la variante, entonces debe declararse "`0|0`".

Sobre el funcionamiento de RLZ, tenemos lo siguiente: A partir de una colección genómica, construye una referencia $\mathcal{R}$ (que puede ser uno de los genomas u otro string externo) y *parsea* cada genoma de la colección con respecto a $\mathcal{R}$, escribiendo cada genoma como una secuencia de *factores* (substrings de $\mathcal{R}$). Con $\mathcal{R}$ y el parsing, construye un índice cuyo tamaño es proporcional al largo de $\mathcal{R}$ más la cantidad total de factores. A partir de esta representación, puede extraer cualquier substring de cualquier genoma en tiempo óptimo, y localizar todas las *occ* ocurrencias de un patrón (un substring) de largo m en tiempo $O((m + occ) \log N)$,

donde N es el largo de la colección representada. La explicación detallada se encuentra dentro de la sección 2.3.

Entonces, teniendo en cuenta los esquemas con los que trabajan VCF y RLZ, la conversión radica en las siguientes etapas:

- **Procesamiento de la referencia**: Se almacena la concatenación de las cadenas que representan a cada cromosoma en la referencia, y solo para la etapa de construcción se guardan las posiciones relativas de inicio de cada cromosoma (Sección 4.1).
- **Parsing de los registros**: Capturamos todas las muestras declaradas en la línea header, para luego caracterizar cada uno de los *edits* en las líneas de registro con una *frase* (Sección 4.2).
- **Ordenamiento de las frases**: Dada la masividad de las *frases* recuperadas, el ordenamiento de estas se hace por medio de una implementación de ordenamiento en disco, proveída por la librería STXXL [8] (Sección 4.3).
- **De frases a factores**: Las *frases* construidas deben ser reinterpretadas a la estructura que requiere la construcción de RLZ, es decir factores de Lempel-Ziv (Véase la sección 2.3.1). En esta etapa, concatenamos los factores que representan el genoma de cada individuo, de esta forma fusionamos los genomas como un único gran string (Sección 4.4).
- **Reinterpretación de ocurrencias en RLZ**: Las ocurrencias reportadas por RLZ son entregadas bajo el contexto de un texto único, entonces en base a restricciones estructurales de la construcción, podemos remapear estas ocurrencias a sus posiciones originales (Sección 4.5).

Notar que la API será desarrollada en C++11 y Python, todo dentro de Linux, y con ello una posterior compatibilidad con este sistema operativo (SO), en virtud de un mejor trabajo y consistencia con el manejo de archivos, aparte de conseguir un producto con dependencias más limpias. Vale destacar que el desarrollo de la solución dentro de Linux no desfavorecerá el alcance de ésta, pues dentro de la bioinformática ya se trabaja con sistemas cuyas compatibilidades son exclusivas a este SO.

1.4. Objetivos

Objetivo general

Desarrollar un software que permita realizar búsquedas de coincidencias sobre grandes colecciones genómicas a partir de su formato VCF, sin la necesidad de descomprimir el archivo. El programa debe proveer una API, la cual internamente transformará los archivos VCF a RLZ y con ello ofrecerá la funcionalidad de búsqueda con reporte de resultados asociados, además de poder almacenar la estructura generada en archivos en disco para su distribución o posterior uso.

Objetivos específicos

Para alcanzar el objetivo general, se hacen necesarios los siguientes pasos:

1. **Comprender el funcionamiento de RLZ**: Se debe estudiar el funcionamiento del índice, con el fin de identificar los aspectos teóricos y prácticos asociados, para luego familiarizarse con la implementación realizada por Diego Arroyuelo y los requerimientos de funcionamiento.
2. **Comprender el funcionamiento de VCF**: Para decidir qué variantes de VCF se trabajarán, se requiere entender la estructura del formato, cómo identificar sus variantes y definir cuáles son compatibles con la transformación a RLZ.
3. **Implementar módulo de conversión**: A partir de la implementación de RLZ, crear un módulo que reciba un archivo VCF y genere un índice de formato RLZ que describa la misma colección de genomas.
4. **Implementar la API**: A partir del índice y construyendo sobre el código de RLZ, generar un módulo capaz de reportar las ocurrencias de un substring dado.

Vale destacar que este trabajo fue presentado en la décimo séptima versión del *Workshop on Compression, Text, and Algorithms* (WCTA 2022 [5]), satélite del *29th International Symposium on String Processing and Information Retrieval* realizado en la Universidad de Concepción en Noviembre de 2022. En virtud de la contribución de este trabajo, todos los códigos asociados se encuentran en repositorios públicos en Github[6]. En el README del repositorio principal se encontrará explicado su uso.

[5] http://spire2022.inf.udec.cl/wcta.html
[6] https://github.com/SanquirinoB/VCF_RLZ

Capítulo 2

Marco Teórico

A continuación se ilustrarán los distintos tópicos que son relevantes para el entendimiento de esta memoria. Como primera parte explicaremos brevemente de qué se trata el estudio de la genética de poblaciones, área que nos permite acotar el alcance de este trabajo; luego la descripción técnica necesaria sobre VCF y cuáles de sus aspectos son relevantes para la conversión, de igual forma qué aspectos quedarán fuera del alcance de este trabajo; y finalmente ilustraremos de qué consta RLZ y conceptos relacionados. Después de una primera parte de conceptos necesarios, dedicaremos una sección sobre trabajos relacionados y cómo se maneja actualmente el problema de la masividad y procesamiento de VCF.

2.1. Genética de poblaciones

Corresponde a una la rama de la genética, la cual responde al estudio de la variación y distribución de la frecuencia alélica dentro de un conjunto de individuos. Es decir, en base a un grupo de individuos de una misma especie recuperamos los genomas y sobre ello se hacen estudios de variantes o preservación. Notar que VCF permite todo tipo de estudios sobre diferentes tipos de muestras y especies, sin embargo para efectos de esta memoria nos acotaremos los tipos de archivos VCF destinados al estudio genético de poblaciones humanas.

2.2. Variant Calling Format

Como explicamos previamente, VCF corresponde a un formato que almacena las variantes genómicas de un conjunto de individuos para un genoma de referencia. Dado que este formato es humanamente leíble, podemos identificar diferentes secciones del archivo que nos permiten interpretar estas variantes.

Como primer acercamiento, ilustraremos la forma en la que se codifica la meta-información del documento, las primeras líneas de especificación; y luego cómo se interpretan las variantes, información posterior a la meta-información.

2.2.1. Meta-información

El formato de las líneas de meta-información se encuentran explicadas en la Sección 1.3, en particular ahora nos interesan las que son antecedidas por "##". Estas líneas estructuradas son de la forma:

```
##FIELD_NAME=<FIELD_1=VALUE_1, ... ,FIELD_n=VALUE_n>
```

`FIELD_NAME` representa el tipo de la línea estructurada, y `FIELD_i` junto a `VALUE_i` el campo y valor asociado. En estas líneas, depende del tipo de línea estructurada la cantidad de campos que ésta posea. Solo es invariante es la forma en la que se declaran: Todos los campos deben poseer valores asociados y los campos con sus valores deben separarse con comas ",", sin espacios.

De todas las líneas estructuradas, las principales son: `reference`, la ubicación del documento que contiene el genoma de referencia; `ALT`, declaración de tipos de alteraciones complejas de los alelos, es decir, formas de variación y sus descripciones, dado que no pueden ser codificadas con variantes simples (Véase la sección 2.2.4); `assembly`, contiene la ubicación de un documento análogo a la referencia, este archivo se hace necesario solo en casos de variantes complejas donde la referencia a utilizar no se encuentra en `reference`; `contig`, son las lineas de caracterización de las secuencias a ser utilizadas en las variantes, suelen referir a las secuencias que se encuentran en `reference` y `assembly`.

2.2.2. La Referencia

El archivo que almacena la referencia de VCF corresponde al formato FASTA[1] (.fa), donde la estructura de estos archivos corresponden a dos tipos de líneas. Una de ellas es la caracterización de la cadena, esta línea posee al menos el ID de la cadena, luego la información puede ser complementada con el largo de la cadena, descripción, etc.

El otro tipo de líneas, que continúan a la línea anterior, corresponden al texto plano de la cadena, donde el alfabeto a utilizar es {ACGTNacgtn}, notar que no es *case sensitive*. Además cada línea de este tipo tiene una cantidad fija de bases por línea, la cual puede variar entre archivos FASTA, pero es la misma dentro del archivo.

Vale destacar que la letra N se utiliza para simbolizar el hecho de que desconocemos cuál de las 4 bases es utilizada en esa posición. Normalmente esta letra ocurre en regiones dentro de la cadena.

Por ejemplo para nuestro caso particular, un archivo FASTA para el cromosoma 1 contendría la siguiente información:

```
>1
```

[1] http://bioinformatics.intec.ugent.be/MotifSuite/fastaformat.php

```
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
ACcTGGggNNAACGTACcTGGggNNAACGT
GTaTGNANTAgNaTaGTaTGNANTAgNaTa
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
```

De tal forma que si buscamos recuperar la cadena completa del cromosoma 1, solo debemos concatenar las 4 líneas de bases.

2.2.3. Líneas de datos

Existen 9 campos obligatorios a declarar para cada variante, donde la línea de header, la única línea antecedida por "#", contiene las siguientes columnas:

1. `CHROM`: Identificador del cromosoma en el que ocurre la variante, se declara con el número de cromosoma, que debe coincidir con el declarado al interior del archivo `reference`.
2. `POS`: La posición dentro de la referencia en la que ocurren la(s) variación(es) declarada en `ALT`, donde la primera base tiene la posición 1. Todas las líneas de datos se encuentran ordenadas por cromosoma, y dentro del mismo cromosoma se ordenan en orden creciente sobre este valor.
3. `ID`: El identificador de la variante, es optativo.
4. `REF`: Base(s) de referencia en la que ocurre la variante, un substring que inicia en `POS` y debe ser alterado según indique `ALT`.
5. `ALT`: La(s) base(s) con la(s) que se altera la referencia, pueden ser varias, las cuales se separan con comas. En caso de ser variantes complejas, puede contener más información, dependiendo el tipo de reemplazo.
6. `QUAL`: La calidad de la variante detectada, que nos indica la probabilidad con la que `ALT` sea verdadero.
7. `FILTER`: Indica si la posición en la que se reconoce la variante cumple con ciertos filtros, en general cuando los edits son suficientemente válidos para estas métricas, la columna se completa con el valor `PASS`.
8. `INFO`: Información adicional, pueden ser parámetros estadísticos de la variante y valores específicos para codificar la alteración aplicada, dependiendo del tipo de esta.
9. `FORMAT`: Declara la estructura de los valores que se encontrarán en la columna asociada al ID del individuo, cada tipo de parámetro es separado por ":". El parámetro de nuestro interés es `GT`, este indica dónde podemos recuperar cuál variante es codificada por cada individuo, o en su defecto si no lo usa (Véase el ejemplo de la sección 1.3).

Luego en esta misma línea se continua con los ID que caracterizarán a cada individuo, cuya cantidad es arbitraria.

Entendiendo la estructura general de las líneas de datos, podemos dar pie a las distintas formas de variantes que podemos representar. Notemos que podemos representar sustituciones, eliminaciones e inserciones, que pueden venir en dos formatos: variantes simples, que se pueden describir autocontenidas en una misma línea de datos (como el ejemplo de la sección 1.3); y las complejas (rearrangements), aquellas cuya variante requiere referenciar una cadena externa. Luego, en las columnas de individuos indicamos si las variantes declaradas se encuentran en el alelo respectivo o no. Además, las variantes son acumulables, es decir, las variantes de un mismo cromosoma se describen en múltiples líneas de datos seguidas.

En las siguientes subsecciones explicaremos con más detalle cada una de estas variantes y después cómo las llevamos a los individuos.

2.2.4. Recuperación de variantes simples

Para ilustrar cada tipo de variante simple, primero observaremos un ejemplo que ilustra cómo se declaran estas variantes sobre cadenas pequeñas y en la explicación de cada variante mostraremos cómo cambia su nomenclatura para cadenas más largas, en caso de aplicar.

	#CHROM	POS	ID	REF	ALT	QUAL	FILTER	INFO	FORMAT
(1)	20	3	.	C	T	.	PASS	.	GT
(3)	20	5	.	CTAG	C	.	PASS	.	GT
(2)	20	6	.	C	CTAG	.	PASS	.	GT
(4)	20	9	.	CTA	C, CTT	.	PASS	.	GT

Primero tenemos la **sustitución**, el caso en el que la base declarada en `ALT` reemplaza directamente a `POS`. Por ejemplo, la línea (1) nos dice que en el cromosoma 20, posición 3, la base C debe ser reemplazada por T. Dada la naturaleza de esta variante, esta no es utilizada para cadenas más grandes.

Por otra parte la **eliminación** es lo contrario, nuestro `ALT` es menor que la referencia, en general solo se escribe en `ALT` la primera base de `REF` para simbolizar la eliminación del resto de la cadena. Entonces la línea (3) se interpreta de la siguiente forma: con respecto al cromosoma 20, posición 5, la cadena `CTAG` se elimina, dejando solo la base `C`. Para cadenas más grandes, en `ALT` se declara `<DEL>` y en la columna `INFO` puede aparecer el atributo `SVLEN` o `END` indicando el largo de la cadena a ser eliminada desde `POS`. Estos dos atributos serán explicados en el ejemplo de la variante duplicación más adelante.

Luego está la **inserción**, que sigue el mismo principio de la sustitución, pero la cantidad de bases en `ALT` son más que las de la referencia y la contienen. Observando la línea (2) la variante sería: con respecto al cromosoma 20, en la posición 6, desde la base `C` se inserta la cadena `TAG`. Para cadenas más grandes, tenemos las variantes complejas descritas en la Sección 2.2.5, que escapan de esta sección, pero si tenemos la **duplicación**.

La **duplicación** puede ser representada de dos formas:

```
    #CHROM  POS ID  REF  ALT    QUAL  FILTER   INFO     FORMAT
(5) 20      10  .   C   <DUP>  .     PASS     SVLEN=5  GT
(6) 20      12  .   C   <CNV3> .     PASS     END=15   GT
```

El formato de la línea (5) corresponde a que en la posición 10 del cromosoma 20 se debe duplicar la cadena que parte en la posición 10, de largo 5 (`SVLEN=5`). O bien se puede usar el formato de la línea (6), que representa una **copia de numero variable** (CNV), que se muestra con la estructura `<CNVi>`, donde la `i` representa el número de copias a realizar, que en este caso indica que debemos copiar consecutivamente 3 veces la cadena. Vale destacar que el largo de la cadena a duplicar se puede caracterizar con su posición de inicio en `POS` y su largo con `SVLEN` o bien posición de cierre con `END`.

Siguiendo estas reglas, podemos hacer líneas de datos combinadas, como muestra la línea (4), dentro del mismo `ALT` declaramos múltiples variantes. En este caso, como primera variante tenemos una eliminación de la cadena `TA`, y la segunda variante es una sustitución de la base `A` por `T`. Si la variantes declaradas en `ALT` requieren atributos específicos de la columna `INFO`, sus valores respectivos serán separados por comas "," dentro de la llave respectiva.

2.2.5. Variantes complejas, Rearrangements

Una forma de entender los rearrangements es que una variante compleja corresponde a la unión de un prefijo de la referencia, junto al sufijo de otra cadena que no es necesariamente la referencia, y viceversa.

Análogo a la subsección anterior, también tenemos la inserción de aquellas cadenas cuya escritura plana en el mismo documento no nos conviene, por lo que se referencia a un *contig*[2] del documento declarado en la línea de meta-información `contig`, y con ello múltiples posibilidades. También se pueden representar inversiones, es decir, una variante corresponde a invertir un substring de la cadena en su misma posición.

Un ejemplo de rearrangement es la situación de la Figura 2.1, que se representa con las siguientes líneas de datos:

```
#CHROM POS    ID    REF ALT          QUAL FILTER INFO
2      321681 bnd_W G   G[13:123457[ 6    PASS   .
2      321682 bnd_V T   ]13:123456]T 6    PASS   .
13     123456 bnd_U C   C[2:321682[  6    PASS   .
13     123457 bnd_X A   ]2:321681]A  6    PASS   .
```

De donde se interpreta lo siguiente: Dentro del cromosoma 2, a partir de la posición 321681, concatenamos el cromosoma 13 a partir de la posición 123457. Luego, nuevamente el

[2]Un contig es una serie de secuencias superpuestas de ADN utilizadas para hacer un mapa físico que reconstruye la secuencia original de ADN de un cromosoma o de una región de un cromosoma.

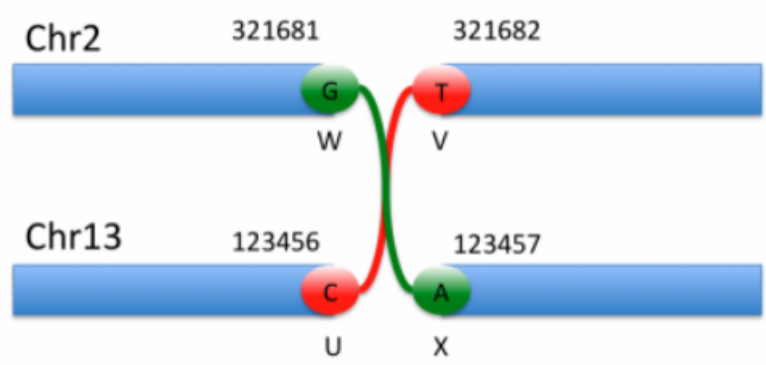

Figura 2.1: Ejemplo de rearrangement, recuperado del documento con la especificación del formato VCF[6]

cromosoma 2, hasta la posición 321682, con respecto a la base T, tenemos que concatenarla al cromosoma 13 desde la posición 123456. Para las últimas dos líneas el proceso es análogo, pero con respecto al cromosoma 13.

La sintaxis $]n_{chr} : p_{chr}]$, o con los paréntesis al revés, se llama *breakpoints*, donde n_{chr} es el número del cromosoma y p_{chr} la posición en este.

Con *breakpoints* podemos representar todas las variantes que impliquen cambios con cadenas grandes. Incluso, cuando trabajamos con *contig*, donde su uso solo se diferencia en que en `ALT` en vez de declarar el número del cromosoma, en n_{chr}, declaramos el id del *contig*. Por ejemplo:

```
#CHROM   POS ID REF ALT           QUAL FILTER INFO
2        32  .  G   G[<ctg_1>:12[ 6    PASS   .
2        32  .  T   ]<ctg_1>:12]T 6    PASS   .
13       12  .  A   ]<ctg_1>:32]A 6    PASS   .
```

Donde `<ctg_1>` es el ID del *contig* referenciado. A su vez, también podemos utilizar estos mismo *contig* como referencia.

Por otra parte, dentro de los rearrangements podemos soportar **inversiones**, las cuales indican que una variante corresponde a invertir un substring de la referencia dentro de su misma posición. Estas variaciones se pueden representar de dos formas.

Una forma es declarar la variante compactada en una línea, en base al tag `<INV>` en la columna `ALT`, donde en la columna `POS` indicamos el inicio de la inversión y en la columna `INFO` se declara la posición final con `END=p`, con `p` la posición, como ilustra el siguiente ejemplo, que representa a la Figura 2.2.

```
#CHROM POS    ID   REF ALT   QUAL FILTER INFO
 2     321682 INV0 T   <INV> .    .      END=421681
```

Estas líneas indican que en el cromosoma 2, debemos invertir la cadena desde la posición 321682 hasta la 421681. Otra forma es hacerlo en base al principio de acumulación de variantes a lo largo de las líneas de datos, donde se necesitan 4 líneas para indicar la reconexión de los puntos. Estas 4 líneas, en base al ejemplo anterior, debiesen ser las siguientes:

```
    #CHROM POS    ID REF ALT           QUAL FILTER  INFO
(1) 2      321681 W  G   G]2 : 421681] .    PASS    MATEID=U;EVENT=INV0
(2) 2      321682 V  T   [2 : 421682[T .    PASS    MATEID=X;EVENT=INV0
(3) 2      421681 U  A   A]2 : 321681] .    PASS    MATEID=W;EVENT=INV0
(4) 2      421682 X  C   [2 : 321682[C .    PASS    MATEID=V;EVENT=INV0
```

Figura 2.2: Ejemplo de inversión

Las que se interpretan de la siguiente forma: (1) Para el cromosoma 2, tomamos el *breakend* `W`, ubicado en la posición 321681 de la referencia, y lo conectamos a su *mate breakend* `U` (`INFO`) de la posición 421681 del mismo cromosoma (`ALT`). (2) Luego tomamos el *breakend* `V` en la posición 321682 y lo conectamos a su *mate breakend* `X`, ubicado en la posición 421682. Las líneas (3) y (4) explican el mismo proceso pero para el extremo derecho de la segmentación.

2.2.6. Referenciación de variantes en individuos

Dentro de cada línea con la declaración de una variante, le siguen los valores que indican si un individuo determinado posee esa variante en su respectivo alelo o no. La sintaxis es $v_1|v_2|\ldots|v_k$, donde $v_\text{i} \in \{0,\ldots,n\} \cup \{.\}$, con n la cantidad de variantes declaradas en la línea, indexando en 1, 0 significa que no aplica la variante y "." que desconocemos la variante que aplica. Además la posición de cada v_i indica el i-ésimo alelo del cromosoma `CHROM` del individuo al que le corresponde la variante. En humanos, por ejemplo, hay normalmente $k = 2$ alelos por cromosoma, salvo en el cromosoma Y y otras excepciones.

Cuando el separador de los v_i es "|" significa que los alelos estań en fase, es decir, estamos seguros de que la variante v_i corresponde al i-ésimo alelo; cuando es "/" indica que están

desfasados, o sea sabemos que la variante v_i ocurre, pero no sabemos en qué alelo. También, aparte de no saber en qué alelo ocurre, puede suceder que sepamos qué alelo es, pero desconocemos la variante que aplica, en ese caso v_i es un ".". Notemos que esta situación induce cierto grado de ambigüedad.

Por ejemplo, observemos las siguientes líneas que ilustran la interpretación de las columnas de individuos:

```
#CHROM   POS ID REF ALT    QUAL FILTER INFO INDV_1 INDV_2 INDV_3
 2       20  .  A   X,Y,Z  .    PASS   .    0|1    3|.    0/2
```

Primero, notemos que hay 3 variantes, `X`, `Y` y `Z`. Luego, para el individuo `INDV_1` en su primer alelo no ocurren variantes, y para el segundo ocurre la primera variante, `X`; el individuo `INDV_2` en su primer alelo ocurre la tercera variante, `Z`, pero para el segundo alelo desconocemos qué variante ocurrió; finalmente para el individuo `INDV_3` sabemos que en uno de sus alelos no hay variante y en el otro ocurre la segunda variante, `Y`, pero desconocemos qué alelos ocurre.

2.3. Relative Lempel-Ziv

Antes de entrar a definir el autoíndice con el que se trabajará, se ilustra el funcionamiento de dos estructuras que sustentan el formato de compresión RLZ [20]. Luego, se introduce el autoíndice RLZ [28].

2.3.1. Factorización de Lempel-Ziv

Dado dos strings $\mathcal{S}$ y $\mathcal{R}$, la factorización de Lempel-Ziv (o parsing) de $\mathcal{S}$ relativo a $\mathcal{R}$, denotado por $LZ(\mathcal{S}|\mathcal{R})$, corresponde a la factorización de $\mathcal{S} = w_0 w_1 w_2 ... w_z$ donde w_0 es el string vacío y para cada i > 0, w_i puede corresponder a dos casos: (a) una letra que no ocurre en $\mathcal{R}$; o bien (b) el prefijo más largo de $\mathcal{S}[|w_0 ... w_{i-1}| + 1..|\mathcal{S}|]$ que ocurre como un substring en $\mathcal{R}$.

Por ejemplo, si $\mathcal{R} = abaababa$ y $\mathcal{S} = aabacaab$, entonces en $LZ(\mathcal{S}|\mathcal{R})$ tenemos $w_1 = aaba$, $w_2 = c$ y $w_3 = aab$. Ahora, es conveniente no representar estos factores como strings, ya que básicamente recreamos $\mathcal{S}$ y no podemos recuperar la factorización, y en particular no estaríamos reduciendo espacio. Para ello, representamos cada factor como $w_i = (p_i, l_i)$, donde p_i es la posición dentro de $\mathcal{R}$ donde inicia w_i y l_i el largo del factor; si w_i es del tipo (a), el par sería la letra acompañada de un 0. De esta forma, el ejemplo anterior se representaría como $LZ(\mathcal{S}|\mathcal{R}) = (3, 4)(c, 0)(3, 3)$.

Por simplicidad, para evitar factores de tipo (a), a $\mathcal{R}$ podemos añadirle a lo sumo $\sigma - 1$ caracteres que no aparecen en $\mathcal{R}$ originalmente, con σ el tamaño del alfabeto.

2.3.2. Sets comprimidos de enteros

Dado un set S de m enteros sobre un universo $[1..u]$, esta estructura soporta las operaciones: $rank(S, \mathrm{i})$, retornando la cantidad de enteros en S menores o iguales que i; y $select(S, \mathrm{i})$, retornando el valor del i-ésimo elemento en S, donde los elementos se encuentran ordenados crecientemente. Esta estructura utiliza espacio $m \log \frac{u}{m} + O(m)$ bits, provee *select* en tiempo constante y $rank$ en $O(\log \frac{u}{m})$ [29].

2.3.3. Formato de compresión RLZ

En base a los conceptos previos, podemos definir en qué consiste el formato. Dada una colección $\mathcal{S}$ de t secuencias $\mathcal{S}_k \in \mathcal{S}$ tales que $|\mathcal{S}_k| = n$ para $1 \leq k \leq t$ y $\Sigma_{k=1}^{t} |\mathcal{S}_k| = t \cdot n = N$, donde definiendo $\mathcal{R} = S_1$ tenemos que:

$$Z^{\mathrm{i}} = LZ(\mathcal{S}_{\mathrm{i}}|\mathcal{R}) = (f_1, l_1), (f_2, l_2), ..., (f_{z_{\mathrm{i}}}, l_{z_{\mathrm{i}}}),$$

Donde z_{i} corresponde a la cantidad de factores que representa a Z^{i}. De esta forma, la unión de $\mathcal{R}$ y la colección de $\{Z^{\mathrm{i}}\}_{\mathrm{i}=2}^{t}$ representa $\mathcal{S}$. Así, para $2 \leq \mathrm{i} \leq t$ y $z = \Sigma_{\mathrm{i}=2}^{r} z_{\mathrm{i}}$, podemos almacenar $\mathcal{S}$ en a lo sumo $n \log \sigma + z \log n + z \log \frac{N}{z} + O(z)$ bits tal que cualquier substring de la forma $\mathcal{S}_k[s..\mathrm{e}]$ se puede retornar en tiempo $O(\mathrm{e} - s + \log \frac{N}{z})$. Esto es posible gracias a la forma en la que almacenamos las factorizaciones de cada $\mathcal{S}_{\mathrm{i}}$. Concretamente, guardamos la referencia $\mathcal{R}$ en $n \log \sigma$ bits, de forma plana, y almacenamos la colección de Z^{i} en dos partes, descritas a continuación.

Primero, las componentes de posición de cada factor son almacenadas en una tabla $F[1..z]$, tomando $z \log n$ bits. Tengamos en cuenta que F es la simple concatenación de los bits que codifican cada $f_1, ..., f_{z_{\mathrm{i}}}$. En virtud de esta concatenación, por simplicidad, para hablar de cada factor $(f_{z_{\mathrm{i}}}, l_{z_{\mathrm{i}}})$ lo referenciaremos en base a (f_j, l_j), donde $j \in [1..z]$. Entonces, como cada f_j es de $\log n$ bits, tenemos acceso a cada $f_j = F[j]$ en tiempo $O(1)$.

Luego, las componentes de largo de cada factor son almacenadas en un set comprimido de enteros L, donde cada elemento es de la forma $L[j] = \sum_{k=1}^{j} l_k$, para todo $j \in 1..z$, es decir, almacenamos los largos acumulados de la factorización, que además representan directamente las posiciones donde empiezan cada uno de los factores. De esta forma, utilizamos $z \log \frac{N}{z} + O(z)$ bits de espacio para almacenar L.

De esta forma, por medio de la consulta *select*, L nos permite tener acceso a cada f_j por medio de $F[j]$; y el largo del j-ésimo factor, l_j, es la diferencia entre el largo acumulado hasta el valor $j+1$ y j, es decir, $select(L, j+1) - select(L, j)$, para $j \in 1..z-1$. Además, podemos saber el factor en que se encuentra cualquier $\mathcal{S}[j]$ consultando $rank(L, j)$.

2.3.4. Autoíndice a utilizar

En base al formato RLZ, se sustenta el auto-índice RLZ [28], el cual corresponde a una versión simplificada de Fast Relative Lempel-Ziv (FRLZ) [9]. El FRLZ usa considerablemente

más espacio que el índice RLZ, a cambio de reducciones modestas en la complejidad de peor caso. Vale remarcar que el FRLZ nunca se ha implementado, mientras que el RLZ sí.

RLZ es capaz de buscar eficientemente las ocurrencias de un patrón $P[1..m]$ dada la caracterización de las ocurrencias y estructuras diseñadas para su identificación. La caracterización de estas ocurrencias se hace vía 3 procesos: recuperar aquellas dentro de la referencia $\mathcal{R}$, luego dentro de un factor del *parsing* de RLZ, y finalmente cruzando factores del mismo *parsing*.

Para la primera etapa, buscamos todas aquellas ocurrencias de P en la referencia, por medio de FM-Index sobre $\mathcal{R}[1..n]$. Notemos que FM-Index [14] [2] es un índice que se construye sobre $\mathcal{R}[1..n]$ sobre un alfabeto $[1..\sigma]$, que nos permite encontrar todas las *occ* ocurrencias de cualquier patrón $P[1..m]$ en $\mathcal{R}$ en tiempo $O(m + occ \log n)$ y retornar cualquier substring $\mathcal{R}[\mathrm{i}..j]$ en tiempo $O(\log n + j - \mathrm{i})$. Vale destacar que ambas complejidades van multiplicadas por $\log \sigma$, mas no se incluye en estas, pues en nuestro caso $\sigma = 5$, lo que transforma este factor en una constante en la práctica.

Luego, en base a las ocurrencias $\mathcal{R}[\mathrm{i}..\mathrm{i} + m - 1]$ reportadas en el paso anterior, buscamos aquellos substrigns $\mathcal{S}[p_u..p_u + l_u - 1] = \mathcal{R}[t_u..t_u + l_u - 1]$ en el *parsing* de RLZ donde esta fuente cubra el factor $\mathcal{R}[\mathrm{i}..\mathrm{i} + m - 1]$, es decir, $[\mathrm{i}..\mathrm{i} + m - 1] \subseteq [t_u..t_u + l_u - 1]$. Entonces se reporta una ocurrencia que inicia en. Gracias al uso de estructuras compactas para permutaciones [25] y RMQs (*Range Maximum Queries*) [15], este proceso se realiza en tiempo $O(occ)$.

Finalmente, se recuperan las ocurrencias que cruzan factores. Para ello, por medio de un wavelet-tree [26] y bit vectors [5], se almacenan en una grilla todos aquellos límites entre factores. De esta forma, por cada partición $P[1..\mathrm{i}]$ y $P[\mathrm{i} + 1..m]$, observando los límites, buscamos aquellos sufijos de un factor que unidos a un prefijo del siguiente correspondan al patrón buscado. De esta forma, en $O(z \log z)$ bits se pueden encontrar las *occ* ocurrencias entre factores en tiempo $O((m + occ) \log z)$.

De esta forma, el tiempo total de búsqueda de *occ* ocurrencias de un patrón P de largo m es $O((m + occ) \log N)$, utilizando espacio $n \log \sigma + o(n \log \sigma) + O(n + z \log N)$ bits.

2.4. Trabajo relacionado

Actualmente, aquellos sistemas que trabajan con VCF tienen que recrear la colección genómica completa a partir de este archivo, para así poder ejecutar los análisis requeridos, como filtrar variantes específicas, comparar archivos, resumir variantes, etc.

Como se enunció en la introducción, el problema aquí es que para acceder al genoma necesitamos descomprimir VCF, donde para analizar los datos se requiere el texto plano, cuyo tamaño es del orden de los giga o terabytes, órdenes de magnitud mayor que el archivo VCF original. En virtud de esto es que nacen dos formas, no necesariamente disjuntas, para resolver esto.

2.4.1. Reducciones de tamaño

Existen soluciones paliativas para contrarrestar el tamaño de estos archivos, que es transferir estos documentos VCF bajo su versión comprimida en GZIP (GZ), más un archivo de índices (TBI), es decir, comprimimos dos veces. Recordemos que GZIP también se sustenta en Lempel-Ziv (LZ), al igual que RLZ, ofreciéndonos un ratio de compresión del 90 % en promedio para este tipo de archivos.

Otra opción es utilizar p7zip en vez de VCF, el cual consigue una relación de compresión de $100:1$, siendo 10 veces mejor que VCF y RLZ, mas no es capaz de brindar algo más allá de un buen almacenamiento. VCF pese a comprimir con una tasa de $4:1$, nos brinda un formato humanamente legible, que permite incluso extraer genomas completos individuales, lo que no es posible con p7zip. Es por esto que nos enfrentamos a un trade-off entre espacio y accesibilidad, el cual ha ganado VCF a los ojos de la comunidad.

Un compresor más específico para genomas es genozip [21], el cual consigue una relación de compresión de $250:1$, valor que puede mejorar en virtud de parámetros de optimización disponibles al momento de comprimir el archivo.

Notemos que nuevamente las propuestas asociadas a reducción de espacio se limitan únicamente a esta funcionalidad, que si bien consiguen excelentes relaciones de compresión, como hemos ilustrado hasta ahora, son formatos no humanamente legibles, que no soportan consultas sobre el genoma sin descomprimir. En esencia, son formatos solo con fines de almacenamiento.

2.4.2. Eficiencia en tiempos de consulta

Considerando el problema del espacio como un tema aparte, otros esfuerzos van por el lado de hacer consultas sobre el documento lo más eficientemente posible. Esta arista ha sido desarrollada principalmente por aquellos software que procesan este archivo, como VCFtools [6] y BCFtools [7], que corresponden a software con interfaz integrada.

Estos programas han optimizado los procesos por medio de la computación paralela y algoritmos probabilísticos tanto para identificar variantes como para su compresión, donde el costo del almacenamiento y procesamiento va a cuenta de la máquina que ejecute estos procesos.

Por otro lado, existen librerías ya sea en R o en Python, como vcfR[3] y Glow[4], respectivamente, que tienen un enfoque análogo al de los software descritos. Donde este último se sustenta en la computación y almacenamiento distribuido por medio del uso de Apache Spark y Delta Lake, arquitecturas conocidas por su trabajo orientado al manejo de datos masivos de forma distribuida.

Nuevamente, el enfoque para resolver el problema de los tiempos de ejecución ante la masividad de los datos reside en diseñar mejores procesos, mas allá de aprovechar la misma

[3] https://cran.r-project.org/web/packages/vcfR/vignettes/intro_to_vcfR.html
[4] https://glow.readthedocs.io/en/latest/introduction.html

naturaleza de los documentos a tratar.

Por otro lado, dentro de las técnicas computacionales para indexar textos muy repetitivos [27] tenemos exponentes como las gramáticas de libre contexto, donde si identificamos una gramática que genere únicamente $\mathcal{S}$, podemos obtener sus substring en tiempos y espacios análogos a los de RLZ. Lo mismo se puede tener con otras técnicas como los *Collage Systems* y la Transformación de Burrows-Wheeler (BTW). Sin embargo, aún teniendo mejores desempeños, sus procesos de construcción también residen en recibir el texto descomprimido y la lógica de sus construcciones son diferentes a la forma de trabajo de VCF, no así RLZ.

Capítulo 3

Diseño de la solución

En este capítulo explicaremos algunas consideraciones generales que se deben tener en mente para el entendimiento del alcance e implementación de este trabajo, para luego explicar el planteamiento del problema a resolver, lo que permitirá ilustrar los requerimientos a los que responden los procesos descritos en el Capítulo 4.

3.1. Consideraciones generales

A continuación veremos el acotamiento de este trabajo en el marco de VCF, es decir qué atributos y variantes serán soportadas en la conversión, y luego qué librerías externas fueron necesarias para el funcionamiento del módulo de conversión.

3.1.1. Alcance dentro de VCF

Como vimos en la Sección 2.2, VCF soporta todo tipo de variantes, lo que permite que este formato sea sumamente versátil. Sin embargo, en la misma medida no hay muchos estándares que caractericen los distintos usos de este formato. Por ejemplo, los archivos VCF se pueden utilizar para analizar muestras de diferente naturaleza, como lo pueden ser tumores o secciones de los órganos.

Para efectos de esta memoria, nos acotaremos a aquellos **VCF utilizados en los estudios de genética de poblaciones**, en particular la restricción es que sean documentos que trabajen con 23 cromosomas, los cuales deben ser diploides[1]. Si bien el módulo implementado soporta una cantidad arbitraria de cromosomas y ploidías, no es correcto asumir a priori que el proceso de conversión respeta la interpretación de cada caso, y dado el tiempo disponible para el desarrollo de este trabajo, preferimos acotarnos a este espectro. Por este mismo motivo, **solo trabajaremos sobre los cromosomas autosómicos**, osea los cromosomas sexuales, X e Y, quedarán fuera de este trabajo, pues su interpretación no es la misma que

[1]Dos alelos por cromosoma

se realiza con los otros 22.

Además, dado que el alfabeto soportado por la implementación de RLZ a utilizar es solo de las 4 bases nitrogenadas, debemos limitar los **VCF procesables a aquellos que no hagan uso de la letra N** en el contexto de las cadenas, y con ello las referencias utilizadas. Como explicamos en la Sección 2.2.2, la letra N tiene un comportamiento neutro, y reemplazarlo arbitrariamente con cualquier otra base no sería técnicamente correcto. Para el alcance de esta memoria y la experimentación sobre el módulo, sin embargo, reemplazaremos todas las N en las cadenas con la letra A.

Finalmente, para esta primera versión de la conversión solo soportaremos las variantes descritas en la Sección 2.2.4, las que ocurren principalmente en los VCF utilizados en los estudios mencionados.

3.1.2. Librerías a utilizar

Con respecto a la implementación de este proceso, notemos que el código de RLZ se sustenta en Succinct Data Structure Library (SDSL) [17], una librería diseñada en C++11 que implementa estructuras de datos sucintas. Entonces, en virtud de esta base, es que toda implementación que se llegue a realizar, trabajará con los mismos recursos.

Por otro lado, considerando que se requerirá ordenar archivos que dada su masividad no conviene trabajar en RAM, como explicamos en la Sección 1.3, haremos uso de la librería STXXL [8]. Es una librería diseñada también en C++11, la cual implementa Standard Template Library (STL) [30] para memoria externa, lo que nos permite desarrollar algoritmos que deban ejecutarse sobre grandes volúmenes de datos, que solo entran en disco.

3.2. Planteamiento del problema

Como hemos mencionado, el problema que busca resolver esta memoria es convertir el formato VCF a RLZ, pero para ello primero necesitamos entender brevemente cómo es que se construye RLZ.

RLZ en su forma actual espera recibir el texto plano, sin saltos de línea, del string que simbolizará la referencia ($\mathcal{R}$); sumado al texto plano que va a ser reconstruido en base a la referencia ($\mathcal{S}$), a este proceso lo llamaremos factorización. La factorización busca detectar los substrings más largos en $\mathcal{S}$ que también se encuentren en $\mathcal{R}$, de tal forma que generemos $LZ(\mathcal{S}|\mathcal{R})$.

Entonces en primer lugar debemos transformar la referencia de VCF en formato FASTA a la estructura necesaria para obtener $\mathcal{R}$, tarea cuya implementación se encuentra descrita en la Sección 4.1, el procesamiento de la referencia.

Por otro lado, no es directo el obtener $\mathcal{S}$ desde VCF. Recordemos que RLZ trata a $\mathcal{S}$ como una única cadena, y VCF por cada línea de registro reporta edits por cada alelo en el

cromosoma correspondiente, para cada individuo. Por lo que no es conveniente leer todo estos registros y después separar todos y cada uno de los edits por alelo, cromosoma e individuo. Sin embargo, esta misma condición de VCF es un punto a favor, la estructura del texto que codifica VCF es fija, independiente de la cantidad y tipo de variantes, entonces si bien no es óptimo separalos, sí podemos ordenarlos.

En términos computacionales, para un individuo i cada uno de sus 22 cromosomas $\{C_{\mathrm{i},j}\}_{j=1}^{22}$ pueden ser representados como la concatenación de sus alelos, es decir $C_{\mathrm{i},j} = A^1_{\mathrm{i},j}A^2_{\mathrm{i},j}$. Luego, para n individuos sus genomas $\{G_\mathrm{i}\}_{\mathrm{i}=1}^{n}$ se pueden representar como la concatenación de sus cromosomas, que a su vez es la concatenación de los alelos correspondientes, de esta forma obtenemos la siguiente estructura:

$$\begin{aligned} G_\mathrm{i} &= C_{\mathrm{i},1}C_{\mathrm{i},2}\cdots C_{\mathrm{i},21}C_{\mathrm{i},22} \\ &= A^1_{\mathrm{i},1}A^2_{\mathrm{i},1}A^1_{\mathrm{i},2}A^2_{\mathrm{i},2}\cdots A^1_{\mathrm{i},21}A^2_{\mathrm{i},21}A^1_{\mathrm{i},22}A^2_{\mathrm{i},22} \end{aligned}$$

Al conjunto de todos los alelos de la forma $A^k_{\mathrm{i}j}$ lo definiremos como $\mathcal{A}$. Así, si concatenamos el genoma de todos los individuos bajo esta estructura, tenemos que:

$$\mathcal{S} = G_1G_2\cdots G_{n-1}G_n$$

Ahora, no esperamos generar el $\mathcal{S}$ propuesto en texto plano, el desafío reside en directamente generar $LZ(\mathcal{S}|\mathcal{R})$, y para ello se hacen necesarias las tareas de recuperar todos los edits desde VCF transformándolos a *frases* (Sección 4.2), luego ordenar estas *frases* para que respeten el orden de escritura que usaremos para $\mathcal{S}$ (Sección 4.3) y finalmente estas *frases* serán interpretadas para generar los factores de $LZ(\mathcal{S}|\mathcal{R})$ (Sección 4.4).

Además, si buscamos ejecutar búsquedas sobre RLZ, pero obtener valores consistentes con la interpretación de VCF, requerimos generar una estructura extra que almacene las posiciones de inicio de cada $A^k_{\mathrm{i}j}$, de tal forma que aprovechando la estructura de $\mathcal{S}$ y en base a esas posiciones seamos capaces de calcular donde pertenece originalmente cada ocurrencia (Sección 4.5).

Capítulo 4

Proceso de conversión e interpretación

En este capítulo explicaremos a detalle la implementación del proceso de conversión. Partiremos por explicar cómo es que procesamos la referencia, y con ello la recuperación de valores necesarios para los pasos posteriores. Luego veremos cómo es que cada línea de registro en VCF es transformada en una estructura a la que denominamos *frases*. Posteriormente ilustraremos cómo unificar los genomas, motivo por el cual es importante ordenar las *frases* generadas en base a las reglas de unificación. Después explicaremos cómo es que las *frases* son transformadas en factores, para dar pie a la construcción de RLZ junto con una estructura complementaria. Finalmente cómo es que se interpretarán estas estructuras, con el fin de retornar resultados de búsqueda que emulen los valores de VCF.

Notar que cada subsección iniciará explicando el procedimiento de cada etapa, para luego concluir en cómo la implementación cumple con el comportamiento descrito.

4.1. Procesamiento de la Referencia

Como explicamos en la Sección 2.2.2, la referencia utilizada por VCF tiene una estructura que no es aceptada por RLZ en su construcción, este proceso requiere la concatenación de las cadenas que representan a cada referencia. Por ejemplo, para el mismo ejemplo de la sección mencionada, el formato de referencia válido en este caso es la concatenación de las 4 líneas de texto plano de la cadena.

Entonces, el procedimiento para recuperar $\mathcal{R}$ consta únicamente de concatenar las líneas que representan a la cadena. Sin embargo, para el paso siguiente, recuperar *frases* a partir de edits, necesitamos almacenar algunas caracterizaciones de la referencia. En virtud de esto, por cada cadena concatenada de la referencia ($\mathcal{R}_\text{i}$) almacenamos su posición de inicio relativa en la concatenación ($\mathcal{R}_\text{i} \rightarrow P_\text{i}$), junto a su ID ($\mathcal{R}_\text{i} \rightarrow ID$), que es el número del cromosoma en este caso, y la cantidad de bases que esta posee ($\mathcal{R}_\text{i} \rightarrow N_b$), todo esto dentro de una estructura a la que llamaremos *MetaReference*.

Por otra parte, notemos que RLZ no soporta declarar factores que contengan texto que no viva en la referencia, por ende, las variantes simples sobre cadenas cortas no pueden

ser codificadas en ese formato. No obstante, el largo de las cadenas tiende a ser alrededor de 4, entonces aprovechando esta situación, al final de la referencia concatenaremos todas las permutaciones de largo 1 a 4 sobre el alfabeto $ACGT$, donde para cada permutación generaremos también una estructura *MetaReference* con su información, cuyo $\mathcal{R}_{\text{i}} \rightarrow ID$ será la permutación misma.

Notemos que es más conveniente insertar este conjunto de permutaciones a buscarlas en la misma referencia para capturar los *MetaReference* respectivos, pues si bien es computacionalmente viable, preferimos pagar espacio a procesamiento, espacio que es insignificante con respecto al tamaño de la referencia. La referencia utilizada en nuestros estudios contiene aproximadamente 2,800 millones de caracteres, y el string a concatenar con todas las permutaciones es solo de 340 caracteres.

Todo este proceso descrito es ejecutado por una clase implementada en Python llamada *ReferenceProcessor*, de esta forma el objetivo de la clase es generar un diccionario ordenado con todas las estructuras *MetaReference* obtenidas (*DictMetaReference*), cuya llave es el $R_{\text{i}} \rightarrow ID$ correspondiente, y crear un archivo de texto que contenga la cadena completa del genoma presente en la referencia.

4.2. Transformación de edits a frases

4.2.1. Recuperación de individuos

Vale destacar que gracias a la construcción de *DictMetaReference* en el paso anterior, hace que la información contenida en las líneas de meta-información de VCF no sea de mayor aporte; en adición a que de todas formas debemos recorrer el documento de la referencia, y escoger los valores arrojados por ese proceso entrega resultados más confiables a los que se encuentran declarados la meta-información. Por ejemplo que la línea de meta-información `contig` no tiene como campo obligatorio el largo de la cadena. De esta forma, podemos descartar la meta-información y pasar directamente a la línea de header.

Sabemos que la estructura del header siempre contendrá el nombre de los 9 campos obligatorios y luego el identificador de todos los individuos a ser representados en los documentos, entonces separando esta línea en una lista, en base a los tabs como separador, recuperamos desde el décimo valor en adelante y así obtenemos la lista de todos los ID, con ello la cantidad de individuos a esperar por registro.

Además, en base a esta lista, virtualmente se le asigna a cada individuo un ID de uso interno, que coincide con su posición dentro de la lista. De esta forma, podemos almacenar los identificadores reales en un archivo de texto, que solo serán consultados para el reporte de ocurrencias, proceso que se encuentra descrito en la Sección 4.5, con ello para el resto de la construcción solo nos basaremos en sus ID.

4.2.2. Frases y su construcción

Después de procesar el header, inicia la etapa de análisis de los registros, pero para entender este proceso, primero ilustraremos de qué consta una *frase*.

En primer lugar, una *frase* (p) corresponderá a una estructura que posee 7 valores:

1. $p \rightarrow indv$: ID numérico de uso interno, que representa el individuo al que pertenece esta *frase*.
2. $p \rightarrow chrom$: ID numérico de uso interno, que representa el cromosoma al que pertenece esta *frase*.
3. $p \rightarrow alele$: Número de alelo al que pertenece esta *frase*, al trabajar sobre cromosomas diploides este valor es 0 o 1.
4. $p \rightarrow pos$: Posición relativa a $\mathcal{R}$ donde ocurre el edit.
5. $p \rightarrow len$: Largo del fragmento que inicia en $p \rightarrow pos$ y será reemplazado por el edit.
6. $p \rightarrow posE$: Posición relativa a $\mathcal{R}$ donde se encuentra la secuencia correspondiente al edit.
7. $p \rightarrow lenE$: Largo del edit que inicia en $p \rightarrow posE$.

En particular, a los primeros 3 valores los caracterizaremos como valores de identificación y los otros 4 restantes como valores de reconstrucción. Por simplicidad, para futuros ejemplos cuando busquemos ilustrar todos los valores de una frase, será por medio de una tupla de 7 valores de la forma $(p \rightarrow indv, p \rightarrow chrom, p \rightarrow alele, p \rightarrow pos, p \rightarrow len, p \rightarrow posE, p \rightarrow lenE)$.

Gracias a esta estructura, podemos representar todas las variantes simples de VCF bajo un mismo formato, solo cambia la forma en la que se calculan los valores de la *frase*. Esto nos permite tener una implementación más robusta y extensible en el tiempo, pues soportar más variantes solo radica en incluir nuevas reglas de construcción de *frases*.

Para ilustrar cómo es que se construye una frase a partir de un registro, veamos el siguiente ejemplo. Limitándonos al caso de estudio, sean los registros:

```
     #CHROM   POS ID REF ALT       QUAL FILTER INFO   INDV_1 ... INDV_I
(0)   c       p   .  r   a1,..,aA  .    PASS   .      i|j    ... x|y
(1)   1       10  .  ACC A, ACCTT  .    PASS   .      1|2
(2)   1       20  .  A   <DEL>     .    PASS   END=25 1|0
```

Sea $V = |\{\texttt{a1,..,aV}\}|$, la cantidad de variantes. `I` es el número de individuos, donde los valores bajo el identificador de cada individuo corresponden a la declaración de variantes de sus alelos respectivos, con `i`, `j`, `x` e `y` $\in [0..V]$, recordemos que 0 simboliza que no aplica ninguna variante.

En primer lugar, debido a que los individuos declararán sobre el mismo conjunto de variantes dentro del registro, nos conviene procesar primero la información necesaria para completar las variables de reconstrucción de las *frases*, información que se encuentra en las columnas `CHROM`, `POS`, `REF` y `ALT`, también `INFO` para variantes simples en cadenas largas. En base a esto, se construyen *frases* template que serán almacenadas en una lista caché C, cuya posición en la lista es correlativa al orden de declaración de la variante en el registro, por ello siempre $|C| = V$ y su acceso es de la forma C[i] con i $\in [1..V]$. De esta forma, cuando se procesen las declaraciones de variantes de cada individuo, se selecciona la *frase* template correspondiente, completándose solo los valores de identificación pendientes. Finalmente, las *frases* completadas se almacenan en una lista llamada $\mathcal{F}$.

Para la construcción de las *frases* tenemos dos familias de reglas de construcción, las que se usan sobre variantes simples en cadenas cortas (Familia *VSCC*), y las que se usan sobre estas mismas variantes en cadenas largas (Familia *VSCL*). Un ejemplo concreto que ilustra la familia *VSCC* es el procesamiento del registro (1), descrito a continuación.

Primero, decidimos si el registro será procesado, esto sucede solo si `FILTER=PASS`, de lo contrario es automáticamente descartado y proseguimos con el siguiente registro. De cumplir con el filtro, procedemos a identificar los valores que tendrán en común todas las *frases* template más allá de la variante que representen, que son: el cromosoma en el que estamos trabajando, $p \rightarrow chrom = 1$; posición de ocurrencia, con $p \rightarrow pos = 10$ y el largo del fragmento de la referencia a reemplazar, con $p \rightarrow len = |\texttt{ACC}| = 3$.

Con los valores comunes completados, luego debemos identificar cuántas variantes son declaradas en `ALT`, con el fin de completar los valores de reconstrucción restantes. En este caso tenemos una eliminación (`A`) y una inserción (`ACCTT`), con ello, tenemos que generar dos conjuntos de *frases* template.

Para la *frase* de eliminación (p_{e}), vemos que la sección de $\mathcal{R}$ declarada será reemplazada por la cadena `A`, entonces buscaremos en *DictMetaReference* un $\mathcal{R}_{\text{i}}$ tal que $\mathcal{R}_{\text{i}} \rightarrow ID = \texttt{A}$, con el fin de encontrar la posición de `A` en la referencia, por medio de $\mathcal{R}_{\text{i}} \rightarrow P_{\text{i}}$. Una vez es recuperado $\mathcal{R}_{\text{i}}$, al que definimos como $\mathcal{R}_{\texttt{A}}$ completamos p_{e} con $p_{\text{e}} \rightarrow posE = \mathcal{R}_{\texttt{A}} \rightarrow P_{\text{i}}$. Luego el largo del edit es $p_{\text{e}} \rightarrow lenE = \mathcal{R}_{\texttt{A}} \rightarrow N_b = 1$. Finalmente, obtenemos el siguiente resultado $p_{\text{e}} = (X, 1, X, 10, 3, \mathcal{R}_{\texttt{A}} \rightarrow P_{\text{i}}, 1)$, con X los valores de identificación a ser completados posteriormente. Ahora $C = \{p_{\text{e}}\}$.

Luego, para la *frase* de inserción, el procedimiento es análogo al de eliminación. No obstante, en esta variante el edit posee $|\texttt{ACCTT}| = 5$ caracteres, y nosotros definimos que para las frases soportaremos edits explícitos cuyos largos pueden ser hasta 4. Para estos casos, cuando el largo del edit es superior a 4, lo que haremos es construir una secuencia de *frases* template, donde por cada partición de largo 4 del edit generaremos una frase, y la ultima *frase* puede representar un edit de largo menor o igual a 4. En el marco de este ejemplo, requeriremos una *frase* para $ACCT$ ($p_{\text{i}1}$) y T ($p_{\text{i}2}$). Luego para cada *frase* repetimos el proceso descrito anteriormente, con $\mathcal{R}_{\texttt{ACCT}}$ *MetaReference* de `ACCT` y $\mathcal{R}_{\texttt{T}}$ *MetaReference* de `T`, teniendo así $p_{\text{i}1} = (X, 1, X, 10, 3, \mathcal{R}_{\texttt{ACCT}} \rightarrow P_{\text{i}}, 4)$ y $p_{\text{i}2} = (X, 1, X, 10, 3, \mathcal{R}_{\texttt{T}} \rightarrow P_{\text{i}}, 1)$. Con ello, $C = \{p_{\text{e}}, \{p_{\text{i}1}, p_{\text{i}2}\}\}$.

Con C construido, procedemos a asignar las *frases* correspondientes según la declaración

de variantes de cada individuo. Para el registro (1), el individuo `INDV_1` declara que su primer alelo codifica la primera variante, entonces su *frase* utilizará como template $C[1] = p_{\mathrm{e}}$, obteniéndose $\mathcal{F} = \{(1, 1, 1, 10, 3, \mathcal{R}_{\mathtt{A}} \rightarrow P_{\mathrm{i}}, 1)\}$; como el segundo alelo codifica la segunda variante, el template a utilizar es $C[2] = \{p_{\mathrm{i}1}, p_{\mathrm{i}2}\}$, obteniendo $\mathcal{F} = \{(1, 1, 1, 10, 3, \mathcal{R}_{\mathtt{A}} \rightarrow P_{\mathrm{i}}, 1), (1, 1, 2, 10, 3, \mathcal{R}_{\mathtt{ACCT}} \rightarrow P_{\mathrm{i}}, 4), (1, 1, 2, 10, 3, \mathcal{R}_{\mathtt{T}} \rightarrow P_{\mathrm{i}}, 1)\}$

Para cadenas más largas, como lo es el ejemplo del registro (2), este proceso tiene como principal diferencia, con respecto al proceso anterior, que cambia la forma en cómo calculamos $p \rightarrow len$ y $p \rightarrow lenE$. Para la eliminación `<DEL>`, al ser una variante extensa, en vez de preservar el $p \rightarrow len$ precalculado, lo definiremos como $p \rightarrow len = \mathtt{SVLEN} = \mathtt{END} - p \rightarrow pos$, con cual de las dos definiciones nos quedemos dependerá de qué valor se encuentre declarado en la columna `INFO` del registro. Además, como es una eliminación completa, $p \rightarrow posE = p \rightarrow lenE = 0$. Para el caso de la duplicación, ya sea simple (`<DUP>`) o variable (`<CNVi>`), como ya sabemos la posición en $\mathcal{R}$ del fragmento a ser duplicado, basta capturar $p \rightarrow lenE = \mathtt{SVLEN} = \mathtt{END} - p \rightarrow pos$ y generar una secuencia de esas *frases* tan larga como copias se soliciten.

Todo el proceso de transformación de edits a frases es llevado a cabo por una clase implementada en Python llamada *VCFParser*, que contiene como miembro a *ReferenceProcessor*. El objetivo de *VCFParser* es ejecutar todas las etapas descritas en esta Sección para finalmente $\mathcal{R}$ en texto plano y $\mathcal{F}$ en un archivo binario, donde al principio y final de este escribiremos una frase de inicio y final *dummies*, respectivamente, requeridos como límites para el proceso de transformación de *frases* a factores (Sección 4.4).

4.3. Ordenamiento de las frases

Con $\mathcal{F}$ almacenado en un archivo binario, podemos hacer uso del soporte de ordenamiento en disco de STXXL [8], donde debemos definir únicamente las reglas de ordenamiento.

En virtud de la estructura que usaremos para $\mathcal{S}$, descrita en la Sección 3.2, tenemos que ordenar nuestras *frases* de tal forma que queden agrupadas por individuo, luego por cromosoma, posteriormente por alelo y finalmente por posición de ocurrencia. Recordemos en que la creación de $\mathcal{F}$, por el orden en el que son descritos los edits, viene ordenado inicialmente por posición de ocurrencia.

Esta tarea es llevada a cabo por una clase implementada en C++11 llamada *VCFParsingSorter*, la que recibe el path del archivo binario de $\mathcal{F}$ y un archivo extra que nos describe cuántas *frases* hay contenidas en $\mathcal{F}$. Como miembro tiene una estructura llamada *phrase*, que es la implementación en C++11 de una *frase*, la que incluye la definición de los operadores booleanos de igualdad ($==$), menor que ($<$) y el operador de escritura ($\ll$), los que son requeridos para el proceso de ordenamiento de STXXL.

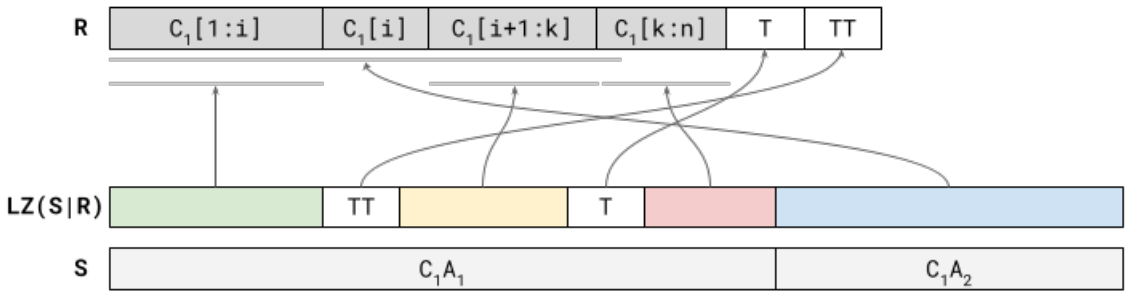

Figura 4.1: Ejemplo representa la relación que existe entre $\mathcal{R}$, $LZ(\mathcal{S}|\mathcal{R})$ y $\mathcal{S}$ para el caso de un solo cromosoma, un individuo y dos edits.

4.4. De frases a factores

Una vez $\mathcal{F}$ está ordenado, es seguro leerlo *frase* por *frase* para recrear $LZ(\mathcal{S}|\mathcal{R})$.

Ahora, notemos que las *frases* de $\mathcal{F}$ representan únicamente los cambios que hay que aplicar sobre la sección de $\mathcal{R}$ que corresponda para la construcción de A^k_{ij}, en cambio los factores describen qué secciones de $\mathcal{R}$ debemos unir para generar $\mathcal{S}$. Por ello, la conversión de *frases* a factores consiste en el proceso descrito a continuación.

En primer lugar, para esta explicación observaremos la Figura 4.1, la que representa la relación que existe entre $\mathcal{R}$, $LZ(\mathcal{S}|\mathcal{R})$ y $\mathcal{S}$ para un caso de un solo cromosoma, un individuo y dos edits. Aquellos fragmentos en blanco en $LZ(\mathcal{S}|\mathcal{R})$ corresponden a los edits conocidos gracias a las *frases*, es decir, en la construcción de $LZ(\mathcal{S}|\mathcal{R})$ hubiésemos tenido un $\mathcal{F}$ del tipo $\mathcal{F} = \{(1, 1, 1, \mathrm{i}, 1, n+2, 2), (1, 1, 1, \mathrm{i}, 1, n+1, 1)\}$.

Notemos que entre las secciones que son informadas por las *frases* debemos completar con aquellas secciones que $\mathcal{R}$ que sí se mantienen. Para identificar cómo se deben completar estas secciones caracterizamos 5 tipos de factores:

1. **Factor Edit (FE)**: Factor que es recuperado de una *frase* p, que es de la forma $(p \to posE, p \to lenE)$. Para los casos en los que $p \to posE = p \to lenE = 0$, no se generará un FE (Fragmento blanco de la Figura 4.1 en $LZ(\mathcal{S}|\mathcal{R})$).
2. **Factor de Inicio (FIni)**: Ocurre cuando se presenta la primera frase p de un A^k_{ij} cuyo edit no ocurre en el inicio de $\mathcal{R}_\mathrm{i}$. Es de la forma $(\mathcal{R}_\mathrm{i} \to P_\mathrm{i}, l)$, con $l = p \to pos$, y $l = p \to pos$ (Fragmento verde de la Figura 4.1 en $LZ(\mathcal{S}|\mathcal{R})$).
3. **Factor Intermedio (FInter)**: Este factor se utiliza para completar la región que se conserva entre dos *frases* p_1 y p_2 cuyos edits no ocurren en la misma posición y viven en el mismo A^k_{ij}. Que es de la forma (f, l), donde $f = \mathcal{R}_\mathrm{i} \to P_\mathrm{i} + p_1 \to pos + (p_1 \to len - 1)$ y $l = p_2 \to pos - (p_1 \to pos + 1) + (p_1 \to len - 1)$ (Fragmento amarillo de la Figura 4.1 en $LZ(\mathcal{S}|\mathcal{R})$).
4. **Factor de Cierre (FEnd)**: Se usa cuando el edit de la última frase p de A^k_{ij} no ocurre

al final de $\mathcal{R}_\text{i}$. Es de la forma (f, l), con $f = p \to pos + p \to len - 1$ y $l = \mathcal{R}_\text{i} \to N_b - ((p \to pos + 1) + (p \to len - 1))$ (Fragmento rojo de la Figura 4.1 en $LZ(\mathcal{S}|\mathcal{R})$).

5. **Factor Completo (FFull)**: Se utiliza cuando los edits de dos *frases* p_1 y p_2 no ocurren en el mismo $A^k_{\text{i}j}$, entonces creamos un FEnd para p_1, luego por cada alelo, cromosoma o individuo de distancia que tengan generamos FFull de la forma (f, l) con $f = \mathcal{R}_\text{i} \to P_\text{i}$ y $l = \mathcal{R}_\text{i} \to N_b$, hasta que alcanzamos el $A^k_{\text{i}j}$ en el que se encuentra el edit de p_2 y generamos un FIni para este (FFull es el fragmento azul de la Figura 4.1 en $LZ(\mathcal{S}|\mathcal{R})$).

En base a estas caracterizaciones, podemos iterar sobre $\mathcal{F}$ usando estas reglas de construcción de factores para generar $LZ(\mathcal{S}|\mathcal{R})$ completo sin nunca descomprimir. Además, por cada $A^k_{\text{i}j}$ representado en $LZ(\mathcal{S}|\mathcal{R})$ almacenaremos en un bit vector comprimido [29] su posición de inicio relativa en $\mathcal{S}$, $\mathcal{P}_\mathcal{A}$. Esta estructura será requerida para el proceso de recuperación de posiciones de RLZ a VCF (Sección 4.5).

El proceso descrito en esta sección es realizado por una clase implementada en C++11 llamada *VCFParsingInterpreter*, la que recibe como parámetro la ubicación de la carpeta en la que se encuentra el resultado de *VCFParser*, después de ser ordenado por *VCFParsingSorter*. El objetivo de esta clase es generar los recursos necesarios para construir RLZ y $\mathcal{P}_\mathcal{A}$, así una vez que los tiene listos pasa directamente a la construcción de las estructuras mencionadas.

4.5. De RLZ a posiciones VCF

Una vez RLZ es construido, cuando consultemos por las ocurrencias de $P[1..m]$, sus posiciones serán relativas a $\mathcal{S}$. Nuestra tarea es tomar estas posiciones y por cada una de ellas generar una 4-tupla de la forma $(Indv, Chrom, Alele, Pos)$. Para conseguir esto, aprovecharemos la estructura que tiene $\mathcal{S}$, descrita en la Sección 3.2, y en base a ello ejecutamos el siguiente procedimiento:

Para una posición $p_{RLZ} \in [1..N]$ buscaremos cuál es el $A^k_{\text{i}j}$ que inicia antes de esta posición con $a = rank_1(\mathcal{P}_\mathcal{A}, p_{RLZ})$, luego recuperamos en qué posición ocurre efectivamente con $pos_a = select_1(\mathcal{P}_\mathcal{A}, a)$ y la posición del siguiente $A^k_{\text{i}j}$ con $pos_{a+1} = select_1(\mathcal{P}_\mathcal{A}, a + 1)$.

Si el patrón P cruza pos_{a+1}, entonces se descarta. De lo contrario, hacemos la división entera de p_a sobre la cantidad de cromosomas por la ploidía, de esta forma obtenemos el índice de del individuo correspondiente, con el que podemos recuperar su identificador original y así se tiene $Indv$. Luego, calculamos la posición de inicio del genoma (G_i) de este individuo y se lo restamos a p_{RLZ}, el resultado (o) al dividirse por la cantidad de cromosomas multiplicada por ploidía nos entrega el cromosoma de G_i, valor que se asigna a $Chrom$. Análogamente, con la posición del cromosoma le restamos a o este valor, que al dividirlo por la ploidía nos entrega el alelo.

De esta forma, definiendo C como la cantidad de cromosomas, y Pl como la ploidía, podemos decir que esta conversión se puede calcular en tiempo $O(\log \frac{N}{I \cdot C \cdot Pl})$ por cada ocurrencia de $P[1..m]$. Notemos que para soportar la conversión solo debemos agregar $O(I \cdot C \cdot Pl \cdot (2 + \log \frac{N}{I \cdot C \cdot Pl}))$ bits de espacio, correspondiente a $\mathcal{P}_\mathcal{A}$.

Capítulo 5

Experimentación y Análisis

En el presente capítulo veremos la descripción y análisis del proceso de experimentación realizado, junto a la caracterización de las dificultades que se presentaron con el constructor de RLZ y cómo fueron solucionadas.

Primero veremos la experimentación realizada para la validación del módulo de conversión de colecciones genómicas, sin construir RLZ. Luego serán descritos los problemas que se presentaron con el constructor de RLZ, dando pie a la ilustración del nuevo sistema de construcción propuesto. Finalizando con los experimentos realizados para evaluar la conversión completa sobre archivos VCF para un solo cromosoma.

5.1. La conversión

5.1.1. Validación

Primero evaluaremos el **tiempo de conversión** del módulo sobre colecciones genómicas. Notemos que dada la masividad de los datos y la longitud de los procesos, el proceso de conversión se encuentra implementado para trabajar en 3 etapas separadas: PARSE corresponderá a las etapas descritas en las Secciones 4.1 y 4.2; SORT es la etapa descrita en la Sección 4.3, PHRASES es el proceso descrito en la Sección 4.4, pero que no considerará la construcción de RLZ, y PARTIAL RLZ corresponde a la etapa PHRASES pero incluye el tiempo de construcción de RLZ a excepción de la grilla de prefijos y sufijos. El motivo de la distinción entre PHRASES y PARTIAL RLZ se encuentra descrito en la Sección 5.2.1.

En virtud de lo descrito, caracterizaremos los tiempos de construcción sobre 6 conjuntos de datos, ilustrados en la Tabla 5.1. Estos conjuntos corresponden a un subconjunto de los VCFs publicados por el proyecto *1000Genomes*[1], el cual dispone de un VCF por cada uno de los cromosomas del genoma humano para 2500 individuos, junto al FASTA utilizado como referencia, el cual tiene 2,881,034,538 bases.

Posterior a estas cuatro etapas, también recuperaremos la cantidad de edits identificados,

Conjunto	**Individuos**	**VCF** (GB)	**Proporción**
S12	12	15	23 %
S24	24	19	14 %
S36	36	22	11 %
S48	48	26	10 %
S60	60	30	9 %
S72	72	33	8 %

Tabla 5.1: Conjunto de muestras a procesar por cada una de las 3 etapas de construcción. Todos los conjuntos se encuentran compuestos por los 22 cromosomas autosómicos, donde la referencia a utilizar pesa 3GB. La última columna representa el porcentaje de espacio que representa VCF con respecto a su versión con datos planos.

cuántos factores fueron generados, el largo del $\mathcal{S}$ correspondiente y el peso de los archivos binarios generados por PARSE y SORT.

5.1.2. Resultados y análisis

Los resultados de la Tabla 5.2 nos muestran los tiempos de conversión para cada conjunto, medido en minutos, desglosados por el tiempo de ejecución de cada una de las etapas. Luego en la Tabla 5.5 tenemos la evaluación parcial de los tiempos de construcción de RLZ sobre 3 conjuntos, recordemos que la grilla de prefijos y sufijos no fue construida en este experimento. Después tenemos la Tabla 5.3, donde para cada uno de los conjuntos tenemos la cantidad de edits y factores generados a partir de ellos, junto al largo de $\mathcal{S}$ y el tamaño en gigabytes (GB) de los archivos binarios que contienen la información generada por los 3 procesos de conversión.

Conjunto	**PARSE** (min)	**SORT** (min)	**PHRASES** (min)	**Total** (min)
S12	84	15	23	122
S24	133	25	26	154
S36	186	26	22	234
S48	228	27	30	289
S60	269	27	34	330
S72	299	28	38	365

Tabla 5.2: Tiempos de conversión para los conjuntos de prueba, medidos en minutos.

Como era esperado, los tiempos de conversión (Tabla 5.2) aumentan conforme incrementamos la cantidad de individuos representados, pues a mayor cantidad de individuos, mayor cantidad de edits y *frases* a procesar. No obstante, notemos que es un crecimiento marginalmente decreciente. Observando más detalladamente, los tiempos de PARSE aumentan a priori en un factor constante, pero SORT a partir de los 24 individuos empieza a converger y PHRASES crece de forma marginalmente decreciente.

Estos comportamientos se pueden explicar en base a que PARSE debe recorrer todos los

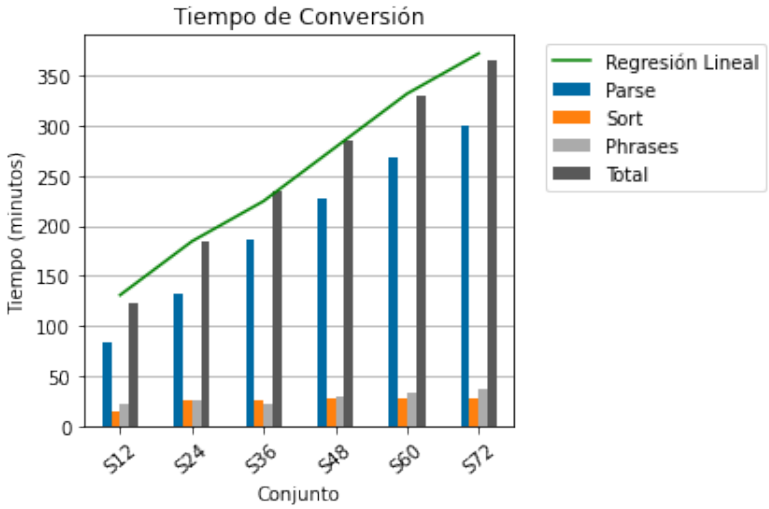

Figura 5.1: Tiempos de conversión separados por etapa. La regresión lineal se calcula en base al tamaño en megabytes de cada conjunto.

registros y caracterizar los edits para cada uno de los individuos, cuya declaración de variantes puede ser efectiva o no, en consecuencia PARSE independientemente del resultado se ejecuta en tiempo $O(|Reg| \cdot I)$, con $|Reg|$ la cantidad de registros e I la cantidad de individuos, valores que se encuentran caracterizados en la Tabla 5.3.

Después de PARSE, solo conservaremos las frases, las que no crecen necesariamente en la misma medida que los individuos, por ende la cantidad de información que debe ordenar SORT crece en menor medida y con ello los tiempos varían menos. En consecuencia de lo anterior, también aumenta en menor medida los tiempos para PHRASES. De todas formas no podemos olvidar que el proceso de PARSE se encuentra implementado en Python, versus SORT y PHRASES que están en C++11, lo que los hace inherentemente más veloces.

Por otro lado, como ilustra la Figura 5.1, a nivel macro, el tiempo total de la conversión tiene una tendencia lineal con respecto al tamaño de los VCF a indexar. Con el fin de aproximar este comportamiento, al realizar una regresión lineal sobre el tiempo totale de conversión en minutos, con respecto al peso en megabytes de cada conjunto, se obtuvo que la relación entre estos es de la forma $T(s) = 0{,}013 \cdot s$, con s el tamaño del conjunto en megabytes y $T(s)$ el tiempo de conversión en minutos. En base a esto, si quisiéramos procesar la colección VCF completa de cromosomas autosómicos de *1000 Genomes* para sus 2.500 individuos, esto requeriría aproximadamente 4 días.

Por otra parte, con respecto a las reducciones de espacio, si bien los archivos binarios producto de PARSE y SORT no son la versión final de la conversión, estos debiesen ser estrictamente más grandes que el producto final. Con ello, como primer acercamiento, observando la Tabla 5.4, notemos que se consigue alguna compresión con respecto a VCF, pero especialmente un ratio de compresión sobre 10:1 con respecto a su versión plana. Notemos que si buscáramos unir los genomas de los individuos estudiados, se obtendría un string cuyo largo sería de del orden de cientos de GBs, como ilustra la Tabla 5.3 en la penúltima columna.

Conjunto	Edits	Factors	Length	Parsing (GB)
S12	70.172.526	137.748.030	69.126.302.064	6,4
S24	140.322.365	275.464.297	138.252.900.317	11
S36	210.543.817	413.313.137	207.379.195.005	14
S48	280.701.363	551.037.548	276.505.394.382	18
S60	350.865.846	688.780.210	345.631.943.744	21
S72	421.257.601	826.959.515	414.758.033.900	25

Tabla 5.3: Caracterización de cada conjunto en base a : Cantidad de edits identificados, cantidad de factores generados, largo final de $\mathcal{S}$ y peso de los archivos binarios generados a partir de PARSE y SORT.

Conjunto	Parsing (GB)	Porción VCF	Porción Plana
S12	6,4	42 %	10 %
S24	11	57 %	8 %
S36	14	63 %	7 %
S48	18	69 %	7 %
S60	21	70 %	6 %
S72	25	75 %	6 %

Tabla 5.4: Porción a la que corresponden los archivos binarios del parsing de cada conjunto, con respecto al conjunto VCF completo original y su versión en texto plano.

5.2. Indexación completa

5.2.1. Dificultades con RLZ y soluciones

No fue hasta que nos enfrentamos a la experimentación que descubrimos que la implementación de RLZ no admitía secuencias $\mathcal{S}$ cuyos largos fueran de una magnitud superior a lo que un *unsigned int* pudiera almacenar (véase como referencia los valores de la cuarta columna de la Tabla 5.3), lo que nos forzó a definir que todas las variables de tipo *unsigned int* pasaran a ser *unsigned long long*. Algo que si bien no es un problema como tal, el hecho de aumentar el orden de magnitud de los datos sí repercutió en el proceso de construcción del índice y con ello la experimentación.

La construcción de RLZ requiere la secuencia $\mathcal{S}$ para una etapa de ordenamiento de la grilla de prefijos y sufijos pertenecientes a $\mathcal{S}$, donde la cantidad de elementos a ordenar es directamente proporcional a la cantidad de factores generados. Entonces, al aumentar la magnitud de los datos y con ello la cantidad de factores producidos, el proceso de ordenamiento incrementó significativamente sus tiempos de ejecución. Por otro lado, dado que nosotros no generamos $\mathcal{S}$ sino virtualmente por medio de $LZ(\mathcal{S}|\mathcal{R})$, se debió reimplementar el comparador del sistema de ordenamiento para que la posición que se buscara en $\mathcal{S}$ fuese recalculada a su ocurrencia en $\mathcal{R}$, a través de la lectura de $LZ(\mathcal{S}|\mathcal{R})$ para recuperar la posición respectiva en $\mathcal{R}$.

Debemos destacar que este comparador incluye optimizaciones que son importantes al ordenar sufijos o prefijos de un texto repetitivo. Al compararse strings en base a su factorización con respecto a una referencia, evitamos comparar secciones repetidas. Para aprovechar esta situación, se implementaron dos optimizaciones.

Una de ellas es que si partimos observando el mismo factor dentro de $LZ(\mathcal{S}|\mathcal{R})$, evitamos consultar el par de prefijos/sufijos asociados. Otra optimización es que, si bien pueden ser factores diferentes, sí puede suceder que estemos observando la misma posición en $\mathcal{R}$. En ese caso, sabemos que a partir de esa posición, los caracteres a leer serán idénticos, entonces podemos parar la comparación en ese punto y saltar el resto de la frase en ambos strings.

Conjunto	**PARTIAL RLZ** (min)
S12	25
S24	27
S36	34

Tabla 5.5: Evaluación del tiempo de construcción de todas las estructuras de RLZ a excepción de la grilla de prefijos y sufijos, medido en minutos.

Sin embargo, aún con las optimizaciones, los tiempos de indexación sobre colecciones de genomas no pudieron ser conocidos. Por ejemplo, las primeras etapas de construcción de RLZ para S12 se llevaron a cabo en 25 minutos (Tabla 5.5), quedando pendiente solo la construcción de la grilla, la cual aún después de 4 días de ejecución no consiguió terminar la etapa de ordenamiento de prefijos.

Por estos motivos, descartamos evaluar el tiempo de construcción del índice para los conjuntos descritos en la Tabla 5.1, optando por evaluarlo sobre conjuntos más acotados.

5.2.2. Validación

En virtud de lo descrito en la sección anterior, validaremos el proceso de conversión y construcción de RLZ sobre los conjuntos de datos descritos en la Tabla 5.6. Análogamente a los conjuntos anteriores, estos corresponden a subconjuntos del VCF para el cromosoma 21 del proyecto *1000 Genomes*.

Con el fin de ilustrar la magnitud de tiempo de cada etapa, capturaremos los tiempos de conversión para PARSE, SORT y BUILD, donde la última etapa corresponde desde la conversión de frases a factores hasta la construcción completa del índice. Además, también capturaremos la caracterización final de los conjuntos (Edits, Cantidad de Factores, Largo de $\mathcal{S}$ y tamaño del producto de las etapas PARSE y SORT), junto al tamaño de la estructura final.

Luego, a partir de los índices construidos para S05-21 y S12-21, evaluaremos los tiempos de consulta para patrones de largos $[4, 6, 8, 10, 12, 14, 16, 18, 20]$. Con el fin de mitigar el efecto de las ocurrencias, para cada largo generaremos 100 muestras aleatorias, donde el tiempo promedio de consulta a reportar será la suma de todos los tiempos de consulta dividido en el total de ocurrencias obtenidas para cada una de esas 100 muestras.

Conjunto	Individuos	**VCF** (MB)	Proporción
S05-21	5	175	35 %
S06-21	6	179	30 %
S07-21	7	183	26 %
S08-21	8	187	23 %
S09-21	9	191	21 %
S10-21	10	196	20 %
S11-21	11	200	18 %
S12-21	12	204	17 %

Tabla 5.6: Conjunto de muestras a indexar. Los conjuntos trabajan en base al cromosoma 21, usando una referencia de 51.305.818 caracteres (50MB). La última columna representa el porcentaje de espacio que representa VCF con respecto a su versión con datos planos.

5.2.3. Resultados y análisis

Cuando la magnitud de los datos a trabajar pasan a ser del orden de megabytes, como los datos de la Tabla 5.6, los tiempos de PARSE y SORT se hacen marginales, mas no así el proceso de construcción del índice. Véase el contraste entre el orden de magnitud de la gráfica de la derecha y la izquierda en la Figura 5.2.

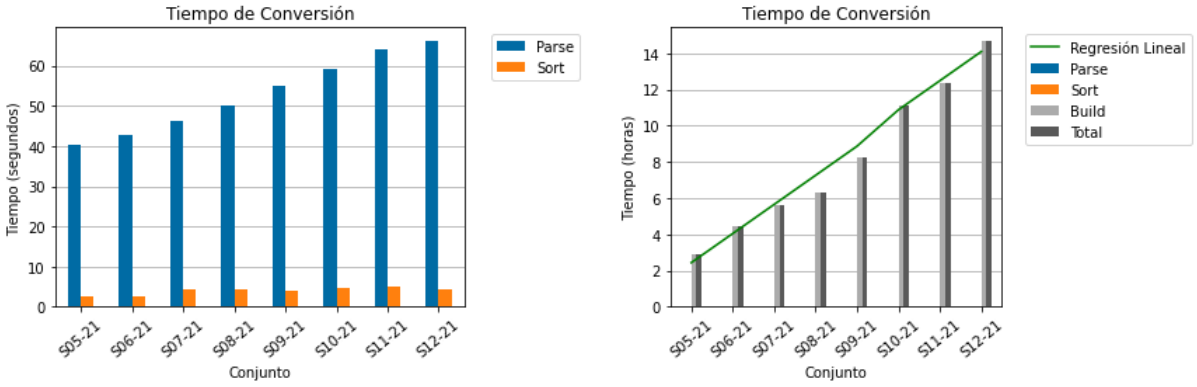

Figura 5.2: Tiempos de conversión separados por etapa. La Figura de la izquierda corresponde a los tiempos de las etapas PARSE y SORT. Luego, la Figura de la derecha también cubre PARSE y SORT, incluyendo a BUILD y el tiempo total.

Por ejemplo, convertir el conjunto S05-21, con un largo de $\mathcal{S}$ de 500 millones de caracteres (Tabla 5.7), tomó aproximadamente 3 horas (Figura 5.2). Y para S12-21 fueron más de 14 horas de procesamiento.

Este comportamiento se debe esencialmente a la etapa de construcción de la grilla de prefijos y sufijos de RLZ, la que es requerida para buscar las ocurrencias de un patrón entre frases. Notemos que el algoritmo de ordenamiento de la grilla realiza $O(F \log F)$ comparaciones de strings, con F la cantidad de factores. No obstante, cada una de estas comparaciones de strings, en el peor caso, puede realizar hasta $O(|\mathcal{S}|)$ comparaciones de caracteres, teniendo una complejidad total de orden $O(|\mathcal{S}| \cdot F \log F)$.

Conjunto	**Edits**	**Factors**	**Length**	**Parsing** (MB)	**Index** (MB)
S05-21	441.035	864.966	512.860.694	73	45
S06-21	528.556	1.036.663	615.415.623	78	47
S07-21	615.529	1.207.325	718.008.641	82	50
S08-21	702.315	1.377.514	820.591.457	87	53
S09-21	791.440	1.552.342	923.170.619	92	55
S10-21	880.031	1.726.064	1.025.726.002	96	58
S11-21	968.569	1.899.916	1.128.299.814	101	61
S12-21	1.058.478	2.076.352	1.230.854.327	106	64

Tabla 5.7: Caracterización de cada conjunto en base a : Cantidad de edits identificados, cantidad de factores generados, largo final de $\mathcal{S}$, peso de los archivos binarios generados a partir de PARSE y SORT, y finalmente el tamaño de los archivos binarios de RLZ y el bitvector comprimido.

En la práctica, esta complejidad es del orden de $O(\mathcal{C} \cdot F \log F)$, gracias a las optimizaciones implementadas en el comparador (véase la Sección 5.2.1), donde $\mathcal{C}$ debería acercarse a la cantidad promedio de caracteres comparados por par de strings. Como el ordenamiento llega a comparar los strings más cercanos lexicográficamente, $\mathcal{C}$ se acerca al largo promedio del prefijo común entre sufijos lexicográficamente consecutivos, que en texto aleatorio es $O(\log N)$ [10].

Para tener un acercamiento a la magnitud de $\mathcal{C}$, sobre la conversión de S05-21 se estudió la cantidad de comparaciones de caracteres realizadas por cada comparación de prefijos y sufijos, obteniéndose que en promedio se leen alrededor de 64 caracteres en S05-21. Esto muestra que $\mathcal{C}$ es en la práctica es suficientemente significativa como para incidir en los tiempos de ejecución de este proceso.

Otro ejemplo concreto de esta situación es el contraste de los tiempos de PARTIAL RLZ (Tabla 5.5) y los obtenidos para la construcción completa en la Figura 5.2. Como vemos en la Tabla 5.5, construir sobre colecciones genómicas todas las estructuras requeridas por RLZ, a excepción de la grilla, no tomó más de 35 minutos. Aún doblando la cantidad de datos, los tiempos de ejecución no variaron significativamente.

Con el fin de tener una aproximación del tiempo que hubiese tomado indexar los conjuntos de la sección anterior, se calcuó una regresión lineal sobre los tiempos de conversión de la Figura 5.2. Obteniendose una función de la forma $T(s) = 0{,}4 \cdot s$, con s el tamaño del conjunto VCF en megabytes y $T(s)$ el tiempo que tomaría indexar el conjunto en horas. Es decir, si quisiéramos indexar el conjunto S12, nos tomaría 6.144 horas, o 256 días. En definitiva, la construcción de la grilla presume un cuello de botella en la construcción de RLZ.

Con respecto a reducciones de espacio, si observamos el peso original de los conjuntos en formato VCF, contra el peso de esta misma información ya indexada, obtuvimos una reducción de espacio de razón entre 4:1 y 3:1 (Tabla 5.8). Además, si contrastamos la diferencia entre las columnas Parsing e Índice, se confirma que el tamaño de los archivos binarios producto de PARSE y SORT son estrictamente más grandes que los ya indexados. Por consiguiente, si se hubiese conseguido indexar las colecciones genómicas de la Tabla 5.1, potencialmente

Conjunto	**Parsing** (MB)	**Índice** (MB)	**Porción VCF**	**Porción Plana**
S05-21	73	45	25 %	9 %
S06-21	78	47	26 %	8 %
S07-21	82	50	27 %	7 %
S08-21	87	53	28 %	7 %
S09-21	92	55	28 %	6 %
S10-21	96	58	29 %	6 %
S11-21	101	61	30 %	5 %
S12-21	106	64	31 %	5 %

Tabla 5.8: Porción a la que corresponden los archivos binarios del índice generado a partir de cada conjunto, con respecto al conjunto VCF completo original y su versión en texto plano.

también se alcanzaría un ratio de compresión análogo.

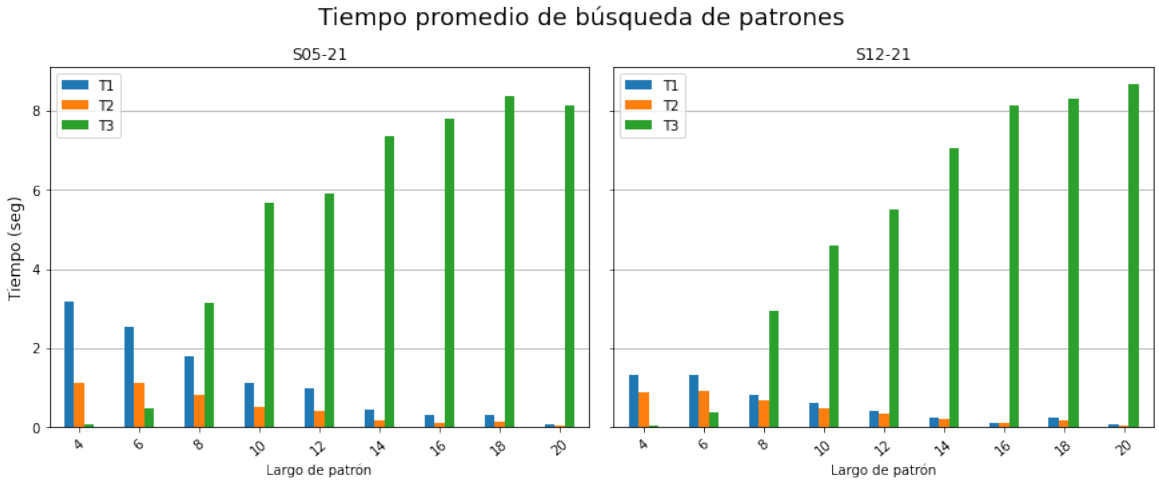

Figura 5.3: Tiempo promedio de búsqueda de patrones con respecto a su largo. Los tiempos se encuentran caracterizados por la búsqueda de ocurrencias: dentro de la referencia (T1), dentro de las frases (T2) y entre las frases (T3). La gráfica de la izquierda es la evaluación del experimento sobre el conjunto S05-21 y el de la derecha es sobre el conjunto S12-21.

Sobre la búsqueda de ocurrencia de patrones, como era esperado, los tiempos de ejecución por ocurrencia no varían significativamente ante el cambio de tamaño del índice en el que se busca (Figura 5.3). Notemos que T1 y T2 decrecen pues los tiempos de las consultas aumentan con el número de ocurrencias, al ser más largo el patrón hay menos ocurrencias. No así T3, pues esta consulta busca en base a todas las particiones del patrón consultado.

Por otra parte, si bien el comportamiento es análogo entre los conjuntos, las diferencias visibles son atribuibles a las características del input utilizado. Por ejemplo, al medir tiempos promedios, como dividimos el tiempo sobre cantidad de ocurrencias detectadas, es más probable encontrar más ocurrencias de patrones pequeños en conjuntos de gran magnitud que en los más pequeños, como lo es la situación entre S05-21 y S12-21.

5.3. Propuestas de mejora

El problema principal a resolver a futuro es el de construcción, descrito en la Sección 5.2.1, puesto que constituye un cuello de botella para la utilización de esta herramienta en colecciones genómicas mayores. El encontrar un algoritmo más eficiente para indexar algunos sufijos (o prefijos reversos) de un texto sumamente largo al que además se tiene acceso sólo vía extracción usando la referencia, es un problema que excede los alcances de esta memoria [13] [3].

En otros aspectos, la primera propuesta de mejora es la utilización de una versión compilada de Python a su versión en C++11, con el fin de optimizar los tiempos de ejecución y preservar la lógica ya implementada. Vale destacar que este proceso fue implementado en Python para reducir la complejidad de implementación del módulo, como VCFParser sustenta el reconocimiento de variantes en expresiones regulares y parsing de líneas de texto, preferimos partir por un lenguaje en el que estos procesos fuesen más amenos.

Otra propuesta es crear nuevas reglas de construcción para el soporte de variantes complejas en VCF, notemos que la estructura de una *frase* sí soporta representar estas variantes. No obstante, soportar inversión requiere un proceso más complejo. Una opción es que además de crear la regla de construcción de *frases* pertinente, necesitaríamos soportar el incluir en la referencia el inverso de la secuencia a usar durante el proceso de captura de edits, en vez de sellar el contenido de $\mathcal{R}$ al inicio de esta etapa.

También con respecto a RLZ, otro espacio de mejora es aumentar el alfabeto soportado de ACTG a ACTGN, condición necesaria para que este trabajo sea aplicable en casos reales. Además, para aumentar la eficiencia en la búsqueda de patrones en este contexto, el evitar que el índice reporte aquellas ocurrencias que cruzan nuestras muestras reside en adaptar la estructura de RLZ que busca ocurrencias entre factores, a través de evitar ingresar a la estructura los bordes entre alelos.

Además, se debe incluir soporte para la recuperación de snippets dentro de RLZ, tanto para mostrar el contexto de las ocurrencias como para hacer browsing de la colección sin necesitar el acceso a los archivos VCF ni de referencia originales.

Con respecto al módulo de conversión, este soporta alfabetos, ploidías y cantidades arbitarias de cromosomas autosómicos, pero falta incluir el soporte para la interpretación de cromosomas sexuales en VCF.

Capítulo 6

Ejemplo de uso

En este capítulo presentaremos dos programas interactivos que ilustran la forma de utilizar el módulo. El primer programa corresponde a la indexación del conjunto, y el segundo es la búsqueda de patrones en base al índice recuperado de los archivos binarios generados por el primer programa.

6.1. Indexación

Para ilustrar el uso de este módulo de conversión, en esta sección presentamos el uso de un programa interactivo que hace uso de este. En primer lugar, los datos a utilizar son: un archivo de referencia con 2 cromosomas (Figura 6.1), un VCF para el cromosoma 1 (Figura 6.2) y otro para el cromosoma 2 (Figura 6.3). En este ejemplo trabajaremos con dos individuos, `F1` y `F2`.

Vale destacar que en la referencia, cada cromosoma contará con 225 caracteres cada uno. Para efectos ilustrativos, solo en el cromosoma 2 se encontrará la cadena `TTT` en la posición 160. Tanto en el resto de la referencia como en los registros no se encontrará la base `T`.

Con respecto al archivo VCF del primer cromosoma (Figura 6.2), en el primer registro tenemos una sustitución en la posición 19, la cual es codificada solo por el individuo `F2` en su primer alelo. Luego, en el segundo registro se presenta una eliminación en la posición 22, codificada solo por el primer alelo del individuo `F1`.

Después, para el archivo VCF del cromosoma 2 (Figura 6.3), tenemos solo un registro correspondiente a una inserción en la posición 99, para el segundo alelo del individuo `F1`.

A la hora de indexar la colección, debemos declarar donde se quiere guardar la información del índice, la ubicación de la referencia, la cantidad de cromosomas a entregar, seguido por la ubicación de cada uno de los VCF a procesar. Para este caso, esa información se encuentra dentro del archivo *example.txt* (Linea de comando en la Figura 6.4), cuyo contenido es el comando impreso en pantalla para Python.

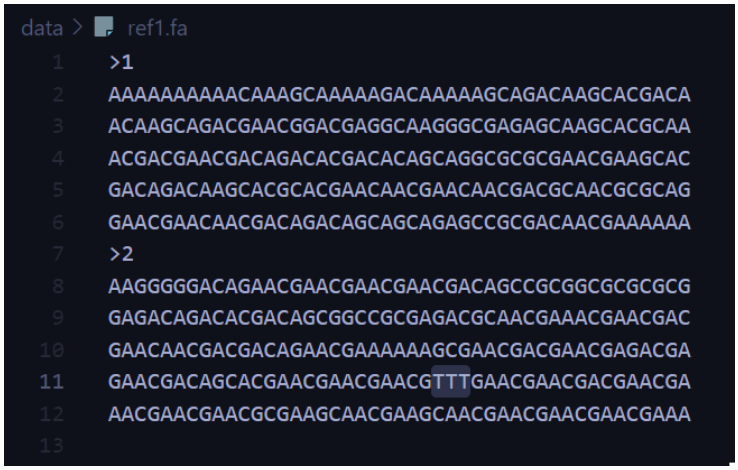

Figura 6.1: Archivo FASTA para dos cromosomas, cuyas referencias son de 225 caracteres.

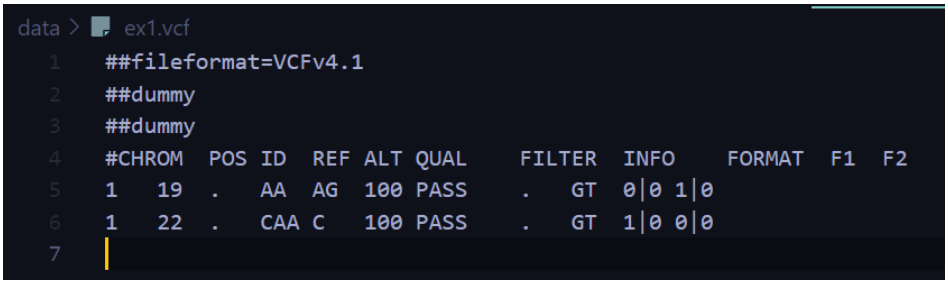

Figura 6.2: Cromosoma 1 con dos registros, uno de sustitución y otro de eliminación.

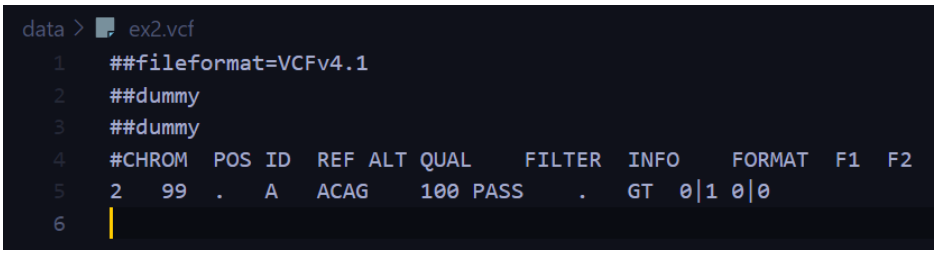

Figura 6.3: Cromosoma 2 con un solo registro de inserción.

Notemos que este programa soporta recibir qué procesos deseamos saltar, por medio de los últimos tres booleanos del comando. Por ejemplo, si previamente ya procesamos nuestros datos a través de PARSE y SORT, podemos continuar con BUILD solo cambiando los ultimos tres valores a `TRUE TRUE FALSE`.

Una vez ejecutado el comando, en pantalla se muestra cada archivo VCF que pasó por el proceso de PARSE exitosamente. Una vez PARSE termina, reporta la cantidad de individuos (samples) identificados, que en este caso es 2. Luego, muestra cuántos registros no fueron procesados por incumplimiento de requisitos; seguido por la cantidad de frases recuperadas (edits). Después, se ilustra el tamaño original de los archivos procesados, versus su versión binaria ya parseada. Para finalmente cerrar con el tiempo de ejecución de esta etapa.

Vale destacar que una vez termina PARSE, todos los archivos binarios son guardados

automáticamente, de esta forma, PARSE puede ejecutarse como un proceso aislado y no necesariamente continuar con SORT.

```
fernanda@knuth:~/VCF_RLZ$ ./build/timeC ../VCF_files/example.txt  ../results.txt FALSE FALSE FALSE
Executing:
        python3 ./VCF_parsing/parsing_process.py /home/fernanda/data/ /home/fernanda/data/ref1.fa -n 2
/home/fernanda/data/ex1.vcf /home/fernanda/data/ex2.vcf
 PARSE starting...
[RLZ]     /home/fernanda/data/ex1.vcf parsed!
[RLZ]     /home/fernanda/data/ex2.vcf parsed!
[RLZ] Resume parsing process:
[RLZ]   Number of samples identified: 2
[RLZ]   Number of dropped records: 0
[RLZ]   Number of valid edits captured: 5
[RLZ]   File size processed: 0.000270843505859375 MB
[RLZ]   Size of parsed file generated: 0.00026702880859375 MB
   PARSE Ended:
/home/fernanda/data/    PARSE   48.4061 (ms)
```

Figura 6.4: Imagen de ejemplo de uso del programa *timeC*, que permite realizar la conversión en 3 etapas. Se muestran los mensajes en pantalla para la primera etapa, PARSE.

Con la etapa de PARSE finalizada, iniciamos SORT (Figura 6.5). Primero se ilustra la inicialización de los recursos destinados para el ordenamiento, proceso que es ejecutado por STXXL directamente. Para efectos de este ejemplo, primero se evalúa si las frases se encuentran ordenadas según nuestros parámetros, de no ser así, procedemos a ordenarlas. Notar que este proceso trabaja directamente sobre el archivo binario que contiene las frases. Una vez esta etapa finaliza, se muestra el tiempo total del proceso.

```
   PARSE Ended:
/home/fernanda/data/    PARSE   48.4061 (ms)
 SORT starting...
[RLZ] Number of identified edits: 5
[STXXL-MSG] STXXL_PARALLEL_MULTIWAY_MERGE
[STXXL-MSG] STXXL v1.4.99 (prerelease/Release) (git b9e44f0ecba7d7111fbb33f3330c3e53f2b75236) + gnu par
allel(20210110)
[STXXL-MSG] Disk '/d2/fernanda/stxxl' is allocated, space: 352859 MiB, I/O implementation: syscall queu
e=0 devid=0 unlink_on_open
[STXXL-MSG] Checking order...
[STXXL-MSG]     WRONG
[STXXL-MSG] Sorting...
[RLZ]   Sort sucessfull! Time elpased: 2.72791ms
[STXXL-MSG] Checking order...
[STXXL-MSG]     OK
   SORT Ended:
/home/fernanda/data/    SORT    7.42018 (ms)
```

Figura 6.5: Imagen de mensajes en pantalla del programa *timeC* para la etapa SORT, posterior a la ejecución de PARSE.

Finalmente, una vez las frases son ordenadas, se inicia la etapa de BUILD (Figura 6.6). Dada la longitud de esta etapa, se busca ilustrar los tiempos de cada uno de los pasos intermedios. En primer lugar, se lee la meta data generada por PARSE, y en base a ello se empieza a leer frase por frase del archivo binario ya ordenado. Por cada frase leída, se generan los factores correspondientes, que son inmediatamente almacenados en otro archivo binario.

Con los factores generados, se pasa a la construcción del índice. Primero se comprime la referencia, y con ello se construye secuencialmente cada una de las estructuras que componen a RLZ. Una vez que BUILD finaliza, guardamos la estructura en la carpeta declarada en el comando original.

```
   SORT Ended:
/home/fernanda/data/    SORT    7.42018 (ms)
 BUILD starting...
[RLZ] Reading metadata
[RLZ]      Metadata readed in 0.147498 ms
[RLZ] Building factors from phrases
[RLZ]      Building factors builded in 0.036974
[RLZ] Building Index
[RLZ]      Load reference from parsing
TextFilter::readReferenceFull - Inicio (leyendo "/home/fernanda/data/Tmp/Parsing/Reference.tmprlz")
TextFilter::readReferenceFull - Fin (chars: 1702 en 0.054404 ms)
[RLZ]      Create index
RelzIndexReference - Inicio (factors: 14, len_text: 1798, len_ref: 1702)
RelzIndexReference - Preparing Factors
RelzIndexReference - Factors Sorted prepared in 0.018241
RelzIndexReference - Preparing Vector S
RelzIndexReference - Vector S prepared in 0.006584
RelzIndexReference - Preparing Permutation PI
RelzIndexReference - PI prepared in 0.027927
RelzIndexReference - Preparing Vector B
RelzIndexReference - Vector B prepared in 0.043187
RelzIndexReference - Preparing fm_index
RelzIndexReference - fm_index prepared in 19.293
RelzIndexReference - Preparing arr X
RelzIndexReference - arr X prepared in 0.033611
RelzIndexReference - Preparing arr Y
RelzIndexReference - arr Y prepared in 0.020734
RelzIndexReference - Preparing WT
RelzIndexReference - WT prepared in 9.96349
RelzIndexReference - End
[RLZ]      Index finished in 29.7085 ms
   BUILD Ended:
/home/fernanda/data/    BUILD   29.9727 (ms)
RelzIndexReference::save - Start (base "/home/fernanda/data/VCF_RLZ/vcf-rlz-index")
CompactedText::save - len: 1702
RelzIndexReference::save - End
```

Figura 6.6: Imagen de mensajes en pantalla para la etapa BUILD dentro del programa *timeC*, posterior a la ejecución de SORT.

6.2. Búsqueda de patrones

En base a los archivos binarios generados por el programa anterior, reconstruimos el índice, de esta forma podemos realizar consultas para un mismo índice en cualquier momento. Además de que estos archivos son fácilmente distribuibles, lo que facilita su difusión.

Para hacer uso de este programa (Figura 6.7), al ejecutable le entregamos únicamente la ubicación de la carpeta donde se almacenaron los archivos previamente. Luego, el programa muestra en pantalla si la carga de archivos fue exitosa o no.

Con las estructuras cargadas, se ofrecen dos opciones: Buscar un Snippet, o bien cerrar el programa. Si se selecciona la primera opción, luego nos solicita ingresar la cadena a ser buscada, que en este caso buscaremos la cadena `TTT`, la que solo existe en el cromosoma 2 (Figura 6.8). Una vez ingresado el patrón, se imprime en pantalla el total de ocurrencias encontradas, caracterizadas por individuo, cromosoma, alelo y la posición correspondiente dentro de este. Finalmente se ofrece nuevamente la consulta.

Para pruebas de mayor escala, a este programa se le puede integrar la declaración del nombre del archivo donde se quieren almacenar los resultados. Notar que la impresión en pantalla de los resultados es optativa.

```
fernanda@knuth:~/VCF_RLZ$ ./build/prebuild /home/fernanda/data/VCF_RLZ/
[STXXL-MSG] STXXL v1.4.99 (prerelease/Release) (git b9e44f0ecba7d7111fbb33f3330c3e53f2b75236) + gnu par
allel(20210110)
[STXXL-MSG] Disk '/d2/fernanda/stxxl' is allocated, space: 352859 MiB, I/O implementation: syscall queu
e=0 devid=0 unlink_on_open
RelzIndexReference::load - Start (base "/home/fernanda/data/VCF_RLZ/vcf-rlz-index")
CompactedText::load - len: 1702
CompactedText::load - n_bytes: 426
RelzIndexReference::load - fm_index
RelzIndexReference::load - rmq
RelzIndexReference::load - bits_s
RelzIndexReference::load - bits_b
RelzIndexReference::load - wt
RelzIndexReference::load - End
Puedes ejecutar las siguientes acciones:
        (1) Buscar un snippet
        (2) Salir
Ingresa el nro de tu opción:
```

Figura 6.7: Imagen de mensajes en pantalla para la ejecución del programa *prebuild*, el cual ilustra la carga exitosa del índice. Luego se ofrece la opción de búsqueda.

```
Puedes ejecutar las siguientes acciones:
        (1) Buscar un snippet
        (2) Salir
Ingresa el nro de tu opción:
1
Ingrese el snippet a consultar:
TTT
Occ: 4
Sample: F1
 - Chrom: 2
   - Alele: 1
     - Pos: 160
 - Chrom: 2
   - Alele: 2
     - Pos: 164
Sample: F2
 - Chrom: 2
   - Alele: 1
     - Pos: 160
 - Chrom: 2
   - Alele: 2
     - Pos: 160
Puedes ejecutar las siguientes acciones:
        (1) Buscar un snippet
        (2) Salir
Ingresa el nro de tu opción:
```

Figura 6.8: Imagen de mensajes en pantalla para el uso del programa *prebuild*, para el patrón `TTT`, donde se reportan las 4 ocurrencias esperadas.

Conclusión

El estudio de variantes en genoma nos permite desde entender historia evolutiva hasta la caracterización de diferentes condiciones de salud. La realización de estos estudios requiere acceder al genoma y buscar aquellos patrones cuyas secuencias sean de interés. Como VCF no soporta búsquedas sin descompresión, el trabajo de esta memoria consistió en implementar una API que nos permitiera llevar la información de VCF a RLZ sin descomprimir, con el fin de dar soporte a estas búsquedas de forma eficiente.

Trabajo cuyo resultado, aún pese a las dificultades encontradas, tuvo éxito. En términos de implementación y diseño se consideran como cumplidos los objetivos específicos. A su vez también se cumplió parcialmente el objetivo general propuesto. Pese a no validar la funcionalidad completa en colecciones genómicas, si se consiguió validar la implementación en colecciones de un solo cromosoma. Vale destacar que hacer esto posible significó reimplementar parte del constructor de RLZ, quedando como problema abierto la construcción de una grilla de prefijos y sufijos sobre textos del orden de gigabytes. Situación que se encontraba fuera del scope original de este trabajo.

En definitiva, el sobrestimar los recursos implementados y subestimar el alcance de los datos dió pie a que la experimentación y validación en datos reales no fuese del todo exitosa. Aún cuando durante toda la implementación de la solución se tuvo en cuenta que la magnitud de los datos debía manejarse con cuidado, en particular por el uso de C++11, no se cuestionó hasta el final si la implementación de RLZ mantendría sus fortalezas en tiempos de construcción para estas magnitudes.

Además, hubo un retraso considerable con respecto a la planificación original, pues el estudio de las especificaciones e interpretación de VCF fue sumamente compleja, dado que es un formato muy versátil y su interpretación pertenece a un campo de estudio que no se vincula con la computación directamente. El encontrar asesoramiento de personas que trabajasen con este formato fue lento y difícil, en consecuencia la etapa más compleja de implementar fue la de transformación de edits a *frases*.

Sin embargo, enfrentar desafíos como lo fue este Trabajo de Título se tradujo en un proceso lleno de aprendizaje, se aprendió desde cómo realizar análisis en genoma hasta cómo definir apropiadamente proyectos escalables en C++11. Conforme aparecían más desafíos, la curva de aprendizaje para resolverlos se hacía cada vez más plana. Gracias a este proceso, cuando se descubrió que la implementación de RLZ debía ser adaptada, la concepción de su solución se realizo en cuestión de horas.

Vale destacar que todos aquellos puntos que en este minuto se tradujeron en una limitante, cuentan con soluciones reconocidas. En consecuencia, este proyecto tiene un espacio de mejora definido, y conseguir su aplicabilidad en datos reales tiene una ruta de trabajo identificada.

Bibliografía

[1] Richard M. Durbin Adam Auton, Lisa D. Brooks. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

[2] Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10:748–759, 2011.

[3] Philip Bille, Johannes Fischer, Inge Li Gørtz, Tsvi Kopelowitz, Benjamin Sach, and Hjalte Wedel Vildhøj. Sparse text indexing in small space. *ACM Trans. Algorithms*, 12(3):article 39, 2016.

[4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[5] David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[6] Petr Danecek, Adam Auton, Gonçalo R. Abecasis, Cornelis A. Albers, Eric Banks, Mark A. DePristo, Robert E. Handsaker, Gerton Lunter, Gabor T. Marth, Stephen T. Sherry, Gilean McVean, and Richard Durbin. The variant call format and vcftools. *Bioinform.*, 27(15):2156–2158, 2011.

[7] Petr Danecek, James K Bonfield, Jennifer Liddle, John Marshall, Valeriu Ohan, Martin O Pollard, Andrew Whitwham, Thomas Keane, Shane A McCarthy, Robert M Davies, and Heng Li. Twelve years of SAMtools and BCFtools. *GigaScience*, 10(2), 2021. giab008.

[8] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: standard template library for xxl data sets. *Software: Practice and Experience*, 38(6):589–637, 2008.

[9] Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative lempel-ziv self-index for similar sequences. In *Frontiers in Algorithmics and Algorithmic Aspects in Information and Management - Joint International Conference, FAW-AAIM 2012. Proceedings*, volume 7285 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2012.

[10] Julien Fayolle and Mark Ward. Analysis of the average depth in a suffix tree under a markov model. *Discrete Mathematics and Theoretical Computer Science*, AD:95–104, 2005.

[11] Hector Ferrada, Travis Gagie, Tommi Hirvola, and Simon J. Puglisi. Hybrid indexes for repetitive datasets. *CoRR*, abs/1306.4037, 2013.

[12] Héctor Ferrada, Dominik Kempa, and Simon J. Puglisi. Hybrid indexing revisited. In *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018*, pages 1–8. SIAM, 2018.

[13] Paolo Ferragina and Johannes Fischer. Suffix arrays on words. In *Combinatorial Pattern Matching*, pages 328–339, Berlin, Heidelberg, 2007.

[14] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52:552–581, 2005.

[15] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40:465–492, 2011.

[16] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1459–1477.

[17] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[18] William T. Harvey, Alessandro M. Carabelli, Ben Jackson, Ravindra K. Gupta, Emma C. Thomson, Ewan M. Harrison, Catherine Ludden, Richard Reeve, Andrew Rambaut, Sharon J. Peacock, and et al. Sars-cov-2 variants, spike mutations and immune escape. *Nature Reviews Microbiology*, 19(7):409–424, 2021.

[19] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013.

[20] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010. Proceedings*, volume 6393 of *Lecture Notes in Computer Science*, pages 201–206.

[21] Divon Lan, Ray Tobler, Yassine Souilmi, and Bastien Llamas. Genozip: a universal extensible genomic data compressor. *Bioinformatics*, 37(16):2225–2230, 2021.

[22] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Trans. Inf. Theory*, 22(1):75–81, 1976.

[23] R. R. Hudson M. Przeworski and A. D. Rienzo. Adjusting the focus on human variation. In *Trends Genetics*, volume 13, pages 296–302, 2000.

[24] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010.

[25] J. Munro, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438, 2011.

[26] Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25, 2013.

[27] Gonzalo Navarro. Indexing highly repetitive string collections, part II: compressed indexes. *ACM Comput. Surv.*, 54(2):26:1–26:32, 2021.

[28] Gonzalo Navarro and Victor Sepulveda. Practical indexing of repetitive collections using relative lempel-ziv. In *Data Compression Conference*, pages 201–210. IEEE, 2019.

[29] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007*.

[30] Alexander Stepanov and Meng Lee. The standard template library. 1999. http://stepanovpapers.com/STL/DOC.PDF.