UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Compresión de datos crudos de secuenciación de ADN por nanoporos

Informe de Proyecto de Grado presentado por

Rafael Agustín Castelli Ottati

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Guillermo Dufort y Álvarez
Álvaro Martín

Montevideo, 4 de diciembre de 2023

Agradecimientos

Quiero agradecer a mi madre Laura y a mi tío Gabriel, así como a mis amigos por acompañarme. También quiero agradecer a mis tutores Guillermo y Álvaro por sus enseñanzas y apoyo continuo.

Resumen

En este proyecto, generamos un framework para el desarrollo de compresores de datos crudos de secuenciación de ADN por nanoporos que permite prototipar de forma sencilla nuevos compresores y modelos estadísticos para dichas señales.

Trabajamos sobre el formato de archivo POD5 y su biblioteca homónima, desarrollados por Oxford Nanopore Technologies (ONT) para suplantar a su predecesor, FAST5. El producto desarrollado permite la implementación de nuevos compresores que se integran a la biblioteca de POD5, así como compresores independientes.

Presentamos dos compresores para este tipo de datos: PGNanoS y PGNano5. Estos compresores logran una mejora porcentual relativa de aproximadamente 2.9 % respecto a Vbz, que es el compresor por defecto para POD5, al costo de considerable incremento de costo computacional. Estos compresores sirven como una línea base para investigaciones futuras, usando el framework desarrollado.

También presentamos un conjunto de pruebas y análisis sobre los datos de secuenciación y posibles codificaciones que nos permiten interpretar el funcionamiento de Vbz, así como proponer alternativas para mejorar los resultados obtenidos por PGNano5 y PGNanoS.

Palabras clave: Compresión de datos sin pérdida, Secuenciación de ADN, Nanoporos, MinION, POD5, Códigos de Golomb, Codificación aritmética.

Índice general

Capítulo 1

Introducción

Las técnicas de secuenciación de ADN (ácido desoxirribonucleico) tienen como objetivo determinar la secuencia de bases de una molécula de ADN (National Human Genome Research Institute, 2023b). Su desarrollo ha tenido un impacto relevante en la investigación biológica, abarcando por ejemplo el "Proyecto Genoma Humano", llamado la investigación biomédica más grande del siglo 20 (National Human Genome Research Institute, 2022). La tecnología ha tenido impacto tanto en investigaciones como en tratamientos médicos, estudios de enfermedades o incluso en estudios sobre la evolución. Además promete tener aplicaciones para el tratamiento de enfermedades como cáncer, detección prematura de enfermedades en niños recién nacidos y mejor diagnóstico médico en general.

Avances en el área han permitido a investigadores, secuenciar de forma más rápida y barata, mayores porciones del genoma de diversos organismos (National Human Genome Research Institute, 2023b). Esto conlleva necesidades de espacio de almacenamiento y poder de cómputo superiores, para poder tratar con los datos recolectados. Para tener idea de los volúmenes de datos presentes en la realidad, el dispositivo PromethION 48 de Oxford Nanopore Technologies (ONT) puede secuenciar hasta $13{,}3 \text{x} 10^{12}$ bases nitrogenadas en 72 horas (Oxford Nanopore Technologies, 2022i). Luego, es necesario contar con algoritmos de compresión sin pérdida eficientes y eficaces que permitan reducir los requerimientos de almacenamiento y transmisión de los datos.

Nos interesa particularmente la secuenciación de ADN por nanoporos, donde ONT aparece como un gran impulsor de la tecnología. Esta tecnología de secuenciación tiene particularidades, que serán exploradas en el capítulo 2, que utilizamos para la compresión de dichos datos. En esta técnica de secuenciación tenemos una hebra de ADN atravesando un nanoporo[1] de una membrana. Junto a esta membrana tenemos sensores que miden la densidad de corriente eléctrica en la membrana, una medida que varía según el ADN pasa por el nanoporo. Estas mediciones de densidad de corriente eléctrica son nuestros datos crudos

[1] Un poro de tamaño del orden de los nanómetros

de secuenciación de ADN que luego serán procesados para inferir la secuencia de bases de la molécula. La figura 1.1 ilustra un fragmento de una señal obtenida mediante esta técnica en función del tiempo. El eje x representa el número de muestreos que se hicieron hasta el momento de obtener un valor de señal. El eje y representa el valor de la densidad de corriente eléctrica medida en DACs (Data Acquisition values), la unidad de medida con la que nos referimos a los valores resultado de muestrear dicho valor de densidad de corriente eléctrica.

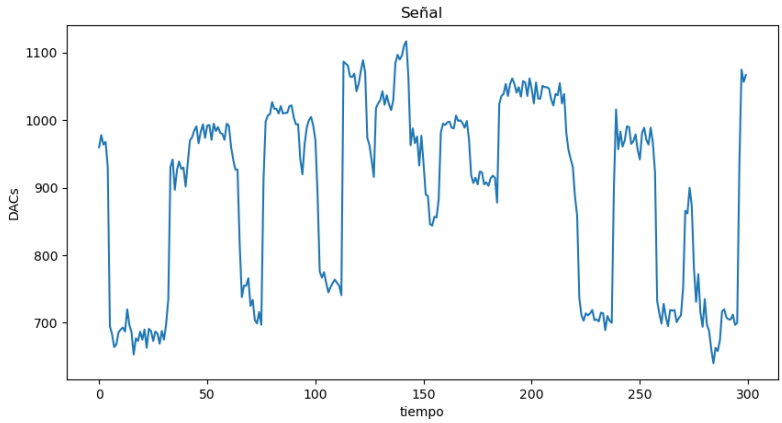

Figura 1.1: Ejemplo de la señal producida por dispositivos de secuenciación de ADN por nanoporos.

Notar que la señal alterna entre mesetas de comportamiento equivalente mediante saltos bruscos. Cada meseta corresponde con la presencia de un kmero en el nanoporo (una subsecuencia de largo k de la molécula de AND), que induce una media característica sobre la señal. Los saltos corresponden al cambio del kmero presente en el nanoporo lo que conlleva al cambio de la media característica.

El formato estándar para almacenar los datos de secuenciación por nanoporos fue FAST5, producido por ONT, hasta 2022 donde la compañía (Oxford Nanopore Technologies, 2023f) anuncia el formato POD5 para sustituir a su predecesor (Oxford Nanopore Technologies, 2022d) (Oxford Nanopore Technologies, 2023a) (Oxford Nanopore Technologies, 2022f). FAST5 tiene una serie de problemas, particularmente relacionados con el acceso en paralelo a los datos y la duplicación de metadatos, que POD5 mejora. A su vez, un equipo independiente propone un formato SLOW5 (Gamaarachchi, Samarakoon, Jenner, y et al., 2022) (Samarakoon, Ferguson, Jenner, Loman, y Quick, 2023) como alternativa a FAST5.

Dado que FAST5 es un formato que se pretende sustituir, y dado que (a diferencia de SLOW5), POD5 es el formato oficial impulsado por ONT que se emite directamente de los dispositivos secuenciadores, en este proyecto nos centramos en POD5. No obstante, es relevante mencionar algunos trabajos previos

que existen.

Sobre el formato FAST5 tenemos el compresor Picopore, desarrollado por Scott Gigante (Gigante, 2022), que se centra en la eliminación de diversos metadados y resultados posteriores a la obtención de datos crudos de secuenciación de ADN. Este producto comprime la señal de secuenciación usando una biblioteca de compresión general llamada gzip[2]. Para FAST5 tenemos también el compresor F5Comp (Nicolás Izquierdo, 2020), desarrollado por Nicolás Izquierdo y Gonzalo Larghero en 2020 como parte de un proyecto de grado con objetivos similares al presente. Este compresor tiene diferentes métodos para reducir la redundancia de metadatos así cómo comprimir los datos crudos. Para comprimir la señal de secuenciación usa la biblioteca gzip y árboles de Huffman (Cover y Thomas, 2006a) construidos para este tipo de datos.

En POD5, tenemos el algoritmo Vbz (Oxford Nanopore Technologies, 2022e), desarrollado por ONT, que corresponde con el estado del arte en métodos de compresión sobre este tipo de datos. Vbz se basa en los algoritmos StreamVByte y ZSTD, así cómo en codificación diferencial y el mapeo de Rice.

Recabando estadísticas sobre POD5, vemos que los datos crudos de secuenciación de ADN representan un 99,892 % del espacio total del archivo. Esto nos motiva a centrar nuestros objetivos de compresión sobre los datos crudos de secuenciación.

Por otro lado, el Núcleo de Teoría de la Información (NTI) también tiene proyectos relacionados con esta área. Este trabajo busca apoyar estos proyectos a través de la creación de un producto bien estructurado que pueda ser extendido con diferentes métodos de compresión. Además, algunos de los experimentos y métodos de compresión implementados fueron sugeridos por el NTI.

Por lo tanto, el proyecto tiene dos objetivos principales: en primer lugar, generar un producto bien estructurado y expandible que facilite la implementación de futuros métodos de compresión para señales de nanoporos. En segundo lugar, buscamos generar y experimentar con diferentes métodos de compresión, que intenten superar los resultados del algoritmo Vbz propuesto por ONT.

En este trabajo presentamos dos compresores para datos crudos de secuenciación de ADN sobre archivos POD5: PGNano5 un compresor integrado a la biblioteca POD5 desarrollada por ONT, así como PGNanoS, un compresor desacoplado de dicha biblioteca. Ambos compresores codifican la diferencia entre muestras consecutivas (en vez del valor de cada muestra propiamente dicho) para aprovechar la correlación estadística entre ellas. A estas diferencias se les aplica el mapeo de Rice (Gadiel Seroussi, 2023a) para llevarlas al rango de enteros no negativos y se separan en dos subsímbolos, uno para el byte alto y otro para el byte bajo de los enteros a comprimir. El resultado se pasa por un codificador aritmético (Alvaro Martín, 2021), que es un algoritmo genérico para comprimir una secuencia de símbolos de a cuerdo a cierto modelo estadístico paramétrico. Estos compresores logran una mejora porcentual relativa de aproximadamente 2.9 % respecto a Vbz, que es el compresor por defecto para POD5, al costo de considerable incremento de costo computacional.

[2]Sitio web de la biblioteca: https://www.gnu.org/software/gzip/

Además, en el capítulo 6 presentamos diversos experimentos que estudian las características de la señal y dan resultados concretos sobre otras posibles codificaciones a usar. Como resultado de estos experimentos, proponemos un esquema general de métodos de compresión que aprovechan las características particulares de las señales de secuenciación por nanoporos, dando varios posibles caminos para futuras investigaciones. Estas vías de investigación futura pueden usar el framework propuesto, que facilita la experimentación y desarrollo de nuevos métodos de compresión. En este contexto, los algoritmos desarrollados sirven de línea base dentro del framework. Por último, en el capítulo 7, analizamos el algoritmo Vbz y explicamos qué características de la señal explota para obtener buenos resultados de compresión.

1.1. Organización del documento

El resto del documento se estructura de la siguiente forma. El capítulo 2 es una revisión de antecedentes, donde presentamos conceptos generales de la secuenciación de ADN. También discutimos en mayor detalle la secuenciación de nanoporos. El capítulo continúa con conceptos generales de compresión, así como una presentación breve de los principales algoritmos utilizados en el proyecto.

El capítulo 3 describe brevemente a Apache Arrow, el formato de archivo en que se basa el formato POD5. Luego, mostramos el formato POD5 y como utiliza los conceptos de Arrow antes presentados, lo que es relevante para entender la decisiones arquitectónicas hechas en la modificación de la biblioteca.

A continuación, en el capítulo 4, introducimos a PGNano5 y PGNanoS. Este capítulo comienza presentando características relevantes de la señal a comprimir, que son el fundamento del algoritmo propuesto, PGNanoAlgo. El capítulo continúa explicando las modificaciones necesarias a POD5 para generar PGNano5, así como las decisiones de diseño más importantes para expandir las funcionalidades de POD5, sin comprometer sus atributos de calidad esenciales. Luego, presentamos PGNanoS junto a una breve explicación de su implementación.

El capítulo 5 presenta resultados experimentales de PGNano5 y PGNanoS, comparándolos con Vbz y dando un análisis de los resultados.

A continuación, el capítulo 6 explora modelos estadísticos alternativos para la señal de secuenciación distintos, al usado en PGNano5 y PGNanoS. El capítulo también presenta experimentos sobre otras codificaciones para la señal computacionalmente más eficientes que un codificador aritmético, en particular usando códigos de Golomb (Gadiel Seroussi, 2023a) y Run Length Encoding (Gadiel Seroussi, 2023b).

Luego, el capítulo 7 analiza el funcionamiento del algoritmo Vbz. Como consecuencia de esto, obtenemos versiones alternativas de Vbz que pueden ser de interés para investigaciones futuras.

Por último, el capítulo 8 expone las conclusiones de este trabajo y documenta un conjunto de posibles futuras vías de investigación.

Capítulo 2

Revisión de antecedentes

Este capítulo es una revisión de antecedentes en donde enmarcamos el trabajo realizado en este proyecto. Además, damos un conjunto de términos y definiciones que necesitaremos a lo largo del documento.

El capítulo comienza dando una introducción a la secuenciación de ADN en la sección 2.1. Allí también, ilustramos diferentes métodos de secuenciación, poniendo énfasis en la secuenciación por nanoporos y sus características principales. La sección culmina profundizando sobre el proceso para el cual se usan los datos crudos de secuenciación, denominado base calling.

La sección 2.2 ilustra conceptos generales sobre la compresión de datos y teoría de la información. Aquí damos un conjunto de conceptos introductorios, los cuales necesitaremos a lo largo del documento. La sección termina dando una primera vista a cómo los conceptos introducidos anteriormente son parte del modelado estadístico que realizamos al comprimir datos, así como plantea algunos de los desafíos principales a los que nos enfrentaremos.

Finalmente, en la sección 2.3 describimos brevemente los algoritmos de compresión y transformaciones de datos más importantes para este trabajo.

2.1. Secuenciación de ADN

Para presentar la secuenciación de ADN citamos la siguiente definición:

"La secuenciación de ADN se refiere a las técnicas generales de laboratorio usadas para determinar la secuencia exacta de nucleótidos, o bases, en una molécula de ADN. La secuencia de bases (a menudo referidas por las primeras letras de sus nombres químicos: A, T, C y G) codifica la información biológica que las células utilizan para desarrollarse y funcionar. Establecer la secuencia de ADN es clave para entender la función de los genes y otras partes del genoma. Ahora hay varios métodos diferentes disponibles para la secuenciación de ADN, cada uno con sus propias características, y el desarrollo de métodos adicionales representa un área activa de investigación en genómica.". (National Human Genome Research Institute, 2023a).

De la definición anterior se desprende que existen diversos métodos para la secuenciación de ADN, todos con el fin de generar la secuencia de bases correspondiente a una molécula de ADN pero cada uno tiene ventajas y desventajas. Para entender las características particulares de la secuenciación por nanoporos, primero necesitamos conocer algunos métodos de secuenciación. La figura 2.1 ilustra diferentes métodos.

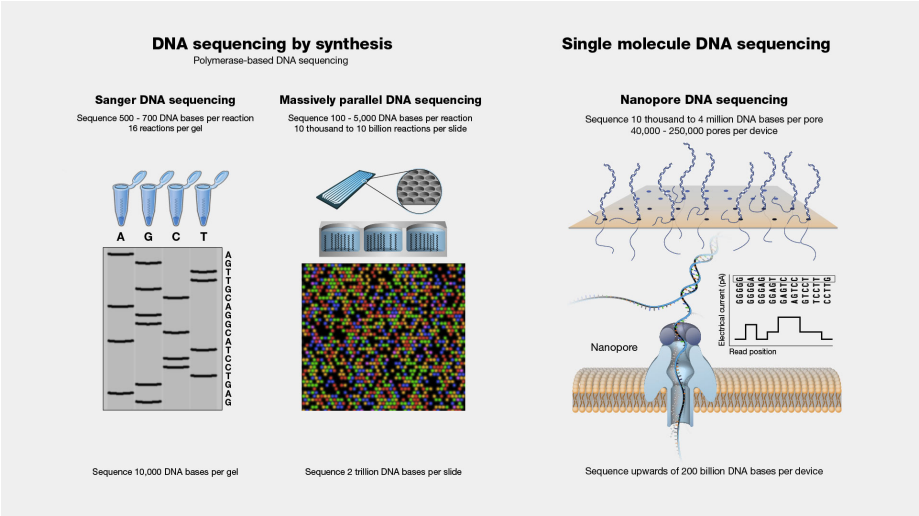

Figura 2.1: Algunos métodos de secuenciación de ADN ordenados por generación. A la izquierda: método de Sanguer (primera generación) y métodos masivamente paralelos (segunda generación). A la derecha: secuenciación por nanoporos (tercera generación).
Figura extraída de (National Human Genome Research Institute, 2023a)

Podemos dividir los métodos de secuenciación en generaciones, según avances tecnológicos relevantes. Tenemos los métodos de primera, segunda y tercera generación, ilustrados en ese orden, de izquierda a derecha en la figura 2.1. Nos importan específicamente los métodos de tercera generación, donde está incluida la secuenciación por nanoporos.

Primera generación El método de Sanger es el método más relevante de la primera generación de métodos de secuenciación (Wikipedia, 2023b). Los métodos de esta generación son los que permitieron los primeros avances. No obstante, estos métodos son lentos y caros respecto a métodos más modernos.

Segunda generación Los métodos de segunda generación, se enfocan en la escalabilidad, permitiendo secuenciar genomas enteros en una pasada. Usualmente se muestrea aleatoriamente un genoma partiéndolo en subsecuencias más pequeñas, y se secuencia cada subsecuencia en paralelo utilizando algún método de secuenciación. Posteriormente las subsecuencias se ensamblan con algoritmos

específicos que intentan reconstruir la secuencia original completa (Wikipedia, 2023b).

Tercera generación Estos métodos también se conocen como métodos de gran escala o métodos de secuenciación "de novo". Se especializan en generar secuencias de piezas de ADN muy largas, como podría ser un cromosoma entero. También pueden generar un gran número de secuencias cortas.

Los métodos de tercera generación que apuntan a generar lecturas largas se clasifican (junto con otros métodos) como "High-throughput". De particular interés para este proyecto son los métodos de secuenciación por nanoporos.

2.1.1. Secuenciación por nanoporos

La secuenciación por nanoporos se enfoca en secuenciar de forma continua grandes segmentos de una molécula de ADN sin usar reacciones químicas complejas. Esto implica que la tecnología promete secuenciaciones de bajo costo, rápidas y capaces de analizarse en tiempo real (Wikipedia, 2023d).

En estos métodos, tenemos un dispositivo que contiene diversos poros (llamados nanoporos) por los cuales pasa una molécula de ADN. Cuando pasan ácidos nucleótidos por el nanoporo, modifican las corrientes eléctricas que existen en el poro. Cada base nitrogenada tiene un efecto diferente sobre dichas corrientes. Estas corrientes se pueden medir, generando una señal analógica que se convierte a una señal digital que codifica la secuencia de nucleótidos (Oxford Nanopore Technologies, 2023b). Para ser más específicos, dado un tipo de poro (por ejemplo R10.4.1), este tiene para cada *kmero*, una subsecuencia de largo k[1] de una molécula de ADN (Wikipedia, 2023c), una densidad de corriente eléctrica característica cuando dicho kmero está presente en el poro. La región del espacio donde un kmero tiene efecto significativo sobre la señal de secuenciación se denomina *región de captura*. La figura 2.2 muestra el proceso.

La densidad de corriente en la superficie del nanoporo depende del nanoporo y la composición de la molécula presente en la región de captura. Concretamente, **para un tipo de poro dado, cada kmero tiene un cambio característico** sobre la densidad de corriente eléctrica en el nanoporo cuando dicho kmero se encuentra en la región de captura. Esto significa que las estadísticas de la densidad de corriente eléctrica tienen distintos comportamientos cuando distintos kmeros están en dicha región. En particular, consideramos que **cada kmero tiene una media y desvio estándar de la densidad de corriente eléctrica característicos**.

Dentro de las compañías que fabrican dispositivos de secuenciación basados en esta tecnología se encuentra Oxford Nanopore Technologies (ONT).

[1] k depende del poro

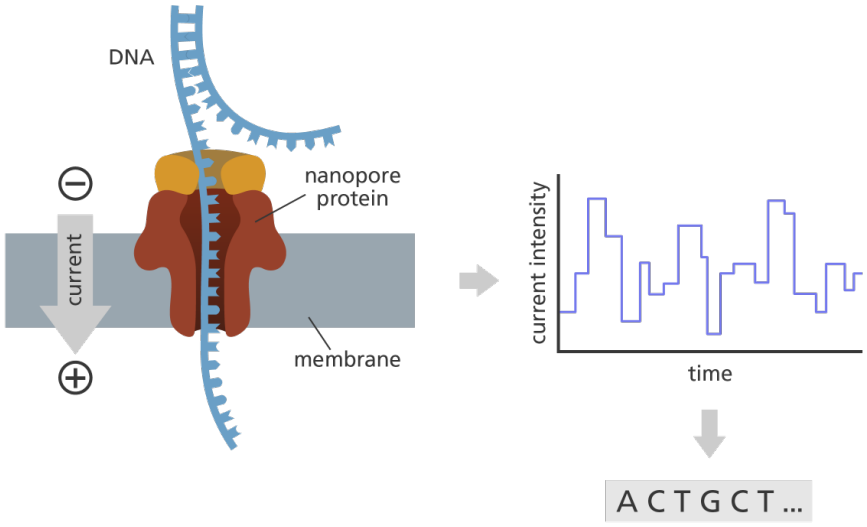

Figura 2.2: Secuenciación de ADN por nanoporos.
Figura extraída de (yourgenome, 2017).

2.1.2. Base calling

Base calling hace referencia al proceso de inferir a partir de un conjunto de datos físicos, como la señal eléctrica producida por los métodos de secuenciación por nanoporos, la secuencia de bases nitrogenadas en ese fragmento de ADN (Wikipedia, 2023a) (Oxford Nanopore Technologies, 2022b). Existen diversos métodos para realizar el proceso de base calling, dependiendo del tipo de señal física subyacente.

En la secuenciación por nanoporos, queremos inducir de una secuencia de mediciones de densidad de corriente eléctrica, una secuencia en el alfabeto $\{A, C, G, T\}$ potencialmente de largo distinto a la secuencia original. Las mediciones siempre se producen cuando determinado kmero está en la región de captura del nanoporo. Además, dado un poro concreto, cada kmero tiene una distribución de probabilidad característica para las mediciones de la señal. Podemos de generar una secuencia de ADN si a cada muestra de la señal le asignamos su kmero correspondiente basándonos en estadísticas de las muestras observadas. Además necesitamos determinar en qué puntos de la señal se produce un cambio de kmero. Con estos dos datos podemos determinar una secuencia de ADN que idealmente coincidirá con la secuencia original.

Para la secuenciación de nanoporos, un primer acercamiento al problema podría ser el siguiente: Cada kmero tiene una media característica. Dada una muestra de la señal, le asociamos el kmero cuya media característica sea más cercana al valor de dicha muestra.

En la práctica, el proceso es mucho más complejo. Por ejemplo, la señal tie-

ne ruido, y existen casos complejos de desambiguar. Supongamos que tenemos kmero de largo 4 ($k = 4$) y la secuencia de ADN es AAAAAAA. En este caso, todos los 4meros de la molécula de ADN tienen las mismas características estadísticas, lo que nos dificulta reconocer cuándo un 4mero termina y empieza el siguiente ("saltos" de kmero). Además no sabemos el largo de la molécula de ADN, por lo que determinar la verdadera secuencia de ADN en este caso no es trivial. Una mejor aproximación al problema usa el algoritmo de Viterbi (Jurafsky y Martin, 2009), que se basa en la técnica de Programación Dinámica.

La solución práctica al problema que ofrece los mejores resultados actualmente utiliza redes neuronales del tipo seq2seq (un método de aprendizaje automático que toma secuencias de datos de entrada y produce secuencias de datos de salida). Algunos ejemplos de estos productos son Guppy, Bonito, Dorado (Oxford Nanopore Technologies, 2023c) y Remora (Oxford Nanopore Technologies, 2023d) de ONT.

2.2. Conceptos generales de compresión de datos

Un compresor es un dispositivo o algoritmo que dado un conjunto de datos, los traduce a una cadena binaria. En la compresión sin pérdida (que es la que nos interesa en este proyecto), esta cadena binaria debe poder interpretarse por un algoritmo de decodificación para recuperar exactamente los datos originales. En este proceso decimos que un codificador le transmite datos comprimidos a un decodificador.

El objetivo principal de la compresión es que el número de bits utilizados para representar a los datos comprimidos sea lo más chico posible y en particular que sea menor que el número de bits usados en la representación original de los datos. Además, queremos que un compresor sea eficiente, tanto en términos de tiempo de ejecución como de consumo de memoria.

En general, pensamos que los datos a comprimir provienen de una fuente de información. En este proyecto pensamos que dicha fuente de información es un proceso estocástico discreto en el tiempo, sobre un alfabeto finito $\mathcal{X}$. Modelamos los datos a comprimir $(x_1, x_2, x_3, ...)$ como una realización de dicho proceso estocástico $X_1, X_2, X_3, ...$ La idea básica de la compresión de datos es asignar menos bits a aquellos símbolos que sean más probables.

Por otro lado, definimos un código como una función invertible $C : \mathcal{X} \to \{0,1\}^*$ que a cada valor de $\mathcal{X}$ le asigna una cadena binaria llamada palabra de código. Llamemos C^{-1} a la inversa de dicha función.

Notemos que C codifica **un** símbolo de la entrada, pero los datos pueden ser una secuencia de símbolos $x_1, x_2, x_3, ...$ del alfabeto $\mathcal{X}$. Entonces definimos la extensión de un código C como $C^* : \mathcal{X}^* \to \{0,1\}^*$ tal que

$$C^*(\epsilon) = \epsilon \tag{2.1}$$

$$C^*(x^{n-1}x_n) = C^*(x^{n-1})C(x_n), \tag{2.2}$$

donde, definimos ϵ como la cadena vacía y $x^n = x_1, x_2, ..., x_n$.

Es decir, C^* concatena las palabras de código para cada x_i. Queremos que C^* sea unívocamente decodificable. Esto es equivalente a que C^* sea invertible. Llamemos C^{*-1} a dicha función inversa.

Una forma de conseguir esto es con códigos de prefijos, también llamados códigos libres de prefijos, en los cuales ningna palabra de código es prefijo de otra.

Los códigos de prefijos son especialmente atractivos porque podemos decodificarlos de forma sencilla.

La desigualdad de Kraft (Cover y Thomas, 2006b) nos asegura que para todo código unívocamente decodificable existe un código de prefijos cuyas palabras de código tienen exactamente los mismos largos que el código unívocamente decodificable. Esto conlleva a que en la práctica, la gran mayoría de compresores usen códigos de prefijos, que es lo que hacemos en este trabajo.

Dado un código C y una distribución de probabilidad p sobre el alfabeto $\mathcal{X}$, definimos el largo de código promedio de C como

$$E_p[l(C)] = \sum_{x \in \mathcal{X}} p(x)l(x), \tag{2.3}$$

donde, $p(x)$ es la probabilidad de x y $l(x)$ es el largo de la palaba de código $C(x)$.

Queremos usar códigos de prefijo que minimicen la ecuación (2.3), para una distribución de probabilidad p dada.

Definamos la siguiente cantidad, denominada entropía, de una variable aleatoria discreta X sobre un alfabeto $\mathcal{X}$, con distribución de probabilidad p como:

$$H(X) = \sum_{x \in \mathcal{X}} -p(x)log_2(p(x)) = E_p[-log_2(p(x))], \tag{2.4}$$

donde, tomamos $p(x)log_2(p(x)) = 0$.

Esta cantidad mide, de manera intuitiva, la incertidumbre que tenemos sobre la variable aleatoria X. En otras palabras, evalúa qué tan bien podemos predecir el resultado de X. Una variable aleatoria más predecible será más fácil de comprimir. Si retomamos la idea de que para minimizar la ecuación 2.3 usamos palabras de código más cortas para los valores más probables, entonces, si la variable aleatoria X es altamente predecible, por ejemplo, porque concentra su probabilidad en un único valor, podremos usar un largo de código menor para el valor más probable. Esto nos permitirá reducir, en promedio, el largo de código esperado.

Para todo código unívocamente decodificable C, para una variable aleatoria X distribuida según p se cumple (Cover y Thomas, 2006c) la siguiente *cota de Shannon*:

$$H(X) \leq E_p[l(C)] \tag{2.5}$$

Para ver la intuición de que cuanto más predecible sea una fuente de información, más fácil es comprimirla, consideramos la figura 2.3. Dicha figura,

grafica una variable aleatoria X sobre un alfabeto de dos símbolos $\mathcal{X} = \{0, 1\}$, que toma el valor 0 con probabilidad θ y el valor 1 con probabilidad $1 - \theta$, la entropía de X en función de θ. Cerca de los bordes, donde estamos "seguros" del valor que va a tomar X, vemos que la entropía es pequeña, nos indica, por (2.5), que podemos potencialmente alcanzar largos de códigos esperados más cortos. Este resultado se generaliza para variables aleatorias sobre alfabetos finitos con más símbolos. En suma, las distribuciones de probabilidad que concentran su masa de probabilidad en unos pocos símbolos son más compresibles que aquellas que la dispersan sobre más símbolos del alfabeto.

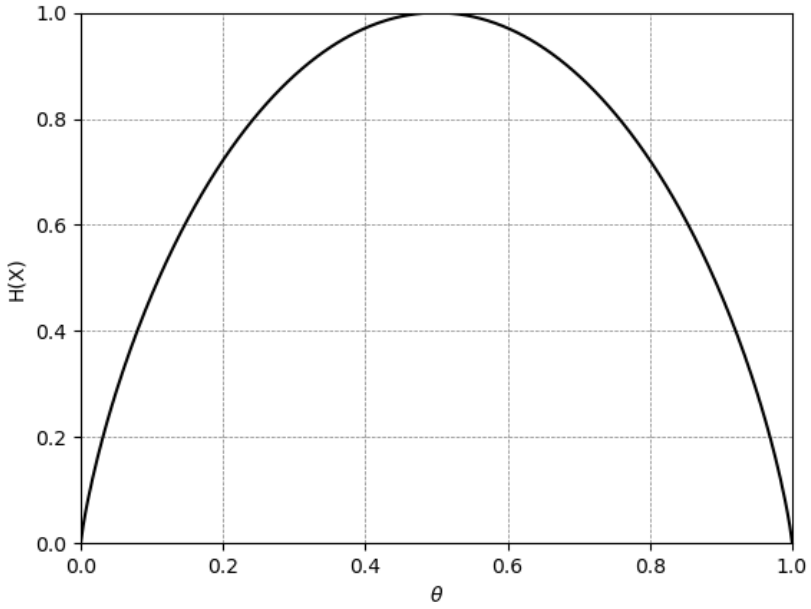

Figura 2.3: Entropía para una variable aleatoria X que toma el valor 0 con probabilidad θ y toma el valor 1 con probabilidad $1 - \theta$.

Notemos que hasta ahora hablamos principalmente de una variable aleatoria aislada, X, pero usualmente nuestros datos van a ser una secuencia de valores $x_1, x_2, ..., x_n$ sobre el alfabeto $\mathcal{X}$. En este caso, no nos importa tanto $p_i(x_i)$, la distribución del iésimo símbolo, sino que nos importa $p_i(x_i|x_{i-1}x_{i-2}...x_1)$. Es decir, queremos predecir el valor de x_i, conociendo los valores anteriores. Por razones de complejidad, tanto estadística como computacional, a menudo nos limitamos a modelos con memoria finita (Markovianos) en los cuales la distribución de x_i dados todos los símbolos anteriores depende en readidad solo de una cantidad finita de ellos. Conocidas las distribuciones de probabilidad condicionales, tenemos algoritmos como los códigos de Huffman y la codificación

aritmética que producen un código con largo esperado cercano a la cota de Shannon. Con todo esto podemos reducir el problema de compresión de datos al problema de obtener un modelo estadístico adecuado para la fuente de información que estamos observando y aplicar sobre dicho modelo un algoritmo de codificación adecuado.

2.2.1. Modelado estadístico

Modelar una fuente de datos es en general una tarea compleja. Usualmente lo que hacemos es, basándonos en conocimiento previo y observaciones empíricas, restringirnos a un modelo probabilístico con un número concreto de parámetros a definir. Por ejemplo, podemos modelar una fuente de información como un conjunto de variables aleatorias independientes e idénticamente distribuidas, donde cada una se distribuye según una distribución geométrica. Entonces, nuestro problema se reduce a calcular el parámetro característico de dicha distribución geométrica. Esto será calcular de algún modo, por ejemplo con estimadores de máxima verosimilitud, los parámetros de nuestro modelo estadístico.

Una alternativa para resolver este problema es la siguiente.

El compresor puede observar toda la secuencia de entrada x^n y calcular los parámetros estadísticos correspondientes. Luego, el compresor transmite una codificación para estos parámetros, seguido de la secuencia de valores de entrada codificados según un modelo con dichos parámetros. Esta solución es un *esquema en dos pasadas*. En esta solución vemos claramente que no sólo tenemos que usar bits para transmitir la secuencia a codificar, sino que tendremos que utilizar algunos bits al principio para transmitirle al decodificador los parámetros del modelo estadístico que estamos usando. Evidentemente cuanto más complejo sea nuestro modelo estadístico (es decir más parámetros tenga), más costoso será transmitirlo. Llamamos a este concepto ***"costo de modelo"***.

Otra alternativa, que es utilizada en este trabajo, es realizar la codificación de forma adaptativa. Esto es, cuando el compresor va a comprimir el símbolo x_i, sólo puede basarse en estadísticas de los símbolos $x_{i-1}, x_{i-2}, ..., x_1$. De esta forma, (y acordando un estado inicial "por convención" entre el codificador y el decodificador), el decodificador puede en cada paso de la descompresión reproducir exactamente las mismas estadísticas que el compresor calculó. Con esta estrategia nos evitamos tener que transmitir explícitamente los parámetros al descompresor.

No obstante, el costo de estos parámetros todavía debemos "pagarlo" de cierta forma. Es decir, cuando usamos codificación adaptativa, no nos estamos ahorrando el costo de modelo. Lo que sucede, es que al principio tenemos un mal modelo de la fuente de datos, lo que nos lleva a tener una mayor ineficiencia en la compresión al inicio. Conforme vemos más datos de la fuente, podemos ajustar mejor estas estadísticas y por tanto comprimir mejor la fuente. Esta ineficiencia al principio de la codificación es lo que entendemos en codificación adaptativa como costo de modelo. Conforme nuestro modelo es más complejo, necesitamos ajustar más parámetros y esto conlleva un mayor costo de modelo.

Podemos pensar que el largo de código total está dado por la suma de dos

términos: el costo de modelo más el costo de codificación de la fuente con ese modelo. El primero aumenta con la complejidad del modelo mientras que el segundo, en general, disminuye. Esta es una relación de compromiso que siempre debemos tener presente al diseñar un compresor.

2.2.2. Codificación predictiva

Existe el concepto de codificación predictiva. En este esquema de compresión, dada una secuencia $x_1, x_2, ..., x_n$ de símbolos a comprimir, no comprimimos $x_1, x_2, ..., x_n$ sino que comprimimos errores de predicción $e_1, e_2, ..., e_n$ que cometemos sobre dicha secuencia. Para este trabajo nos restringimos a una versión simplificada de dicho esquema donde la predicción de x_i sólo puede depender de a lo sumo m valores anteriores.

Concretamente, tenemos una función $p(x_{i-1}, x_{i-2}, ..., x_{i-m})$ que intenta predecir el valor de x_i. Esta función es conocida por el codificador y decodificador.

Definimos la secuencia $\hat{x_1}, \hat{x_2}, ..., \hat{x_n}$ como:

$$\hat{x_i} = p(x_{i-1}, x_{i-2}, ..., x_{i-m}), i > m,$$

$$\hat{x_1} = C_1(constante)$$

$$...$$

$$\hat{x_m} = C_m(constante)$$

Definimos la secuencia $e_1, e_2, ..., e_n$ como:

$$e_i = x_i - \hat{x_i}$$

El compresor codifica la secuencia de valores e_i en vez de codificar explícitamente la secuencia de valores x_i.

Como p es causal (la predicción de x_i sólo depende de valores anteriores), y como los valores iniciales $C_1, ..., C_m$ son conocidos tanto por el compresor como el descompresor, dadas las muestras $x_{i-1}, ..., x_{i-m}$ el descompresor puede calcular $\hat{x_i}$ y despejar $x_i = \hat{x_i} + e_i$, donde e_i es el error de predicción que el compresor le transmitió.

En este caso, la codificación predictiva tiende a concentrar los errores de predicción alrededor de un único valor (típicamente cero), lo cual resulta en una distribución con baja entropía para los errores de predicción.

2.3. Algoritmos de compresión

Esta sección presenta de forma simplificada los algoritmos de compresión, codificaciones y transformaciones que usamos recurrentemente a lo largo del trabajo. Estas incluyen: StreamVByte, mapeo de Rice, códigos de Golomb, Run Length Encoding (RLE), codificación aritmética, ZSTD y Vbz.

2.3.1. StreamVByte

StreamVByte es un algoritmo de codificación de enteros centrado en enteros chicos (Daniel Lemire, Nathan Kurz, Christoph Rupp, 2017). La idea central es no mandar bytes que no estoy utilizando. En general, si tenemos un entero (sin signo para simplificar) de 64 bits (8 bytes), y el rango promedio de nuestros enteros es de 0 a 2048, la mayoría de nuestros enteros van a usar el byte menos significativo, y potencialmente el segundo menos significativo; en particular casi no tendremos enteros que usen los bytes 3-8. La idea entonces es omitir los bytes 3-8, agregando alguna información que diga cuantos bytes fueron usados.

Los algoritmos de "Variable Byte" (en general los algoritmos que explotan la idea antes mencionada) pueden tener diversas variaciones en como codifican exactamente los datos. Además, usualmente se brindan diferentes implementaciones según el tamaño de los enteros a comprimir.

Para este trabajo, nos interesan particularmente enteros de 16 bits y la implementación que usan las bibliotecas para manipulación de archivos de secuenciación de nanoporos. Entonces, presentamos en el algoritmo 1 un pseudocódigo que representa la idea de alto nivel de StreamVByte16. Implementaciones reales usarán extensivamente instrucciones SIMD para acelerar el desempeño y tratarán con otros detalles de bajo nivel como el almacenamiento eficiente de un array de bits.

Algorithm 1 StreamVByte16

```
Require: entrada: Array[uint16]
Ensure: Array[bytes]
máscaras ← Array[bits]
datos ← Array[bytes]
for all x ∈ entrada do
    if OcupaUnByte(x) then
        máscaras.append(0)
        datos.append(ByteMenosSignificativo(x))
    else
        máscaras.append(1)
        datos.append(ByteMenosSignificativo(x))
        datos.append(ByteMásSignificativo(x))
    end if
end for
return concat(máscaras, datos)
```

2.3.2. Mapeo de Rice

Hay veces en compresión de datos que nos interesa convertir enteros con signo a enteros sin signo, sin perder información, porque determinado algoritmo de compresión funciona mejor con enteros sin signo. Supongamos enteros de

8 bits. Un casteo directo de enteros con signo a enteros sin signo del valor -1 lo transforma en 255 (representación en complemento a 2). Esto tiene como desventaja que enteros negativos con un valor absoluto chico se mapean a enteros muy grandes, y a veces nos interesa conservar el valor absoluto de dichos enteros. Una posible solución es el mapeo de Rice (a veces llamado zigzag encoding), dado por la siguiente función:

$$Rice(x) = \begin{cases} 2x, & \text{si } x \geq 0 \\ -2x - 1, & \text{en otro caso} \end{cases} \tag{2.6}$$

Esencialmente estamos alternando los valores como la siguiente secuencia: 0, -1, 1, -2, 2, ...

2.3.3. Códigos de Golomb

Los códigos de Golomb son códigos óptimos para distribuciones geométricas sobre los enteros no negativos. La gran ventaja de estos códigos es su gran eficiencia computacional (Gadiel Seroussi, 2023a).

Antes de definir los códigos de Golomb más formalmente definamos dos notaciones. $binario_m(x)$ es la representación más corta en binario del número x usando $\lfloor log\ m \rfloor$ o $\lceil log\ m \rceil$ bits. $unario(x)$ es la representación en unario de x. Esto es, emitimos x 1's seguidos de un 0.

Los códigos de Golomb son paramétricos en $k \in \mathbf{Z}^+$ y codifican un entero no negativo x como: $G_k(x) = binario_k(x\ mod\ k)\ unario(x\ div\ k)$.

Estos códigos son especialmente eficientes cuando nos restringimos a potencias de 2 ($k = 2^n$ para algún $n \in \mathbf{N}$) y son los que utilizamos en este proyecto.

2.3.4. Run Length Encoding

Run Length Encoding (RLE) es un algoritmo de compresión que funciona generando macrosímbolos cuando se repiten símbolos en la entrada (Gadiel Seroussi, 2023b). Supongamos que tenemos una secuencia de entrada $x_1, x_2, ...x_n$. Si tenemos una sección de muestras tales que $x_i = x_{i+1} = ... = x_{i+m}$ con $m \geq 1$, en vez de codificar $x_i, x_{i+i}, ..., x_{i+m}$ codificando cada símbolo de forma independiente, podemos: Emitir x_i, emitir una indicación (usualmente x_{i+1}) de que va a empezar una sección de símbolos iguales y luego emitir $m - 1$, el largo de run. Esto nos permite codificar m símbolos de forma eficiente.

Supongamos que codificamos cada x_i de forma independiente (es decir, el código de x^n es la concatenación de los códigos para x_i). Entonces, para cada x_i tenemos que emitir al menos un bit. Supongamos que X es una variable aleatoria tal que $x^n \sim X$. Si $0 \leq H(X) \ll 1$, entonces por el primer teorema de Shannon un código óptimo para X tiene un largo de código normalizado arbitrariamente cerca de $H(X)$, pero codificando símbolo por símbolo el largo de código normalizado siempre será mayor a 1. Para comprimir a menos de un bit por símbolo necesitamos agrupar los símbolos en macrosímbolos de 2 o más símbolos del alfabeto original.

RLE entonces nos permite generar macrosímbolos de una secuencia de símbolos que todos tenga el mismo valor. Si es común que cuando dos símbolos consecutivos tienen el mismo valor, entonces una porción larga del resto de los símbolos consecutivos tengan también el mismo valor, RLE nos permite codificar de muy buena forma una secuencia de m símbolos, lo que en condiciones adecuadas nos permite mejorar las tasas de compresión bajándola a menos de un bit por símbolo para determinadas secciones.

2.3.5. Codificación aritmética

Un compresor aritmético codifica una secuencia x^n secuencialmente, símbolo por símbolo, basándose en las distribuciones de probabilidad $P(X_i|x^{i-1})$ dadas por nuestro modelo.

El largo de la palabra de código generada por el codificador será cercano a $-logP(x_1, ..., x_n)$ que, por la cota de Shannon (2.5), es esencialmente óptimo. Usando un codificador aritmético, el problema de comprimir una fuente se reduce al de estimar secuencialmente $P(X_i|x^{i-1})$ para $i = 1, 2, ..., n$.

2.3.6. ZSTD

ZSTD es un algoritmo de compresión sin pérdida desarrollado por Yann Collet en Facebook (Wikipedia, 2023e), perteneciente a la familia de algoritmos Lempel-Ziv. En particular toma ideas de DEFLATE, un algoritmo de compresión sin pérdida, que en su corazón utiliza el algoritmo Lempel-Ziv 77 (LZ77) (Y. Collet; M. Kucherawy, Ed., 2021) (Yann Collet; Chip Turner, 2016).

Para evitar entrar en detalles del algoritmo ZSTD, nos centramos brevemente en su componente principal, LZ77.

LZ77 es un algoritmo de compresión sin pérdida del tipo *"sliding window"*. En esencia, LZ77 mantiene una ventana de los últimos n símbolos vistos en la secuencia de entrada. Codifica nuevas entradas como un puntero a esta ventana de símbolos anteriores más un largo dentro de esta. Por ejemplo, una parte de la secuencia de entrada puede comprimirse como: ir al símbolo visto 5 muestras atrás en la ventana y copiar 3 símbolos (Seroussi, 2021).

LZ77 es asintóticamente óptimo para procesos estacionarios y ergódicos, aunque su velocidad de convergencia es "lenta". Si los datos de entrada tuvieran largo infinito, entonces no habría problema en usar LZ77, pero dado que los datos son finitos, usualmente buscamos algoritmos de compresión que no sea universales para una familia de procesos tan grande (como la familia de todos los procesos estacionarios y ergódicos) pero que tengan velocidades de convergencia mayores.

2.3.7. Vbz

Vbz es un algoritmo para compresión de datos crudos de secuenciación por nanoporos desarrollado por ONT (Oxford Nanopore Technologies, 2022e). Se

basa en StreamVByte y ZSTD. Presentamos en el algortimo 2 un pseudocódigo de Vbz.

Algorithm 2 Vbz

Require: entrada: Array[uint16]
Ensure: Array[bytes]
1: intermedio ← StreamVByte(entrada)
2: **return** ZSTD(intermedio)

Capítulo 3

Formato y biblioteca de funciones para archivos POD5

En este capítulo explicamos el formato de archivos POD5 y su biblioteca asociada, así como su dependencia más importante: Apache Arrow. Primero explicamos Apache Arrow, en la sección 3.1, para establecer las bases que permiten entender la arquitectura de POD5 y sus decisiones de diseño. En la sección 3.2 explicamos específicamente el formato POD5, comentando la estructura de sus archivos, las tablas asociadas y la arquitectura de alto nivel de la biblioteca que nos permite manipular dichos archivos.

3.1. Apache Arrow

POD5 utiliza Apache Arrow IPC en sus bases (Oxford Nanopore Technologies, 2022h). Arrow es una biblioteca y un formato de archivo, que intentan unificar el acceso a datos y memoria de distintas fuentes. Arrow está diseñado de forma de ser eficiente para grandes volúmenes de datos, mediante el uso de tablas. Físicamente estas tablas serán representadas en POD5 como uno o más objetos de clase arrow::RecordBatch.

Es importante comprender el funcionamiento de Arrow pues además de usarse para serializar los datos en POD5, también se usa en una gran porción de la representación en memoria de estos datos. En lo que sigue, presentamos los aspectos de Apache Arrow más relevantes para este proyecto.

3.1.1. Beneficios de Arrow

El uso de Arrow en POD5 presenta varias ventajas. En particular Arrow facilita la recuperación de archivos cuya escritura se cortó de forma abrupta.

Además contiene algoritmos e implementaciones orientadas al acceso eficiente de datos y permite la compresión de buffers usando rutinas estándar. Arrow es un formato extensamente usado y estable.

Otros beneficios generales de Arrow incluyen su definición agnóstica del lenguaje de programación. Dentro de su especificación se incluyen estructuras en memoria, serialización de metadatos y protocolos para serializar y deserializar datos.

Un aspecto destacado de Arrow es su orientación a columnas, que permite obtener mejores resultados de compresión (a diferencia de comprimir fila por fila), para compresores integrados en la biblioteca de Arrow. Al tener contiguos todos los datos de una misma columna de una tabla, el algoritmo de compresión que se aplique podrá enfocarse en las estadísticas de una columna específica, lo que puede facilitar la convergencia del algoritmo en caso de que los datos de la columna estén correlacionados. La disposición de datos en Arrow se ilustra en la figura 3.1. Esta figura ilustra arriba, una vista lógica de los datos, organizados en tablas; a la izquierda un buffer de memoria "tradicional", donde todos los atributos de cada fila están consecutivos en memoria; a la derecha un buffer de memoria de Arrow, donde todos los datos de la primera columna están consecutivos en memoria, seguidos de todos los datos de la segunda columna, seguidos de la tercera. La figura muestra además que la disposición de datos en memoria de un buffer de Arrow permite el uso de instrucciones SIMD de manera más efectiva.

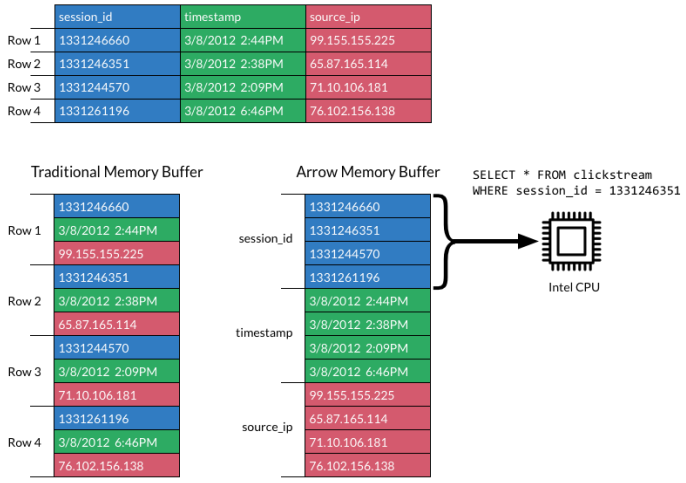

Figura 3.1: Estructura en columnas de Arrow y como esto facilita el uso de instrucciones SIMD. Figura tomada de (The Apache Software Foundation, 2016a).

Por último es relevante mencionar que la gran mayoría de objetos de Arrow

son inmutables, es decir, una vez construidos no se pueden modificar. Esta decisión de diseño tiene como fin facilitar las aplicaciones multi-threaded e impacta en cómo se escriben valores en este formato. En particular, un objeto de Arrow (particularmente un Array) se escribe incrementalmente (usando el patrón builder), agregando valores uno a uno, hasta que en un momento dicho objeto se consolida. Desde ese punto, el Array es inmutable y por tanto no se puede modificar.

El patrón builder es un patrón de diseño que facilita la creación de objetos complejos. Desde lo que nos es relevante en este proyecto, un builder permite construir incrementalmente un objeto. En particular, para construir un objeto usualmente llamamos al constructor de objetos de dicha clase, que es una invocación a **una** función. Cuando el objeto es complejo, este puede tener muchas configuraciones. El patrón builder nos permite configurar el objeto en **varias** llamadas, potencialmente separadas en el tiempo, y luego de que terminamos de configurarlo le delegamos al builder la responsabilidad de construir el objeto.

3.1.2. Especificación de Arrow

Como mencionamos anteriormente, los datos en Arrow se representan en tablas como la que se presenta en la figura 3.2. Estas tablas tienen un *Schema* (esquema de la tabla) que define sus columnas. Las columnas se definen mediante *Fields* (campos) dentro del Schema.

Un campo denota una columna particular de una tabla y contiene un tipo de datos, un nombre y, opcionalmente, metadatos.

Un esquema describe la estructura de un dataset bidimensional (una tabla) y está formado por una secuencia de campos. El esquema también puede tener opcionalmente metadatos comunes a todo el esquema.

Los datos se almacenan en un tipo particular de array específico a Arrow, es decir, cada columna de una tabla es un array de Arrow.

Una tabla se compone físicamente por uno o más *record batches*, que es un subconjunto de registros de la tabla. Para este proyecto podemos pensar que una tabla es una lista ordenada de record batches. Cada record batch tiene a su vez un esquema que es idéntico al esquema de la tabla. Notar que cuando serializamos un conjunto de record batches a un archivo, el esquema se coloca dos veces en dicho archivo, una vez al comienzo (header) y una copia redundante al final (footer), sin importar el número de record batches escritos.

En general la dinámica de lectura y manipulación de datos en Arrow (en lo que concierne al funcionamiento interno de POD5) es como sigue:

- Se abre el archivo a leer en un objeto f de clase io::ReadableFile.
- Se liga un objeto fr de clase ipc::RecordBatchFileReader (simplemente un lector de record batches) al archivo f.
- Se lee un record batch de f en un objeto r de clase arrow::RecordBatch.
- usando r se accede a una columna del record batch, que representa una porción continua de una columna específica del archivo.

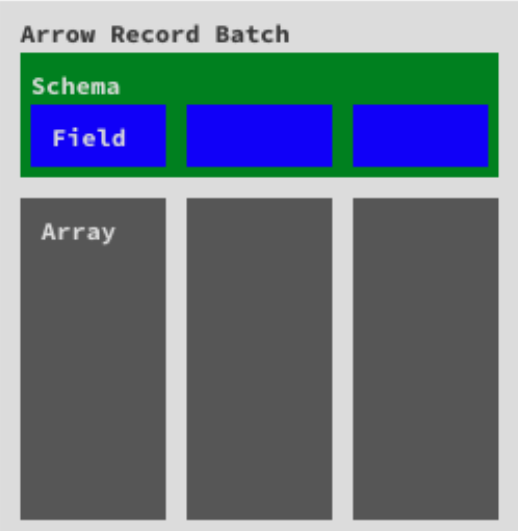

Figura 3.2: Estructura de un Arrow Record Batch. Figura tomada de (The Apache Software Foundation, 2016b).

- Se lee o manipula el array recuperado.

Por ejemplo, para acceder al registro 157 de la columna pore, de la tabla de reads se tiene que obtener el record batch que contiene el registro 157 de la tabla, para después pedirle a ese record batch que devuelva los datos asociados a la columna pore y dentro de esa columna (un array) indexar el registro 157.

Entonces tenemos que la clase arrow::RecordBatch es la unidad mínima de serialización en Arrow, y por tanto en POD5. Notamos además que una tabla puede estar formada por varios record batches y podemos escribir y leer las tablas de forma incremental, de a record batches.

Mencionamos antes que un record batch tiene como columnas Arrays **de Arrow**. En Arrow, un Array es un conjunto de uno o más buffers, que tienen un tipo de datos asociado y están continuos en memoria. Esto nos permite interpretar a un Array como una secuencia lógica de valores.

La forma usual de operar sobre un array en Arrow utiliza el patrón visitor. El patrón visitor es un patrón de diseño utilizado para implementar una operación sobre elementos de una estructura de objetos. El patrón invierte la responsabilidad de invocar a la operación representada como un visitor; la transfiere desde el cliente de la estructura de objetos a los elementos de dicha estructura. Supongamos que tenemos una estructura que puede tener un número entero o un número en punto flotante, pero no ambos y queremos implementar una operación que multiplique el número por dos. En este caso generaríamos un visitor PorDos, que dado un número entero lo sabe multiplicar por dos y dado un número en punto flotante lo sabe multiplicar por dos. Luego, un cliente

aplica el visitor sobre la estructura. Decimos que el visitor "visita" a la estructura y la estructura "acepta" al visitor. Cuando la estructura acepta al visitor, la estructura se fija si el valor que ella contiene es un entero o un número en punto flotante. Según esa condición, la estructura aplica el visitor sobre el valor correspondiente. Dado el valor, el visitor le aplica la operación al elemento. Lo importante es que el cliente de la estructura y el visitor no tienen que conocer cómo la estructura determina si tiene dentro un entero o un número en punto flotante. Además, el cliente se abstrae de las diferentes implementaciones que pueda tener la operación de multiplicar por dos para cada uno de los números.

Por su parte, un buffer es una región continua de datos. Arrow abstrae el concepto de buffer, de forma que nos independizamos del almacenamiento físico de ese buffer siempre que sus datos estén continuos. Así, un buffer puede ser una porción de memoria RAM, un archivo local, un archivo remoto o incluso parte de la memoria de una GPU, entre otros.

Un buffer es un contenedor mutable de datos sin tipo, es decir, en un buffer podemos agregar, quitar y modificar valores. En algún momento, el buffer se termina de escribir y se le asigna un tipo a los elementos que contiene. A partir de dicho buffer obtenemos un Array inmutable, que va a potencialmente ser una columna de un record batch, que a su vez puede ser una sección continua de datos de una tabla.

Por otro lado, Arrow define también su propio sistema de tipos. Por ejemplo, los Arrays en Arrow tienen determinado tipo, supongamos enteros de 16 bits, y esos tipos están manejados por Arrow. Un usuario puede extender Arrow agregando nuevos tipos. En particular, POD5 genera tipos que modelan la señal de secuenciación comprimida, teniendo un tipo por método de compresión soportado.

Los tipos de datos en Arrow se pueden representar de tres formas: usando un enumerado, usando una subclase concreta de la clase arrow::DataType, o usando instancias de arrow::DataType.

3.2. POD5

Como mencionamos anteriormente, POD5 es el formato que se utiliza actualmente para almacenar datos de secuenciación de nanoporos. Este formato guarda tres estructuras tabulares que contienen los datos: las tablas de Read, Signal y RunInfo, todas representadas como un conjunto de record batches de Arrow.

Cada registro de la tabla RunInfo identifica un *experimento* de secuenciación. Un experimento se refiere a la preparación de una flowcell y su uso, desde que empieza el proceso de secuenciación hasta que termina. Algunos datos que se guardan en estos registros pueden ser: el nombre del preparado insertado, el nombre del dispositivo secuenciador y su tasa de muestreo. Un archivo POD5 puede tener varios experimentos.

Asociado a cada experimento tenemos una o más lecturas del aparato secuenciador, llamadas *reads*. Cada registro de la tabla Read tiene información de

un read específico, es decir, un fragmento entero de una hebra de ADN pasando por un nanoporo. Dentro de estos datos tenemos, por ejemplo, el tipo de poro usado y parámetros del conversor analógico-digital.

Además sabemos que cada read tiene exactamente **una** señal de secuenciación asociada. Esta señal se guarda en los registros de la tabla Signal. En particular, la señal de secuenciación se cuantiza y muestrea para digitalizarla a una secuencia de enteros con signo de 16 bits. La unidad de medida de estos valores la denominamos DACs, que corresponde con "Data acquisition values" (Oxford Nanopore Technologies, 2023e).

Por otro lado, **en POD5** un read tiene uno o más registros de señal asociados. Esto se debe a que internamente la señal de un read se fragmenta en bloques, denominados chunks, de un largo máximo, fijado al momento de crear el archivo. Como la secuenciación por nanoporos tiende a generar lecturas muy largas, fragmentar la señal en chunks facilita el acceso aleatorio a diferentes partes de la señal.

Para ilustrar este proceso de fragmentación, consideremos un ejemplo: si tenemos un read con una señal que tiene 990 muestras pero decidimos trabajar con un tamaño de chunk de 100 muestras, entonces internamente se fragmenta la señal en 10 registros, los primeros 9 conteniendo 100 muestras consecutivas y el último las 90 restantes.

Además, la tabla RunInfo apunta a la tabla Read que a su vez apunta a la tabla Signal. Cada experimento está asociado a varios reads, pero cada read proviene de exactamente un solo experimento. Por otro lado, cada read tiene una o más señales, pero cada señal pertenece a un solo read. La figura 3.3 muestra informalmente estas relaciones, mientras que la figura 3.4 explicita las claves y claves foráneas de cada tabla.

Notar que las tablas están estructuradas de forma de reducir la duplicación de datos. Además algunos datos (como el tipo de poro) no se almacenan directamente, sino que se representan como un índice a un diccionario contenido en el archivo, lo que ayuda a evitar más aún la duplicación de datos.

Las tablas tienen el esquema que se muestra en la figura 3.5. Los valores en azul son claves (primarias o foráneas) mientras que los valores en verde representan datos. Explicamos los atributos más relevantes a continuación. Para más detalles ver (Oxford Nanopore Technologies, 2022g).

Atributos de cada registro de la tabla Signal

- read_id: Es un GUID (Globally-Unique IDentifier) del read que produjo la señal de este registro.
- signal: Secuencia de enteros de 16 bits que representa la señal digitalizada.
- samples: Número de muestras de la señal en este registro.

Atributos de cada registro de la tabla Reads

- read_id: GUID que identifica al read en este registro.

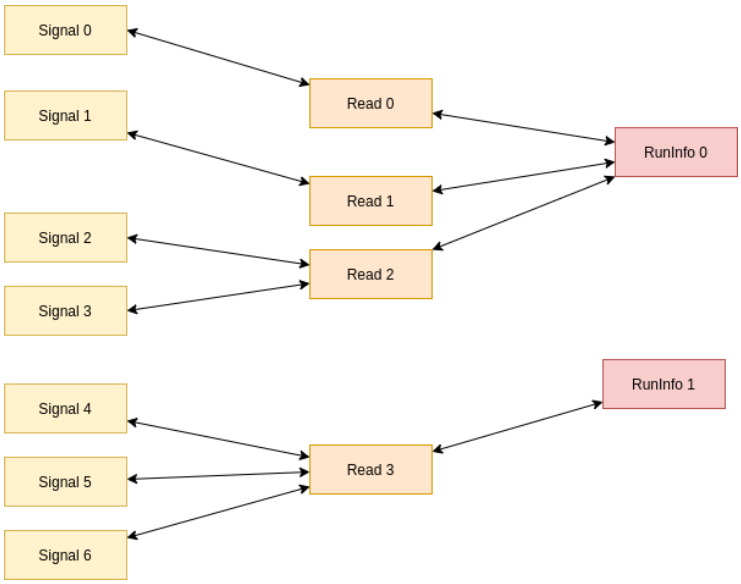

Figura 3.3: Relación entre los diferentes registros almacenados en un archivo POD5.

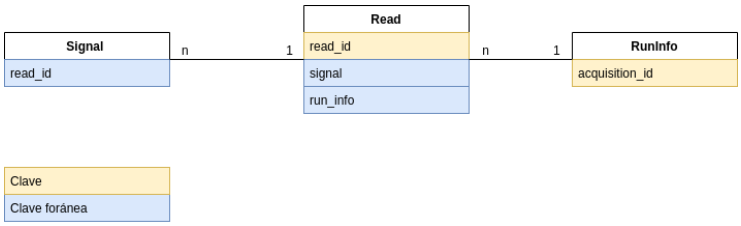

Figura 3.4: Claves y claves foráneas de las tablas correspondientes.

- signal: Lista de índices de las filas en la tabla Signal que corresponden a chunks de la señal producida por este read. La lista está ordenada de forma que al recorrerla se reconstruye la señal completa a partir de los chunks.
- pore_type: Nombre del poro presente.
- num_samples: Número total de muestras en el read, calculado como la suma de todos los campos samples de la tabla Signal que tienen el mismo read_id.

Atributos de cada registro de la tabla RunInfo

- acquisition_id: Un GUID para el run descripto.

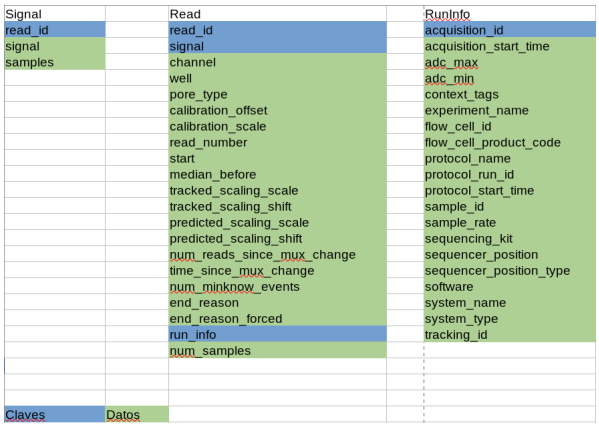

Figura 3.5: Tablas con sus atributos.

- adc_max: Máximo valor de la conversión analógico digital. Representa valores fijos según el hardware del dispositivo secuenciador.
- adc_min: Mínimo valor de la conversión analógico digital. Representa valores fijos según el hardware del dispositivo secuenciador.
- sample_rate: Tasa de muestreo de la conversión analógico digital utilizada en el experimento.

3.2.1. Arquitectura de alto nivel de la biblioteca POD5

La figura 3.6 muestra la organización de más alto nivel de POD5, que tiene componentes en C, C++ y Python.

El componente "biblioteca" implementa las funcionalidades provistas sobre el formato POD5, como puede ser lectura, escritura y recuperación de archivos. Es el componente principal donde se implementan los mecanismos de compresión, descompresión, serialización y deserialización del formato. En este componente se concentran la mayoría de los cambios que realizamos en el proyecto.

Luego, tenemos el componente "API de C", que contiene algunas funciones para acceder a las funcionalidades provistas por la biblioteca en C++, proporcionando una API completamente en C y añadiendo una capa adicional de abstracción. En la subsección 3.2.2 presentamos brevemente esta API porque hacemos algunas modificaciones en el proyecto.

Por otro lado, el componente "API de Python" expone una API para el manejo del formato de archivos POD5 en Python, usando las funcionalidades de la biblioteca base. Este componente no es modificado en el marco del proyecto.

Los componentes "Aplicaciones en Python" y "Aplicaciones en C" interactuan con POD5 mediante las APIs expuestas. Esta estrategia permite centralizar las modificaciones de funcionalidad en la biblioteca de C++, además de mantener la integridad de las invariantes del dominio.

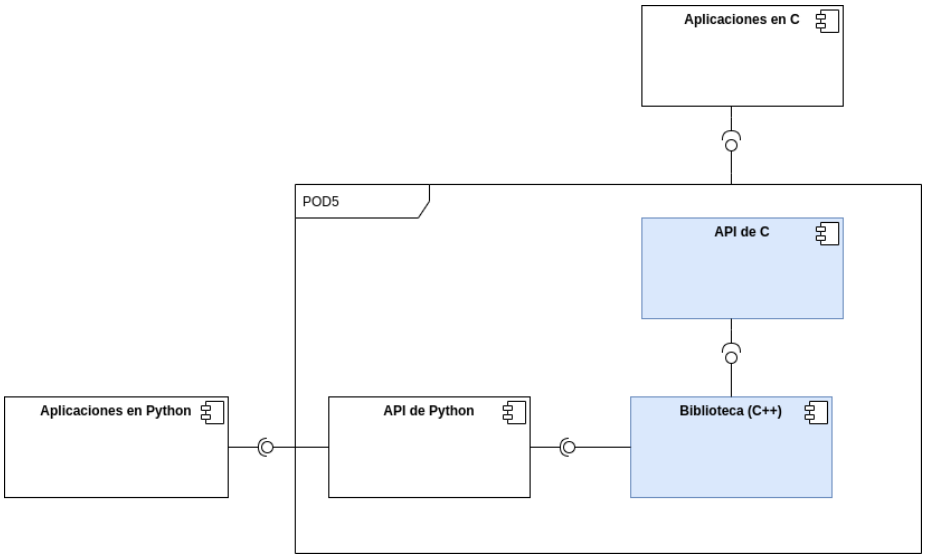

Figura 3.6: Estructura base de POD5.

3.2.2. API de C

La API de C expone en C las funcionalidades principales de POD5. Presenta un modelo de datos que se acerca más a lo que un usuario desearía utilizar respecto a las funciones internas de la biblioteca, en lugar de mostrar directamente las tres tablas que componen el formato POD5. En este modelo, un usuario puede considerar un read como una unidad, aunque conceptualmente incluye datos de la tabla Reads junto con la señal de secuenciación.

En esta subsección enumeramos algunas de las funciones más importantes de dicha API y cuando sea relevante damos algún detalle de sus parámetros o funcionamiento.

Funciones de inicialización (abrir archivos):

- pod5_open_file()
- pod5_create_file(writer_options)

Funciones de lectura:

- pod5_get_read_batch(reader)
- pod5_get_file_run_info(reader)
- pod5_get_read_complete_signal(record batch)
- pod5_get_end_reason(record batch)
- pod5_get_pore(record batch)

Funciones de escritura:

- pod5_add_reads_data()
- pod5_add_run_info()
- pod5_add_pore()

Para abrir un archivo para escritura usamos la función pod5_create_file, que recibe como parámetro un conjunto de opciones, entre las que se encuentra el método de compresión que vamos a usar.

Notamos que según la API expuesta, la señal, el end_reason, el tipo de poro y los datos de un experimento (run info) se pueden obtener independientemente uno de otros. Además, para obtener la señal necesitamos un record batch.

Por otro lado, si bien tenemos funciones para agregar tipos de poro (pod5_add_pore) y experimentos (pod5_add_run_info) separadas de la escritura de reads, POD5 fuerza implícitamente un protocolo de acceso que exige que al momento de escribir un read, tenemos que ya haber escrito el tipo de poro y experimento a los que hace referencia el read. Detallamos más este punto en la subseccion D.1.2. Además, cuando un cliente quiere escribir datos sobre un read, mediante la función pod5_add_reads_data(), POD5 le exige que pase en conjunto los datos del read junto con su señal de secuenciación.

Realizamos muy pocas modificaciones a esta API como parte de este proyecto. Estas modificaciones no alteran la API externa para un cliente. De las funciones nombradas sólo modificamos a pod5_create_file, para que admita como opcion un nuevo método de compresión, PGNano5.

3.3. Representación de POD5 mediante objetos Arrow

Como dijimos anteriormente, POD5 se construye por arriba de Arrow. Esta sección explica cómo se implementan los diferentes elementos de POD5 utilizando primitivas de Arrow. Esto es importante porque los objetos que presentamos en esta sección son los objetos que tenemos que modificar para generar un nuevo método de compresión.

3.3.1. El record batch en POD5

POD5 se basa en las estructuras tabulares de Arrow, pero adaptándolas al dominio de secuenciación de ADN. De esta forma, tenemos estructuras como SignalTableRecordBatch, ReadTableRecordBatch y RunInfoTableRecordBatch. Todas estas estructuras se basan en un record batch, pero cada una agrega semántica específica para el tipo de registros guardados.

En particular, las clases antes mencionadas heredan de la clase TableRecordBatch que expande el record batch con métodos que se usan recurrentemente

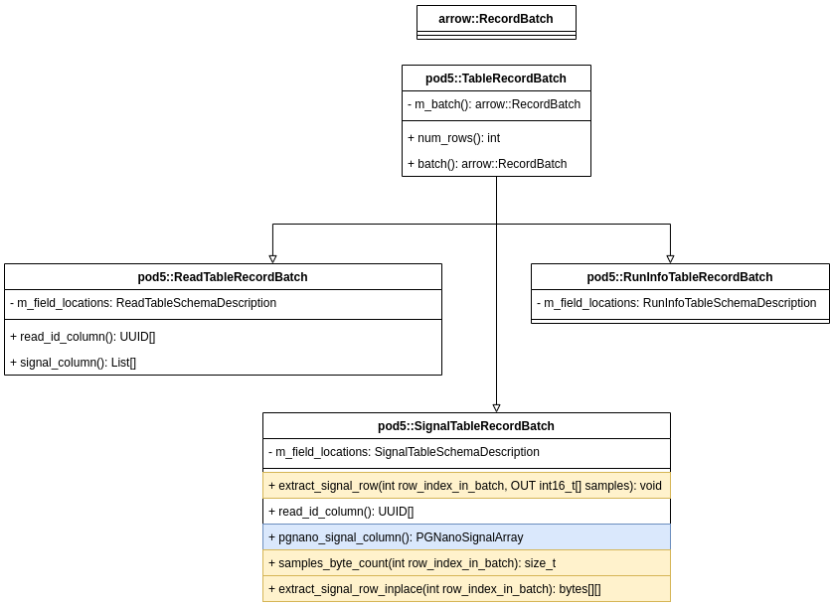

Figura 3.7: Diagrama de clases simplificado de los diferentes tipos de batch. En amarillo: métodos modificados en el proyecto. En azul: métodos agregados en el proyecto.

en POD5. El diagrama de clases de la figura 3.7 ilustra los diferentes tipos de record batch indicando con un color los métodos modificados en este proyecto.

Aquí hay varias cosas para notar. En primera instancia la clase TableRecordBatch encapsula un record batch de Arrow. Este TableRecordBatch luego se extiende para especializarse en cada una de las tablas existentes. Por su parte, cada una de estas subclases expone en métodos separados las columnas "principales" de ese tipo de registro, lo que simplifica el manejo de los datos al incorporar información adicional de tipado y nombres pertinentes al dominio. Por ejemplo, el método read_id_column de un objeto de clase SignalTableRecordBatch recupera el read_id de cada registro contenido en ese record batch.

Por otro lado, cada tipo de TableRecordBatch expone funcionalidades particulares. Por ejemplo, SignalTableRecordBatch es quien se encarga de descomprimir la señal, mediante el método extract_signal_row.

Otro detalle importante de POD5 relacionado al uso de record batches, es que las tablas están formadas por uno o más record batches de tamaño acotado. Es decir, una tabla en POD5 es una colección de record batches manejada por POD5 donde cada record batch tiene un tamaño máximo preestablecido. Si tenemos 1099 tuplas en un archivo POD5 y definimos que cada record batch tiene a lo sumo 100 registros, la biblioteca genera 10 record batches con 100 registros cada uno, y un último record batch con 99 registros. Todas estas características permiten un acceso aleatorio eficiente a nivel de record batch en POD5.

El record batch para C Desde la API de C, el concepto de record batch es distinto a como lo tratamos internamente en la biblioteca. Esto será importante al momento de implementar nuestras modificaciones a la biblioteca. En la subsección D.1.2, entramos más en detalle, pero es bueno tener una primera idea de las diferencias.

Desde esta API, un cliente sólo conoce un tipo de record batch, Pod5ReadRecordBatch, y con ese record batch y un lector de archivos de tipo Pod5FileReader_t puede leer datos. Para escribir, el cliente no necesita formar record batches antes de pasárselos a la biblioteca. El cliente sólo necesita tener un escritor de archivos POD5 de tipo Pod5FileWriter_t y **seguir un protocolo de escritura** (por ejemplo, los tipos de poros deben escribirse antes que escribir la información de un read). Este protoloco de escritura, junto al modelo de datos de la API de C generan un conjunto de restricciones al cliente. Estas restricciones permiten que nuestras modificaciones a POD5 requieran solo cambios menores, que logran mantener las responsabilidades asignadas originalmente a cada componente. Además, nos permite conservar los atributos de calidad más importantes de la biblioteca: el acceso aleatorio y el acceso en paralelo a datos.

3.3.2. Builders

En POD5, la unidad mínima de serialización es un record batch. Este formato permite operar en línea, lo que significa que los clientes pueden pasar señales en subconjuntos o individualmente, en lugar de todas a la vez. Esto es útil, por ejemplo, en dispositivos de secuenciación que invocan repetidamente a la biblioteca a medida que se detectan nuevas señales.

Para facilitar esto, POD5 permite la construcción incremental de un record batch a través del uso del patrón de diseño "builder", especialmente en la columna Signal. Un "builder", en el contexto que nos interesa, almacena señales hasta que se alcanza el tamaño de record batch deseado. Una vez alcanzado este tamaño, se genera un record batch y se reinicia el estado interno del "builder", permitiendo la adición de nuevas señales.

Además, los builders en POD5 son quienes **determinan internamente el método de compresión utilizado**. En particular POD5 tiene las clases UncompressedSignalBuilder y VbzSignalBuilder para construir record batches sin comprimir o comprimidos con Vbz, respectivamente.

Una vez que se abre un archivo POD5 para escritura y se indica el tipo de compresión deseada, el tipo de builder queda definido hasta que se cierre dicho archivo, y con ello el tipo de compresión usado.

3.3.3. Visitors

Como mencionamos anteriormente, la manipulación de datos en Arrow y por tanto internamente en POD5, se hace mediante el patrón visitor. En particular tenemos los visitors ilustrados en la figura 3.8.

Cada visitor implementa una operación particular sobre un conjunto de señales todavía en construcción. Nos interesa especialmente el visitor

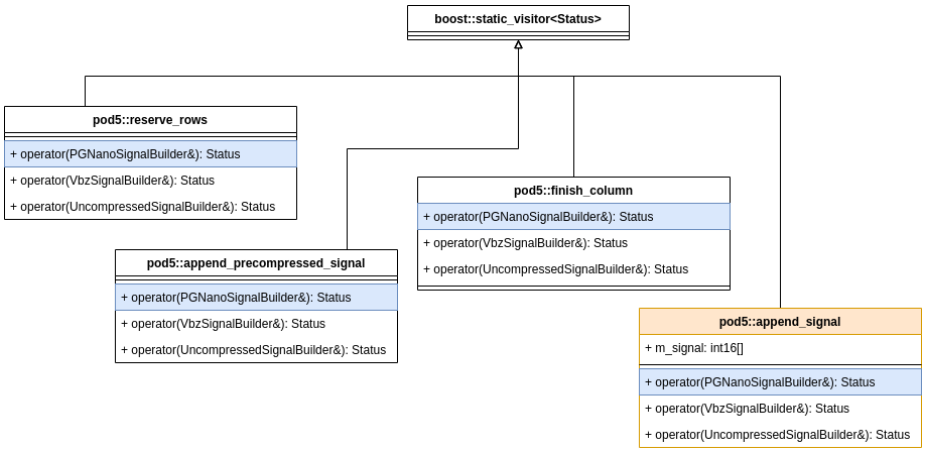

Figura 3.8: Diferentes visitors en POD5. En azul: métodos agregados en el proyecto. En naranja: clase append_signal, el visitor donde se producen los cambios más relevantes.

append_signal. Este visitor toma una señal sin comprimir por parte del cliente y la comprime según el método de compresión indicado por el tipo del builder. Como podemos ver de la figura, esto se hace implementado un método operator()(T x) para cada tipo de señal.

3.3.4. Representación de los diferentes métodos de compresión

El tipo de señal tiene dos representaciones posibles en POD5.

Enumerado La primera corresponde a un enumerado SignalType que contiene los valores SignalType = {UncompressedSignal, VbzSignal}.

Representación como tipo de Arrow La otra forma de representar a la señal es instanciando clases hijas de la clase arrow::ExtensionType. En Arrow tenemos que declarar los tipos de extensión antes de poder usarlos, y para declararlos necesitamos instanciar estas clases.

En POD5 se registra el tipo VbzSignalType que usa detrás el tipo arrow::LargeBinary (una secuencia de bytes de largo arbitrario) y representa una señal concreta. El tipo VbzSignalArray representa un array de señales y se basa en arrow::LargeBinaryArray (un array donde cada elemento es una secuencia de bytes de largo arbitrario).

Estas definiciones de tipo permiten declarar metadatos asociados y permiten trabajar con un tipo más adecuado al dominio que el tipo genérico de Arrow, además de facilitar el acceso a datos. En particular, ahora podemos hablar de

señales eléctricas de secuenciación comprimidas con Vbz y no tratar todo como una secuencia de enteros de 16 bits con signo o una secuencia de bytes.

El tipo de señal se utiliza junto al patrón visitor para determinar qué algoritmos ejecutar en el transcurso del programa. En particular **el tipo de señal indica el método de compresión a utilizar**.

3.3.5. Jerarquía de escritura

Para la escritura de un archivo POD5 hay más de un objeto involucrado. Estos objetos se estructuran en un árbol jerárquico donde los hijos tienen responsabilidades más específicas que los padres, quienes coordinan las actividades de los hijos.

Para empezar, cuando abrimos un archivo POD5 para escritura, debemos generar un objeto de clase FileWriter. Este objeto contiene a su vez objetos de clase SignalTableWriter, ReadTableWriter y RunInfoTableWriter que especializan su comportamiento en escribir la tabla correspondiente. Por su parte, el FileWriter toma las entradas del código cliente y coordina estos tres writers para implementar la operación deseada (delegándoles responsabilidades), como por ejemplo agregar un nuevo read con su señal de secuenciación correspondiente.

Por otro lado, los writers, y en particular el SignalTableWriter manipularán SignalTableRecordBatches, o sus análogos para las respectivas tablas. Esto implica que al momento de escribir un archivo POD5, es el SignalTableWriter (junto al visitor append_signal) quien comprime la señal. Cuando el SignalTableWriter tiene acumulado en el builder tantas filas como se haya especifiado en el tamaño de record batch, el SignalTableWriter encapsula la señal en un SignalTableRecordBatch para después escribirla. Los visitors aplicados sobre el builder desacoplan al SignalTableWriter de las responsabilidades específicas para cada tipo de señal.

3.3.6. Jerarquía de lectura

En la lectura de archivos POD5, al igual que en la escritura, tenemos una jerarquía de objetos interactuando. En la cima de esta jerarquía se encuentra un objeto de la clase FileReader. Por cada archivo que se abre para lectura, se crea un FileReader. Este FileReader se compone de los siguientes readers: SignalTableReader, ReadTableReader y RunInfoTableReader, junto a otros auxiliares. Cada uno de estos readers se especializa en una de las tablas presentes en POD5.

Dada una invocación a un método desde la API externa el FileReader delega y coordina operaciones a uno o más de estos readers. Cada uno de estos readers manipulará los tipos de record batch correspondientes, de forma análoga a la jerarquía de escritura.

En caso de la lectura, es el SignalTableRecordBatch quien descomprime la señal.

Capítulo 4

Implementación de compresores para archivos POD5

Este capítulo presenta dos compresores distintos para POD5: PGNano5, que es una modificación de la biblioteca de POD5 y PGNanoS que permite comprimir archivos POD5 por fuera de la biblioteca. Estos compresores tienen su implementación en un componente PGNanoLib, de forma de compartir código común a ambos compresores. Además presentamos el algoritmo de compresión sobre el cual se basan estos compresores: PGNanoAlgo.

Principalmente nos interesa PGNano5, porque modificar la biblioteca para mejorar la compresión produce en general una solución más completa, además de facilitar su adopción por terceros y facilitar la inserción de los cambios en el ecosistema de tecnologías existentes.

La primera sección presenta a PGNanoAlgo, comenzando primero con un análisis de la señal de secuenciación que motiva las decisiones de diseño tomadas. La segunda sección presenta a PGNanoLib. La tercera sección presenta a PGNano5. La cuarta sección presenta a PGNanoS.

4.1. PGNanoAlgo

PGNanoAlgo es un algoritmo de compresión de datos crudos de secuenciación de ADN específicamente diseñado para explotar las características de las señales crudas de secuenciación por nanoporos. Primero explicamos características relevantes de la señal y luego describimos a PGNanoAlgo basándonos en dichas características.

4.1.1. Análisis de la señal de secuenciación

Recordemos que una señal de secuenciación es una secuencia $x_1, x_2, ..., x_n$ de enteros con signo de 16 bits. La figura 4.1 da un ejemplo de un fragmento de una señal. En esta figura tenemos en el eje x el número de muestreos que el aparato secuenciador ha tomado hasta este momento, en el read actual. En el eje y tenemos la señal de secuenciación cuantizada. Recordemos que la unidad de medida de dicha señal es por definición un DAC (Data Acquisition value), que representa la mínima medición que podemos obtener del aparato secuenciador.

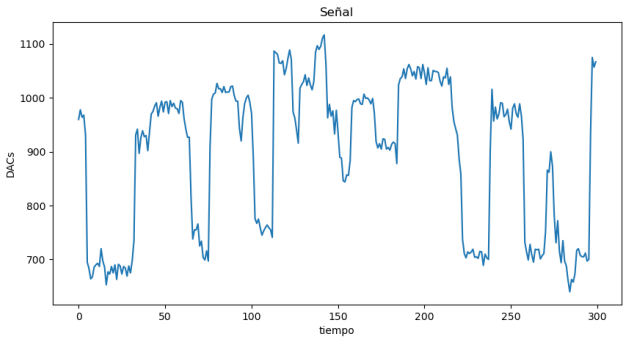

Figura 4.1: Un ejemplo de un fragmento una señal de secuenciación.

De la sección 2.1.1, sabemos que la señal de secuenciación tiene estadísticas características según el kmero presente en la región de captura del nanoporo. En particular, cada kmero tiene una media y un desvío estándar característicos distintos. En la figura 4.1 vemos que la señal alterna entre regiones donde tiene el mismo comportamiento.

Lo que se observa en la figura va en línea con la idea de que cada elemento de la entrada x_i es producido cuando un kmero K está presente en la región de captura del nanoporo. Entonces podemos asignar a cada muestra de la señal un kmero K correspondiente. Como el movimiento de los kmeros por el nanoporo es un fenómeno progresivo y ordenado en el tiempo, es razonable dividir la señal de secuenciación $x_1, x_2, ..., x_n$ en subsecuencias $x_i, x_{i+1}, ..., x_{i+m}$, donde todas las muestras de una misma subsecuencia provienen del mismo kmero.

Si retomamos la intuición sobre base calling que presentamos en la sección 2.1.2, los dos puntos anteriores nos dan la intuición de que la señal está dividida en subsecuencias donde los valores x_i tienen comportamientos similares que llamamos **subsequencias equivalentes**. En particular, podemos pensar que todos los valores x_i pertenecientes a una de estas subsecuencias tienen la misma distribución de probabilidad, que según ONT es una normal (discretizada). Si retomamos la figura 4.1, esta nos indica que la idea de subsecuencias equivalentes parece razonable.

Considerando que todos los valores dentro de una misma subsecuencia tienen una media característica, si tomamos la diferencia entre dos muestras consecuti-

vas, entonces para cada una de estas subsecuencias tendríamos que, excepto en las transiciones entre subsecuencias, las diferencias de muestras consecutivas se distribuyen centradamente alrededor de cero, independiente del kmero presente en la región de captura. Además, si pensamos que el kmero está presente un período de tiempo relativamente largo en el nanoporo[1], es razonable suponer que las transiciones entre subsecuencias se encuentren separadas en el tiempo.

Entonces, analizamos la secuencia $d_1, d_2, ..., d_n$ donde

$$d_1 = x_1,$$
$$d_i = x_i - x_{i-1}.$$

Es decir, $\{d_i\}$ es la secuencia de restas de valores consecutivos de $\{x_i\}$. Notar que calcular la secuencia $\{d_i\}$ es equivalente a un esquema de codificación predictiva (ver 2.2.2) con $\hat{x}_n = x_{n-1}$, donde $\{d_i\}$ es el error de predicción. La figura 4.2 ilustra un ejemplo de señal alineada a la respectiva diferencia de muestras consecutivas. En esta figura podemos ver nuevamente las intuiciones antes presentadas, en particular notando que el error de predicción se centra en cero.

En la figura 4.3 graficamos para un conjunto de señales de ejemplo, el histograma de los errores de predicción. Esto refuerza la intuición antes presentada de que la distribución de la resta de muestras consecutivas se centra en cero.

Por otro lado, que los valores de las diferencias de muestras consecutivas se concentren en 0 independientemente del kmero asociado produce un fenómeno de ***"independización de la media"***. Es decir, en las diferencias entre valores consecutivos ya no es relevante la media característica del kmero asociado, excepto en "saltos"; lugares donde tenemos una transición de un kmero a otro. Además, podemos interpretar que cuando una diferencia de dos muestras consecutivas toma un valor muy alto, probablemente estemos en presencia de un salto de kmero.

4.1.2. Descripción del algoritmo

PGNanoAlgo es un algoritmo de compresión de datos crudos de secuenciación de ADN por nanoporos que se basa en las propiedades estadísticas de la señal presentadas en la sección 4.1.1. PGNanoAlgo se presenta en el algoritmo 3.

La primera línea del algoritmo almacena cero en la variable señal_anterior. En particular, PGNanoAlgo es un algoritmo que comprime errores de predicción sobre la señal usando el predictor $\hat{x}_n = x_{n-1}$, que equivale a comprimir la diferencia de muestras consecutivas. La variable señal_anterior guarda la historia de la señal que necesitamos para el predictor. Al inicializar el algoritmo esta variable toma el valor cero por convención.

En la sección 2.3.5 vimos que dado un modelo estadístico para los símbolos a comprimir, un codificador aritmético produce un código cercano al óptimo.

[1] El valor medio del tiempo que está un kmero presente en el poro depende de varios factores como la tasa de secuenciación del aparato.

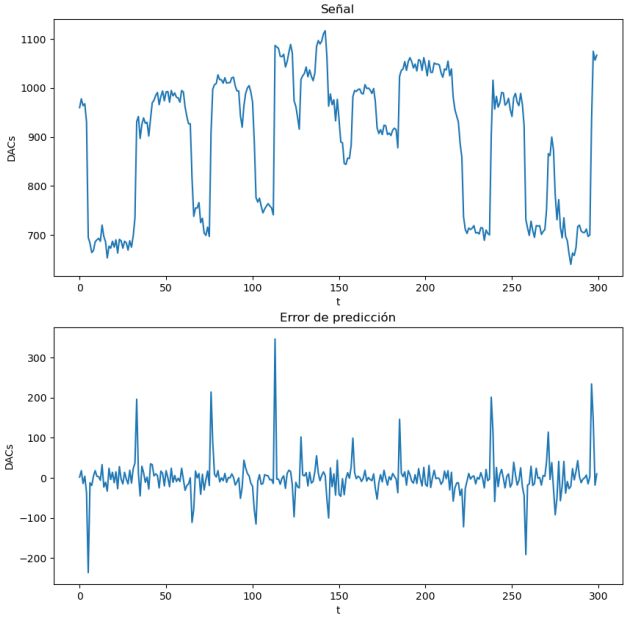

Figura 4.2: Una sección de secuencia de diferencias muestras consecutivas alineada con su respectiva señal.

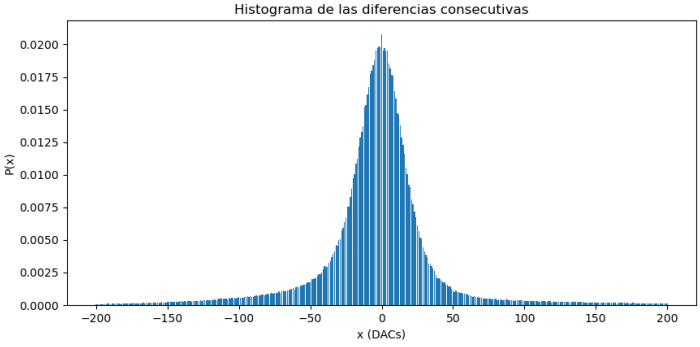

Figura 4.3: Histograma normalizado de la diferencia de muestras consecutivas en el rango [-200, 200].

PGNanoAlgo usa un codificador aritmético para esto. En particular, PGNanoAlgo codifica por separado el byte alto y el byte bajo de la secuencia $\{Rice(d_i)\}$. Recordar que el mapa de Rice es una transformación que lleva enteros con signo

Algorithm 3 PGNanoAlgo

Require: señal: Array[int16]
Ensure: salida: Array[bytes]
1: *señal_anterior* ← 0
2: *salida* ← new Array[bytes]
3: *modelo_parte_alta* ← new Modelo(256)
4: *modelo_parte_baja_0* ← new Modelo(256)
5: *modelo_parte_baja_1* ← new Modelo(256)
6: **for** x **in** *señal* **do**
7: $error \leftarrow x - señal\_anterior$
8: **if** $error < 0$ **then**
9: $simbolo \leftarrow -2 \times error - 1$
10: **else**
11: $simbolo \leftarrow 2 \times error$
12: **end if**
13: $byte\_alto, byte\_bajo \leftarrow$ separarBytes($simbolo$)
14: *salida*.append(codificadorAritmético(*byte_alto*, modelo_parte_alta))
15: actualizarModelo(modelo_parte_alta, byte_alto)
16: **if** $byte\_alto == 0$ **then**
17: *salida*.append(codificadorAritmético(*byte_bajo*, modelo_parte_baja_0)
18: actualizarModelo(modelo_parte_baja_0, byte_bajo)
19: **else**
20: *salida*.append(codificadorAritmético(*byte_bajo*, modelo_parte_baja_1)
21: actualizarModelo(modelo_parte_baja_1, byte_bajo)
22: **end if**
23: $señal\_anterior \leftarrow x$
24: **end for**
25:
26: **return** *salida*

a enteros sin signo de forma que enteros con el mismo valor absoluto quedan consecutivo, dada por la ecuación: $Rice(x) = 2x \,\text{if}\, x \geq 0 \,\text{else}\, 2x - 1$. El algoritmo usa un modelo para el byte alto de dicha secuencia y dos modelos para el byte bajo de la secuencia, condicionando según si el byte alto es cero o no. Cada uno de estos modelo establece un valor de probabilidad para cada uno de los 256 valores que puede tomar un byte. Las líneas 3 a 5 inicializan estos tres modelos estadísticos con distribuciones uniformes. Los modelos son adaptativos y van evolucionando a medida que la secuencia es comprimida.

El algoritmo sigue con un loop sobre todos los valores de la señal. Para cada valor de la señal, PGNanoAlgo calcula su predicción dada la historia para luego computar el error de predicción, guardado en la variable error. A dicho error de predicción, le aplica el mapa de Rice y almacena el resultado en la variable símbolo. Luego, separa el símbolo a comprimir en el byte alto y bajo.

Como el byte bajo está condicionado según el byte alto, primero comprimi-

mos el byte alto usando el modelo para dichos bytes y un codificador aritmético. Luego, comprimimos el byte bajo, usando un modelo distinto según si el byte alto era cero o no. Inmediatamente después de comprimir algún byte con su modelo correspondiente, actualizamos dicho modelo en función del valor visto, con el objetivo de aprender los parámetros de dicho modelo de forma adaptativa.

Motivación del algoritmo

Es relevante mencionar que PGNanoAlgo codifica $\{Rice(d_i)\}$. El mapa de Rice es una transformación que lleva enteros con signo a enteros sin signo de forma que enteros con el mismo valor absoluto quedan consecutivos. Además los valores quedan en orden creciente de valor absoluto. Esto logra que los valores con mayor probabilidad (enteros positivos o negativos con un valor absoluto pequeño) queden consecutivos en la representación en bytes y por tanto las distribuciones de probabilidad de cada byte independiente tendrán menor entropía.

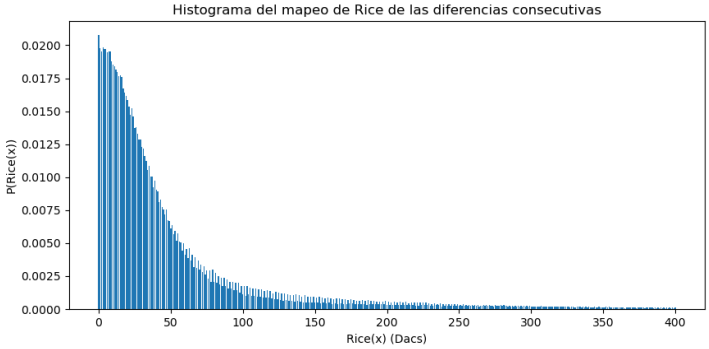

Figura 4.4: Histograma normalizado del mapeo de Rice de la diferencia de muestras consecutivas.

Cuando aplicamos el mapa de Rice al error de predicción obtenemos un histograma como el de la figura 4.4, que fue obtenido a modo de ejemplo de una señal de 646322 muestras de nuestra base de experimentos (ver capítulo 5 para obtener detalles sobre de los conjuntos de datos). Vemos que esto hace que la mayor masa de probabilidad se concentre en enteros menores que 256 y que por tanto su byte alto sea usualmente cero. Podemos forzar este conocimiento a priori de la señal separando las muestras de a bytes, como hace PGNanoAlgo. La figura 4.5 grafica histogramas normalizados para el byte bajo y alto del mapeo de Rice de muestras consecutivas, ilustrando cómo aplicando en conjunto estas técnicas logramos tener una entropía muy baja en el byte alto.

Notamos además que PGNanoAlgo agrupa los valores del byte bajo en dos clases de contexto, según si el byte alto es cero o no. Intuitivamente, si el byte alto no es cero, entonces el byte bajo de la señal no tiene estadísticas relevantes porque la predicción fue muy mala. Por otro lado, si el byte alto es cero la predicción es de mejor calidad. Intuitivamente, las estadísticas de la parte

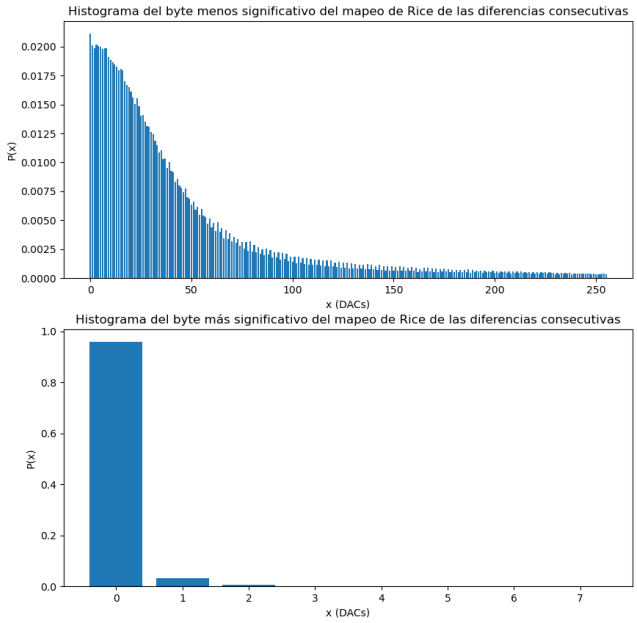

Figura 4.5: Arriba: Histograma normalizado del byte menos significativo del mapa de Rice de la diferencia de muestras consecutivas. Abajo: Histograma normalizado del byte más significativo del mapa de Rice de la diferencia de muestras consecutivas.

baja son distintas en cada caso. Otra intuición que motiva el uso de dos clases de contexto en la parte baja es que cuando el byte alto no es cero, entonces, como vimos en la sección 4.1.1, estamos implícitamente detectando un evento de cambio de kmero. Es razonable pensar que las estadísticas en los saltos entre kmeros sean distintas a las estadísticas dentro de las regiones de equivalencia.

Con esto logramos usar $3 * (2^8 - 1) = 765$ parámetros para modelar la señal, en vez de los $2^{16} - 1 = 65535$ necesarios en caso de codificar los valores como enteros de 16 bits, al mismo tiempo que tenemos una aproximación razonable de la señal. En suma, esto nos permite reducir el costo de modelo con poca penalidad por aproximar las estadísticas de la señal.

4.2. PGNanoLib

PGNanoLib es el componente principal en la implementación de PGNano5 y PGNanoS. La figura 4.6 expone los diferentes componentes así cómo su relación

con PGNano5 y PGNanoS.

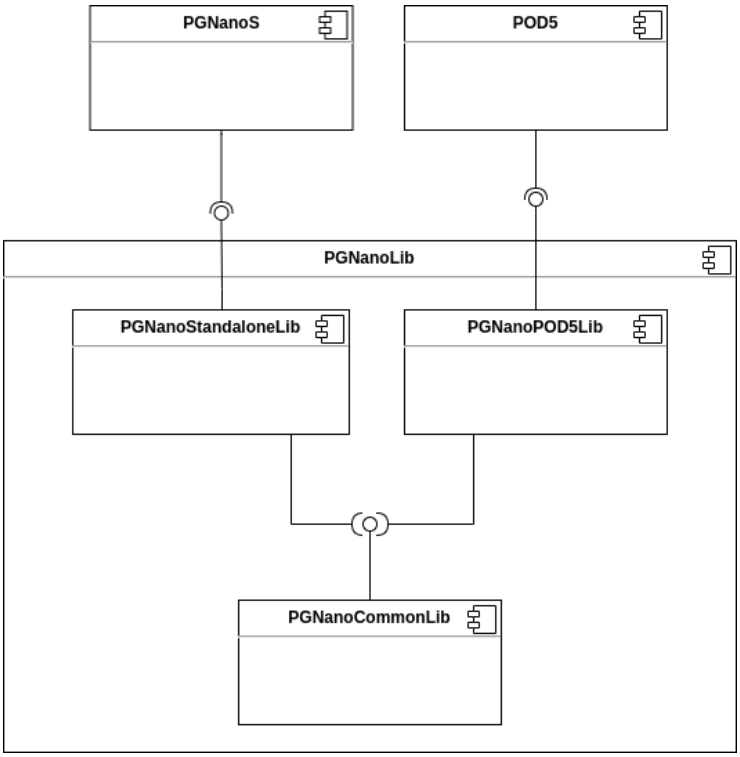

Figura 4.6: Estructura de PGNanoLib y su relación con PGNano5 y PGNanoS.

El componente PGNanoCommonLib es el componente base de PGNanoLib que implementa funcionalidades comunes a los compresores PGNano5 y PGNanoS. En particular implementa el mapeo de Rice y un modelo estadístico que representa una distribución de probabilidad como un histograma que será usado por el codificador aritmético. Además define un conjunto de tipos comunes a ambos compresores e implementa un parser para el tipo de poro.

El componente también implementa un conjunto de clases auxiliares que nos sirvieron para experimentar con ideas intermedias exploradas en el proyecto. Por ejemplo, implementa una clase LevelTable que interpreta los archivos que ONT da donde para cada kmero define su media característica. Como estas clases no se usan en la versión final de los compresores, no entramos en detalle en ellas. No obstante, las mantenemos porque aportan al framework generado.

Notamos que tanto PGNanoPOD5Lib como PGNanoStandaloneLib son componentes que implementan de forma separada a PGNanoAlgo. El motivo detrás de esto es que, como explicaremos en la sección 4.4, hay algoritmos que son más apropiados implementarlos integrados a POD5, siguiendo la arquitectura y restricciones de la biblioteca, pero hay otros algoritmos que necesitan evitar algunas restricciones de POD5 para implementarlos de forma razonable.

De esta forma, el componente PGNanoPOD5Lib implementa el algoritmo PGNanoAlgo para ser utilizado en el compresor PGNano5. Este componente también proporciona un conjunto de adaptadores que facilitan la integración con POD5, por ejemplo traduciendo las invocaciones de POD5 que usan arrays de Arrow a arrays de C. Por último, el componente implementa algunas clases que son necesarias para que en la interacción con POD5 podamos recuperar más metadatos de la señal. En particular, implementa un estado compartido entre POD5 y el compresor integrado a dicha biblioteca, de forma que por ejemplo POD5 notifique al componente PGNanoPOD5Lib cuando un usuario registra un nuevo tipo de poro.

Por su parte, PGNanoStandaloneLib implementa el algoritmo PGNanoAlgo para ser utilizado en el compresor PGNanoS.

El componente PGNanoS implementa un par de programas compresor y descompresor que nos permiten comprimir un archivo POD5 por fuera de dicha biblioteca. Esto nos permite comprimir señales enteras y evitar algunas restricciones de POD5. La responsabilidad fundamental de este componente es leer un archivo POD5 y comprimirlo a un formato específico del proyecto, integrando si es necesario otras fuentes de datos como archivos BAM.

En la figura 4.6, el componente POD5 es la biblioteca POD5 con las modificaciones del proyecto. Definimos a PGNano5 como el componente PGNanoPOD5Lib, más las modificaciones a POD5 agregadas en el proyecto.

4.3. PGNano5

Esta sección presenta a PGNano5 y las modificaciones realizadas a POD5 para agregar este nuevo método de compresión. Queremos minimizar los cambios que hacemos a POD5.

POD5 tiene una arquitectura jerárquica. Los módulos de compresión se encuentran al final de la jerarquía. Esto implica que los cambios se concentran al final de la jerarquía, excepto por las APIs donde tenemos que permitir un nuevo método de compresión.

Notamos que, en PGNano5 mantenemos la compresión independiente de chunks de señal. Correlacionar distintos chunks de la misma señal daña el acceso aleatorio que POD5 brinda. Además puede perjudicar notoriamente la eficiencia computacional, introduciendo puntos de sincronización que antes no existían, o implementando de costosos algoritmos de búsqueda que además implica cambios mayores a la arquitectura de POD5. Estos cambios significativos en POD5 son precisamente lo que buscamos evitar.

4.3.1. Modificaciones a las APIs existentes

POD5 expone dos APIs, una en C y otra en Python. En el proyecto sólo vamos a trabajar con la API de C.

La principal preocupación es no modificar lo que un cliente conoce de la API. Un cliente que utilice POD5 no puede perder soporte, y el acceso a los nuevos

métodos de compresión debe ser lo más similar al resto del manejo de las APIs preexistentes.

También es importante notar que no todas las funcionalidades de la API son soportadas. En particular POD5 permite comprimir la señal por fuera de la biblioteca, invocando al algoritmo de compresión Vbz que implementa, y luego pasar un read con la señal pre-comprimida. No exponemos una funcionalidad análoga para el compresor PGNano5 por simplicidad. Esta restricción no impide el uso correcto de la biblioteca. Lo que si soportamos es leer los datos de un read y poder escribir datos de un read, **delegándole la responsabilidad de compresión a la biblioteca**.

Cambios en la definición de la API de C

A nivel de definición de la API, lo único que cambiamos es que ahora el cliente tiene una opción más de compresión. El cliente elige como comprimir los datos que escribe mediante un enumerado, ese enumerado ahora se expande con el valor PGNANO_SIGNAL_COMPRESSION.

Cambios en la implementación de la API de C

El único cambio que hacemos en la implementación de la API de C mediante el uso de la biblioteca en C++ de POD5, es que al generar un writer, ahora que tenemos un nuevo formato de compresión, entonces tenemos una nueva opción para la construcción de dicho writer. Debemos reconocer cuando el cliente pide comprimir con PGNanoAlgo y configurar el tipo de señal internamente para que use PGNano5. Esto sucede en la función pod5_create_file de la API.

4.3.2. Cambios a la biblioteca

Como primer paso para introducir un nuevo método de compresión a la biblioteca POD5, agregamos un nuevo tipo de señal. Esto significa crear un nuevo tipo de Arrow que representa una nueva codificación de la columna samples de la tabla Signal. Debemos representar el nuevo tipo de señal de dos formas: como un enumerado y como una clase de Arrow.

Enum Para expandir esta representación agregamos un valor más al enumerado SignalType (PGNanoSignal) y agregamos código que maneje dicho valor siempre que corresponda. Esto conlleva los cambios en la API de C ya mencionados, y en las clases SignalTableReader, SignalTableRecordBatch y SignalTableWriter. Los cambios introducidos son menores y simplemente terminan determinando a qué método especializado llamar.

Como sucedía con la relación entre reader y record batch en Arrow, donde es el record batch quien tiene la responsabilidad de recuperar los datos, al agregar un tipo de señal, el reader no se ve modificado sustancialmente porque es el record batch quien descomprime la señal. **El record batch debe llamar al componente PGNanoLib para descomprimir los datos.**

Por su parte, en el SignalTableWriter debemos generar un nuevo builder (PGNanoSignalBuilder), para construir la columna de señal asociada con los datos que se le pasan. Asociado a ese builder, tenemos los visitors expuestos en 3.3 que también extendemos.

En general las modificaciones que tengan que ver con patrones de diseño, como las mencionadas anteriormente, consisten en código muy similar a su correspondiente implementación para una señal comprimida con Vbz. Esto se debe a que las abstracciones y tipos de datos usadas por los componentes encargados de manejar señales comprimidas con Vbz son adecuadas pues esencialmente se restringen a tratar las muestras como bytes.

Representación como tipo de Arrow Debemos además generar un nuevo tipo de Arrow para representar la señal. Por tanto creamos las clases PGNanoSignalType y PGNanoSignalArray, cuyo código es nuevamente muy similar a VbzSignalType y VbzSignalArray, respectivamente.

Por último, como POD5 ya introducía nuevos tipos de Arrow, entonces en el mismo código que ONT registra los nuevos tipos usados, registramos también el tipo PGNanoSignalType.

Implementación de un compresor genérico

Modificar la lógica de compresión implica modificar todos los visitors que se utilizan para manipular la señal de un SignalTableRecordBatch. El cambio más importante es en append_signal donde cambiamos el método de compresión llamado. En la nueva implementación usamos el método pgnano::compress_signal().

Continuando a niveles de abstracción superiores, a nivel del SignalTableWriter, lo único para hacer es declarar un nuevo builder (PGNanoSignalBuilder), luego el SignalTableWriter se abstrae del tipo de señal justamente usando el patrón visitor.

Implementación de un descompresor genérico

Recordando el acceso a datos de Arrow (y por tanto POD5), debemos modificar la clase SignalTableRecordBatch, ya que es la encargada de descomprimir los datos. En particular se deben modificar los métodos extract_signal_row, extract_signal_row_inplace y samples_byte_count. Además se debe agregar el método pgnano_signal_column. Los cambios notables suceden en extact_signal_row.

El método, extract_signal_row es quien recupera la columna signal (la señal) del record batch de Arrow subyacente y la descomprime. Simplemente lo que hay que hacer aquí es, cuando el tipo de señal sea PGNanoSignal interpretamos la columna de señal como un PGNanoSignalArray y llamamos al algoritmo de descompresión implementado en PGNanoLib.

4.3.3. Pasaje de metadatos

Otro cambio importante de PGNano como framework es que éste permite usar metadatos sobre los reads para mejorar los algoritmos de compresión, incluso dentro de la biblioteca POD5, no era posible esto. El anexo D profundiza sobre los cambios hechos y las características de POD5 que nos determinan una solución concreta que respeta la arquitectura de POD5 así como los atributos de calidad más importantes: concurrencia y acceso aleatorio.

Para entender los cambios, debemos recordar que POD5 tiene una estructura jerárquica, la cual vamos a aprovechar para enriquecer el flujo de datos interno de POD5 con mínimo esfuerzo.

Otro punto importante es que POD5 tiene un protocolo de acceso para escribir los datos, que en particular le permite escribir datos redundantes (como el tipo de poro, que es una cadena de caracteres arbitraria dado por parte del usuario) como índices en un diccionario. Esto permite reducir la duplicación de datos interna en POD5, pero dificulta el acceso a estos metadatos como puede ser el tipo de poro. De todos estos datos nos enfocamos en recuperar el tipo de poro pues es el que consideramos más relevante.

Para resumir el cambio en el flujo de datos, presentamos la figura 4.7 que muestra el flujo de datos relevante antes y después de las modificaciones del proyecto.

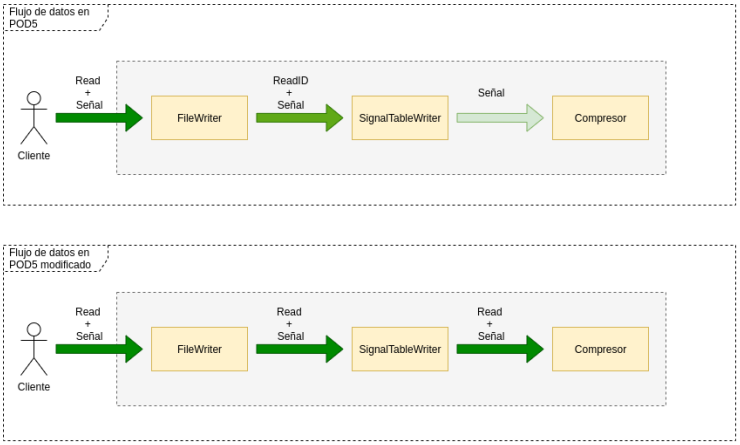

Figura 4.7: Arriba: Flujo de datos dentro de POD5 antes de las modificaciones a la biblioteca. Abajo: Flujo de datos dentro de POD5 con las modificaciones a la biblioteca que permiten agregar metadatos de un read a la compresión de una señal.

Como vemos en la figura, desde la invocación de la biblioteca por parte del cliente hasta la invocación del algoritmo de compresión, POD5 pasa cada vez menos metadatos a capas inferiores de la jerarquía, en particular cada capa tiene menos información sobre el read de la señal. Justamente la arquitectura en capas de POD5, nos permite simplemente que podamos propagar los metadatos

contenidos en un read, a las capas inferiores simplemente modificando firmas de funciones.

Por otro lado tenemos el protocolo de acceso de POD5, que ilustramos en la figura 4.8.

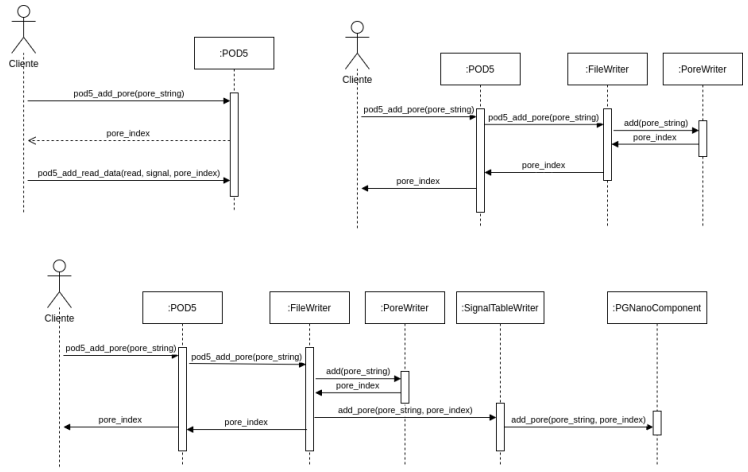

Figura 4.8: Arriba a la izquierda: Protocolo de acceso simplificado de POD5 para el tipo de poro. Arriba a la derecha: Interacción interna en POD5 para registrar el tipo de poro antes de las modificaciones de la librería. Abajo: Interacción interna en POD5 para registrar el tipo de poro luego de las modificaciones de la librería. Notar que en este caso POD5 escribe en el estado compartido con PGNano5 el nuevo tipo de poro.

Lo relevante de este protocolo es que aquellos metadatos que se escriben como índices a un diccionario (por ejemplo el tipo de poro), deben ser declarados por el cliente antes de poder usarlos en la construcción de un read. Esto implica que podamos hacer los cambios que están a la derecha de la figura 4.8, donde las declaraciones de dichos metadatos ahora son notificadas a PGNano5 mediante el estado compartido que expone PGNanoPOD5Lib.

Recordar que, como mencionamos en la sección 3.2.2, POD5 solo permite a un cliente **escribir datos de una señal pasando en la misma llamada tanto los datos de la señal como del read correspondiente**. Esto es crucial para las modificaciones presentadas.

4.4. PGNanoS

PGNanoS es un compresor que opera por fuera de la biblioteca POD5. Concretamente, este compresor nos permite experimentar con la compresión de señales enteras (en vez de chunks) y facilita la implementación de modelos estadísticos que utilizan datos generados en etapas posteriores al proceso de obtención de datos crudos. En particular, podemos pensar que la secuencia de

nucleótidos generada a partir de la señal puede tener información importante para el modelado estadístico de la señal y por tanto para su compresión. Por otro lado, base callers como Remora (un base caller de investigación construido por ONT) emiten una secuencia de valores llamados "move tag". Estos valores sirven para alínear de forma aproximada, la secuencia de señales con la secuencia de nucléotidos. Es decir, los primeros 100 valores de la señal corresponden al kmero 'AGT', los segundos 50 valores de la señal corresponden al kmero 'GTC'. Con esto se pueden plantear modelos que contextualicen las estadísticas en el k-mero que actualmente está atravesando el poro en la señal de secuenciación.

Tener la señal segmentada en chunks y comprimir los chunks de forma independiente no es conveniente para implementar esta idea. En principio, se tienen que procesar todos los chunks de una misma señal en orden, o en su defecto tener calculado algún offset que dado un chunk indique donde empezar a interpretar la move tag. Este offset debería ser calculado por el compresor, quien nuevamente ve a los chunks de forma independientes. Todo esto implicaría secuencializar el acceso a chunks dentro de una misma señal, para que se hagan en orden. Esto atenta fuertemente contra el procesamiento paralelo interno que hay dentro de la biblioteca, además de causar disrupciones arquitectonicas mayores. Por otro lado, el esfuerzo de desarrollo involucrado es sustancial, sumando a los nuevos desafíos que involucra el manejo de un nuevo tipo de archivo de secuenciación y los potenciales problemas de concurrencia que puedan surgir.

Por todo esto, se decide entonces generar un par de programas compresor y descompresor que no estén directamente integrados a POD5.

En virtud de mantener lo más posible el formato de archivo existente, y reutilizar gran parte de la infraestructura y aspectos de calidad brindados por POD5, dichos programas escribirán un archivo POD5, pero la compresión se hará fuera de la biblioteca.

Por todo lo antes mencionado, esta solución tiene como desventaja que desaprovecha algunos de los aspectos de calidad de POD5 que se quieren mantener expandiendo la biblioteca. En particular, se pierde el acceso aleatorio a nivel de chunk. Otros aspectos como la compresión y descompresión paralela también se pierden, pero esto es simplemente porque los programas planteados no se implementan de forma paralela; la algoritmia se puede paralelizar ahora a nivel de read y recuperar dichas ventajas. Por otro lado, este enfoque facilita la recuperación de todos aquellos metadatos que fueron ignorados en la implementación anterior.

Luego, ambas soluciones tienen ventajas y desventajas tales que ambas tienen valor en contextos diferentes.

4.4.1. Implementación

En concreto, el compresor debe poder generar un formato de archivo intermedio, tal que el descompresor pueda recuperar el archivo POD5 original de este formato intermedio. En la práctica, este formato es exactamente un archivo POD5, pero podría expandirse para contener juntos datos de secuenciación y basecalling.

Recordemos que uno de los "tipos de señal" que ofrece POD5 es una señal no comprimida. Por otro lado, para POD5 una señal no es más que una secuencia de largo fijo de enteros con signo de 16 bits. En particular, si se tiene un conjunto de bytes comprimidos por un compresor externo, este se puede pasar a POD5 para que lo almacene internamente junto a los metadatos correspondientes. Al momento de la lectura POD5 interpreta esos bytes como una señal descomprimida, por lo que retorna exactamente los bytes comprimidos originales. Luego un descompresor externo puede recuperar la señal original de estos bytes. Además el descompresor tiene acceso a los metadatos que se encuentran en el archivo POD5 utilizado como representación intermedia. Luego, el descompresor puede recuperar el archivo POD5 original.

Esta idea es la que se implementa para esta solución. Un compresor standalone abre un archivo POD5 de entrada, comprime la señal con un algoritmo de compresión arbitrario, recuperando los metadatos asociados. Luego, el compresor escribe un archivo POD5 intermedio que contiene los metadatos del archivo original, pero como señal le pasa a POD5 el stream de bytes comprimidos. Esta es la representación intermedia utilizada. Por otro lado, el descompresor va a poder leer de este POD5 intermedio el stream de bytes comprimidos, que para la biblioteca de POD5 es una señal de secuenciación. Dado este stream de bytes comprimidos y los metadatos correspondientes, el descompresor puede recuperar la señal de secuenciación original y escribirla en un archivo POD5 de salida.

Si bien esta solución no es la más elegante, tiene un gran número de ventajas prácticas. Concretamente, permite mantener todos los metadatos de un archivo POD5 junto a la señal de secuenciación sin necesidad de generar un nuevo formato de archivo. Esto por su parte acelera enormemente el tiempo de desarrollo, permitiendo explorar otras ideas que potencialmente mejoren aún más la compresión. Es decir, nos permite concentrarnos en el algoritmo de compresión propiamente dicho, teniendo resuelto por POD5 la serialiazación de todo el resto de metadatos auxiliares que son relevantes para el verdadero archivo de secuenciación.

Capítulo 5

Evaluación experimental del desempeño de los compresores

Este capítulo presenta resultados experimentales sobre los compresores PGNano5 y PGNanoS. Para ello primero presentamos los datasets utilizados y sus características. Luego presentamos las métricas sobre las que evaluamos a los compresores. Seguimos con la presentación de los resultados junto con su análisis. Por último, extraemos algunas conclusiones de estos resultados.

5.1. Datasets utilizados

En este proyecto nos concentramos en los tipos de poro R9.4.1, R10.3 y R10.4.1, por lo que necesitamos al menos un conjunto de archivos para cada uno de estos tipos de poro.

Por otro lado, nos interesan principalmente las señales de secuenciación sobre humanos. Entonces, para cada tipo de poro tendremos un dataset que corresponde a la secuenciación de alguna parte del genoma humano. De todas formas, no descartamos datasets de otros tipos de organismos.

La tabla 5.1 muestra las características de cada dataset. Para obtener los datos y reproducir los experimentos, referirse a las tablas A.3, A.2 y A.1 del anexo. Los datasets mostrados pueden consistir en cientos de gigabytes de datos, por lo que realizamos un muestreo aleatorio de cada dataset.

Nombre abreviado	Autor	Archivos muestreados	Tipo de poro	Tamaño total (GB)
GIAB10.4.1	ONT	15	R10.4.1	12.1
GIAB9.4.1	ONT	15	R9.4.1	12.1
GIAB10.3	ONT	14	R10.3	12.4
Fly	Standford University	14	R10.4.1	8.8

Tabla 5.1: Datasets utilizados junto a sus características.

En el documento, cuando hablemos de un conjunto de datos particular, nos referiremos siempre o bien por el nombre abreviado presente en la tabla 5.1 o por el tipo de poro que representan (por ejemplo todos los archivos con poro R10.4.1). De esta forma definimos:

- R9.4.1 = GIAB9.4.1,
- R10.3 = GIAB10.3,
- R10.4.1 = GIAB10.4.1 ∪ Fly,
- Todos = R9.4.1 ∪ R10.3 ∪ R10.4.1

Notar que los datasets GIAB10.3 y GIAB9.4.1 están publicados como un único dataset que compara ambos tipos de poro. Aquí los separamos en dos datasets diferentes para analizar si hay diferencias o no según el tipo de poro.

También es relevante señalar que los tamaños de los archivos dentro de cada dataset pueden variar significativamente. Por esto, en el muestreo aleatorio de los archivos nos aseguramos de obtener archivos con diferentes tamaños.

5.1.1. Preprocesado y normalización de datos

Dado que POD5 es un formato relativamente nuevo, los datos usualmente están publicados en formato FAST5. Además POD5 permite configurar un largo de chunk, que cambia cómo se segmenta la señal y por tanto afectará a los algoritmos de compresión. Por esto, es necesario realizar un paso de preprocesado y normalización de datos.

Las etapas más importantes de este proceso son convertir los archivos FAST5 a POD5 y normalizar el largo de chunk.

Para convertir los archivos FAST5 a POD5 utilizamos una herramienta brindada por ONT disponible en (Oxford Nanopore Technologies, 2023g).

Para la normalización del largo de chunk lo que hacemos es copiar los archivos POD5 generados de la etapa anterior con el largo de chunk por defecto en la biblioteca de POD5.

5.2. Métricas de evaluación

La primer métrica que presentamos es la tasa de compresión (tc) en bits por símbolo de 16 bits, es decir, en bits por muestra. Abreviamos bits por símbolo mediante bps. Valores más pequeños representan una mejor tasa de compresión.

Primero definimos la tasa de compresión para un archivo:

$$tc(archivo) = \frac{\sum_{señal \in archivo} largo(comprimir(señal)) * 8}{\sum_{señal \in archivo} largo(señal)},$$

donde, $largo(comprimir(señal))$ es el largo de la señal comprimida en bytes y $largo(señal)$ es el la cantidad de muestras de la señal.

La tasa de compresión de un dataset la definimos como el promedio de las tc de los archivos que lo componen,

$$tc(dataset) = \frac{\sum_{archivo \in dataset} tc(archivo)}{|dataset|}.$$

Luego, presentamos la mejora porcentual relativa a Vbz. Esta métrica no tiene unidades y la abreviamos como mpr. Valores más grandes de la métrica son mejores. Valores positivos indican que PGNanoAlgo comprime mejor que Vbz. Valores negativos indican que Vbz comprime mejor que PGNanoAlgo. La definimos como:

$$mpr(dataset)_{método} = \frac{tc(dataset)_{Vbz} - (tc(dataset)_{método}) * 100}{tc(dataset)_{Vbz}},$$

donde métodos puede ser PGNanoS o PGNano5.

Por último, medimos la velocidad de compresión de una secuencia X_{in} en MB/s. Es decir, medimos el tiempo que demora en ejecutarse el método que comprime los datos, en segundos, y dividimos dividimos por el tamaño de la representación original de la secuencia (antes de comprimir) expresada en MB.

Definimos la velocidad de compresión del algoritmo como el promedio simple de la velocidad de compresión sobre un conjunto de secuencias de entrada. Estas medidas son sólo sobre el algoritmo de compresión y se miden con los datos ya cargados en memoria.

5.3. Pruebas de corrección

Las pruebas de corrección se basan en comprimir un archivo con PGNano5 o PGNanoS y descomprimirlo al formato original (es decir comprimido con Vbz). Luego, usamos un script provisto por ONT para verificar si los archivos son equivalentes. El script está disponible en (Oxford Nanopore Technologies, 2022a). Esto significa que para un cliente de la biblioteca, es indiferente tener el archivo original o el archivo luego de ser comprimido y descomprimido con PGNano. Recordemos que el formato binario de POD5 tiene partes que son generadas aleatoriamente para ayudar a identificar a los archivos. En particular, (prácticamente) nunca tendremos dos archivos POD5 que tengan el mismo contenido binario. Además, el orden de los reads y la segmentación de la señal en chunks es irrelevante para un cliente de la biblioteca. Este script toma en cuenta estos detalles para considerar si dos archivos son equivalentes o no.

Para las versiones finales de PGNano5 y PGNanoS, las pruebas de corrección son exitosas.

5.4. Resultados experimentales

Para obtener los resultados experimentales corrimos los compresores PGNano5, PGNanoS y Vbz sobre los datasets descritos en la sección 5.1. Recor-

damos que PGNano5 comprime la señal de a chunks, mientras que PGNanoS comprime la señal sin segmentarla en chunks. Entonces el algoritmo Vbz lo corrimos segmentando la señal en chunks, para compararlo contra PGNano5, y también sin segmentar la señal, para compararlo contra PGNanoS. En lo que sigue denotamos respectivamente Vbz chunks o Vbz full a estas versiones de Vbz.

Las pruebas las corremos sobre un sistema con las siguientes características:

- CPU: Intel(R) Core(TM) i9-10940X CPU @ 3.30GHz (1.20GHz-4.80GHz).
- 14 cores con hyperthreading activado (28 threads).
- RAM: 256GB.
- Disco: 8TB NVMe.

Presentamos los resultados en dos tablas. La tabla 5.2 muestra las métricas *tc* y *mpr* para los diferentes algoritmos de compresión. La tabla 5.3 da la velocidad de compresión en MB/s para Vbz y PGNano.

	Chunks		Full	
Dataset	PGNano5 (bps)	Vbz (bps)	PGNanoS (bps)	Vbz (bps)
R10.4.1	6.908 (3.597 %)	7.166	6.898 (3.538 %)	7.151
R10.3.1	**6.844 (3.661 %)**	7.104	**6.832 (3.611 %)**	7.088
R9.4.1	5.221 (0.948 %)	5.271	5.215 (1.07 %)	5.271
Promedio	6.324 (2.917 %)	6.514	6.314 (2.910 %)	6.503

Tabla 5.2: Resultados de tasa de compresión para PGNano5, PGNanoS, Vbz chunks y Vbz full sobre los conjuntos de prueba presentados. Para cada algoritmo damos su tasa de compresión *tc* en bits por símbolo (bps). Además, para PGNano5 y PGNanoS damos su mejora porcentual relativa (*mpr*) respecto a Vbz entre paréntesis.

En la tabla 5.2, observamos que los algoritmos PGNano5 y PGNanoS mejoran la tasa de compresión de Vbz en todos los datasets. Esto lo podemos ver porque para todos los dataset obtenemos una *mpr* positiva. Notar que las mayores ventajas en tasa de compresión se obtienen para los poros más nuevos con los que trabajamos.

En promedio reducimos el tamaño de las señales comprimidas un 2.917 % para la compresión de señales enteras y un 2.910 % para la compresión de chunks individuales. Esto se traduce en una mejora de 0.190 bps para el caso de señales comprimidas por chunks y 0.189 bps para las señales enteras.

En el mejor caso (dataset R10.3.1) obtenemos una reducción del tamaño de los chunks comprimidos un 3.661 % del tamaño que ocupan al comprimirse con Vbz, lo que se traduce en una mejora de 0.26 bps. Para el caso de las señales enteras, reducimos el tamaño un 3.611 % y obtenemos una mejora de 0.256 bps.

En el peor caso (dataset R9.4.1) reducimos el tamaño de los chunks un 0.948 %, obteniendo una mejora de 0.050 bps. Para las señales enteras reducimos su tamaño a un 1.07 %, obteniendo una mejora de 0.056 bps.

Además, notamos que las compresiones de las señales enteras son siempre mejores que la compresión por chunks, tanto para Vbz como para PGNanoAlgo. Si consideramos el dataset R10.4.1, tenemos que la compresión de señales enteras en Vbz mejora la tasa de compresión en 0.015 bps, mientras que para PGNanoAlgo lo hace en 0.01 bps. Si bien la diferencia es irrelevante, podemos suponer que la compresión por chunks afecta más a Vbz que a los algoritmos aquí presentados, lo cual no es extraño dado que Vbz utiliza ZSTD que tiene una tasa de convergencia "lenta", mientras que el modelo estadístico de los algoritmos propuestos es bastante más simple. No obstante, en cualquiera de los casos las diferencias entre comprimir por chunks o las señales enteras es menor.

La pequeña diferencia entre la compresión por chunks y la señal entera reafirma la decisión de diseño tomada en la sección 4.3, donde restringimos a PGNano5 a comprimir chunks individuales. En el caso de PGNano5, la compresión de chunks de forma independiente minimiza el número de cambios a realizar y simplifica la arquitectura, sin comprometer significativamente la tasa de compresión del algoritmo. En el caso de PGNanoS, dado que estamos comprimiendo los datos por fuera de la biblioteca, podemos aprovechar esta pequeña diferencia sin problemas adicionales.

Como conclusión, tenemos que el mejor compresor en términos de tasa de compresión es PGNanoS, seguido de PGNano5.

Por último, presentamos resultados sobre la velocidad de los algoritmos y su consumo de memoria. En la tabla 5.3 vemos que Vbz es mucho más rápido que PGNanoAlgo. En particular, vemos que Vbz es aproximadamente 19 veces más rápido que PGNanoAlgo, en una prueba de velocidad en memoria sin contar overhead de las bibliotecas.

Algoritmo	Velocidad (MB/s)
Vbz	**921**
PGNanoAlgo	49

Tabla 5.3: Velocidad de compresión en MB/s consumidos del buffer de entrada par Vbz y PGNanoAlgo.

Para el consumo de memoria utilizamos el programa de copia usado en las pruebas de corrección de la parte 5.3. En particular, copiamos un archivo POD5 comprimido con Vbz hacia un archivo comprimido con PGNano5. También copiamos el mismo archivo pero comprimiendolo con Vbz. Notar que entonces estas mediciones solo las hacemos sobre las señales fragmentadas en chunks y los resultados contienen el overhead del programa de copia y las bibliotecas involucradas. Destacamos que el consumo de memoria de los algoritmos Vbz y PGNano5 no depende del tamaño de archivo considerado sino del tamaño de los chunks individuales. Además, como hacemos exactamente la misma secuencia de pasos en una ejecución del programa cuando comprimimos con Vbz o

con PGNano5, excepto el método de compresión usado, inferimos que la mayor diferencia en el consumo de memoria proviene de los algoritmos de compresión usados.

En este marco tenemos que Vbz tiene un consumo máximo de 2547 MB de memoria residente mientras que PGNano5 tiene un consumo máximo de 2526 MB de memoria. Obtenemos que PGNano5 usa ligeramente menos memoria que Vbz.

5.4.1. Diferencias de compresión entre archivos de distinto poro

Como podemos observar en la tabla 5.2, los datasets con poro R9.4.1 reportan mejores tasas de compresión respecto a aquellos con poro R10.3 o R10.4.1. Por otro lado, los archivos con poro R10.3 logran marginalmente una mejor tasa de compresión respecto a aquellos con poro R10.4.1. Esto lo podemos explicar porque el rango de conversión analógico digital de los poros R9.4.1 y R10.3 es menor que el de los archivos con poro R10.4.1. Esto implica que vamos a tener una cuantización más fuerte y que por lo tanto el alfabeto a comprimir es de menor tamaño.

Otro factor que podría justificar las diferencias de tasa de compresión entre archivos de distinto poro corresponde con la "complejidad" del poro. Si los kmeros que captura un poro tienen un largo mayor a los de otro tipo de poro, podemos pensar que más factores entran en juego en las estadísticas de la señal. Es importante destacar que los poros R9 tienen un único lector en el poro, mientras que los poros R10 tienen dos lectores en el nanoporo, lo que incrementa el número de bases que determinan la señal capturada y mejora los resultados del base calling en regiones donde se tienden a repetir nucléotidos (Oxford Nanopore Technologies, 2022c).

Si bien ambos factores pueden estar afectando la diferencia en tasa de compresión entre poros, parece más razonable suponer que esta diferencia se ve más afectada por el rango de conversión analógico digital.

Por otro lado, vemos que las mayores ganancias en términos de tasa de compresión respecto a Vbz se dan para los poros R10.4.1 y R10.3 (ambos poros R10.X). Si analizamos el comportamiento de Vbz, encontramos que esto también tiene sentido, nuevamente por el mayor rango de conversión analógico digital de los instrumentos. En particular, como veremos en la sección 6.3, no necesariamente se cumple que el byte alto de la señal mapeada es generalmente 0. Cuando el rango de digitalización aumenta, como tenemos más bits de precisión, es menos probable que el byte alto se haga 0 y por tanto es menos probable que Vbz pueda omitir el byte alto mediante el paso de StreamVByte.

Algo a notar es que los resultados para R10.3 y R9.4.1 son sustancialmente diferentes, incluso cuando estos provienen del mismo experimento y de la misma muestra biológica pero alterando el tipo de poro. Esto reafirma la idea de que el tipo de poro usado es relevante para la compresión.

Capítulo 6

Modelos de compresión alternativos

Con lo presentado en los capítulos 4 y 5, desarrollamos un framework que nos permite prototipar rápidamente nuevos métodos de compresión. Además obtenemos una línea base sobre la cual comparar los nuevos métodos desarrollados que mejora la línea base de Vbz. Este capítulo presenta cómo usamos ese framework, que incluye software adicional a PGNano5 y PGNanoS, para mejorar nuestros algoritmos de compresión y estudiar las características de la señal. Para estos análisis proponemos una serie de experimentos, presentados en la sección 6.1.

Como los poros R10.4.1 son los poros más nuevos con los que trabajamos, entonces en este capítulo trabajaremos con el conjunto de datos R10.4.1.

6.1. Experimentos propuestos

En el capítulo 4 analizamos la señal concentrándonos en los aspectos relevantes para PGNanoAlgo. En esta sección presentamos un conjunto de experimentos que tienen como objetivo buscar otros aspectos de la señal de secuenciación que podamos usar para mejorar la compresión y que no fueron usados en PGNanoAlgo.

A modo de resumen, proponemos los siguientes experimentos:

- Experimentar con los predictores $p_{dif-1-negativa}$ y $p_{dif-1-positiva}$: La señal parece alternar constantemente entre crecimiento y decrecimiento. Estos predictores intentan aprovechar esta característica para mejorar las predicciones.

- Modelo de contexto "Signo": En línea con el punto anterior, presentamos un modelo de contexto para la parte baja que intenta ver si la señal tiene estadísticas diferentes al crecer o decrecer.

- Modelo de contexto de “bandas de frecuencia”: Estos modelos intentan aproximar mejor las estadísticas de la señal reduciendo el número de kmeros probables que podemos inferir que existen en el poro para una muestra dada.

- Nuevas particiones de la señal: Intentamos partir los símbolos de la señal en subsímbolos distintos correspondientes a 2 partes, alta y baja, pero con diferente tamaño (no necesariamente bytes).

- Parametrizar las distribuciones de probabilidad observadas. En particular, ajustar mediante distribuciones geométricas y triangulares para reducir el costo de modelo y aumentar la eficiencia computacional.

- Utilizar otras codificaciones para la señal: Proponemos experimentos con códigos de Golomb potencia de dos, RLE y ZSTD.

- Codificar con ZSTD los cinco bits menos significativos del error de predicción mapeado para ver la existencia de correlaciones temporales en forma de cadenas de Markov.

- Otros modelos de contexto para los cinco bits menos significativos del error de predicción mapeado.

6.2. Experimentos sobre el predictor

El algoritmo propuesto, PGNanoAlgo, se basa en predicciones: dada la historia de la señal, intenta predecir el siguiente valor y codifica el error de predicción. Intuitivamente, si mejora el predictor, la tasa de compresión mejorará.

En esta sección analizamos tres predictores concretos que nos dan los resultados más relevantes. El anexo B profundiza el análisis. Es relevante comentar que el predictor utilizado por PGNanoAlgo obtiene mejores resultados que todos los otros analizados en el anexo.

Para evaluar los predictores, utilizamos el mismo esquema de compresión de PGNanoAlgo, reemplazando únicamente el predictor. Estimamos el largo de código como logaritmo del inverso de la probabilidad asignada al símbolo por nuestro modelo, teniendo un único modelo para el byte menos significativo del error de predicción mapeado y otro para el byte más significativo.

Por último, estos experimentos se corren utilizando la API de Python de POD5, compilada y linkeada contra la biblioteca original de POD5 (es decir, sin modificaciones).

Para motivar los predictores que presentamos, la figura 6.1 grafica la señal en función del tiempo, así cómo la diferencia entre dos muestras consecutivas.

De la figura podemos intuir que la señal parece alternar entre crecer y decrecer, lo que equivale a que la diferencia de muestras consecutivas alterne entre valores positivos y negativos. Los predictores que presentamos intentarán mejorar su predicción de la señal, tratando de anticipar este cambio.

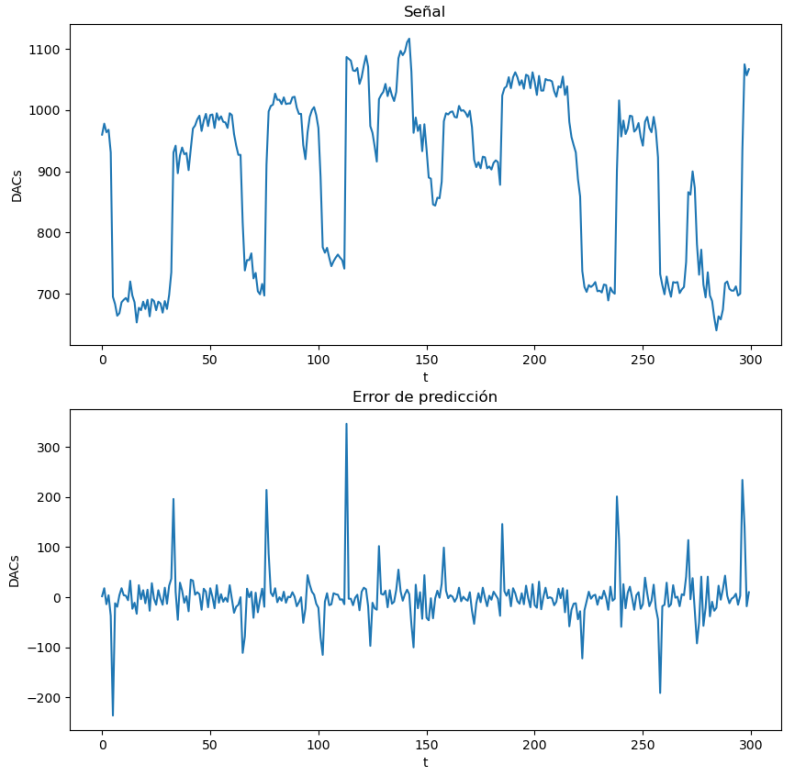

Figura 6.1: Arriba: Señal en función del tiempo. Abajo: Error de predicción en función del tiempo.

6.2.1. Presentación y análisis de resultados

En concreto, tenemos tres predictores que nos interesan:

- $p_{base} = x_{n-1}$,
- $p_{dif-1-negativa} = x_{n-1} - signo(x_{n-1} - x_{n-2})$,
- $p_{dif-1-positiva} = x_{n-1} + signo(x_{n-1} - x_{n-2})$

p_{base} es el predictor con que trabajamos hasta el momento. El predictor $p_{dif-1-negativa}$ se basa en la intuición de que la señal alterna entre crecimientos y decrecimientos. Este predictor intenta predecir este cambio incorporando un sesgo sobre la predicción original. Si la señal disminuyó anteriormente, predecimos que ahora aumentará, y viceversa.

Método	largo de código de chunks (bps)	largo de código de señal entera (bps)
base	7.12747	7.12061
dif-1-negativa	7.11647	7.10972
dif-1-positiva	7.14134	7.13420

Tabla 6.1: Estimación del largo de código esperado del error de predicción para diferentes predictores.

Del predictor $p_{dif-1-positiva}$ lo único relevante es que si la hipótesis que motiva el uso de $p_{dif-1-negativa}$ es cierta, entonces $p_{dif-1-negativa}$ debe mejorar la predicción pero $p_{dif-1-positiva}$ debe empeorarla. En particular nos interesa como experimento de control.

$p_{dif-1-negativa}$ es el mínimo cambio que podemos hacerle a p_{base}, es decir es la mínima diferencia que podemos predecir respecto a la muestra anterior. Experimentamos también con otros predictores que intentan predecir la magnitud de esta diferencia, pero los resultados no son buenos y por tanto solo los adjuntamos en el anexo B. Podemos pensar que al usar $p_{dif-1-negativa}$ asumimos que el crecimiento de la señal va a cambiar pero no sabemos en cuánto y por tanto tomamos una posición conservadora.

La tabla 6.1 presenta los resultados.

De los resultados podemos ver que al comparar $p_{dif-1-negativa}$, $p_{dif-1-positiva}$ y p_{base} podemos inferir que la intuición de que la señal alterna entre crecimientos y decrecimiento puede ser explotada para mejorar los resultados de compresión. En particular, porque el predictor $p_{dif-1-negativa}$ ofrece mejores resultados que p_{base}, mientras que el predictor $p_{dif-1-positiva}$ da peores resultados. Esto sugiere que la alternancia entre crecimiento y decrecimiento es una característica de la señal.

Sin embargo, las diferencias entre estos predictores son mínimas, y los experimentos con otros predictores basado en la misma idea no retorna resultados relevantes. Con esto concluimos que p_{base} es adecuado para continuar usándolo en el proyecto, pero una potencial vía de investigación futura correspondería con afinar la intuición antes presentada.

Para ganar un poco más de intuición sobre el comportamiento del predictor base, graficamos una parte de una señal contra el tiempo, e inmediatamente debajo graficamos el error de predicción cometido para ese tiempo (figura 6.1).

Como podemos ver en la figura 6.1, la señal alterna entre regiones "estables" mediante saltos bruscos, que inferimos que son responsabilidad del cambio del kmero presente en el poro. Además vemos que los mayores errores de predicción ocurren en estos saltos. Entonces, para poder lograr una mejor tasa de compresión, sería bueno poder predecir dichos saltos y el nuevo valor de media al que se llega. Una idea para intentar aprovechar estos puntos es, integrar, de alguna forma, al predictor nociones de tiempo que indiquen hace cuánto fue el último cambio brusco.

La figura 6.1 indica que estos saltos, ya sean en regiones estables o inestables, no parecen ocurrir cada un tiempo regular, lo que dificulta implementar un

predictor que se adelante a estos saltos en base al tiempo transcurrido. Esto tiene sentido pues la velocidad con la que un kmero atraviesa el poro no es constante. Un experimento interesante podría ser el uso de algún tipo de red neuronal recurrente (u otros modelos de predicción para secuencias, como las cadenas de Markov ocultas) que capturen de alguna manera el tiempo relativo transcurrido desde el último salto.

6.3. Parametrización de las estadísticas de la señal

En esta sección, buscamos explorar otras formas de partir los símbolos de la señal en dos subsímbolos y cómo podemos ajustar las distribuciones de probabilidad de dichos símbolos mediante distribuciones de probabilidad conocidas (laplaciana, geométrica, triangular). Si podemos parametrizar parte de los símbolos con una distribución de probabilidad conocida, podemos implementar modelos más eficientes para el codificador aritmético, respecto a usar un histograma de los símbolos. Esto mejoraría la eficiencia computacional del algoritmo y reduciría el costo de modelo, lo que puede ser especialmente beneficioso para señales cortas y si usamos más clases de contexto.

En la figura 6.2 graficamos los 11 bits más significativos del mapeo de Rice del error junto a las mismas gráficas para los 8 bits más significativos.

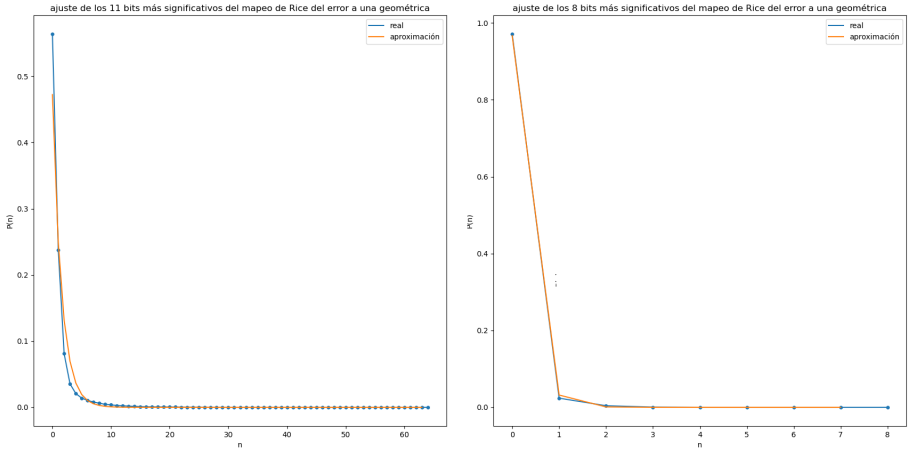

Figura 6.2: A la izquierda: Distribución de probabilidad de los 11 bits más significativos del error mapeado, y su aproximación por una geométrica. A la derecha: Distribución de probabilidad de los 8 bits más significativos del error mapeado, y su aproximación por una geométrica.

Observando los resultados, parece razonable aproximar la distribución de probabilidad de la parte alta mediante una distribución geométrica.

Para la parte baja, graficamos para los 8 bits menos significativos del error, el ajuste de su histograma normalizado a una distribución triangular en la figura

6.3. Graficamos el mismo escenario para los 5 bits menos significativos del error de predicción en la figura 6.4.

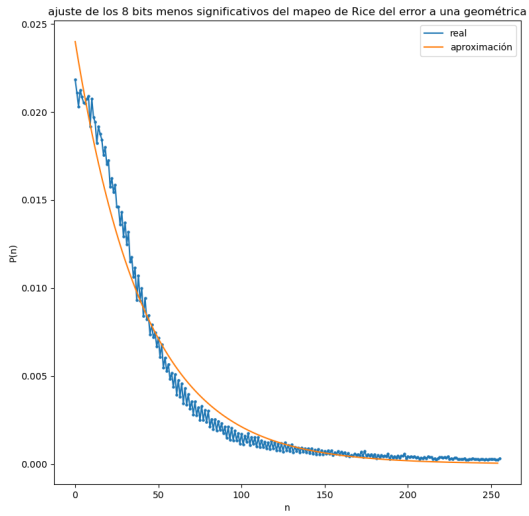

Figura 6.3: Distribución de probabilidad de los 8 bits menos significativos del error mapeado, y su aproximación por una distribución geométrica.

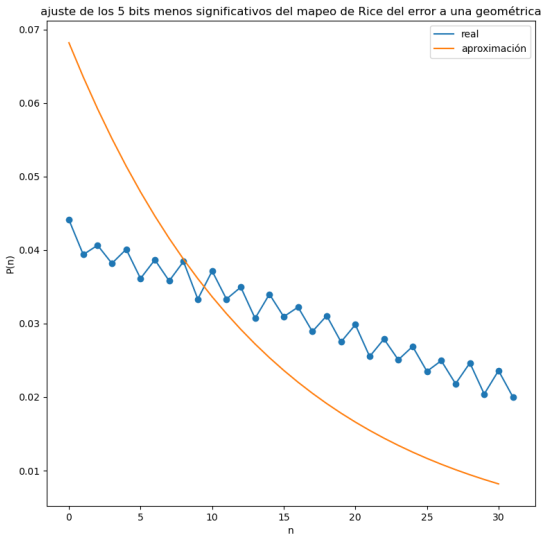

Figura 6.4: Distribución de probabilidad de los 5 bits menos significativos del error mapeado, y su aproximación por una distribución geométrica.

Para el caso de 8 bits, en general parece razonable aproximar la distribución

del error mapeado con una geométrica.

Por otro lado, el ajuste para los primeros 5 bits es muy poble. Sin embargo, si miramos el resultado de invertir el mapeo de Rice sólo a los primeros 5 bits del error ($Rice^{-1}(Rice(error)\&0x1F)$), que se muestra en la figura 6.5, vemos que éste se puede aproximar razonablemente por una distribución triangular.

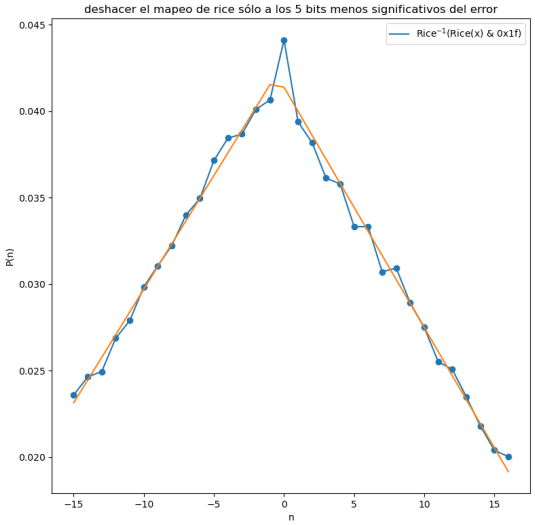

Figura 6.5: Distribución de probabilidad de los 5 bits menos significativos del error mapeado, luego de deshacer el mapeo de Rice, y su aproximación por una distribución triangular.

De toda esta sección podemos concluir que si bien separamos el error mapeado en el byte bajo y el byte alto, esta no es necesariamente la mejor separación posible, en particular porque no necesariamente separa de la mejor forma posible partes del error que tienen estadísticas diferentes. Por ejemplo tomando los 5 menos significativos bits como la parte baja y el resto como la parte alta, podemos parametrizar las distribuciones de probabilidad correspondientes, lo que permite simplificar la codificación. Por otro lado, podemos codificar dicha parte con códigos de Golomb potencia de dos.

Otras alternativas a considerar a partir de este análisis podrían ser:

- Usar Run Length Encoding para la parte alta, siempre que se trate de un 0.
- Codificar la parte baja con un código de Huffman precalculado sobre un conjunto de archivos.
- Emitir la parte baja del error sin comprimir.
- Codificar la parte baja del error con un codificador aritmético, pero ahora utilizando una tabla de 2^5 valores en vez de 2^8.

6.4. Otras codificaciones para la señal

En esta sección exploramos otras codificaciones para la señal, diferentes a la realizada en PGNanoAlgo. Preparamos un programa de benchmark que lee de un formato binario un subconjunto de las señales (sin usar POD5) y calcula el largo de código normalizado producido por diferentes codificaciones. Además medimos la eficiencia computacional en MB/s consumidos del buffer de entrada, una vez que los datos ya están en memoria.

Presentamos los resultados más relevantes en la tabla 6.2. En el anexo E se pueden encontrar los resultados de más experimentos. La tabla usa la siguiente nomenclatura para los métodos de compresión: los métodos que ya se han nombrado (Vbz, Variable Byte, PGNano) se escriben con su nombre entero; los demás métodos (que utilizan composiciones de codificación aritmética, códigos de Golomb, RLE y ZSTD) se notan con la siguiente sintaxis: < codificación parte baja > / < codificación parte alta > (<bits parte baja> - <bits parte alta>). Por ejemplo, Rice + R / G - RLE(15; R) (10-6) significa que comprimimos la diferencia de muestras consecutivas luego de aplicarles al mapa de Rice. Estas diferencias las partimos en los 10 bits menos significativos y en los 6 bits más significativos. Codificamos cada una de estas partes de los símbolos de la señal de forma distinta. Los bits menos significativos los codificamos con un codificador aritmético (R). Los bits más significativos los codificamos con códigos de Golomb (G) siempre que estos no sean 0 y en caso de ver un 0 los codificamos con RLE, permitiendo un largo de run máximo de 15 bits y comprimiendo el largo de run con un codificador aritmético (RLE(15; R)).

Las codificaciones pueden ser:

- R: Codificación aritmética.
- G: Códigos de Golomb potencia de 2.
- U: Mandamos el símbolo sin modificarlo. Cuando se aplica sobre la parte alta de la señal mapeada, antes calculamos el número máximo de bits que ocupan los enteros a codificar. Supongamos que todos los enteros son menores a 2^{12}, y separamos los 5 bits menos significativos para la parte baja, entonces en vez de usar 11 bits para la parte alta usamos 7.
- RLE(n, codif.): Codificación RLE (largo de run) cuando se detecta un entero igual a 0 donde n es el largo máximo de run permitido y codif. es la codificación (alguna de R, G o U) para los largos de run. Si el entero no es 0, se manda sin codificar.
- G - RLE(n, codif.): Si el entero es 0, se codifica como RLE(n, codif.). Si el entero no es 0, se codifica con códigos de Golomb potencia de 2.

Dada una señal $x_1, x_2, ..., x_n$, agregamos el prefijo "Rice + " a los métodos de compresión que codifiquen $Rice(x_i - x_{i-1})$ siempre que dicha transformación no esté implicita en el método de compresión. Por ejemplo, en Vbz o PGNano

Método	Tasa de compresión (bits/símbolo)	Velocidad (MB/s)
PGNano	6.95871	48.3774
Rice + R / G - RLE(15; R) (10-6)	7.10198	57.1402
Rice + R / G (5-11)	7.13079	81.5154
Rice + U / G (5-11)	7.15303	339.864
Rice + ZSTD / G (5-11)	7.16169	7.33436
Vbz	7.19872	834.849

Tabla 6.2: Resultados de diferentes algoritmos de compresión sobre un conjunto de prueba.

no incluimos el prefijo "Rice +" porque estos métodos tienen la transformación implícita en su definición.

Como podemos ver, los códigos de Golomb para la parte alta dañan la tasa de compresión respecto a PGNano pero puede ser una aproximación razonable, especialmente considerando su eficiencia computacional.

De todas las particiones de los bits en parte alta y parte baja, la partición óptima se da para 5 bits en la parte baja cuando no usamos RLE y en 10 bits para la parte baja cuando usamos RLE, que son las particiones que presentamos en esta sección.

Notamos que para el caso de la partición con 5 bits para la parte baja, la codificación con códigos de Golomb para la parte alta logra mejorar un 1 % la tasa de compresión de Vbz a un 37,88 % de su velocidad, cuando no comprimimos la parte baja.

Concentrándonos en códigos de Golomb para la parte alta, tenemos 3 variaciones: codificar la parte baja con un codificador aritmético, enviar los bits sin codificar y comprimirlos con el máximo nivel de compresión ofrecido por ZSTD.

Es interesante ver que para una misma partición en parte alta y parte baja, comprimir la parte baja con un codificador aritmético mejora muy poco respecto a enviar los bits sin codificar. A priori, esto significa que la parte baja tiene una entropía cercana a 5 bits. Si estamos usando aproximadamente 7,15 bits para el código entero, esto implica que la parte alta la comprimimos a aproximadamente 2,15 bits por símbolo. Parece razonable que intentemos mediante otras técnicas tratar de mejorar la tasa de compresión concentrándonos en la parte baja, bajo la hipótesis de que la entropía de los símbolos individuales es cercana a 5. En particular proponemos: Usar clases de contexto para la parte baja (esta idea se explorará brevemente en la sección 6.5) y usar macrosímbolos y memoria, analizando las probabilidades conjuntas y/o condicionales de la parte baja.

Si vemos los resultados que tiene correr ZSTD sobre la parte baja, vemos que los resultados empeoran. Muy probablemente ZSTD esté dañando la tasa de compresión por el costo de modelo que tiene. Es claro, es que la memoria (en la forma de cadenas de Markov) que da ZSTD no ayuda a comprimir la parte baja. En particular, inferimos que las variables aleatorias de la parte baja tienen muy baja correlación.

Por otro lado, vemos que la codificación de la parte baja con códigos de Golomb, seguida de codificar la parte alta con un codificador aritmético, tiene

resultados peores a comprimir la parte alta con códigos de Golomb cuando no usamos RLE. Luego, concluimos que en general no es adecuado comprimir la parte baja con códigos de Golomb.

Analizando los códigos de Golomb, tenemos que para cada símbolo, un código de Golomb va a utilizar al menos 2 bits. Considerando que la parte alta se comprime a 2.15 bits por símbolo, entonces estamos alcanzando resultados bastante buenos con esta codificación. Si queremos mejorar la compresión de la parte alta, lo que probablemente de más resultado es cambiar la codificación por otra. Dado que tenemos una buena tasa de compresión para la parte alta, parece razonable inclinarse por mecanismos que usen macrosímbolos para codificar o en general algun código que pueda comprimir a menos de un bit por símbolo. De particular interés es utilizar Run Length Encoding (RLE) bajo la hipótesis de que si la parte alta se hace 0 en algún momento, va a haber un fragmento de tiempo "largo" donde va a seguir siendo 0.

Si bien no hacemos un análisis estadístico de los largos de run, si los comprimimos usando un codificador aritmético tenemos una tasa de 7,10 bits por símbolo a aproximadamente un 140 % la velocidad de PGNano. Experimentando más, podría llegarse a conseguir un mejor compromiso entre la velocidad de compresión y tasa de compresión respecto a lo que ofrece PGNano y por tanto constituyen una via de investigación futura.

En conclusión obtenemos codificaciones diferentes a PGNanoAlgo que de todas formas superan a Vbz. En particular tenemos dos principales caminos: Asignar pocos bits a la parte baja, no codificarla y codificar la parte alta con códigos de Golomb, indicándonos que de cierta forma los bits menos significativos de los datos sin contextualizar son prácticamente ruido. Por otro lado podemos asignar muchos bits a la parte baja, y comprimir la parte alta con mecanismos especializados en distribuciones de probabilidad con entropía muy baja, en particular RLE.

Sería interesante intentar comprimir la parte alta con técnicas como ZSTD para analizar los resultados. Además deberíamos profundiar el análisis sobre largos de run para la parte alta y ver las estadísticas de los mismos.

6.5. Modelos de contexto para la parte baja

Como mencionamos anteriormente, otra forma de mejorar nuestros resultados es usando modelos contextuales. Nos enfocamos en la parte baja de la señal (tomada como los 5 bits menos significativos del error de predicción mapeado) ya que es difícil de comprimir con el esquema de códigos de Golomb que planteamos en la sección 6.4. En el anexo C presentamos un análisis de clases de contexto previo al análisis de la señal de la sección 6.3. Lo más notable es que el análisis de clases de contexto depende de la partición de la señal realizada, pues de cierta forma ésta partición fuerza conocimiento a priori que tenemos sobre la señal. En particular, vemos que separando en los 8 menos significativos bits como parte baja, no tenemos buenos modelos de contexto, pero sí los tenemos para los 5 bits menos significativos.

Los modelos contextuales utilizados fueron los siguientes:

- Base
- ParImpar
- ValorAbsoluto
- Signo
- 2BandasFrecuencia
- 4BandasFrecuencia

El modelo Base simplemente no contextualiza la entrada. Es la línea base de nuestro experimento.

El modelo ParImpar contextualiza según si el símbolo anterior era par o impar. Proponemos este modelo porque gráficamente parecerían favorecerse a los valores pares frente a los impares.

Los modelos ValorAbsoluto y Signo refieren a la diferencia $d_i = x_{i-1} - x_{i-2}$. De forma sencilla estos modelos intentan capturar el crecimiento o decrecimiento del error de predicción mapeado.

El modelo ValorAbsoluto separa según el predicado $p(d_i) = 1 \text{ if } abs(d_i) > 7 \text{ else } 0$, intentado ver si estuvimos recientemente en una zona de inestabilidad donde la señal cambió mucho. El modelo Signo separa según el predicado $p(d_i) = 0 \text{ if } d_i > 0 \text{ else } 1$, tratando de ver si hay una diferencia en las estadísticas cuando el error decrece o crece.

Los modelos 2BandasFrecuencia y 4BandasFrecuencia contextualizan la parte baja del error de predicción mapeado, según la **señal** antes vista. En particular, estos modelos calculan el valor máximo y mínimo de la señal y dividen ese rango en 2 (ó 4) particiones iguales, marcando "bandas de frecuencia" en la señal.

El modelo de bandas de frecuencia tiene la siguiente intuición. Según ONT, cada kmero tiene una media y desvío estándar diferente. Por el fenómeno de independización de la media explicado en la sección 4.1.1, excepto en los cambios de kmero, que podemos pensar que corresponden a la minoría de las muestras, la media característica del kmero ya no es relevante pero podemos refinar los modelos estadísticos usando su desvío estándar.

Relacionar cada muestra de señal con su kmero correspondiente presentó problemas prácticos y no produjo resultados suficientemente buenos. Por lo tanto, proponemos una solución no supervisada (modelos de bandas de frecuencia) de bajo costo computacional que tiene como objetivo inferir información estadística de los kmeros sólo a partir de la señal de secuenciación.

El modelo intenta ganar información sobre el kmero al que corresponde la muestra realizando una partición del rango de valores posibles. Como los kmeros tienen una distribución normal con una media característica, esto implica que en cada subconjunto de dicha partición, solo un subconjunto de kmeros tienen probabilidades significativas de ocurrir. Entonces, ganamos información sobre a

qué kmero corresponde cada muestra y por tanto podemos refinar su modelo estadístico.

La principal ventaja del modelo es que podemos experimentar con diferente número de bandas de frecuencias de forma de obtener un balance entre el costo de modelo que agrega tener más franjas respecto a la mejor aproximación de las estadísticas que podemos hacer. La idea es que particionando lo suficiente las muestras, podemos inferir cambios de kmero significativos y refinar nuestro conocimiento sobre la desviación estándar de las muestras. Esto puede permitirnos ajustar mejor los modelos estadísticos en cada clase de contexto.

Si bien esta es la intuición por detrás, la realidad es que en tramos no consecutivos de la señal que corresponden al mismo kmero, la desviación estándar de dichas muestras puede ser significativamente diferente. La figura 6.6 ilustra la situación.

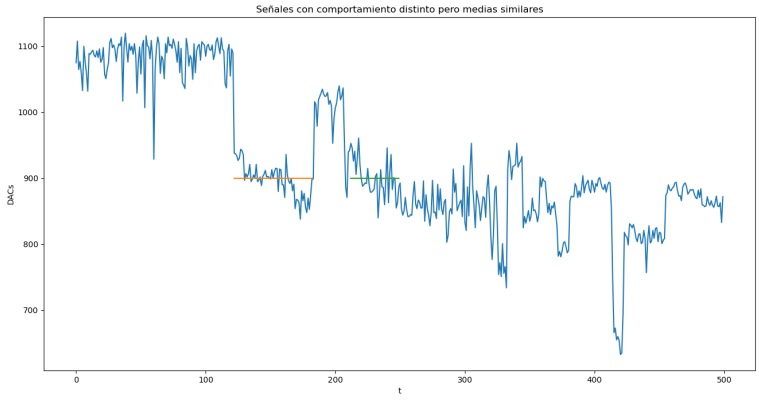

Figura 6.6: Para medias muy similares la desviación estándar puede ser diferente.

No obstante, es una idea que al menos parece lo suficientemente razonable como para probarla.

Presentamos entonces los diferentes resultados en la tabla 6.3.

Método	tasa
Base	4.95591
ParImpar	4.92054
ValorAbsoluto	4.95640
Signo	4.95636
2BandasDeFrecuencia	4.95505
4BandasDeFrecuencia	4.95065

Tabla 6.3: Resultados de modelos de contexto sobre los 5 bits menos significativos del error mapeado.

Como vemos, los modelos ValorAbsoluto y Signo dañan el largo de código. El modelo de ParImpar obtiene una mejora de 0.035 bits/símbolo posicionándose como el mejor modelo de contexto de los presentados. No obstante, la mejora obtenida no es muy grande. Respecto a los modelos de frecuencias, para 2 bandas de frecuencia el modelo no mejora, pero para 4 bandas tenemos una mejora de 0.005 bits/símbolo. Nuevamente, los resultados están lejos de ser relevantes. Queda a futuro investigar hasta que punto agregar bandas de frecuencia mejora las tasas de compresión.

Capítulo 7

Análisis sobre el desempeño de Vbz

Este capítulo intenta dar intuiciones sobre por qué Vbz tiene un buen desempeño, algo que consideramos especialmente relevante dada la falta de información que hay respecto a este tema. Adicionalmente analizar Vbz nos permite proponer un par de algoritmos para trabajos futuros, sobre los cuales no nos centramos en este proyecto porque en principio no eran compatibles con la generación de un framework con las características deseadas, particularmente con la habilidad de utilizar datos adicionales sobre las lecturas para mejorar la compresión.

En la tabla 7.1 mostramos los resultados de diferentes algoritmos de codificación de enteros. Los resultados se dan sobre el conjunto de datos R10.4.1. En particular analizamos la tasa de compresión de Vbz, así como de StreamVByte y Rice + StreamVByte (StreamVByte sobre el error de predicción mapeado con Rice). Recordamos que Vbz es el resultado de comprimir la señal con Rice + StreamVByte y ese resultado comprimirlo con ZSTD, por lo que para entender Vbz es relevante entender StreamVByte.

Observamos que StreamVByte no logra mejoras cuando lo aplicamos sobre la señal original. Es decir, generalmente usamos algún bit de cada byte de la señal, lo cual concuerda con el análisis estadístico mencionado en la subsección 6.3.

Teniendo en cuenta que el rango de digitalización de los dispositivos de

Método	Tasa de compresión (bits/símbolo)	Velocidad (MB/s)
Vbz	7.199	834
Rice + Variable byte	9.187	4282
Variable byte	16.986	5153

Tabla 7.1: Resultados de diferentes algoritmos de codificación de enteros sobre un conjunto de prueba.

secuenciación utiliza al menos 10 bits en los conjuntos de datos obtenidos, y considerando que tenemos enteros con signo codificados en complemento a 2; es muy probable que el byte alto de la señal tenga al menos un bit diferente de 0. Por lo tanto, StreamVByte casi siempre se encontrará con enteros de 2 bytes, los cuales no puede comprimir.

Por otro lado, cuando aplicamos StreamVByte a las diferencias de muestras consecutivas luego de aplicarle el mapeo de Rice, StreamVByte logra una mejora de aproximadamente 7,8 bits por símbolo. Al recordar las distribuciones de probabilidad del error de predicción mapeado, observamos que estas concentran su mayor masa de probabilidad en valores cercanos al 0. En enteros sin signo esto implica que los valores ocupan casi siempre solo el byte menos significativo del entero. De esta manera, StreamVByte usualmente podrá ahorrarse el byte más significativo de cada entero de 16 bits, lo que es parte importante de las mejoras que logra Vbz.

Considerando Vbz vemos que el paso de compresión con ZSTD produce una mejora de aproximadamente 2 bits por símbolo.

Es importante recordar que la implementación de StreamVByte usada en Vbz escribe primero en el buffer de memoria todas las máscaras para saber el número de bytes ocupado por cada entero, y despues escribe el stream de enteros codificado.

Supongamos arbitrariamente que un 0 en las máscaras de StreamVByte indica que se usa un solo byte y 1 que se usan dos bytes para el entero correspondiente. Es razonable plantear como hipótesis que el stream de datos que consiste en la máscaras escritas al principio por StreamVByte, es altamente compresible si usamos codificaciones con macrosímbolos o RLE. En particular, parece razonable suponer que $P(X_i = 0) \gg P(X_i = 1)$, pues la mayoría de los enteros son menores a 256, donde X_i es la iésima máscara escrita por StreamVByte. Esto implica entonces que el paso con ZSTD de Vbz va a comprimir muy bien las máscaras escritas por StreamVByte, y StreamVByte usualmente podrá enviar sólo un byte de cada entero.

En lo que respecta a los bytes, si las máscaras de StreamVByte ocupan un bit por símbolo, pero Vbz mejora en aproximadamente 2 bits respecto a Rice + StreamVByte, entonces concluimos que Vbz también logra identificar algunos patrones en la secuencia de bytes emitidos. No obstante, parecería que los bytes en sí no son muy compresibles. Para ver esto razonemos (informalmente) de la siguiente manera: Vbz comprime toda la señal a aproximadamente 7,2 bits por símbolo. Las máscaras no ocupan más de un bit por símbolo. Entonces, los bytes de los enteros ocupan al menos 6,2 bits por símbolo de 16 bits. Si asumimos que la mayoría de los enteros son de un byte, entonces "tenemos" que los bytes al menos ocupan aproximadamente 6,2 bits por símbolo de 8 bits.

Resumiendo, Vbz es efectivo porque StreamVByte le permite evitar un gran porcentaje de los bytes de los enteros de la señal mapeada. Además, StreamVByte escribe continuas todas las máscaras para indicar cuantos bits ocupa cada entero, lo que permite a ZSTD inferir rápidamente que la mayoría de estas máscaras son 0 y por tanto comprimirlas a menos de un bit por símbolo, usando macrosímbolos. Por último, ZSTD logra encontrar algunas regularidades en los

bytes de la señal. Notemos que si Vbz usara otras implementaciones de Variable Byte, por ejemplo alguna implementación que escribiera para cada símbolo su máscara seguido de sus bytes correspondientes, entonces ZSTD estaría analizando al mismo tiempo dos tiras de datos con estadísticas diferentes, lo que intuitivamente puede afectar el desempeño del algoritmo, dadas las características del mismo.

Con esto, una primera mejora que podemos proponer a Vbz es en la etapa de compresión con ZSTD, comprimir por separado a las máscaras respecto a los bytes, reiniciando la memoria de ZSTD para cada secuencia de datos.

Notamos que hay un paralelismo entre el análisis de la sección 6.3 y lo que hace Vbz. Este paralelismo genera las mismas dudas exploradas en dicha sección. En particular, el punto de corte óptimo para este algoritmo va a depender del tipo de poro y rango de digitalización de los aparatos, pero Vbz siempre va a partir la señal en bytes. Esto nos motiva a presentar el algoritmo 4, StreamVBit que de cierta forma generaliza a StreamVByte. Sería interesante experimentar cómo Vbz se comporta al reemplazar StreamVByte con StreamVBit, agregando algún método que permita eficientemente determinar buenos puntos de corte para separar la señal en una parte alta y baja.

Algorithm 4 StreamVBit16

Require: entrada: Array[uint16]
Require: k: uint (punto de corte)
Ensure: Array[uint16]
1: máscaras ← Array[bits]
2: datos ← Array[bytes]
3: **for all** x ∈ entrada **do**
4: **if** $x \geq 2^k$ **then**
5: máscaras.append(1)
6: datos.append(BitsMenosSignificativos(x, k))
7: datos.append(BitsMásSignificativos(x,k))
8: **else**
9: máscaras.append(0)
10: datos.append(BitsMenosSignificativos(x,k))
11: **end if**
12: **end for**
13: **return** concat(máscaras, datos)

Evidentemente, este algoritmo necesita que su resultado sea codificado posteriormente con otro algoritmo de compresión que logre codificar de forma eficiente secuencias de enteros de k bits (por ejemplo un codificador aritmético o ZSTD).

Otra idea que podemos plantear, especialmente para reducir el costo de modelo de ZSTD se basa en lo siguiente: El mayor problema del predictor presentado se da en los saltos entre kmeros, los que podemos suponer razonablemente espaciados en el tiempo. Con todo esto podemos asumir que independientemen-

te de si el valor anterior era un 0 o un 1, el stream de bits probablemente seguirá con una corrida de 0's. Entonces, aplicando codificaciones de macrosímbolos esperamos obtener una buena tasa de compresión para las máscaras. Esto nos permitiría no usar ZSTD para las máscaras y en su lugar utilizar RLE. Para los datos podríamos usar, por ejemplo, alguna de las codificaciones propuestas en 6.4, especialmente condicionando es la máscara correspondiente a dicho símbolo. Con esto podemos proponer el siguiente esquema de algoritmo "RVbit", presentado en el algoritmo 5.

Algorithm 5 RVbit

Require: entrada: Array[int16]
Require: algoritmoParteBaja: Compresor
Require: algoritmoParteAlta: Compresor
Ensure: Array[uint16]

```
deltas ← Rice(DeltaEncode(entrada))
k ← DeterminarMejorPartición(deltas)
máscaras, datos ← StreamVBit(deltas, k)
máscaras_comprimidas ← RLE(máscaras)
datos_comprimidos ← new Array[bytes]
for i ∈ {0..len(deltas) − 1} do
    if máscaras[i] == 0 then
        datos_comprimidos.append(algoritmoParteBaja(parteBaja(deltas[i])))
    else
        datos_comprimidos.append(algoritmoParteBaja(parteBaja(deltas[i])))
        datos_comprimidos.append(algoritmoParteAlta(parteAlta(deltas[i])))
    end if
end for
return concat(máscaras, datos)
```

No implementamos ninguno de estos algoritmos en el proyecto, principalmente porque no estaban alineados con la generación de un framework basado en metadatos de reads. Además, sería necesario realizar prototipos que validen si los algoritmos alcanzan buenas tasas de compresión y luego sería un reto práctico lograr que estos algoritmos tuvieran un desempeño computacional comparable con Vbz. Adicionalmente, deberíamos evaluar si la complejidad de este algoritmo tiene ganancias que justifiquen su uso, en particular dado que consiste en mucho más pasos que Vbz, un algoritmo que ya explota esta idea pero de forma muy eficiente tanto al comprimir como al descomprimir. Recordemos que los algoritmos basados en LZ77 tienen velocidades de descompresión excelente. Por último, queda abierto el problema de cómo determinar la mejor partición de la señal. No obstante, son posibles vías de investigación futura.

Capítulo 8

Conclusiones y Trabajo Futuro

En el proyecto tuvimos dos ejes centrales: generar un producto bien estructurado que permitiera modificaciones posteriores con otros métodos de compresión y realizar experimentos con diferentes métodos de compresión, junto a la realización de un análisis de la señal a comprimir, de forma de obtener mejores tasas de compresión que Vbz.

En lo que respecta al primer eje del proyecto, tenemos como resultado un framework que permite prototipar rápidamente compresores integrados a POD5, levantando algunas de las restricciones que esta biblioteca impone a los algoritmos de compresión (como por ejemplo no conocer el tipo de poro).

Este framework facilita la implementación e integración de nuevos métodos de compresión a POD5. El framework también previene la implementación de métodos de compresión no integrados a la biblioteca, de forma de independizarse de las restricciones de POD5, si la implementación particular lo requiriera. Además, se incluyen los algoritmos PGNano5 y PGNanoS como líneas base dentro del framework desarrollado.

Por otro lado el framework presenta un conjunto de scripts de análisis de resultados y automatización de pruebas que agiliza los ciclos de desarrollo de nuevos métodos de compresión.

Como resultado de dicha implementación, obtenemos un algoritmo de compresión que mejora marginalmente la tasa de compresión de Vbz bajando el espacio ocupado a un 97 % a 99 % de lo ofrecido por Vbz.

De la parte experimental del proyecto logramos comprender mejor cómo funciona Vbz (y por qué es eficaz). También obtenemos posibles caminos para mejorar a los algoritmos presentados.

En particular vimos que el error de predicción de la señal (calculado como la diferencia de dos muestras consecutivas) lo podemos separar en dos partes: una parte alta, donde la codificación de macrosímbolos parece crucial y una parte baja, que es inherentemente difícil de comprimir cuando ocupa pocos bits (por

ejemplo en la separación 5-11 de la sección 6.4), y no parece beneficiarse mucho de algoritmos con memoria. No obstante, esta parte baja se puede comprimir mejor si utilizamos modelados estadísticos que tengan una convergencia más rápida que ZSTD.

Además vimos que la separación óptima del error de predicción en parte baja y parte alta no necesariamente se da en separar el byte alto del bajo.

En conjunto a esto, vimos que si bien cada kmero tiene diferentes medias características para la señal de secuenciación, al comprimir la resta de dos muestras consecutivas nos independizamos de esta media característica, excepto en regiones cercanas al cambio de kmero. No obstante, sería interesante contextualizar los modelos estadísticos según el kmero presente en el nanoporo. El modelo de bandas de frecuencia presentado, puede dar una aproximación no supervisada al problema.

Por otro lado, si bien el mejor predictor para la siguiente muestra dadas las anteriores no es $p(x_{n-1}) = x_{n-1}$, este predictor es muy eficiente y logra resultados razonablemente buenos.

Todo esto nos lleva a proponer la búsqueda de algoritmos dentro del esquema presentado en el algoritmo 6.

Algorithm 6 *compress*

Require: $xs : int16_t$ es la señal de secuenciación

Require: $modelo_{alto}$: $Modelo$ es un algoritmo de compresión para la parte alta del error de predicción que utiliza macrosímbolos. Ejemplos podrían ser ZSTD o RLE.

Require: $modelo_{bajo}$: $Modelo$ es un algoritmo de compresión para la parte baja del error de predicción. Este algoritmo debe concentrarse en tener una convergencia rápida. En este modelo concentramos el estudio de clases de contexto.

1: $ys \leftarrow DeltaZigZag(xs)$
2: Estimar k número de bits para la parte baja
3: Separar ys en ys_{baja} (k bits menos significativos) y ys_{alta} (bits restantes)
4: Comprimir ys_{alta} con $modelo_{alto}$
5: Comprimir ys_{baja} con $modelo_{bajo}$

Además presentamos un conjunto de experimentos e ideas que pueden ser un buen punto de partida para posteriores investigaciones. En particular, planteamos análisis sobre los predictores de la señal, sobre clases de contexto involucradas y sobre diferentes codificaciones.

Algunas posibles vías de investigación y trabajo futuro podrían ser:

- Analizar las clases de contexto para la separación 10-6 de los símbolos en parte alta y parte baja (codificación Rice + R / G - RLE(15, R) (10-6)).
- Mejorar el predictor para obtener distribuciones de probabilidad del error de predicción con menor entropía.

- Experimentar con predictores preentrenados o entrenados sobre todo el archivo POD5 (predictores de clases B y E).
- Iterar en el desarrollo de modelos de contexto, en particular sobre la idea de separación en bandas de frecuencia.
- Desarrollar e implementar algoritmos para predecir cambios de kmero.
- Aproximar la distribución de los errores de predicción como mezclas de otras distribuciones de probabilidad.
- Estudiar las estadísticas de largo de run para la parte alta de la señal.
- Desarrollar e implementar algoritmos eficientes que permitan determinar los puntos de corte óptimos para la señal en una parte baja y una parte alta.
- Ampliar los experimentos con otros datasets.
- Optimizar las implementaciones existentes utilizando paralelismo SIMD a nivel de bit.
- Exponer los nuevos métodos de compresión en la API de Python dada por POD5.
- Explorar el esquema de algoritmo RVBit.
- Proponer mejoras a la detección de eventos incluida en PGNanoAlgo. En particular explorar el uso del algoritmo de Viterbi y cómo esto impacta la eficiencia computacional.
- Mejorar la detección de eventos de salto de kmero para contextualizar la señal. En particular, proponer un modelo de clases de contexto que agrupe estadísticas para todas las muestras cercanas a la muestra que produce el evento (por ejemplo una ventana de tres muestras) bajo la intuición de que el movimiento de un kmero por un nanoporo es un fenómeno continuo.

Referencias

Alvaro Martín. (2021). *Compresión de Datos sin Pérdida Codificación Aritmética.* https://eva.fing.edu.uy/pluginfile.php/339430/mod_resource/content/3/CodificacionAritmetica.pdf. (Ultimo acceso: 2023-29-10)

Cover, T. M., y Thomas, J. A. (2006a). *Elements of information theory* (2nd ed.). John Wiley & Sons.

Cover, T. M., y Thomas, J. A. (2006b). *Elements of information theory* (2nd ed.). John Wiley & Sons.

Cover, T. M., y Thomas, J. A. (2006c). *Elements of information theory* (2nd ed.). John Wiley & Sons.

Daniel Lemire, Nathan Kurz, Christoph Rupp. (2017). *Stream vbyte: Faster byte-oriented integer compression.* https://arxiv.org/abs/1709.08990. (Ultimo acceso: 2023-29-10)

Gadiel Seroussi. (2023a). *Applications of information theory in image processing 4. the loco-i lossless image compression algorithm and the jpeg-ls standard.* https://eva.fing.edu.uy/pluginfile.php/354813/mod_resource/content/12/ATIPI2023-4.pdf. (Ultimo acceso: 2023-29-10)

Gadiel Seroussi. (2023b). *Applications of information theory in image processing 4. the loco-i lossless image compression algorithm and the jpeg-ls standard.* https://eva.fing.edu.uy/pluginfile.php/354813/mod_resource/content/12/ATIPI2023-4.pdf. (Ultimo acceso: 2023-29-10)

Gamaarachchi, H., Samarakoon, H., Jenner, S., y et al. (2022). Fast nanopore sequencing data analysis with slow5. *Nat Biotechnol*, *40*, 1026–1029. Descargado de https://doi.org/10.1038/s41587-021-01147-4 (Ultimo acceso: 2023-3-12)

Gigante, S. (2022). *Picopore: A tool for reducing the storage size of oxford nanopore technologies datasets without loss of functionality.* https://github.com/scottgigante/picopore. (Ultimo acceso: 2023-22-11)

Jurafsky, D., y Martin, J. H. (2009). *Speech and language processing.* Pearson Education International.

National Human Genome Research Institute. (2022). *Human genome project fact sheet.* https://www.genome.gov/about-genomics/educational-resources/fact-sheets/human-genome-project. (Ultimo acceso: 2023-07-05)

National Human Genome Research Institute. (2023a). *Dna sequencing.* https://www.genome.gov/genetics-glossary/DNA-Sequencing. (Ultimo acceso: 2023-07-05)

National Human Genome Research Institute. (2023b). *Dna sequencing fact sheet.* https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Fact-Sheet. U.S. National Academy of Sciences. (Ultimo acceso: 2023-07-05)

Nicolás Izquierdo, G. L. (2020). *Compresión de datos generados por secuenciación de adn por nanoporos* (Tesis de Master no publicada). Universidad de la República (Uruguay). Facultad de Ingeniería. (Ultimo acceso: 2023-22-11)

Oxford Nanopore Technologies. (2022a). *check_pod5_files_equal.py.* https://github.com/nanoporetech/pod5-file-format/blob/master/python/pod5/test_utils/check_pod5_files_equal.py. (Ultimo acceso: 2023-22-10)

Oxford Nanopore Technologies. (2022b). *How it works - basecalling.* https://nanoporetech.com/how-it-works/basecalling. (Ultimo acceso: 2023-29-10)

Oxford Nanopore Technologies. (2022c). *How it works - basecalling - nanopores.* https://nanoporetech.com/how-it-works/basecalling. (Ultimo acceso: 2023-29-10)

Oxford Nanopore Technologies. (2022d). *London calling 2022 collection.* https://nanoporetech.com/london-calling-2022-collection video "Update from Oxford Nanopore Technologies" minuto 11:30.

Oxford Nanopore Technologies. (2022e). *nanoporetech/vbz_compression: Vbz compression plugin for nanopore signal data - github.* https://github.com/nanoporetech/vbz_compression. (Ultimo acceso: 2023-4-11)

Oxford Nanopore Technologies. (2022f). *Oxford nanopore delivers technology update at annual london calling conference: bringing together years of innovation to showcase one sensing platform for all biological analyses.* https://nanoporetech.com/about-us/news/oxford-nanopore-delivers-technology-update-annual-london-calling-conference-bringing. (Ultimo acceso: 2023-29-10)

Oxford Nanopore Technologies. (2022g). *Pod5-file-format/docs/tables.* https://github.com/nanoporetech/pod5-file-format/tree/master/docs/tables. (Ultimo acceso: 2023-11-06)

Oxford Nanopore Technologies. (2022h). *Pod5 format specification.* https://pod5-file-format.readthedocs.io/en/latest/SPECIFICATION.html. (Ultimo acceso: 2023-07-06)

Oxford Nanopore Technologies. (2022i). *Promethion 24 & 48.* https://nanoporetech.com/products/sequence/promethion-24-48. (Ultimo acceso: 2023-4-11)

Oxford Nanopore Technologies. (2023a). Minion mk1b it requirements [Manual de software informático]. https://community.nanoporetech.com/requirements_documents/minion-it-reqs.pdf. (Ultimo acceso: 2023-29-10)

Oxford Nanopore Technologies. (2023b). *Nanopore dna sequencing.* https://en.wikipedia.org/wiki/Sanger_sequencing. (Ultimo acceso: 2023-07-06)

Oxford Nanopore Technologies. (2023c). *nanoporetech/dorado: Oxford nanopore's basecaller - github.* https://github.com/nanoporetech/dorado. (Ultimo acceso: 2023-29-10)

Oxford Nanopore Technologies. (2023d). *nanoporetech/remora: Methylation/modified base calling separated from basecalling - github.* https://github.com/nanoporetech/remora. (Ultimo acceso: 2023-29-10)

Oxford Nanopore Technologies. (2023e). *nanoporetech/remora: Methylation/modified base calling separated from basecalling - github.* https://github.com/nanoporetech/remora#python-api. (Ultimo acceso: 2023-29-10)

Oxford Nanopore Technologies. (2023f). *Pod5: a high performance file format for nanopore reads.* https://github.com/nanoporetech/pod5-file-format. (Ultimo acceso: 2023-07-05)

Oxford Nanopore Technologies. (2023g). *pod5 convert fast5.* https://github.com/nanoporetech/pod5-file-format/tree/master/python/pod5#pod5-convert-fast5. (Ultimo acceso: 2023-22-10)

Samarakoon, H., Ferguson, J. M., Jenner, S. P., Loman, N. J., y Quick, J. (2023). Flexible and efficient handling of nanopore sequencing signal data with slow5tools. *Genome Biology*, *24*(69). Descargado de https://doi.org/10.1186/s13059-023-02910-3 (Ultimo acceso: 2023-3-12) doi: 10.1186/s13059-023-02910-3

Seroussi, G. (2021). *Lossless source coding lempel-ziv coding lempel-ziv 1977 (lz77).* https://eva.fing.edu.uy/pluginfile.php/29106/mod_resource/content/3/Diapositivas/2021/charla5LZ-LZ77.pdf. (Ultimo acceso: 2023-29-10)

The Apache Software Foundation. (2016a). *Apache arrow overview.* https://arrow.apache.org/overview/. (Ultimo acceso: 2023-07-06)

The Apache Software Foundation. (2016b). *Tabular data.* https://arrow.apache.org/docs/cpp/tables.html. (Ultimo acceso: 2023-07-06)

Wikipedia. (2023a). *Base calling.* https://en.wikipedia.org/wiki/Base_calling. (Ultimo acceso: 2023-29-10)

Wikipedia. (2023b). *Dna sequencing.* https://en.wikipedia.org/wiki/DNA_sequencing. (Ultimo acceso: 2023-07-06)

Wikipedia. (2023c). *K-mer.* https://en.wikipedia.org/wiki/K-mer. (Ultimo acceso: 2023-29-10)

Wikipedia. (2023d). *Nanopore sequencing.* https://en.wikipedia.org/wiki/Nanopore_sequencing. (Ultimo acceso: 2023-07-06)

Wikipedia. (2023e). *zstd.* https://en.wikipedia.org/wiki/Zstd. (Ultimo acceso: 2023-29-10)

Y. Collet; M. Kucherawy, Ed. (2021). *Zstandard compression and the 'application/zstd' media type.* https://datatracker.ietf.org/doc/html/rfc8878. (Ultimo acceso: 2023-29-10)

Yann Collet; Chip Turner. (2016). *Smaller and faster data compression with zstandard.* https://engineering.fb.com/2016/08/31/core-infra/smaller-and-faster-data-compression-with-zstandard/. (Ultimo acceso: 2023-29-10)

yourgenome. (2017). *What is oxford nanopore technology (ont) sequencing?* https://www.yourgenome.org/facts/what-is-oxford-nanopore-technology-ont-sequencing. (Ultimo acceso: 2023-24-11)

Anexo A

Datos para recuperar datasets

Esta sección del anexo presenta las tablas A.1, A.2 y A.3 que son de utilidad para recuperar los conjuntos de prueba sobre los que trabajamos.

Nombre abreviado	Link de obtención
GIAB10.4.1	s3://ont-open-data/giab_lsk114_2022.12/
GIAB9.4.1	s3://ont-open-data/gm24385_2020.09/
GIAB10.3	s3://ont-open-data/gm24385_2020.09/
Fly	s3://ont-open-data/contrib/melanogaster_bkim_2023.01/flowcells/

Tabla A.1: Enlaces para obtener dataset utilizados.

Nombre abreviado	Página web
GIAB10.4.1	https://labs.epi2me.io/askenazi-kit14-2022-12/
GIAB9.4.1	https://labs.epi2me.io/gm24385_2020.09/
GIAB10.3	https://labs.epi2me.io/gm24385_2020.09/
Fly	https://www.ncbi.nlm.nih.gov/bioproject/?term=PRJNA914057

Tabla A.2: Páginas web donde se describen más en detalle los datasets.

Nombre abreviado	Nombre
GIAB10.4.1	Genome in a Bottle - Ashkenazi Trio
GIAB9.4.1	GM24385 (Genome in a bottle; 9.4.1)
GIAB10.3	GM24385 (Genome in a bottle; 10.3)
Fly	Drosophila melanogaster Nanopore Q20 sequencing

Tabla A.3: Nombre completo de cada dataset.

Anexo B

Experimentos sobre el predictor

En este capítulo del anexo detallamos los experimentos a los que refiere la sección 6.2, detallando algunos más de los experimentos realizados y ampliando los aspectos que consideramos de cada uno de ellos, particularmente su facilidad de implementación en PGNano5 y PGNanoS así como su costo de modelo.

Concretamente, podemos pensar en al menos cinco clases de predictores con los cuales experimentar, según como se adapten a los datos ya vistos.

A. Predictores fijos no entrenables.

B. Predictores fijos preentrenados.

C. Predictores adaptativos sobre la señal entera.

D. Predictores adaptativos sobre chunks individuales.

E. Predictores adaptativos sobre todo un archivo de secuenciación.

Para aclarar, damos un ejemplo de cada clase.

Un predictor de la clase A es el predictor ya implementado (p_{base}).

Un predictor de la clase B puede ser un predictor de la forma $\hat{x}_n = x_{n-1}w_1 + x_{n-2}w_2$, donde, w_1 y w_2 estén precalculados sobre un conjunto razonable de archivos de secuenciación y queden fijos mediante alguna convención, por ejemplo definidos en tiempo de compilación.

Los predictores de las clases C, D y E son análogos a los predictores de la clase B, únicamente que definimos algún esquema de inicialización de los parámetros del predictor y luego los ajustamos conforme la señal es procesada.

La clase A de predictores es especialmente atractiva por su simplicidad y es la única clase que tratamos en el cuerpo principal del informe.

La clase B de predictores tiene algunas dificultades de implementación práctica. La forma más directa de implementarlos implicaría fijar los parámetros en tiempo de compilación. Para que esto sea una solución viable, requerimos que

los parámetros sea estables en el tiempo. Conforme la tecnología de secuenciación por nanoporos avanza, se crean nuevos poros y mejoran los aparatos existentes, todo lo que puede alterar esos parámetros predefinidos en tiempo de compilación. Luego, esta alternativa no es viable dada la rápida evolución que existe en el área. Otra solución sería incluir los parámetros de alguna forma en el archivo. Con poco esfuerzo adicional sobre esta solución, podemos implementar predictores de las clases C, D o E, que intuitivamente deberían dar mejores resultados.

Con esto, los predictores de clase B se descartan excepto por la idea de inicializar de alguna forma conveniente los parámetros de un predictor.

Los predictores de clase C se pueden implementar tanto en PGNano5 como en PGNanoS. Si el cálculo puede hacerse on-line conforme los datos son pasados al compresor, no hay que tener mayores consideraciones. La alternativa es utilizar un cabezal al principio del stream comprimido que le indique al descompresor los parámetros a usar. Evidentemente el número de parámetros y por tanto la complejidad del predictor no puede ser muy grande para no incurrir en un costo de modelo muy alto.

Los predictores de clase D alivianan este problema, pudiendo inferir los parámetros sobre una señal entera. Conforme a todo lo explicado anteriormente, un predictor de este estilo sería apropiado para implementarse solo en PGNanoS, dada la independencia en el procesamiento de chunks distintos que existe en POD5.

Por último, los predictores de clase E se pueden implementar extendiendo la representación intermedia utilizada por PGNanoS. Si en vez de utilizar como formato intermedio un archivo POD5 "plano" con la señal comprimida, se le agrega a este un cabezal con los parámetros necesarios, resolvemos el problema.

Calcular los parámetros sobre todo un archivo de secuenciación puede bajar aún más la penalidad por símbolo del costo de modelo; peligrando que los parámetros no se ajusten particularmente bien a ninguna de las señales independientes.

Por otro lado, su implementación no es directa si se quiere reutilizar la API de POD5 de forma eficiente. Por esto, en este proyecto no trabajamos sobre esta clase de predictores, aunque son una posible vía de investigación futura.

Entonces trabajamos con las clases A, C y D de predictores.

Recordamos que para la evaluación de los predictores presentamos la tasa de bits por símbolo aplicada al mapeo de Rice de la predicción, separando los 8 bits menos significativos de los 8 bits más significativos.

Por último, recordamos que los experimentos se corren utilizando la API de Python de POD5, compilada y linkeada contra la biblioteca original de POD5. Esta biblioteca presenta suficientes abstracciones tales que es trivial reutilizar la implementación de un predictor de clase C para uno de clase D, lo único que hay que hacer es pedir la señal entera de un read en vez de pedirla de a chunks. Por esto hablamos indistintamente de predictores de clase C y D.

Método	largo de código de chunks (bps)	largo de código de señal entera (bps)
vbz	7.40624	7.40589
base	7.12747	7.12061
lineal	7.23374	7.22820
neuronal	7.54878	7.35492
diferencias-negativa	7.66367	7.66312
diferencias-positiva	7.81905	7.81086
dif-1-negativa	7.11647	7.10972
dif-1-positiva	7.14134	7.13420

Tabla B.1: Estimación del largo de código esperado del error de predicción para diferentes predictores.

B.1. Descripción de los predictores involucrados

Antes de analizar los resultados, definimos que predictores vamos a comparar.

B.1.1. Predictores de clase A

En esta clase tenemos cinco predictores.

Primeramente tenemos los predictores p_{base}, $p_{dif-1-negativa}$ y $p_{dif-1-positiva}$ ya definidos en la sección 6.2.

Recordamos que $p_{dif-1-negativa}$ y $p_{dif-1-positiva}$ intentan predecir que la señal va a cambiar respecto al valor anterior, pero no hacen suposiciones fuertes de cuanto va a cambiar. Para ello definimos los predictores $p_{diferencias-negativa}$ y $p_{diferencias-positiva}$ que intentar predecir el valor absoluto de este cambio como $x_{n-1} - x_{n-2}$.

Luego $p_{diferencias-negativa} = x_{n-1} - (x_{n-1} - x_{n-2}) = x_{n-2}$ y $p_{diferencias-positiva} = x_{n-1} - (x_{n-1} - x_{n-2}) = -x_{n-2}$.

B.1.2. Predictores de clase C y D

En esta clase trabajamos con dos predictores, un predictor lineal de la forma $p_{lineal} = x_{n-1}w_1 + x_{n-2}w_2 + x_{n-3}w_3 + b$. El otro predictor que usamos es una red neuronal ($p_{neuronal}$) simple con 3 capas ocultas, cada una con 3 neuronas. Esta red toma como entrada las 3 muestras anteriores. Todas las capas tienen como función de activación la función ReLU (Rectified línear Unit), una función que vale x si $x > 0$ y vale 0 si $x <= 0$.

B.2. Análisis de resultados

La tabla B.1 presenta los resultados, donde estimamos el largo de código esperado como el menos logaritmo de la probabilidad de los símbolos. Incluimos a vbz como punto de referencia.

De los resultados podemos ver que, excepto el predictor $p_{dif-1-negativa}$, el predictor de base supera al resto de predictores. Con esto concluimos que predecir una muestra como el valor de la muestra anterior parece ser razonablemente efectivo.

Viendo el resultado de $p_{diferencias-negativa}$, para proponer un predictor que aproveche mejor el fenómeno en que se basa $p_{dif-1-negativa}$ deberíamos ajustar el parámetro que representa cuanto varía la señal respecto a la muestra anterior. Esto requeriría más investigación que decidimos no hacer en este proyecto pero es una potencial vía de trabajo futuro.

Anexo C

Experimentos sobre las clases de contexto del modelo

Como mencionamos en 6.5, otra técnica que puede ayudarnos a mejorar nuestro modelo estadístico es usar clases de contexto. Este capítulo del anexo presenta un análisis de clases de contexto para la partición original de la señal en el byte bajo y alto de la misma. Principalmente lo incluimos por completitud y para respaldar la afirmación de que el punto de corte de la señal impacta en la efectividad de las clases de contexto, especialmente porque vimos que particionar la señal de forma acorde separa valores que tienen estadísticas muy distintas. Además, este capítulo documenta algunos modelos de contexto e ideas que no produjeron buenos resultados, pero que son importantes para ganar intuición sobre las estadísticas de la señal y posiblemente retomar algunas ideas en futuras iteraciones.

Experimentamos con los siguientes modelos:

- Base
- Vbz
- TiempoParImpar
- ParImpar
- Estabilidad
- Signo
- Thresholds
- SignoYThresholds

Todos los modelos (excepto Vbz) operan con el predictor base ($p(x_{n-1}) = x_{n-1}$), codificando el error de predicción con el mapeo de Rice y recolectando estadísticas separadas para el byte alto y bajo del código. Esto simula lo implementado en C++, pero contextualizando la señal de forma diferente.

El modelo Vbz corresponde simplemente a comprimir la señal con Vbz.

El modelo base no implementa clases de contexto. Es la línea base de nuestro experimento.

El modelo TiempoParImpar recolecta estadísticas diferentes según si $n\,\%2 = 0$, donde n es el número de muestras vistas. Este modelo surge de la misma intuición de que la señal decrece inmediatamente luego de crecer y viceversa. Además, si los dos sensores de los poros R10 produjeran muestras intercaladas, este modelo asignaría clases de contexto diferentes a cada sensor.

Los modelos ParImpar y Signo ya se presentaron en la sección 6.5.

El modelo Thresholds es un modelo que se parametriza al inicializarse y dada una lista ordenada de valores límite $[t_1, t_2, ..., t_m]$ de largo m, donde, $t_i > 0 \forall i \in 1..m$ y $\forall i \in 2..m . t_{i-1} < t_i$. Si existe un i tal que $t_i <= abs(x_{n-1} - x_{n-2}) < t_{i+1}$ la clase de contexto es $i - 1$, en caso contrario, la clase de contexto es m.

Es decir, contextualiza según que tanto decreció o creció la señal en el pasado cercano, de cierta forma intentando separar los saltos bruscos de aquellas regiones más planas de la señal. Simula de forma muy básica un modelo de "rugosidad" o de nivel de actividad de la señal. Cuando presentemos los resultados, nos referiremos por Thresholds$[t_1; t_2; ...]$ para el modelo de Thresholds que usa los valores $t_1, t_2, ...$

El modelo SignoYThresholds mezcla el modelo de Thresholds con el de SignoDiferencias. Tiene una clase de contexto por cada valor límite. Además tiene clases de contexto diferentes según si $x_{n-1} - x_{n-2}$ es negativo o positivo.

Por último, el modelo de estabilidad intenta capturar de alguna forma si la señal es estable o no.

Intuitivamente, intenta ver que tanto cambió la señal en el pasado cercano. Mientras la señal sea "estable", codificamos sólo la parte baja del código del error de predicción. Si la señal se vuelve "inestable", emitimos un símbolo de escape de 9 bits, para emitir durante un período máximo de 5 muestras la parte baja y alta de la señal, de forma separada. Luego de esas 5 muestras (o antes, si la historia de la señal lo admite), consideramos la señal como estable nuevamente, y emitimos un nuevo símbolo de escape.

Evidentemente el modelo de estabilidad tiene un conjunto de parámetros que se podrían ajustar, pero dados experimentos posteriores y los resultados de este experimento, decidimos no continuar sobre este camino.

Presentamos en la tabla C.1 los resultados de los diferentes modelos de contexto.

Como se observa de los datos, estos modelos de contexto no dan buenos resultados. Los resultados empeoran pues al agregar clases de contexto, el número de parámetros del modelo crece y por tanto crece el costo de modelo. Además, si podemos comprimir mejor estas clases de contexto por separado, la mejora que obtenemos no compensa el costo de modelo adicional.

Nombre	H(X) de chunks (bps)	H(X) de señal entera (bps)
Base	7.11361	7.11171
Signo	7.12980	7.11560
Thresholds-[5]	7.14531	7.13104
Thresholds-[3]	7.14651	7.13345
ParImpar	7.16194	7.14633
TiempoParImpar	7.16207	7.14642
Thresholds-[3; 5]	7.17621	7.15389
SignoYThresholds-[3]	7.18459	7.15312
SignoYThresholds-[5]	7.18585	7.15246
Thresholds-[3; 5; 10]	7.20510	7.17140
SignoYThresholds-[3; 5]	7.23366	7.18784
SignoYThresholds-[3; 5; 10]	7.28738	7.22386
Vbz	7.38983	7.40003
Estabilidad	7.94622	7.93160

Tabla C.1: Resultados de modelos de contexto.

Particularmente relevante es el modelo de estabilidad. Sus resultados dejan claro que es necesario ajustar parámetros para que tenga sentido dicho modelado.

Por otro lado vemos que no hay diferencias significativas en estos modelos entre aplicar los modelos a la señal entera o por chunks.

Anexo D

Protocolo de acceso POD5

La intención de este anexo es ilustrar los aspectos más importantes del protocolo de escritura de datos de POD5 que describimos brevemente en la sección 4.3.3. Vale la pena notar que algunos de los cambios requeridos no se introducen ni en dicha sección ni en el anexo por brevedad.

D.1. Protocolo de acceso de POD5

El protocolo de acceso es relevante porque restringe el flujo de datos desde un cliente hacia la biblioteca, lo que nos permite hacer cambios puntuales que permitan agregar metadatos sobre los reads al compresor. Presentamos algunos puntos particulares del protocolo, basándonos en la API de C, que son relevantes para la solución propuesta.

D.1.1. Protocolo de acceso de POD5 - Comentarios generales

Cuando tratamos con el protocolo de acceso de POD5, empezamos a ver diferencias entre qué es un record batch para POD5 internamente y qué se le expone a un cliente.

Primeramente, hay diferencias semánticas. Un cliente pierde toda la semántica de un record batch relacionada a Arrow y a las diferentes tablas subyacentes. Esto evita que el cliente pueda escribir de forma arbitraria en un record batch y por tanto permite a la librería forzar las invariantes de dominio que existen (por ejemplo toda señal pertenece a un read). De hecho POD5 sólo permite la escritura de datos mediante un par de funciones, ninguna de las cuales usa el concepto de batch.

Conceptualmente, los datos de la tabla Read y Signal son datos estrechamente relacionados, casi inseparables. Por esto, el cliente solo tiene acceso a un record batch de tipo Pod5ReadRecordBatch_t que esencialmente combina los datos de la tabla Read y la tabla Signal. En esencia, para un cliente el concepto

de record batch existe simplemente por motivos de eficiencia computacional y de uso de memoria, sólo representa una separación lógica de un stream continuo de datos, de forma análoga a los bloques de disco en un archivo.

Por otro lado, tenemos los datos de la tabla RunInfo. Estos datos quedan expuestos de forma separada a los datos de secuenciación mediante el tipo RunInfoDictData_t. En la API de C esto es evidenciado por la función pod5_get_file_run_info quien no necesita un record batch de señal para recuperrar los datos.

Vale la pena destacar que POD5 permite a un cliente comprimir los datos con Vbz y pasarlos comprimidos a la API. En este escenario, es responsabilidad del cliente asegurarse que los datos estén comprimidos. Por tanto, nos centramos en el caso donde un cliente pasa a POD5 una señal no comprimida.

D.1.2. Impacto del protocolo de acceso

Lo relevante de este protocolo es que cuando la señal ingresa al sistema para su escritura, el protocolo le exige al cliente que ingrese para cada read, los datos de la tabla de Read **junto** con la señal correspondiente. La señal solo puede ingresar al sistema mediante la función pod5_add_reads_data, que requiere la señal junto con los metadatos asociados. Esto hace inmediata la recuperación de metadatos basados en reads para el compresor. Es decir, todos los datos que están en la tabla de Read y se pasan explicitamente (a diferencia del tipo de poro que corresponde a un índice en un diccionario), se pasan siempre en conjunto a la correspondiente señal que queremos comprimir.

Luego, en el flujo de datos, cuando un objeto de clase FileWriter le delega una señal para comprimir a un objeto de clase SignalTableWriter, le adjunta también algunos datos del read correspondiente. Podemos modificar la biblioteca para que se le pasen todos los datos del read. Este razonamiento se extiende mediante la jerarquía de clases que presentamos en la subsección 3.3.5 hasta el módulo de compresión. El FileWriter puede pasarle, sin esfuerzo adicional, al SignalTableWriter los metadatos del read correspondiente para que este pueda pasárselos al algoritmo de compresión.

Por otro lado tenemos los datos que se almacenan como índices en un diccionario, por ejemplo el tipo de poro. Estos datos se ingresan en dos pasos. Ilustramos el proceso con el tipo de poro.

Supongamos que un programa cliente de POD5 quiere escribir reads que tengan el tipo de poro “R10.4.1”. El cliente tiene primero que invocar a una función de la API que “registra” el tipo de poro en el sistema. Esta es la función pod5_add_pore que presentamos en la subsección 3.2.2. Concretamente, el cliente le pasa a POD5 la cadena de caracteres “R10.4.1” y POD5 le devuelve el entero 0. Luego, el cliente le pasa los datos del read a POD5 (mediante pod5_add_reads_data), pero en vez de pasarle “R10.4.1” como tipo de poro, le pasa 0. Internamente POD5 registra en un diccionario que el entero 0 corresponde al tipo de poro “R10.4.1”. En la tabla de Read, POD5 escribe un 0 en el tipo de poro. Esto reduce la duplicación de datos.

Lo más importante aquí es que para que este mecanismo funcione, POD5 obliga al cliente a "declarar" el tipo de poro "R10.4.1" antes de poder escribir datos que referencien ese tipo de poro. Luego, POD5 puede usar el índice que le dió al cliente para recuperar el valor original. Con este protocolo, podemos hacer que POD5 le comunique dichos cambios de estado a PGNano5 (en particular PGNanoPOD5Lib), en las mismas invocaciones que ya existen en POD5, de forma que ahora PGNano5 puede usar el tipo de poro para comprimir los datos, porque el mismo mapeo entre el índice de diccionario 0 y el tipo de poro real "R10.4.1" que construye POD5 ahora también es conocido por PGNano5 y por tanto puede recuperar el valor "R10.4.1".

Entonces, podemos recuperar una gran cantidad de metadatos sobre la señal y naturalmente pasárselos al compresor, siempre que provengan de un read, porque el protocolo de acceso a POD5 exige: Los índices de diccionarios deben ser declarados con anterioridad a su uso y cuando un cliente quiere escribir una señal de secuenciación debe, en la misma llamada a POD5, dar todos los datos del read correspondiente.

Sin embargo, algunos metadatos requieren interpretarlos para potencialmente ser útiles en un algoritmo de compresión. Entonces, quedan excluidos todos aquellos metadatos relacionados con un RunInfo y end_reason.

Interpretar los valores de end_reason llevaría un esfuerzo no trivial y sospechamos que es poco relevante para las estadísticas de la señal.

Tenemos además los datos asociados con un RunInfo. Decidimos no utilizar estos metadatos porque: No tienen una interpretación directa que nos ayude en la compresión. Muchos de los datos allí presentes son irrelevantes para la compresión (cómo por ejemplo el software que escribe el archivo). Esperamos que algunos de los datos potencialmente relevantes los podemos inferir de la señal y la tabla de Read, por ejemplo el rango de digitalización.

En resumen, la escritura ordenada de los datos exigida por POD5 permite la recuperación de metadatos para la señal de secuenciación de forma que no es necesario modificar la API expuesta a un cliente, ni agregar sincronizaciones internas en la biblioteca, ni agregar dependencias que no existían entre clases. Esto se logra explotando los datos que ya existen en el flujo de datos de POD5, particularmente, cuando un objeto de jerarquía superior, delega una responsabilidad a un objeto de jerarquía inferior, le pasamos adicionalmente otros de los datos que el objeto de jerarquía superior ya tiene disponibles en dicha llamada.

Por último, notamos que la solución para sincronizar tipos de poros entre POD5 y PGNano5 es reutilizable para otros tipos de datos que también se representen cómo índices en un diccionario.

Modificaciones a la interfaces para agregar metadatos

Con las observaciones de la sección D.1.2 tenemos un camino viable y prolijo para agregar metadatos extra al algoritmo de compresión. Observamos que, en el flujo de datos de escritura de POD5, desde la llamada a pod5_add_reads_data (primera llamada relevante del cliente) hasta el SignalTableWriter, los métodos pertinentes siempre conocen al menos el read_id del read al que corresponde la

señal, pues deben escribirlo (o deben delegar su escritura) en la tabla Signal. Además, un read conlleva su propio read_id, tanto en el modelo de datos en memoria como en el modelo tabular guardado en disco (tabla Read), luego, dado un read tenemos su ID. Entonces lo que hacemos es, en todo método que esté en la cadena de llamadas al compresor y que reciba el read_id como parámetro, ahora recibe el read entero. Concretamente pasamos ahora un objeto de clase ReadData que constituye la representación en memoria de los datos contenidos en la tabla Read.

Por otro lado, tenemos aquellas clases que no recibían ningún dato del read. Para la escritura, la única clase a modificar es el visitor append_signal, a quien le agregamos en el constructor como parámetro una referencia a un objeto de clase ReadData.

Desde el lado de la lectura necesitamos los mismo metadatos para descomprimir la señal. Optamos por enviar los metadatos que usemos en el algoritmo de compresión como un cabezal antes de la señal comprimida. Esto nos permite evitar cambios que aumentan el acoplamiento interno de las clases, complejizan los algoritmos e incrementan el tiempo de ejecución total. Además, comprimir un cabezal es útil para otros objetivos y brinda una opción más al framework.

Anexo E

Otras codificaciones para la señal

Este capítulo del anexo presenta en la tabla E.1 los resultados completos del experimento de la sección 6.4.

Método	Tasa de compresión (bits/símbolo)	Velocidad (MB/s)
PGNano	6.95871	48.3774
Rice + R / G - RLE(15; R) (10-6)	7.10198	57.1402
Rice + R / G (5-11)	7.13079	81.5154
Rice + R / G - RLE(15; R) (9-7)	7.137	67.0539
Rice + U / G (5-11)	7.15303	339.864
Rice + R / G - RLE(15; R) (8-8)	7.15314	57.7556
Rice + ZSTD / G (5-11)	7.16169	7.33436
Rice + R / G - RLE(15; R) (7-9)	7.16698	71.3654
Vbz	7.19872	834.849
Rice + R / G (4-12)	7.21975	96.7593
Rice + U / G	7.22334	297.075
Rice + R / G (6-10)	7.25368	68.2916
Rice + R / G - RLE(15; R) (6-10)	7.31826	76.3711
Rice + G / R (8-8)	7.32681	187.813
Rice + G / R (9-7)	7.33178	181.179
Rice + G / R (10-6)	7.35285	186.998
Rice + G / R (11-5)	7.35476	191.268
Rice + G / R (12-4)	7.35591	191.953
Rice + G / R (7-9)	7.36802	175.149
Rice + U / G (6-10)	7.38068	381.388
Rice + G / R (6-10)	7.41532	153.676
Rice + G / R (5-11)	7.45349	127.31
Rice + G / R (4-12)	7.55324	107.832
Rice + R / G - RLE(15; R) (5-11)	7.8356	78.5551
Rice + U / G - RLE(15; U) (6-10)	7.94399	328.852
Rice + R / G - RLE(15; R) (4-12)	8.09915	92.1761
Rice + G / U (11-5)	8.37349	234.869
Rice + U / G - RLE(15; U) (5-11)	8.59255	260.147
Rice + U / G - RLE(15; U) (4-12)	8.82556	251.54
Rice + G / U (10-6)	9.36924	234.671
Rice + G / U (9-7)	10.3118	236.0
U / G (6-10)	10.9173	462.934
R / G (5-11)	10.9349	86.0287
U / G (5-11)	10.9446	395.821
ZSTD / G (5-11)	10.9525	6.52554
U / G (4-12)	10.9782	346.585
Rice + G / U (8-8)	11.1644	238.3
Rice + G / U (7-9)	11.8937	240.783
Rice + G / U (6-10)	12.336	249.808
Rice + G / U (5-11)	12.4407	267.765
Rice + G / U (4-12)	12.4841	290.965
Rice + G / U (1-15)	12.5004	364.938
Rice + G / U (3-13)	12.5026	317.331
Rice + G / U (2-14)	12.5126	347.634

Tabla E.1: Resultados de diferentes algoritmos de compresión sobre un conjunto de prueba.