UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE

"IMPLEMENTACIÓN Y ANÁLISIS DE ALGORITMOS PARA INTERSECCIÓN DE CONJUNTOS DE INTERVALOS EVALUANDO SECUENCIAS GENÓMICAS"

ALONSO NICOLAS RODRÍGUEZ ZAMBRANO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Diego Arroyuelo
Profesor Correferente: Juan Pablo Castillo

Noviembre - 2024

DEDICATORIA

Las horas de trabajo que tomaron este informe no hubieran sido posibles por el constante apoyo de mi familia y amigos, en los cuales siempre he tenido una mano para levantarme cuando no queda la energía y un lugar donde buscar la alegría y la calma necesaria para reponerse ante las arduas jornadas de trabajo y estudio.

AGRADECIMIENTOS

Pasaron cientos de cosas en paralelo a la escritura de esta memoria y la pasión inicial se vio abrumada por malas fortunas y la gran presión de la misma. De gran euforia paso a agotamiento a lo largo de los meses, hasta que finalmente logre retomar la voluntad para terminar este documento.

Volver a retomar fue un proceso duro, que rápidamente hizo que me diera cuenta de que el valor de esta memoria no solo es por el desarrollo científico y mi deber educacional, sino en por el deber del trabajo, la dedicación, la pasión y la necesidad de cierre de ciclos.

Mi madre me ha enseñado de todo a lo largo de la vida, de ella herede el amor a la naturaleza, el arte y la curiosidad ante la vida. También me ha entregado el cariño y las reprehendas que he necesitado en los momentos que más bajo y nunca deja de estar para entregar esa chispa de amor y alegría en el momento donde más se anhela.

Mi padre me ha enseñado a encaminarme en esta vida y sin él no podría ni hacer una caja, me enseño a nadar, recorrer cerros y el valor del trabajo. Me ha mostrado el valor del esfuerzo, la dedicación y la pasión a lo que se hace, quizás a veces demasiado, pero siempre lo necesario para darnos lo que necesitamos y lo que deseamos.

Hay muchas personas que me gustaría agregar aquí, que aportaron su granito de arena, pero que serian difíciles de plasmar. Pero a todos ellos, los tengo en mi mente con mucho cariño y me han dado mucha felicidad. Ustedes saben.

También en el transcurso de la carrera, conocí a buenos amigos con los que compartí horas y noches de estudio, no hubiera llegado hasta aquí sin ellos. Sin olvidar a muchos profesores que enseñaban o guiaban de forma espectacular, sus clases despertaban el deseo de aprender.

Finalmente, debo agradecer a mis profesores guías de esta memoria, que me permitieron seguir el camino de este tema de estudio, dándome las pautas y consejos necesarios para el desarrollo de este documento, a pesar de mi agitado compromiso.

Muchas gracias.

RESUMEN

La presente memoria presenta la implementación y análisis comparativos de dos algoritmos de intersección de conjuntos de intervalos, considerando la solución estado del arte desarrollada por Layer y Quinlan ella cual compararemos contra el algoritmo de búsqueda propuesto por Juan Pablo Castillo con una modificación realizada por el autor para que considere la intersección de múltiples conjuntos.

Buscaremos calcular la eficiencia computacional de ambos algoritmos bajo una muestra de conjuntos de intervalos basada en el primer cromosoma de diferentes versiones de secuenciación de 3 especies de mamíferos, el ser humano, el chimpancé y el ratón. Estos datos fueron extraídos cómo archivos BEDs utilizando Table Browser, una potente herramienta del UCSC Genome Browser.

Los análisis nos permitieron concluir cómo el algoritmo de búsqueda es una alternativa secuencial eficiente para la resolución del problema presentado, demostrando su eficacia con respecto al algoritmo de Layer y Quinlan y cómo este resulta de utilidad para el estudio comparativo de secuencias genómicas

Palabras Clave—Intersección de Intervalos, Bioinformática, Computación Científica.

ABSTRACT

The present report consists of the implementation and comparative analysis of two interval set intersection algorithms, considering the state-of-the-art solution developed by Layer and Quinlan, which we will compare against the algorithm proposed by Juan Pablo Castillo with a modification made by the author to consider the intersection of multiple sets.

We aim to calculate the computational efficiency of both algorithms using a sample of interval sets based on the first chromosome of different versions of the sequencing of 3 mammalian species: Human, Chimpanzee, and Mouse. These data were extracted as BED files using Table Browser, a powerful tool of the UCSC Genome Browser.

The analyses allowed us to conclude that the modified algorithm by Juan Pablo Castillo is an efficient sequential alternative for solving the presented problem, demonstrating its effectiveness compared to the Layer and Quinlan algorithm. Moreover, it proved to be useful for the comparative study of genomic sequences.

Keywords—Interval Intersection, Bioinformatics, Scientific Computing.

GLOSARIO

- Algoritmo: Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.
- Genoma: Secuencia total de ADN que posee un organismo, la cual se organiza en múltiples cromosomas.
- Cromosoma: Estructura donde se organiza una secuencia de ADN con forma doble hélice de espiral.
- ADN: Ácido desoxirribonucleico. Ácido nucleico que contiene las instrucciones usadas en el desarrollo y funcionamiento de todos los seres vivos.
- Nucleótido: Molécula pequeña sintetizada por los organismos vivos. Los nucleótidos de base nitrogenada componen el ADN son la Adenina, la Timina, la Citosina y la Guanina.
- Fenotipo: Características o rasgos observables de un organismo.
- BED: Browser Extensible Data.
- $\boldsymbol{S}$: Conjunto de conjunto de intervalos.
- N : Magnitud del conjunto de conjuntos de intervalos $\boldsymbol{S}$, equivalente a $|\boldsymbol{S}|$.
- S_i : Conjunto de intervalos i - ésimo.
- $s_{i,j}$: Intervalo j perteneciente al conjunto i .
- NW : Solución de un problema de intersección de múltiples conjuntos.
- I : Total de intersecciones encontradas por los algoritmos.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ALGORITMOS

INTRODUCCIÓN

En el campo de estudio interdisciplinario del genoma, conocido como la *genómica*, no es extraña la necesidad de comparar dos o más secuencias de nucleótidos para identificar similitudes o diferencias que nos permitan generar un entendimiento de la estructura o evolución del genoma de un organismo. Encontrando igualdades entre distintas especies podemos generar un camino evolutivo sobre el cual estudiar los cambios genéticos entre especies y cómo estos generan aparición de características fenotípicas dentro de un organismo.

Este tipo de comparaciones es de gran complejidad considerando que cada genoma fácilmente se compone de millones o hasta miles de millones de pares de nucleótidos, creando la necesidad del uso de alguna simplificación para representar la información contenida dentro del genoma, que nos entregue una visión más acotada y concisa sobre alguna propiedad interes de estudio.

Uno de estos métodos es la identificación y localización de genes, partes del genoma que producen la aparicion de alguna característica en un ser vivo, estas tienen punto de inicio y término bien definidos dentro de la secuencia y podemos identificarlos precisamente. Los genes identificados pueden almacenarse en un formato estándar denominado Browser Extensible Data (BED), los cuales almacenan el cromosoma y los intervalos de inicio y términos que ubican genes dentro de un genoma. En la figura 2 siguiente se encuentra un ejemplo, donde un gen es identificado en el cromosoma 1 (chr1) y adyacente tiene su intervalo de inicio y término dentro de ese cromosoma.

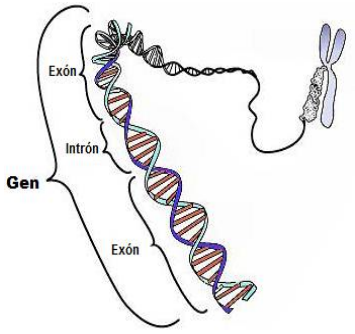

Figura 1: Descomposición de un cromosoma a un gen. Fuente: Wikipedia

1	chr1	11868	14409	ENST00000456328.2	0
2	chr1	12009	13670	ENST00000450305.2	0
3	chr1	14403	29570	ENST00000488147.1	0
4	chr1	17368	17436	ENST00000619216.1	0
5	chr1	29553	31097	ENST00000473358.1	0
6	chr1	30266	31109	ENST00000469289.1	0
7	chr1	30365	30503	ENST00000607096.1	0
8	chr1	34553	36081	ENST00000417324.1	0
9	chr1	35244	36073	ENST00000461467.1	0
10	chr1	52472	53312	ENST00000606857.1	0

Figura 2: Primeros 10 líneas de un archivo BED. Fuente: Origen Propio

Con esta representación, orientada a evidenciar la localización de cada gen, podemos buscar un enfoque de comparación y centrarnos en identificar si es que existe alguna intersección de genes entre dos o más genomas. Esta comparación se puede entender cómo un problema de intersección de conjuntos de intervalos y ha recibido bastante atención la solución particular para dos conjuntos[1], pero la problemática se vuelve mucho más compleja para cuando se busca encontrar una intersección que se encuentre simultáneamente entre más de 2 conjuntos de intervalos.

Las técnicas para dos conjuntos, se pueden extender para resolver el problema de intersección de múltiples conjuntos, iterando de forma consecutivas los resultados entre pares de conjuntos, pero es fácil notar cómo esta solución es ineficiente y considerara reiteradas veces intervalos donde es imposible una intersección en todos los conjuntos. Por esto, es necesario el uso de nuevas técnicas cómo las desarrolladas por A. Rayan y R. Quinlan[2] o Juan Pablo Casto[3] los cuales modificaron algoritmos previos para intersección de segmentos e intervalos respectivamente para ser utilizados para el problema de múltiples conjuntos.

Nos interesa el estudio de estos algoritmos dado que es común que un genoma contenga miles de genes los cuales corresponderían a miles de intervalos dentro de un archivo BED, los cuales, dado a su envergadura de datos no son sencillos de comparar y es necesario un algoritmo eficiente para realizar estas comparaciones. Por ejemplo, considerando un caso de 3 conjuntos de intervalos cada uno con 1000 intervalos, si quisiéramos encontrar a fuerza bruta las intersecciones, en el peor de los casos, tendríamos miles de millones de comparaciones que realizar, esto nos hace ver fácilmente la necesidad de alternativas.

Por lo anterior, la presente memoria realizara un estudio de la implementación de ambos algoritmos mencionados anteriormente, comparando de forma cuantitativa la eficacia computacional de estos, en búsqueda de un análisis que nos presente las propiedades de ambos algoritmos para el problema de comparación de secuencias genómicas, cómo a su vez, su comportamiento ante conjuntos de datos sintéticos preparados bajo una cantidad definida de intersecciones.

Partiremos introduciendo ciertos conceptos clave para comprender con mayor profundidad la motivación del estudio genómico y los conceptos matemáticos relevantes, que incluyen los intervalos e intersecciones. A continuación, presentaremos una recapitulación del estado del arte de los algoritmos y herramientas pertinentes en el desarrollo de nuestra memoria. Continuaremos con el proceso de desarrollo de nuestro estudio y realizaremos una comparativa y análisis de los resultados. Estos análisis nos guiarán en la formulación de nuestras conclusiones y reflexiones, las cuales se presentarán cómo posibles comentarios para finalizar el presente informe.

CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA

Detallaremos brevemente conceptos respecto a genómica que motivan este estudio, su modelado en la bioinformática y los fundamentos matemáticos en los que los iremos asociando para el desarrollo de esta memoria.

1.1. GENOMA, CROMOSOMAS, GENES Y BED

Un genoma es la secuencia completa de ADN que posee un organismo en particular, el cual se organiza en cromosomas dentro del núcleo celular, empaquetado cómo una cadena de pares de bases nucleotídicas con forma de doble hélice en espiral, donde una base nucleótida, y en particular una nitrogenada, consiste en grandes polímeros[1] que son codificados cómo A, T, C y G[2]. Esto nos permite hacer una analogía del ADN cómo un arreglo de caracteres, donde sus posibles valores son las 4 bases nucleotídicas[3]. Esta secuencia es el pilar fundamental de la herencia, permitiendo la transmisión de información genética entre seres vivos y la producción de moléculas utilizadas en el desarrollo y mantenimiento de un organismo celular.

Dentro de cada cromosoma, podemos identificar genes, zonas del código genético que permiten la manifestación de una característica en el ser vivo, encapsulando la información necesaria para producir alguna macromolécula con alguna función celular específica. Estas son áreas definidas dentro del genoma y reconociéndolo cómo una cadena de caracteres, es de utilidad anotar los puntos de inicio y término de cada gen, para acceder a ellos rápidamente dentro de la cadena, esta ubicación particular dentro de un cromosoma es denominada *locus*[4].

Estos fundamentos nos permiten hacer la analogía de un gen, cómo un intervalo dentro de un conjunto, esto es algo que ya se ha transparentado en el ámbito de la bioinformática y uno de los varios formatos estandarizados para almacenar estos intervalos genéticos son los archivos de extensión BED ó Browser Extensible Data, los cuales mínimamente requieren de 3 columnas para identificar una característica o gen dentro de un genoma, siendo estos, el cromosoma donde se encuentra y el locus que lo

[1] Polímero: Sustancia compuesta por moléculas de gran tamaño.

[2] Adenina, Timina, Citosina y Guanina respectivamente.

[3] Aun que la secuencia de bases son pares de nucleótidos, cada uno de estos solo tiene un nucleótido opuesto compatible, así que podemos reconocer el ADN completo solo secuenciando una de las cadenas de nucleótidos.

[4] En plural *loci*.

ubica dentro del cromosoma, también puede contener diversas columnas opcionales, cómo un nombre, algún identificador único o datos relevantes.

Tabla 1: Ejemplo de datos en BED
Fuente: hg38 - GENOME

chrom	*chromStart*	*chromEnd*	*name*
chr1	11868	14409	ENST00000456328.2
chr1	14403	29570	ENST00000488147.1
chr1	34553	36081	ENST00000417324.1
...	...	...	...

Utilizando estos datos en formato BED cómo intervalos, compararemos diversos genomas resolviendo un problema de intersección de múltiples conjuntos de intervalos, identificando los genes que comparten ubicaciones en los genomas que se están comparando, así, encontrando zonas de interés para el estudio genómico de las cuales se puede inferir la existencia de una función o origen biológico compartido.

Este problema no es trivial, en particular, para el genoma humano[4] se identificaron más de 3 billones de pares nucleótidos, reportando más de 60.000 genes con cerca de 230.000 transcripciones[1], los cuales si buscáramos comparar con técnicas poco eficientes, tendríamos un tiempo de cómputo el cual no seria competitivo para el flujo de datos que se esta creando actualmente.

En la última década, el coste de secuenciación completa de un genoma ha disminuido considerablemente[2], pasando de $10.000.000 de dólares para mediados del 2007, a tan solo menos de $1000 dólares a finales del 2021. Esto ha producido un aumento de interés considerable en el estudio del genoma y sus aplicaciones para en medicina, agricultura, industria, etc. Haciendo necesario la creación de técnicas informáticas que permitan trabajar con estos extensos flujos de datos.

Bajo estos conceptos genómicos cimentaremos la motivación para dar resolución a un problema de intersección de conjuntos de intervalos, el cual podemos aplicar para realizar comparación entre múltiples secuencias genómicas y obtener información de valor respecto al conjunto de datos que buscamos comparar y zonas de valor de estudio dado a su concurrencia en múltiples conjuntos.

[1] Un gen esta compuesto de transcripciones, secciones que permiten codificar RNA y que son únicas para cada gen.
[2] *The cost of Sequencing Human Genome* https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost

1.2. CONJUNTOS DE INTERVALOS

Hemos mencionado la motivación bioinformática del problema, ahora definiremos los conceptos matemáticos clave que utilizaremos. Primero necesitamos saber que es un intervalo, el cual, se define cómo el continuo de valores entre un punto de inicio y término. Por ejemplo, dado un intervalo a con inicio a_{start} y término a_{end}, este comprenderá todos los valores entre ambos puntos. En particular nos mantendremos en los números naturales, dado que no trabajaremos con ningún intervalo decimal. Con esta definición, podemos hacer la analogía de un gen identificado con *locus* definido, cómo un intervalo dentro de un conjunto.

Cuando tomamos n de estos intervalos y los agrupamos, tenemos un conjunto de intervalos. Por ejemplo, sean a_i múltiples intervalos, podemos formar el conjunto $A = \{a_1, a_2, \ldots, a_n\}$ el cual comprende todos los números naturales dentro de los intervalos a_i.

Es importante notar que un conjunto de intervalos, podría contener intervalos que se solapan entre sí, pero no nos enfocaremos en estos casos y supondremos que nuestros conjuntos no los posee, y por lo tanto, estos conjuntos los llamaremos disjuntos. Es posible que existan transcripciones de genes que se solapen entre sí, pero, para estos casos *aplanaremos* estos intervalos en uno solo que los englobe.

Como mencionamos anteriormente, no nos interesa la composición de cada gen, sino más bien su ubicación dentro de la cadena y bajo estos supuestos, formalmente diremos que trabajaremos solo con conjuntos de intervalos disjuntos, esto es, que ningún intervalo del conjunto contiene algún elemento dentro de otro intervalo. Finalmente, asumiremos que los intervalos dentro de estos conjuntos están ordenados de forma ascendente según su punto de partida, dado que en caso de no estarlos, es una operación no costosa de realizar.

1.3. INTERSECCIÓN DE INTERVALOS

Dado dos intervalos a y b, diremos que a interseca con b si se produce que $a_{start} \leq b_{end}$ y $a_{end} \geq b_{start}$. Esto quiere decir que, si el inicio de a es anterior al final de b y el final de a es posterior al inicio de b, estos intervalos se intersecan. Con esta definición, podemos atender 4 posibles casos de intersección, siendo estos cuando a interseca por la izquierda, a interseca por la derecha, a contiene b y cuando a es contenida por b. Estos quedan ilustrados en la figura 1:

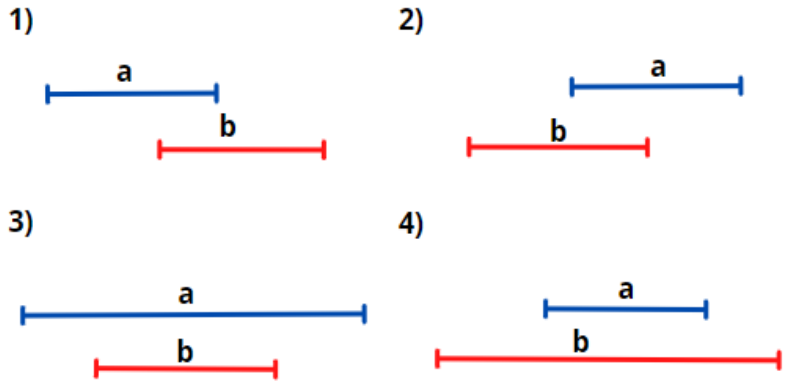

Figura 3: Posibles intersecciones de intervalos.
1) a intersecando por la izquierda a b, 2) a interseca por la derecha a b, 3) a contiene a b y 4) a es contenida por b.
Fuente: Origen propio

Definiremos la operación intersección $I(A,B)$, la cual tomará 2 conjuntos de intervalos A y B y retornara un nuevo conjunto con los intervalos de las intersecciones. Definida matemáticamente cómo:

$$I(A,B)=\{(a,b) \vee a \in A, b \in B, (a_s \leq b_e \wedge a_e \geq b_s)\}$$

Esta operación la podemos utilizar para comparar dos conjuntos de intervalos y obtener los intervalos que intersecan. Podemos notar que comparar 2 conjuntos de intervalos a lo menos podría tomar $O(n^2)$ comparaciones con n el total de intervalos de cada conjunto, siendo el peor caso cuando comparamos cada intervalo del primer conjunto, con cada otro intervalo del otro conjunto.

Pero buscamos encontrar la relación entre múltiples conjuntos, por lo tanto, debemos definir una nueva operación que nos permita extender esto a una operación $I(\boldsymbol{S})$ que recibirá de entrada un conjunto de conjuntos de intervalos $\boldsymbol{S}$ y nos entregue todas sus intersecciones múltiples.

1.4. INTERSECCIÓN DE CONJUNTOS DE INTERVALOS

Utilizaremos la definición desarrollada por R. Layer y A. Quinlan[2] en donde, dado el conjunto $\boldsymbol{S} = \{S_1, S_2, \ldots, S_N\}$ donde cada elemento S_i es un conjunto disjuntos de intervalos. Ademas, diremos que dado dos intervalos a y b, si estos se intersecan lo anotaremos cómo $a = b$. Esto último es solo por simplicidad y no posee las mismas propiedades matemáticas que la igualdad. Utilizaremos esta notación para definir un nuevo operador intersección $I(\boldsymbol{S})$ cómo se ve en la Figura 2.

$$\mathscr{I}(\mathbf{S}) = \left\{ s_{1,j}, \ldots, s_{N,l} \,\middle|\, \begin{array}{l} s_{a,b} \in S_a, \\ s_{a,b} = s_{c,d} \forall s_{a,b}, s_{c,d} \in \{s_{1,j}, \ldots, s_{N,l}\} \end{array} \right\}$$

Figura 4: Intersección de conjuntos de intervalos.
Fuente: [2]

El cual toma toma el conjunto $\boldsymbol{S}$ de largo N y retorna N - tuplas[1], las cuales contienen los N índices de los intervalos pertenecientes a cada $S_i \in \boldsymbol{S}$, tal que, cada uno de ellos interseca entre sí. Utilizaremos la Figura 3 para ejemplificar un resultado.

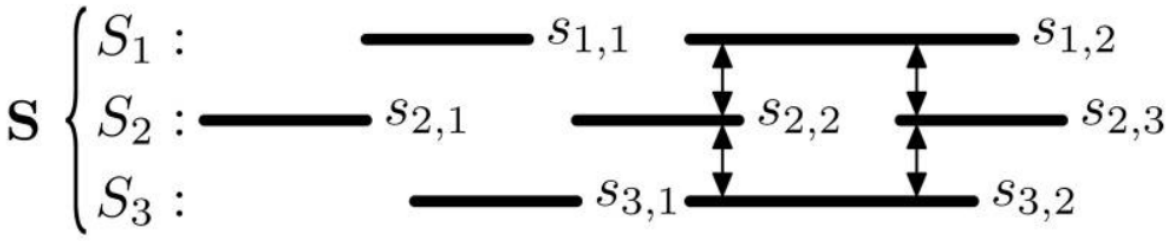

Figura 5: Conjuntos de Intervalos y sus intersecciones.
Fuente: [2]

Para este caso, obtenemos que $I(\boldsymbol{S}) = \{ (s_{1,2}, s_{2,2}, s_{3,2}) , (s_{1,2}, s_{2,3}, s_{3,2}) \}$, donde podemos notar visualmente cómo estos intervalos intersecan. También es interesante ver que $s_{1,1}$ interseca tanto con $s_{2,1}$ y $s_{3,1}$, pero cómo estos últimos no intersecan entre sí, no son una solución valida.

Si suponemos que cada uno de los N conjuntos, tiene n intervalos, podemos notar que tenemos una cota superior asintótica dada por $O(n*n*n*\ldots)=O(n^N)$, dado que esto es lo mismo que ir realizando intersecciones a pares entre los conjuntos hasta obtener las intersecciones finales. Vemos que esto escala rápidamente con el numero de conjuntos y una solución exponencial es totalmente inutilizable.

[1]Tupla con N elementos.

1.5. OBJETIVOS

1.5.1. OBJETIVO GENERAL

- Implementar algoritmos para intersección de conjuntos de intervalos evaluando intervalos genéticos para comparar sus eficiencia de cómputo.

1.5.2. OBJETIVOS ESPECÍFICOS

- Implementar el algoritmo de Layer y Quinlan para intersección de conjuntos de intervalos.
- Implementar y adaptar el algoritmo de búsqueda de intersección de pares de conjuntos disjuntos de intervalos de Juan Pablo Castillo para múltiples conjuntos.
- Evaluar experimentalmente dichos algoritmos de intersección de intervalos y comparar su eficiencia de cómputo.

CAPÍTULO 2: MARCO CONCEPTUAL

El problema de intersección de intervalos se utiliza en diversos campos de estudio y es de particularidad utilidad para la comparación de secuencias genómicas, por lo tanto, es necesario reconocer las investigaciones y el trabajo previo realizado en torno a esta problemática para basarnos en fundamentos sólidos para cumplir con los objetivos planteados en esta memoria. Dado esto, el presente capítulo se encargará de presentar los algoritmos bajo los que nacieron las principales fuentes para este algoritmo y los cuales son las bases de los algoritmos que implementaremos y presentaremos más adelante.

En esta sección, presentaremos dos documentos de los cuales se desarrollaron las soluciones algorítmicas que se presentarán en el siguiente capítulo, uno enfocado en problemas de intersección de segmentos en el plano y el otro en intersección de múltiples conjuntos de enteros. Finalmente, presentaremos las dos herramientas clave para la extracción y manejo de datos.

2.1. ALGORITMO *SWEEP* EN INTERSECCIONES GEOMÉTRICAS

Desarrollado en 1978, este algoritmo es la base de múltiples algoritmos de intersecciones geométricas, descrito por Bentley y Ottoman en "*Algorithms for reporting and counting geometric intersections[5]",* donde nos presentan diversas soluciones al problema geométrico de encontrar intersecciones entre segmentos de línea o planos en el espacio.

Nos centraremos en el algoritmo de barrido o *sweep* para segmentos en el plano que nos presenta. Este algoritmo es una idea más generalizada de intersecciones, donde un segmento tiene tanto una componente horizontal cómo una vertical en el plano, por lo tanto, la intersección suele ser un punto en el plano. Esta solución se basa en la idea de ordenar los *endpoints*[1] de cada segmento según su posición x en el plano y hacer un *sweep*[2] de izquierda a derecha trazando líneas verticales en estos puntos.

Buscamos los segmentos que intersecan con estas líneas verticales y así establecer una razón de orden entre los segmentos, permitiéndonos reconocer que existen segmentos sobre o bajo otros. Con esto podemos notar, que si un *endpoint* de un segmento A está sobre otro segmento B y el otro *endpoint* de A está bajo B, significa que estos se

[1]Los puntos en el plano que definen el segmento.
[2]Barrido

debieron haber cruzado y por lo tanto, se intersecan en alguna parte dentro del área de estas líneas verticales y así estas se utilizan cómo base para desarrollar encontrar una solución.

Bentley y Ottoman demuestran que este algoritmo tiene una complejidad de $O(N \log N + k \log N)$, con N el total de segmentos y k el total de intersecciones. Es importante notar que esta solución es peor que $O(N^2)$ para el caso que k se aproxime a N^2, pero es una solución bastante eficiente y útil para comparar la eficiencia de las otras soluciones que se presentarán en en esta memoria.

En el mismo documento nos presentan otros algoritmos para otros tipos de intersecciones, en particular nos interesa su el algoritmo para contar las intersecciones entre segmentos horizontales y verticales, el cual es similar a la intersección de conjuntos de intervalos, si consideramos el caso donde no existan segmentos verticales. Demostraron que esta solución tiene complejidad $O(N \log N + k)$ donde N es el total de segmentos y k es el total de pares que intersecan. Es importante este valor, dado que Shamos y Hoey[6] demostraron que este es una cota inferior para el problema de intersección de segmentos.

Los algoritmos presentados en este documento son clave para el desarrollo de soluciones eficientes para el problema de algoritmos para intersección de intervalos y es la idea base de donde nace el algoritmo de intersección de múltiples conjuntos desarrollado por Layer y Quinlan en "*A parallel algorithm for N-way interval set intersection"*[2].

2.2. ADAPTATIVE INTERSECTION

En el paper "*Adaptative Intersection and t-Threshold Problems"[7]* los autores Claire Kenyon y Jérémy Barbay nos presenta una solución eficiente al problema de intersección de múltiples conjuntos enteros, en donde tenemos conjuntos ordenados sobre los cuales utilizamos búsquedas exponenciales y binarias para encontrar sus intersecciones entre conjuntos. Esta solución ha demostrado tener una cota superior de $O(N\delta\log(n/\delta))$, donde N son el total de conjuntos, n el largo de los conjuntos de intervalos y δ un parámetro denominado dificultad de entrada, el cual depende del total de intersecciones encontradas y el largo de la mayor serie de desigualdades que representan los índices de intersección.

El cálculo la dificultad de entrada se define cómo $\delta = I+min(P)$ donde I es el total de intersecciones y P el largo de la serie mínima de desigualdades estrictas requeridas para contener los elementos de la intersección. Por ejemplo, suponiendo dos conjuntos S_1 y S_2 sin intersecciones, I es 0 y P es 1, por que solo se necesita la desigualdad $s_{1,1} > s_{2,1}$ para representar las intersecciones, implicando que ningún elemento con índice mayor a 1 del conjunto 2, es intersección al primer elemento $s_{1,1}$ del conjunto 1. El calculo de la desigualdad estricta es un trabajo no menor y no sera incluida en esta memoria, en particular, nos interesa la función $\delta(I)$, la dificultad con respecto a las intersecciones.

También nos presenta una solución al *t-Threshold Problem*, el cual es una generalización del problema de intersección de conjuntos, con la diferencia de que un intervalo no necesita intersecar con los N conjuntos de $\boldsymbol{S}$, sino que basta con obtener al menos t intersecciones. Podemos notar que en el caso de que $t = N$ tenemos nuestro problema inicial de N intersecciones.

Estos algoritmos nos interesan por que los podemos adaptar para intersecar intervalos, para esto, necesitamos considerar la diferencia entre los elementos enteros de los conjuntos y los intervalos que manejaremos, por lo tanto, será necesario establecer un nuevo criterio de búsqueda basado en intersecciones de intervalos. Estas modificaciones fueron desarrolladas por Juan Pablo Castillo en su memoria[3], pero más enfocado a la intersección de pares de conjuntos, por lo tanto, se utilizará el algoritmo original y el de Juan Pablo cómo referencias para implementar una versión que sea compatible con el caso de múltiples conjuntos.

2.3. UCSC TABLE BROWSER

Podremos obtener un conjunto de datos valioso y cuantitativo a través de la herramienta libre *Table Browser*[1] [8], que funciona utilizando la vasta base de datos del *UCSC Genome Browser*. Esta plataforma ofrece múltiples herramientas para extraer datos en formato texto de secuencias genómicas de diversas especies secuenciadas, permitiendo acceder a diferentes versiones de secuenciación, aplicar filtros de elementos, y mucho más.

Table Browser permite realizar consultas personalizadas según las necesidades específicas de los datos, brindando opciones para definir la estructura, el formato de exportación y los cromosomas que queremos extraer. Ofrece una gran flexibilidad al seleccionar qué secuencias obtener, permitiendo ajustar parámetros como la longitud de las secuencias, aplicar filtros por identificadores específicos, e incluso realizar intersecciones entre genomas.

Esta herramienta cuenta con varias capacidades útiles para la manipulación de datos, como el filtrado de regiones específicas, la búsqueda de intersecciones y correlaciones entre conjuntos, y la personalización del formato de archivo en el que deseamos descargar las secuencias. Además, permite acceder a múltiples versiones de secuenciación de diversos organismos, incluyendo virus.

Nos resulta especialmente valiosa su capacidad para exportar secuencias en formato BED, lo cual será crucial para generar nuestras muestras de conjuntos, que utilizaremos en la evaluación de los algoritmos de intersección de intervalos.

[1]Enlace de herramienta: https://genome.ucsc.edu/cgi-bin/hgTables

2.4. BEDTOOLS

BEDTools[1] es una *suite* de software[1] que posee las herramientas necesarias para comparar, manipular y anotar características[2] genómicas en formato BED. Posee funcionalidades para convertir formatos BED a otros, realizar operaciones de intersección a pares, unión de características que se sobreponen, cambiar el orden de las características, entre otros.

Una de las funcionalidades esenciales para el desarrollo de esta memoria, es la capacidad de intersecar pares de BED y encontrar las características que se solapan, ofreciendo diversos parámetros de intersección, e incluso la capacidad de comparar un archivo base, con diversos otros e identificar sus intersecciones a pares de características.

Dado que un archivo BED puede contener diversas características de un genoma, y no necesariamente solo genes, utilizaremos BEDTools para manejar estos intervalos y *aplanar* los intervalos que se superponen dentro del mismo cromosoma. Se nos provee de la función *merge*[3], la cual nos da diversas alternativas para unir intervalos, cómo por ejemplo por proximidad o directamente por intersección, pero en particular, no nos interesa agrupar todas las características que se sobreponen en una sola.

[1] Conjunto de archivos y aplicaciones
[2] Decimos característica dado que un BED puede contener elementos del gen, por ejemplo, un gen contiene exones e intrones.
[3] Documentación en línea: [enlace]

CAPÍTULO 3: PROPUESTA DE SOLUCIÓN

La propuesta de esta memoria consiste en evaluar comparativamente el tiempo de cómputo de una modificación del algoritmo búsqueda de Juan Pablo Castillo[3] de intersección de intervalos para el caso no estudiado de N conjuntos de intervalos, contrastado con el algoritmo de *sweep* estado del arte descrito por Layer y Quinlan[2]. Implementaremos ambos algoritmos mencionados en el lenguaje compilado C++, siendo este idóneo para este tipo de problemas donde se requiere de gran capacidad de cómputo.

Ambos algoritmos tras ser implementados, serán evaluados comparativamente utilizando conjuntos de datos genómicos, en formato BED. Estos serán manejados para ajustarse a un problema de intersección de conjuntos de intervalos, utilizando BEDTools para volverlos disjuntos y finalmente, ser procesados por nuestros algoritmos. Los resultados de estos experimentos serán presentados en el próximo capítulo.

El presente capítulo se enfocara en explicar el funcionamiento de ambos algoritmos y exponer un estimado de su cota asintótica o notación O grande, que nos apoye a evaluar los resultados que obtendremos en la experimentación.

3.1. ALGORITMO *SWEEP* PARA CONJUNTOS DE INTERVALOS

Describiremos en Algoritmo 3.1 la solución propuesta por Layer y Quinlan para el problema de intersección de múltiples intervalos, el cual es una extensión del algoritmo *sweep* presentado previamente, pero que fue modificado para ser compatible con el problema de N conjuntos de intervalos.

Considerando la definición previa de conjuntos de intervalos, tenemos cómo *input* de entrada un conjunto de conjuntos de intervalos $\boldsymbol{S} = \{S_1, S_2, \ldots, S_N\}$ con largo $\boldsymbol{N}$. Estos conjuntos podrían tener todos distinta cantidad de intervalos, pero por simplicidad diremos que cada conjunto tiene n intervalos, además, estos son disjuntos y están ordenados de forma ascendente según su punto de partida.

Este algoritmo destaca por el uso dos colas de prioridad $\boldsymbol{pqQ}$ y $\boldsymbol{pqC}$, un tipo de cola o *queue* modificada, en donde el primer elemento a retirar o desencolado (operación *pop*) será siempre el que tenga mayor prioridad, por lo tanto, cada elemento a insertar (operación *push*) deberá ser ubicado en donde corresponda según su prioridad. También

para el caso en donde las prioridades sean la misma, retiramos los elementos por su orden de inserción. El uso de estas colas de prioridad evita la necesidad de recorrer continuamente el conjunto de intervalos, y nos sirven

Podemos representar una cola de prioridad cómo una lista ordenada que se llena con tuplas $(p,\ v)$, donde p es la prioridad que nos permite definir el orden de desencole de los elementos priorizado con valor v. Para ambas colas de prioridad que utilizaremos, diremos que el elemento priorizado o con mayor prioridad es aquel con menor valor de p.

Primer definiremos la cola de prioridad ***pqQ***, la cual se inicializa con elementos de prioridad $(p,\ v) = (x_{start},\ i)$, donde x corresponderá al primer intervalo de cada conjunto S_i, los cuales tendrán los menores punto de partida. Esta cola será la encargada de guardar los indices de los siguientes intervalos de cada conjunto con menor punto de partida que aun no hemos revisado.

La iteración principal del algoritmo continuará mientras existan elementos para retirar de la cola de prioridad ***pqQ***, desencolando un elemento q_0 e intentando encolar un nuevo elemento q_n retirado del mismo conjunto $S_{q_0.v}$ de donde se retiró q_0 y solo si es que este no se encuentra vació.

Antes de continuar, debemos introducir el concepto de intervalos en contexto, por esto, diremos que a está en contexto con algún intervalo b si se cumple que $a_{start} > b_{end}$, o sea, si a inicia antes que b termine, estos intervalos están en contexto. Como todos los conjuntos son disjuntos y están ordenados, se puede definir el conjunto O cómo las N duplas o_i de índices de intervalos dentro de lo conjuntos S_i, que están en contexto con mi punto de inicio $q_0.p$, donde cada elemento de o_i corresponderá a un par de índices $(o_{i.start}, o_{i.end})$ que representan el rango índice de intervalos en contexto.

Por ejemplo, sea $o_4 = (3,\ 5)$, significa que los intervalos $s_{4.3}$, $s_{4.4}$ y $s_{4.5}$ pertenecientes a S_4, se encuentran en contexto con q_0. Estas tuplas se inicializan con valor de $(0,-1)$ y agregamos intervalos al contexto aumentando $o_{i.end}$ en uno y los quitamos aumentando $o_{i.start}$ en uno. Si tenemos que $o_{i.end} > o_{i.start}$ significa que no hay intervalos en contexto.

Cuando desencolamos q_0 y queremos buscar un nuevo elemento en contexto, debemos asegurarnos de retirar todos los intervalos que no se encuentren en contexto de nuestro conjunto O, para esto, utilizamos otra cola de prioridad llamada ***pqC***, la cual será similar a ***pqQ***, con la diferencia de que guardará finales de intervalos. De esta forma, desencolaremos elementos de prioridad c pertenecientes de ***pqC***, quitando elementos

en contexto del o_i correspondiente y aumentando $nEmpty$ si se produce que un o_i se quede sin contextos.

Utilizamos $nEmpty$ cómo una variable auxiliar, que nos permite saber cuántos elementos de O se encuentran sin contexto, de esta forma, inicializando $nEmpty$ cómo N, podemos saber que encontramos una intersección múltiple cuando $nEmpty$ llega a 0.

Continuando con el algoritmo, luego de retirar los elementos en contexto y ajustar $nEmpty$, revisaremos si es que el contexto correspondiente al q_0 desencolado, se encuentra con elementos en contexto y así disminuir $nEmpty$. Si $nEmpty$ llega a 0, hemos encontrado una intersección múltiple y debemos calcular una la solución cómo una N tupla que almacenamos en NW.

Para calcular los índices de la intersección múltiple, es necesario del operador $sequence(o_i)$, el cual tomará un contexto o_i y retornara el completo de valores que se encuentran dentro de ese rango de intervalos. Por ejemplo, $sequence((1,4))$, retornara el rango de valores $(1,\ 2,\ 3,\ 4)$, luego utilizaremos los resultados s_i de aplicar el operador para calcular un producto cartesiano que generará el conjunto nw formado por tuplas de tamaño N, donde cada elemento de esta es el índice de un intervalo perteneciente al conjunto S_i, donde i es el índice del valor dentro de esa tupla. Por ejemplo, considerando el siguiente producto:

$$(1,\ 2,\ 3) \times (1,\ 2) = (1,\ 1),\ (1,\ 2),\ (2,\ 1),\ (2,\ 2),\ (3,\ 1),\ (3,\ 2)$$

Podemos tomar cualquier tupla, cómo podría ser la $(2,\ 1)$ e inferir que hay una intersección entre los intervalos $s_{1,2}$ y $s_{2,1}$. El resultado nw del producto deberá ser anexado al conjunto de soluciones finales NW.

Con este proceso, ya hemos realizado casi toda la iteración y solo nos falta agregar un nuevo elemento al contexto para continuar, para esto agregaremos un elemento c_n a la cola $\boldsymbol{pqC}$, con prioridad p igual al fin del intervalo con índice $o_{q_0.v}.end$ perteneciente al conjunto $S_{q_0.v}$ y v equivalente a este conjunto $q_0.v$. Finalmente c_n quedaría cómo $c_n = (p, v) = (S_{(q_0.v),(o_{q_0.v}.end)}.end,\ q_0.v)$, el cual correspondería al nuevo elemento que se esta agregando al contexto.

Así, completamos una iteración y continuaremos hasta que no nos queden elementos en la cola de prioridad $\boldsymbol{pqQ}$, cuando esta se vacié, retornaremos todos los índices de las intersecciones encontradas en el conjunto NW.

3.1.1. NOTACIÓN O GRANDE

Considerando los valores N y M cómo el número de conjuntos y el total de intervalos de S respectivamente, la cola de prioridad ***pqQ*** toma $O(logN)$ [1] comparaciones en sus inserciones y remociones, dándonos una cota de $O(M\,logN)$, además, cada intervalo es agregado y removido de la cola de prioridad ***pqC*** exactamente una sola vez, entregándonos otra cota para recorrer los M intervalos cómo $O(M\,logM)$.

Sabemos que por cada intersección, debemos haber iterado al menos por sobre los N elementos que la componen, lo que agrega un factor WN donde W es el total de intersecciones, lo que nos permite concluir con una cota asintótica de $O(M\,logN + M\,logM + WN)$, donde, asumiendo que $M \gg N$, podemos finalmente llegar a $O(M\,logM + WN)$.

La cota calculada anteriormente no considera el costo del producto cartesiano, el cual, para un caso de producto entre $A \times A$, con A un conjunto con $|A|$ elementos, tenemos una cota optima $O(|A|^2)$ dada por el total de combinaciones, por lo tanto, retomando el algoritmo, en cada intersección deberemos realizar N productos cartesianos, en donde, con seq_i los resultados de cada $sequence(o_i)$, podemos concluir que este producto viene acotado por $O(max(|seq_i|)^N)$, la cual crece exponencialmente con el total de conjuntos. Considerando que este producto se realiza a lo más, una vez por cada intersección (dado que es posible que el producto cartesiano encuentre múltiples) y considerando $X = max(|seq_i|)$, tenemos una cota final para todo el algoritmo de:

$$O(M\,logM + WN + WX^N)$$

Este último análisis, nos permite ver que el algoritmo podría ser extremadamente ineficiente para cuando X y N son muy grandes, situación que debería ser anómala para la gran mayoría de los conjuntos. Un valor de X alto se podría producir si es que existen intervalos grandes que contengan intersección múltiple con muchos otros intervalos, provocando que los contextos tengan que generen secuencias de gran tamaño. En contraparte, es razonable pensar que a mayor N, más difícil es que un solo intervalo interseque con muchos otros, dificultando que el valor de X sea grande, haciendo que estos dos valores se contrapongan entre sí, pero aún así, es un factor a considerar que podría afectar de forma clave el rendimiento en ciertos conjuntos.

[1] Esto es debido a que esta cola a lo más tiene un elemento por conjunto.

Algoritmo 3.1: Algoritmo sweep por Layer y Quinlan

INPUT: Un conjunto $\boldsymbol{S} = \{S_1, S_2, \ldots, S_N\}$ de N conjuntos de intervalos ordenados.
OUTPUT: Un conjunto formado por N - tuplas con los índices de los intervalos que intersecan para cada conjunto.

FUNCTION *Sweep* **START**:

NW → Conjunto vació de N -tuplas con índices de intersecciones.
O → Conjunto de N intervalos con los índices en contexto.
pqQ → Cola de prioridad de inicios de intervalos por barrer.
pqC → Cola de prioridad de términos de intervalos en contexto.

FOR $i=1$ **TO** N **DO:** *// Inicializamos* O *y* pqQ

$o_{i.start} = 0$
$o_{i.end} = -1$
$x = pop(S_i)$ *// Sacamos el menor intervalo de* S_i
$q_n = (x_{start}, i)$ *// Nuevo elemento* q_n *con prioridad* p *igual y* v *igual a* i.
$pqQ.push(q_n)$

$nEmpty = N$ *// Total de ordenamiento vacíos*

END FOR

WHILE $\neg\, empty(Q)$ **DO:**

$q_0 = pop(Q)$

IF $\neg\, empty(S_{q_0.v})$ **THEN**: // Si el conjunto al que pertenece q_0 no esta vació

$x = pop(S_{q_0.v})$
$q_n = (x_{start}, q_0.v)$
$pqQ.push(q_n)$ // Agregamos un nuevo elemento q_n a pqQ

WHILE $(q_0.p \geq top(C).p)$ **DO:**

$c_0 = pop(C)$
$o_{(c_0.v).start} = o_{(c_0.v).start} + 1$

IF $o_{(c_0.v).end} \leq o_{(c_0.v).start}$ **THEN:**

$nEmpty = nEmpty + 1$

END WHILE

IF $o_{(c_0.v).end} \leq o_{(c_0.v).start}$ **THEN:**

$nEmpty = nEmpty - 1$

IF $nEmpty = 0$ **THEN:**

$nw \rightarrow N$ – Tupla de índices de intervalos
FOR $i = 1$ **TO** N **DO:**
$s = sequence(o_i)$
$nw = nw \times s$
$NW = NW \cup nw$
END FOR
$o_{(c_0.v)}.end = o_{(c_0.v).end} + 1$
$i = q_0.v$
$j = o_{i.end}$
$c_n = (s_{i,j.end}, i)$
$pqC.push(c_n)$
END WHILE

RETURN NW
END FUNCTION

3.2. ALGORITMO DE BÚSQUEDA

El algoritmo de búsqueda de intersecciones se basa en conceptos obtenidos del algoritmo de Juan Pablo Castillo[3] para el manejo de intervalos y por Barbay y Kenyon[7] para el algoritmo de búsqueda, algoritmo que se encuentra descrito en Algoritmo 3.4. Este utiliza la estructura original propuesta por Barbay y Kenyon apoyándose de las mejoras a la búsqueda de intervalos desarrolladas por Castillo.

Para explicar el Algoritmo 3.4, partiremos definiendo x cómo el segmento de intervalo actual que se esta intersecando, el cual se inicializa cómo el intervalo con mayor inicio entre los primeros elementos de cada conjuntos, e iremos acotando este x cómo la intersección de cada conjunto que revisamos. De esta forma, x no solo nos permitirá contabilizar la intersección actual y revisar si es que existe una intersección, sino que además, si este se encuentra en todos los conjuntos significa que encontramos una intersección entre todos los conjuntos.

El algoritmo utiliza de una búsqueda exponencial para encontrar un índice de partida que permita acelerar una búsqueda binaria sobre x, de la cual encontraremos el índice del intervalo $s_{i,j}$ que interseca con x dentro de un conjunto S_i, ambos algoritmos quedan descritos en el Algoritmo 3.2 y 3.3 respectivamente. Importante notar, que nuestro algoritmo de búsqueda binaria buscara la intersección más a la izquierda posible de nuestro intervalo x, de no existir, retornara la posición en donde se debería encontrar ese intervalo. Esta propiedad es importante para evitar saltar intervalos contenidos en otros intervalos y además nos ayuda a definir un nuevo intervalo de partida en caso de no haber intersección.

Si la búsqueda binaria nos encuentra un nuevo intervalo x_n perteneciente a S_i que interseque con nuestro previo x, reemplazaremos este último con el resultado de la intersección entre x y x_n, anotando el índice de x_n dentro de S_i y marcando este conjunto para recordar que se encontró una intersección dentro de el. Por el contrario, si x_n no interseca con x, diremos que ya no existe intersección entre los conjuntos y utilizaremos el nuevo x_n obtenido cómo nuevo punto de partida, eliminando nuestras marcas anteriores y anotando el índice y el conjunto cómo se hizo previamente.

Si al intersecar un intervalo x_n y marcar el conjunto S_i nos encontramos con todos los conjuntos marcados, significa que hemos encontrado una intersección y anotamos todos los índices de los intervalos en cuestión en una tupla nw de largo N, donde cada elemento de esta tupla será el índice del intervalo correspondiente según el conjunto perteneciente al intervalo. Por ejemplo, si tuviéramos $nw = (1,1,3)$, significa que tenemos una intersección entre $S_{1,1}$, $S_{2,1}$ y $S_{3,3}$.

Cuando encontramos una intersección múltiple debemos definir un nuevo punto de búsqueda, para esto, tenemos que marcar un nuevo conjunto y definir un nuevo x perteneciente a él para continuar iterando. Si es que el intervalo x_n del que encontramos la última intersección aún tiene una porción al final sin intersecar, podemos definir x cómo esta parte sin intersecar, que corresponde a $x = (x_{end}+1,\ x_{n.end})$ y marcar este mismo conjunto y índice. Por el contrario, si nos encontramos al final de x_n, continuaremos con el intervalo siguiente a x_n dentro de su mismo conjunto.

Finalmente, para poder iterando entre cada conjunto, utilizamos un contador que vamos incrementando en uno por conjunto, y nos aseguramos de no pasarnos del total de conjuntos utilizando el operador modulo sobre este contador.

Este algoritmo es bastante intuitivo de implementar y utiliza conceptos bastante conocidos en el área de la informática para entregar una solución bastante directa y efectiva para el problema de intersección de múltiples intervalos.

3.2.1. NOTACIÓN O GRANDE

Ya hemos introducido la complejidad del algoritmo propuesto por Barbay y Kenyon cómo:

$$O(N\,\delta \log(n/\delta))$$

con el parámetro de dificultad $\delta = I+min(P)$, la cual se mantiene para nuestro caso de intervalos múltiples. Es clave destacar que este algoritmo escala rápidamente ante instancias de dificultad muy altas, o sea, con gran cantidad de intersecciones, dado que este aporta un factor dependiente de no solo del total de conjuntos N, sino que además del intervalo con mayor elementos n a la cota asintótica.

El cálculo de $min(P)$ no es trivial y se requiere de un procesamiento del conjunto solución NW, este trabajo no se estudiará en la presente memoria, pero nos interesa mas el comportamiento general. Por lo tanto, podemos concluir que tenemos un algoritmo que se ve bastante eficiente para la gran mayoría de las instancias, y solo se aproximaría a $O(n^2)$ para casos donde N se aproxime a δ^2, lo cual es una poco común dado que N generalmente será mucho menor a δ. Pero podría ser muy ineficiente cuando hay muchos conjuntos con muchas intersecciones.

Algoritmo 3.2: Búsqueda Exponencial del intervalo x en S_i.

INPUT:

1. Un conjunto de intervalos S_i.
2. Un intervalo x que buscamos dentro de S_i.

OUTPUT: Indice j del último intervalo encontrado con búsqueda exponencial cuadrática dentro de S_i que no termina antes que x inicia.

FUNCTION *ExponentialSearchLeft* **START**:

IF $intersect(x, s_{i,1})$ **RETURN** $i = 1$

$j = 1;$

$n = |S_i|;$

WHILE $(j < n)$ **AND** $(S_{i,j.start} \leq x_{start})$ **DO:**

$j = 2j$

RETURN $floor(j/2)$

Algoritmo 3.3: Búsqueda Binaria del intervalo x más a la izquierda.

INPUT:

1. Un conjunto de intervalos S_i.
2. Un intervalo x que buscamos dentro de S_i.
3. El índice de partida a obtenido con la búsqueda exponencial.

OUTPUT: Indice i del intervalo encontrado con búsqueda exponencial que no termina antes que x inicia.

FUNCTION *BinarySearchLeft* **START**:

$iLeft = a$

$iRight = |S_i|$

WHILE $(iLeft < iRight)$ **DO:**

$iMid = floor((iLeft + iRight)/2)$

IF $(S_{i,iMid.end} \leq x_{start})$ **THEN:** $iLeft = iMid + 1$

ELSE THEN: $iRight = iMid$

END WHILE

RETURN $iLeft$

Algoritmo 3.4: Algoritmo de búsqueda de intersección con conjuntos de intervalos.

INPUT:

1. Un conjunto de conjuntos de intervalos $\boldsymbol{S}$.

OUTPUT: Un conjunto NW de índices intervalos dentro de $\boldsymbol{S}$ que hacen una intersección de múltiples conjuntos de intervalos.

FUNCTION *SearchIntersection* **START**:

$NW \rightarrow$ Conjunto de resultados

$nw \rightarrow$ Lista de tamaño N con los índices de la intersección actual

$marked = 0$ Cantidad de conjuntos marcados.

$lastEnd \rightarrow$ Valor del intervalo con mayor término $s_{i,j.end}$

$Inter \rightarrow$ Resultado de última intersección, se inicializa cómo el intervalo $s_{i,j}$ con mayor $start$

$currIdx \rightarrow$ Indice del intervalo candidato a intersección.

$currSet \rightarrow$ Indice i del conjunto S_i al que pertenece $currIdx$

WHILE $currInt_{start} < lastEnd$ **DO:**

$currSet = (currSet+1)\%N$

$currIdx = exponentialSearchLeft(S_{currSet}, Inter)$

$currIdx = binarySearchLeft(S_{currSet}, Inter, currIdx)$

IF $currIdx > |S_{currSet}|$ **THEN: BREAK WHILE**

IF $intersect(Inter, s_{currSet,currIdx})$ **THEN:**

$marked = marked+1$

$nw_{currSet} = currIdx$

$Inter = intersectionInterval(Inter, Sets_{currSet,currIdx})$

ELSE THEN:

$marked = 1$

$Inter = Sets_{currSet,currIdx}$

$nw_{currSet} = currIdx$

IF $marked = |\boldsymbol{S}|$ **THEN:**

$NW.push_{back}(nw)$

$marked = 1$

IF $Inter_{end} < s_{currSet,currIdx.end}$ **THEN:**

$Inter = (Inter.end+1, s_{currSet,currIdx.end})$

ELSE THEN:

$nw_{currSet} = nw_{currSet}+1$

IF $|S_{currSet}| = currIdx+1$ **THEN: BREAK WHILE**

$Inter = s_{currSet,currIdx+1}$

END WHILE

RETURN NW

END ALGORITHM

CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN

4.1. METODOLOGÍA DE TRABAJO

Para comparar el tiempo de cómputo y obtener un análisis comparativo consistente que nos permita evaluar la eficiencia de ambos algoritmo, utilizaremos una librería heredada de C++, denominada *chrono*, la cual nos permite obtener una hora con bastante precisión, al nivel de nano segundos, en el antes y después de la ejecución de nuestros algoritmos de intersección, así, podemos comparar las diferencias de ambos, y calcular el tiempo de ejecución. La implementación y los resultados de los experimentos se encuentran disponibles en github[1].

Estos cálculos serán realizados en un computador con las siguientes características:

- Intel(R) Core(TM) i5-10300H CPU @2.50GHz.
- 24Gb de RAM instalada (23,8Gb usables, ~16Gb disponibles).
- Sistema Operativo Windows 10 Home Single versión 21H2.
- Compilado con g++ (Rev10, built by MSYS2 project) 11.2.0.

Las pruebas se realizarán sobre diversos conjuntos de datos genómicos, los cuales serán trabajados con BEDTools y Python 3 y preparados para ser utilizados por nuestros código. En más detalle, compararemos el tiempo de cómputo de ambos algoritmos con las siguientes muestras que definirán diversos casos de prueba para el valor de nuestros conjuntos $\boldsymbol{S}$, dados por los siguientes datos:

- Cuatro versiones del primer cromosoma del genoma humano (hg38, gh19, hg18, hg17).
- Cinco versiones del primer cromosoma del genoma del ratón (mm39, mm10, mm9, mm8, mm7).
- Tres versiones del primer cromosoma del genoma del chimpancé (panTro6, panTro5 y panTron4).

Ademas, agruparemos cada una de las muestras anteriores a pares y con el total de ellas para obtener comparaciones entre las diversas especies que se mencionaron anteriormente. Estas últimas entonces corresponderían en más detalle a:

[1]

- Las cuatro versiones del primer cromosoma del genoma humano y las cinco versiones del primer cromosoma del ratón.
- Las cuatro versiones del primer cromosoma del genoma humano y las tres versiones del primer cromosoma del chimpancé.
- Las tres versiones del primer cromosoma del chimpancé y las cinco versiones del primer cromosoma del ratón.
- Las cuatro versiones del primer cromosoma del humano, las tres versiones del primer cromosoma del chimpancé y las cinco versiones del primer cromosoma del ratón.

Con nuestros resultados, realizaremos y analizaremos las tablas y gráficos que nos permitan comparar visualmente los resultados de los algoritmos implementados, particularmente, nos interesa ver cómo la magnitud de intervalos, conjuntos e intersecciones afecta en el tiempo de ejecución de estos.

4.2. GENERACIÓN DE DATOS

Utilizaremos la herramienta Table UCSC Table Browser para obtener los datos brutos BEDs que necesitaremos para nuestras pruebas. Aquí podemos exportar secuencias genómicas en diversos formatos y bajo diversas especificaciones, pero nos centraremos en descargar las secuencias previamente listadas en formato BED.

Antes de poder iterar sobre nuestras secuencias en BED, necesitamos procesarlos con BEDTools y su función de *merge*, para aplanar las secuencias que se sobrelapan y asegurarnos de crear conjuntos de datos disjuntos. Este proceso es necesario, dado a que el concepto de gen se ha mencionado es impreciso y se utilizó para presentar de forma más familiar los conceptos genómicos en cuestión, pero realmente, un BED contiene *features*[1], que son secciones del genoma que contienen alguna función anotada y no son necesariamente genes. Como estas *features* pueden intersecar con elementos de su mismo cromosoma[2], con BEDTools nos aseguramos de crear el conjunto disjuntos que necesitamos para nuestros algoritmos.

Con nuestros conjuntos de intervalos preparados, utilizaremos un script en Python para separar los intervalos por cromosoma, esta operación nos permitirá extraer los intervalos, del primer cromosoma de los distintos genomas que formarán S para cada prueba, los cuales corresponderán a un mismo cromosoma de una o varias especie, con diferentes versiones de secuenciación. Podemos ver un detalle del total de intervalos de cada conjunto en la tabla 2.

[1]Un *feature* es similar a la idea de un gen, pero es más generalizada, considerando partes del genoma que no necesariamente son genes, pero que poseen alguna función en el proceso celular.

[2]Por ejemplo, un *feature* puede ser el gen y otro *feature* una sección dentro del mismo gen con un propósito modularizado o especializado.

La selección de estos conjuntos se debe a la similitud genética entre el humano y el chimpancé, compartiendo un más del 98% del ADN[9]. Estas similitud nos da un punto de comparación entre la otra especie, el Ratón, la cual presenta muchas diferencias morfológicas y genómicas a las otras dos especies, permitiéndonos suponer que es un conjunto de datos el cual presentara pocas intersecciones cuando es comparado a las otras dos especies que poseen más similitud entre ellas.

En contraste, es muy esperable la existencia de grandes similitudes entre las diferentes versiones de un mismo genoma de la misma especie, por lo tanto, cuando realicemos las pruebas entre las diferentes versiones de una misma especie, deberíamos esperar un elevado número de intersecciones.

Finalmente, la comparación cruzada entre Humano, Chimpancé y Ratón se hace para evaluar el funcionamiento en un caso donde la magnitud de S es alta y con múltiples conjuntos que poseen grandes diferencias entre ellos, por lo tanto, es de esperarse un bajo número de intersecciones.

Tabla 2:
Conjuntos de intervalos conformado por primer cromosoma de distintas versiones y especies.

CONJUNTO (versión)	INTERVALOS
HUMANO (Homo Sapiens)	
Hg38	3018
Hg19	2164
Hg18	2010
Hg17	1833
CHIMPANCÉ (Pan Troglodytes)	
panTro6	2395
panTro5	2291
panTro4	2242
RATÓN (Mus Musculus)	
mm39	2221
mm10	2176
mm9	1235
mm8	1050
mm7	1026

4.3. EXPERIMENTACIÓN

Para comenzar el proceso de experimentación, es importante conocer las propiedades clave de cada experimento, partiremos presentando una de las más básicas, esta es la cantidad de intervalos de cada conjunto o experimento $\boldsymbol{S}$, por esto, se ha rellenado la tabla 3 con cada uno de estos valores, los cuales nos dan una visualización clara de la composición cada experimento $\boldsymbol{S}$ y la magnitud de los conjuntos de intervalos S_i que lo conforman.

Tabla 3: Cantidad de intervalos de cada conjunto

	Intervalos de cada Conjunto											
Experimento **S**	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11
Humano	1833	2010	2164	3108								
Chimpancé	2242	2291	2395									
Ratón	2176	2221	1026	1050	1235							
Humano Chimpancé	1833	2010	2164	3108	2242	2291	2395					
Humano Ratón	1833	2010	2164	3108	2176	2221	1026	1050	1235			
Chimpancé Ratón	2242	2291	2395	2176	2221	1026	1050	1235				
H/C/R	1833	2010	2164	3108	2242	2291	2395	2176	2221	1026	1050	1235

Ejecutaremos los algoritmos implementados en 8 iteraciones para obtener diversos valores de tiempo de ejecución y el total de intersecciones encontradas para cada muestra, y aún que es claro que ambos algoritmos entregan las mismas intersecciones cómo solución, los tiempos de ejecución varían considerablemente. Hablaremos más en detalle sobre los tiempos de ejecución en los capítulos siguientes, dado que primero es clave comprender otras propiedades relacionadas a los conjuntos y sus intersecciones resultantes.

Generaremos la tabla 4, la cual nos entrega propiedades que nos servirán para comparar cómo se relacionan el conjunto de entrada y las intersecciones encontrada con respecto a su impacto en el tiempo de ejecución según. En detalle, las columnas de la tabla vienen definidas en orden cómo:

- N : Cantidad de conjuntos de intervalos que componen nuestra conjunto $\boldsymbol{S}$ en cada experimento.

- Total Intervalos: Suma total de la cantidad de intervalos de cada S_i del experimento.
- Promedio de intervalos por conjunto: Total de intervalos sobre la cantidad de conjuntos N del experimento.
- I : Total de intersecciones encontradas en el experimento.
- $I*N$: Producto entre total de intersecciones y la cantidad de conjuntos N .
- $\%\ NW$ / Promedio de Intervalos: Porcentaje del total de intersecciones respecto al promedio de intervalos del conjunto.

Tabla 4: Resultados y propiedades de cada experimento

Experimento	N	Intervalos	Promedio Intervalos	I	I * N	% I / Promedio Intervalos
Humano	4	9115	2278,75	1708	6832	74,95 %
Chimpancé	3	6928	2309,33	2036	6108	88,16 %
Ratón	5	7708	1541,60	541	2705	35,09 %
Humano Chimpancé	7	16043	2291,86	660	4620	28,80 %
Humano Ratón	9	16823	1869,22	191	1719	10,22 %
Chimpancé Ratón	8	14636	1829,50	241	1928	13,17 %
H/C/R	12	23751	1979,25	70	840	3,54 %

Nos serán de utilidad el número de conjuntos N , el total de Intervalos I y el promedio de intervalos para graficar y comparar cómo afectan al tiempo de ejecución con respecto a estos. El promedio de intervalos será útil para poder ver que tanto afectan en proporción ambos parámetros al tiempo de ejecución, aún que este valor podría ser no muy conclusión.

Los propiedades I y $I*N$ las utilizaremos para comprobar el impacto de las intersecciones encontradas, dado que, es claro que el tiempo de ejecución se ve afectado por estas, cómo ha sido presentado en las cotas de ambos algoritmos. Finalmente, el valor $\%I/Promedio$ es de utilidad para darnos una idea que tan similares son cada experimento.

Con esta tabla, podemos ver que la mayor similitud entre conjuntos se dio con los experimentos del Humano y el Chimpancé, en donde un 74,95% y 88,15% de los intervalos

promedios intersecan respectivamente. Y a pesar de que en el caso del ratón se esperaba un resultado similar, estas diferencias podrían ser explicadas por la mayor variabilidad genética derivada por la superior capacidad reproductiva del ratón y a que los conjuntos S_2 y S_3 tienen muchos menos intervalos menos que el promedio.

Es interesante notar que en la comparación entre pares de especies, el mayor porcentaje de similitud se dio entre el Humano y el Chimpancé, con un 28,80% de intersecciones con respecto al promedio de intervalos, en contraste al 10,22% y 13,17% de los experimentos Humano/Ratón y Chimpancé/Ratón respectivamente.

El bajo porcentaje de intersecciones con respecto al promedio de intervalos encontrado en el experimento H/C/R era esperable dada la presente diversificación de especies del experimento, el mayor número de conjuntos N y el alto número de intervalos totales, todas estas propiedades y las diferencias entre especies dificultan la posibilidad de una intersección múltiple, pero esas pequeñas encontradas son de valor para estudio al destacar ante una situación de intersección complicada..

4.3.1. ALGORITMO *SWEEP*

Ejecutando el algoritmo s*weep* presentado en Algoritmo 3.1, pudimos obtener los tiempos de ejecución para cada iteración sobre cada experimento, estos resultados se encuentran presentes en la tabla 5, en donde las mediciones marcadas en rojo y verde representan los dos mayores y menores tiempos respectivamente.

Podemos notar que el algoritmo tuvo un tiempo de ejecución promedio de 24,96ms y los mayores tiempos promedio correspondieron a los experimentos Chimpancé y Chimpancé/Humano con 51,89ms y 32,96ms respectivamente. Estos también presentaron las desviaciones estándares más altas de 24,70 y 21,63. Podemos suponer que mientras más intersecciones tengan los conjuntos, mayor es el aporte del factor exponencial IX^N, lo que es consistente para el caso del chimpancé, el cual posee múltiples intersecciones a pesar de su bajo M, dándonos a ver que el factor $I(N+X^N)$ de la cota, tiene un efecto significativo en el tiempo de cómputo, incluso cuando M sea más pequeño que I.

El resultado de desviaciones estándar altas da a entender que se produjo algún fenómeno que incremente considerablemente el tiempo de cómputo, es interesante notar que a pesar de que el experimento del Chimpancé tomo tiempos muchos mayores al promedio en casi todos sus casos, existe un resultado, de la 2da iteración, que demostró un tiempo considerablemente bajo a los demás, lo que podría dar una evidencia que el alto tiempo fue afectado por algún factor externo.

Los experimentos con menor tiempo de ejecución promedio corresponden a los del Ratón y Chimpancé/Ratón cada uno con 10,73ms y 16,26ms respectivamente. La desviación estándar para estos casos también corresponden a las más bajas siendo estas 2,71 y 2,84.

Tabla 5: Tiempos de ejecución algoritmo Sweep

	Tiempo de ejecución iteraciones del Algoritmo [ms]									
	Iteración									
Experimento	1ra	2da	3ra	4ta	5ta	6ta	7ma	8va	Promedio	Desv. Estandar
Humano	17,90	16,99	27,40	16,70	31,23	13,00	14,62	15,47	19,16	6,53
Chimpancé	52,19	10,18	42,02	24,97	69,68	63,36	70,79	81,95	51,89	24,70
Ratón	9,83	12,13	16,50	10,73	7,28	10,42	10,08	8,90	10,73	2,72
Humano Chimpancé	72,57	19,11	34,54	17,04	59,69	26,40	16,09	18,25	32,96	21,63
Humano Ratón	15,47	20,33	24,30	16,08	18,76	18,81	13,09	16,96	17,97	3,42
Chimpancé Ratón	16,92	17,42	20,06	18,87	11,95	13,61	13,72	17,49	16,26	2,84
H/C/R	24,74	24,35	37,97	27,63	20,08	24,39	26,45	20,10	25,71	5,64
PROMEDIOS	29,95	17,22	28,97	18,86	31,24	24,28	23,55	25,59	24,96	

4.3.2. ALGORITMO DE BÚSQUEDA

Realizando múltiples ejecuciónes del algoritmo de búsqueda, pudimos rellanar la tabla 6, la cual corresponde a los tiempos de ejecución en milisegundos para cada iteración del algoritmo, sobre los experimentos presentados. Al igual que en el ejemplo anterior, los números destacados en verde y rojo son los dos menores y mayores tiempos de ejecución respectivamente.

Podemos notar que el tiempo de ejecución del algoritmo en la gran mayoría de los casos no supero los 4 milisegundos, con excepciones muy particulares que se repitieron constantemente en las iteraciones del experimento Humano/Chimpancé, el cual presento mayores tiempos promedio que las demás muestras y una gran desviación estándar, lo que da a ver la disparidad entre los tiempos.

El experimento con menor tiempo de ejecución promedio corresponde al del Humano / Chimpancé / Ratón con 1,43m. Los experimentos en este caso tuvieron valores muy similares, y es interesante ver cómo la mayor cantidad de intervalos, no implica necesariamente mayor tiempo de ejecución, esto se ha deber, que a mayor número de conjuntos, es más difícil encontrar una intersección múltiple, por lo tanto, más fácil saltar de intervalo en intervalo y es menos lo que hay que verificar.

Comparando ambas cotas asintóticas es esperable suponer un mejor tiempo de cómputo, pero la diferencia fue significativa, dando a ver que este algoritmo es una alternativa bastante eficiente para resolver la intersección múltiple entre conjuntos disjuntos, o en particular, la comparación de secuencias genómicas cómo hemos realizado.

Tabla 6: Tiempos de ejecución algoritmo de búsqueda.

	Tiempo de ejecución iteraciones del Algoritmo [ms]									
	Iteración									
Experimento	1ra	2da	3ra	4ta	5ta	6ta	7ma	8va	Promedio	Desv. Estandar
Humano	3,97	3,10	4,37	3,68	4,48	3,84	4,06	4,56	4,01	0,48
Chimpancé	2,32	2,29	2,43	2,26	2,26	2,26	2,29	2,32	2,30	0,06
Ratón	1,49	1,81	2,30	2,00	1,95	2,42	1,84	1,97	1,97	0,29
Humano Chimpancé	2,87	4,44	8,10	4,47	3,94	2,89	5,86	4,57	4,64	1,70
Humano Ratón	1,86	5,50	2,13	2,04	1,38	1,87	1,83	1,46	2,26	1,34
Chimpancé Ratón	2,09	1,91	1,60	2,22	1,59	1,64	1,91	2,82	1,97	0,41
H/C/R	1,17	1,22	1,53	1,24	1,17	1,95	1,37	1,84	1,43	0,31
PROMEDIO	2,25	2,90	3,21	2,56	2,40	2,41	2,74	2,79	2,66	

4.4. ANÁLISIS DE RESULTADOS

Utilizaremos las tablas 4, 5 y 6 para generar gráficos que nos permitan comparar visualmente cómo se comportaron nuestros algoritmos ante distintos conjuntos de entrada. El eje y de los gráficos corresponderán a los tiempos de ejecución que hemos presentados en las tablas 5 y 6 y nuestro eje x serán las diversas propiedades que se han presentado en la tabla 4 que iremos analizando en los siguientes subcapitulos de forma individual. Todos los gráficos tendrán el eje de tiempo en escala logarítmica para facilitar la visualización de las variaciones de ambos algoritmos.

4.4.1. TIEMPO DE EJECUCIÓN PROMEDIO.

El algoritmo de búsqueda basado en los desarrollo de Juan Pablo Castillo y Barbay y Kenyon demostró un tiempo de cómputo superior a los tiempos obtenidos con el algoritmo de *sweep*, en donde podemos notar un tiempo de ejecución promedio casi 10 veces mayor para este último.

Esta mejora significativa hace sentido con las cotas de complejidad presentadas para ambos algoritmos, en donde, podemos ver que para el algoritmo de búsqueda tenemos una cota más compleja, con un factor dependiente del total de intervalos y un factor exponencial dependiente de N. A demás, el algoritmo de búsqueda itera al menos sobre todos los elementos M, incluso viendo intervalos que probablemente nunca tengan intersección, pero, si consideráramos la cota ofrecida por el factor del producto cartesiano fuera de la notación O, podríamos ver que el algoritmo de *sweep* tendría una cota mejor para cuando M es menor al factor δN.

Un tema que no se ha mencionado aún respecto al algoritmo de *sweep,* es que el algoritmo propuesto en el *paper* de Layer y Quinlan fue diseñado cómo una solución considerando procesamiento paralelo, enfocándose en la partición de los conjuntos en subconjuntos que aseguren la existencia de intersecciones. Este principio de desarrollo pudo tener un impacto significativo al presentar comparaciones de forma secuencial, pero el análisis comparativo en una ejecución paralela no será abordado en esta memoria.

Utilizando la cota presentada en [2] podemos ver que el algoritmo *sweep* es más dependiente de M, el total de intervalos y en menor medida del factor total de intersecciones, cómo es en el caso del algoritmo de búsqueda, donde δ juega un papel más significativo en el crecimiento de la cota. Por lo tanto, al separar los conjuntos en intervalos que seguramente tienen intersección, es probable encontrar un M muy pequeño con un total de intersecciones W no necesariamente pequeño, dado que con n intervalos en un conjunto, es hasta posible tener hasta n^2 intersecciones con los otros.

Estos análisis nos permiten concluir que es probable que el algoritmo de búsqueda sea menos eficiente para conjuntos donde existen muchas más intersecciones que intervalos y

el algoritmo *sweep* sería más eficiente para cuando existen pocos intervalos y muchas intersecciones, pero para corroborar esta hipótesis se requiere del estudio de ambos algoritmos en el caso paralelo.

Podemos hacer una aproximación básica haciendo una igualdad entre ambas cotas asintóticas. Haremos un cambio en la variable n, para que quede respecto a M y sean ambas más fácil de comparar, para esto, aproximaremos n cómo el promedio de intervalos, lo cual corresponde a la igualdad $n \approx M/N$. También, ignoraremos el factor WX^N para el algoritmo de sweep, dado que los contextos (sin realizar el producto cartesiano) también son una representación de la solución, quedamos finalmente con una igualdad entre cotas de:

$$M\,logM + I\,N \;=\; \delta\,N \log\left(\frac{M}{\delta\,N}\right)$$

Esta igualdad nos permite concluir que ambos algoritmos escalan diferente ante los elementos de entrada y las intersecciones encontradas, el algoritmo de *sweep* será más ineficiente que el algoritmo de búsqueda mientras el producto δN sea menor que M o sea, un caso con pocas intersecciones y muchos intervalos, lo cual es una situación común para el caso secuencial que hemos estado estudiando. Pero, cuando las intersecciones son mucho mayores que M, vemos que el factor $I\,N$ debería crecer más lentamente que $\delta N \log\left(M/(\delta N)\right)$, esta situaciones es bastante más común para el caso paralelo del algoritmo, donde debemos de particionar los conjuntos en zonas las cuales aseguren la intersección, disminuyendo M y aumentando I para cada partición.

4.4.2. TOTAL DE INTERVALOS VS TIEMPO DE EJECUCIÓN PROMEDIO

Uno de los parámetros que más intuitivamente podría afectar en el tiempo de ejecución es el total de intervalos, en la figura 6 tenemos el gráfico comparativo podemos revisar la comparativa entre el total de intervalos y el logaritmo de los tiempos de ejecución promedio, donde hemos obtenido dos líneas de tendencias a la baja del tiempo con el aumento de intervalos.

Para el caso del algoritmo de sweep, el primer punto del gráfico es atípico y lo podemos considerar un *outlier*, haciendo notar notar una clara tendencia a la alta al tiempo con el aumento de intervalos, lo cual es consistente con la cota asintótica que hemos planteado para este algoritmo, y el alto valor primer punto puede ser explicado con que es consistente con el factor $I\,N$, considerando que el primer valor es la medición correspondiente al chimpancé, el cual posee el mayor número de intersecciones.

El algoritmo de búsqueda tiene una clara tendencia a la baja con el aumento del total de intervalos, y esto también es esperable dado a que este no es afectado directamente por el total de intervalos, sino más bien, por el promedio de este respecto al total de

conjuntos. Las variaciones del tiempo de ejecución también se dan por el total de intersecciones y el parámetro de dificultad.

Finalmente, es interesante notar que existen dos picos de tiempo para el experimento con 16000 intervalos, este punto corresponde al experimento Humano / Chimpancé, el cual es uno de los conjuntos con mayor promedio de intervalos, un alto número de intersecciones y un N de 7, lo que lo convierte en un candidato costoso de resolver para ambos algoritmos.

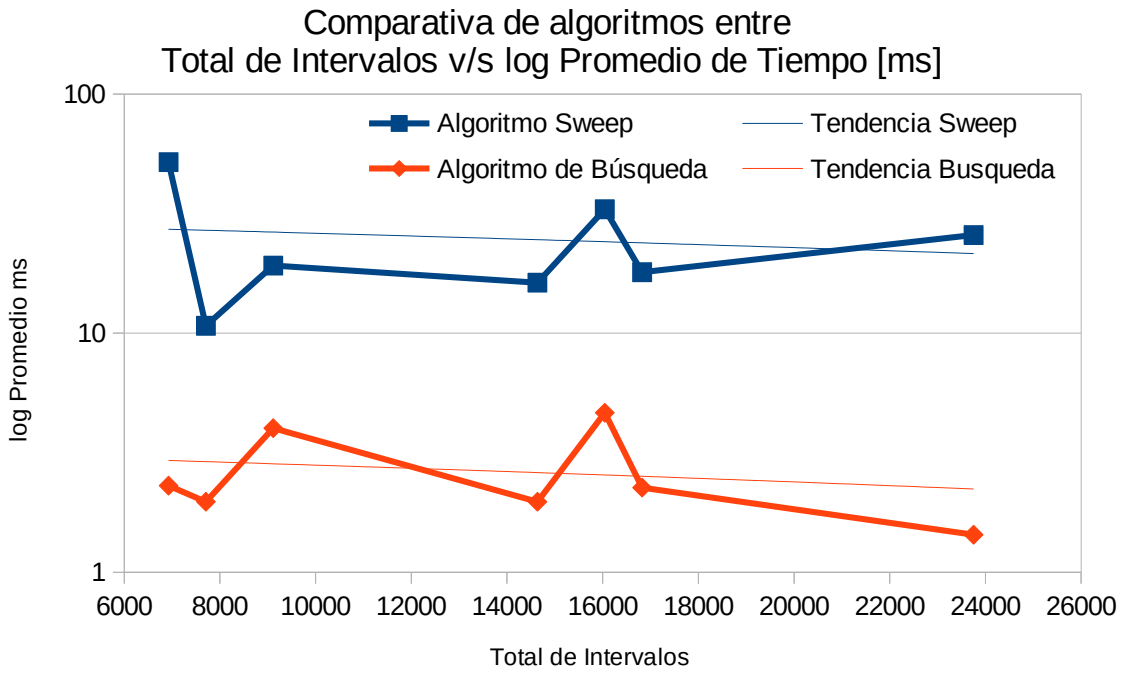

Figura 6: Gráfica comparativo entre el total de intervalos y el promedio de tiempo. Fuente: Origen Propio

4.4.3. PROMEDIO DE INTERVALOS VS TIEMPO DE EJECUCIÓN PROMEDIO

El promedio de intervalos es otro valor de interés para ver cómo se relacionan los algoritmos bajo el supuesto de que los conjuntos tienen cantidad de intervalos al menos similares, para esto, en la figura 7 tenemos un gráfico comparativo de estos algoritmos, en donde podemos notar una tendencia a la alta ante el aumento del promedio de intervalos. Para el algoritmo de búsqueda esto es esperado, y podemos notar que este factor también es alterado por la dificultad δ, lo que hace que el crecimiento no sea constante a pesar del aumento del promedio de intervalos.

En la cota asintótica del algoritmo de *sweep* no tenemos una relación directa con el promedio de intervalos, pero la gráfica nos deja ver un crecimiento aparentemente exponencial al aumento del mismo, pero se requieren de más puntos estudios para comprender el por que de este comportamiento. Es importante destacar que el promedio

de intervalos no se presenta en las cotas, pero siempre es un valor de interés a revisar para todo tipo de estudios comparativo.

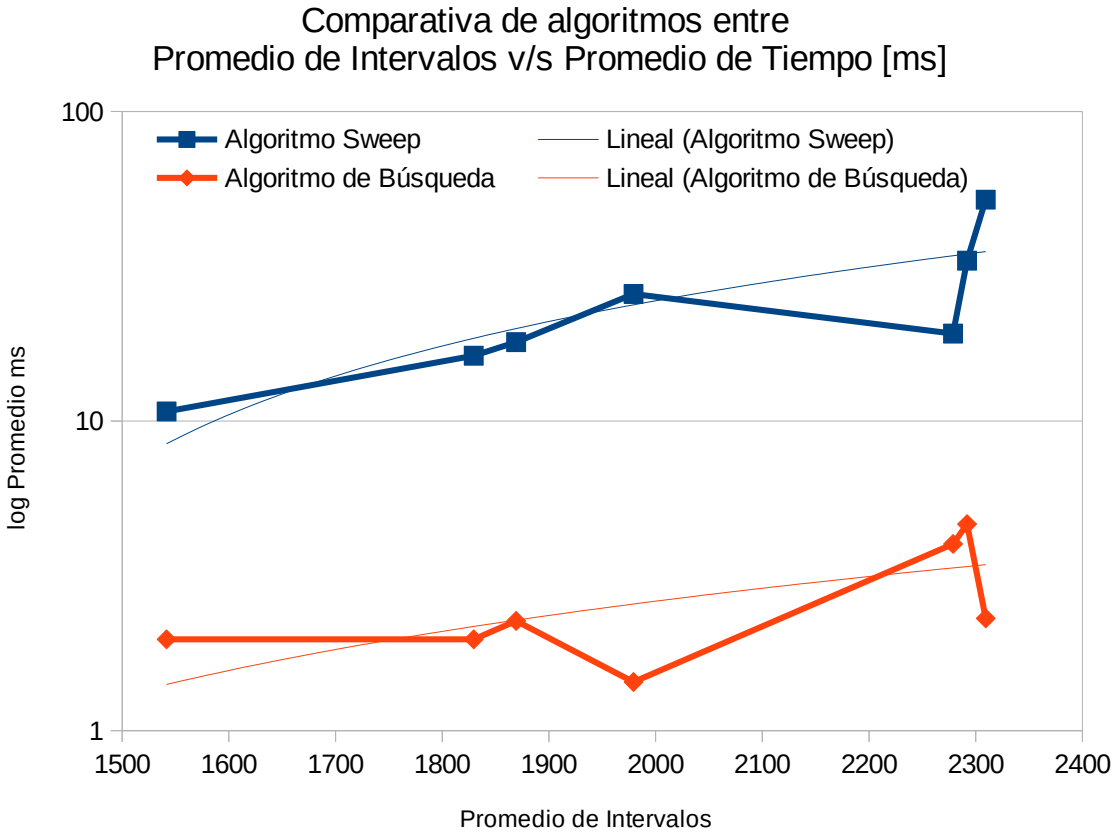

Figura 7: Gráfica comparativa Promedio de intervalos vs Promedio de tiempo.
Fuente: Origen Propio.

INTERSECCIONES POR CONJUNTO VS PROMEDIO DE EJECUCIÓN

La última gráfica que analizaremos es cómo afectan las intersecciones y el total de conjuntos al tiempo de ejecución, en la figura 8 tenemos la gráfica del producto entre el total de intersecciones I y el número de conjuntos N con respecto al logaritmo del promedio del tiempo de ejecución.

Sabemos que tenemos al menos una expectativa de crecimiento lineal bajo el total de intersecciones para ambos algoritmos, para el de *sweep* un factor de IN y para el de búsqueda el factor δ, el cual depende de I. Esto nos deja ver una tendencia de crecimiento lineal para ambos algoritmos ante el producto IN.

Es obvio que el tiempo de cómputo no depende exclusivamente del total de intersecciones, pero podemos ver cómo este producto posee una tendencia clara de subida para ambos conjuntos, los que no da a ver cómo juega un papel clave en el tiempo

de ejecución. Los algoritmos con mayor producto corresponden a los conjuntos del Humano, Chimpance y Humano / Chimpance, los cuales, todos tienen un elevado número de intersecciones, y a pesar de que el tamaño de conjuntos que los compone no es tan grande, el total de intersecciones si lo es.

Podemos notar que el algoritmo de búsqueda se comporto más estable ante las variaciones del producto IN, lo cual es esperable, dado que es el principal factor del cual depende la cota del algoritmo de búsqueda y el factor $\log(n/\delta)$ solo ajustara más la curva, a diferencia del elemento $M\,logM$ que es independiente de este producto, lo que hace que se presenten más variaciones en el mismo.

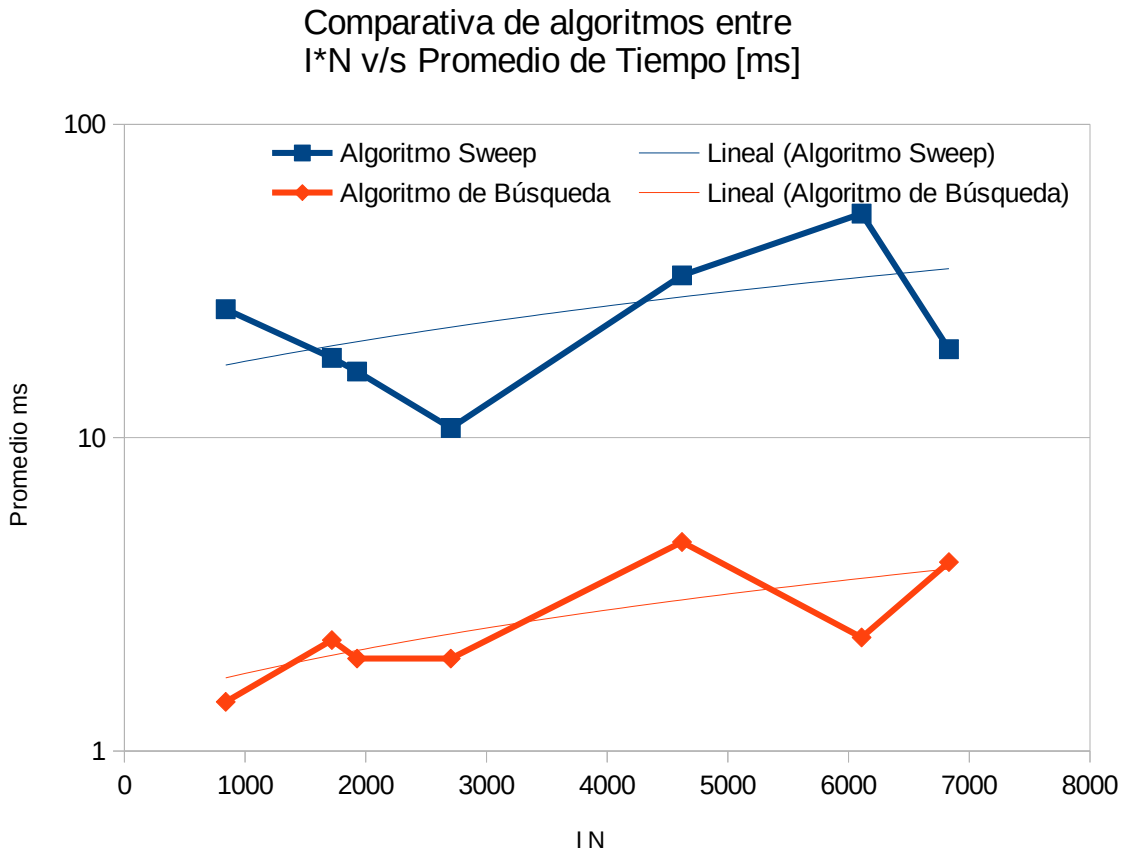

Figura 8: Gráfica comparativa producto total de intersecciones y conjuntos vs log Promedio de Tiempo.

Fuente: Origen propio.

CAPÍTULO 5: CONCLUSIONES

Es de gran valor para el análisis comparativo de conjunto de datos el uso de la intersección para identificar rápidamente elementos que similares entre ellos, sin importar si los elementos que trabajamos son números enteros, segmentos de línea o superficies en el plano, o el particular de nuestro caso de estudio, conjuntos de intervalos. Se ha profundizado bastante en el problema de intersección a pares, pero considerar las intersecciones entre los multiples conjuntos requiere de refinar nuevas técnicas para que esto se resuelva en un tiempo aceptable, sobre todo por que en este tipo de problemas es muy común que existan intervalos que no son parte de una solución.

Dado a la gran complejidad del ADN, es una tarea excesivamente costosa comparar secuencias de nucleótidos utilizando la cadena de caracteres que comprenden el genoma, por lo tanto, una primera aproximación utilizando intersección de múltiples conjuntos nos permite encontrar eficientemente cuales son las áreas del genoma con valor de estudio, permitiéndonos detectar similitudes entre especies, comparar versiones de genomas o analizar una posible línea evolutiva de algún ser vivo.

La representación BED de secuencias genómicas cómo genes con un *locus* definido dentro de cada cromosoma, nos permite hacer una analogía de un gen cómo un intervalo dentro de un conjunto, lo que nos permite realizar aproximaciones de similitud entre conjuntos obteniendo su intersección y resolviendo el problema de intersección múltiple de conjuntos de intervalos.

A pesar de que el formato BED es una simplificación efectiva del genoma, no deja de ser costosa de analizar, más aún en el caso de intersección múltiple, donde la cantidad de conjuntos de comparación puede ser muy elevada y eso no limita que las intersecciones sean muy altas, donde, dado a la naturaleza del genoma las especies pueden compartir grandes similitudes entre sí a pesar de ser muy diferentes.

El interés por el estudio del genoma ha aumentado considerablemente en los últimos años gracias a la disminución del costo de la secuenciación, y esto nos hace ver que es clave investigar algoritmos que nos aseguren tiempos de computación eficientes para abordar problemas de estudio cada vez más complejos, que involucran grandes cantidades de intervalos y conjuntos.

En la propuesta de esta memoria, se compararon diferentes algoritmos en experimentos realizados con distintas versiones de secuenciación de cada uno de los primeros cromosomas de las especies seleccionadas. Estos experimentos permitieron realizar análisis y gráficos que nos permitieron visualizar la efectividad en términos de tiempo de ejecución de los algoritmos implementados.

El algoritmo de búsqueda demostró ser eficiente en los experimentos estudiados, mostrando tiempos de ejecución inferiores a 5 ms en la gran mayoría de los casos. Esto es consistente con las cotas asintóticas presentadas para ambos algoritmos y los experimentos realizados, en donde el algoritmo *sweep*, tenía un impacto mucho mayor el total de intervalos en comparación al de búsqueda, lo que hizo demostrar la superioridad de este último.

Es importante destacar que ambos algoritmos fueron comparados desde un punto de vista secuencial, lo que pudo generar una desventaja en el cómputo para el algoritmo *sweep* presentado en *"A parallel algorithm for N-way interval set intersection"*, debido a que considera particiones de conjuntos para resolver la intersección múltiple de forma paralela y cómo este depende principalmente del total de intervalos M y el producto IN, podemos ver que para instancias con pocos intervalos y muchos conjuntos este algoritmo sería bastante eficiente, incluso pudiendo ser más eficiente que el algoritmo de búsqueda.

El algoritmo de búsqueda también se puede paralelar aplicando las mismas técnicas de partición que presentan los autores Layer y Quinlan en [2], examinando la cota estudiada para este algoritmo, es probable que sea eficiente y una alternativa más sencilla de implementar para conjuntos cómo los experimentos presentados en esta memoria, pero en las situaciones donde N e I son grandes y M es pequeño, podemos notar que el algoritmo de *sweep* debería ser más eficiente que el de búsqueda. Esto último requiere de mayor estudio y es relevante para la resolución de casos muchos más complejos, donde la cantidad de intervalos y conjuntos podría ser mucho mayor, por ejemplo, buscando una trazabilidad a lo largo de todo un genoma para una gran variedad de especies, la reducción del tiempo de cómputo entregado por el paralelismo es clave para estudiar estas grandes cantidades de datos.

A pesar de que el paralelismo es una alternativa, no deja de ser importante el análisis secuencial y el algoritmo de búsqueda demostró su efectividad en esta área, no solo siendo mejor en todos los casos estudiados, sino que además dejando margen de mejora, por ejemplo, utilizando de memoria para acotar la búsqueda exponencial según un índice de partida ya analizado.

Finalmente, podemos concluir que el algoritmo de búsqueda para la resolución de intersecciones de intervalos es una alternativa eficiente al algoritmo estado de arte de

sweep presentado por Layer y Quinlan, convirtiéndolo en una solución secuencial eficiente y fácil de implementar para resolver un problema de gran complejidad, cómo es la intersección de múltiples conjuntos de intervalos, problemática que es común para en el ámbito de la genómica, un área de estudio innovador y de gran complejidad, pero con mucho potencial para resolver problemas en otros ámbitos cómo podrían ser la ganadería, agricultura, medicina o incluso, la industria.

REFERENCIAS BIBLIOGRÁFICAS

[1] A. R. Quinlan y I. M. Hall, "BEDTools: a flexible suite of utilities for comparing genomic features", *Bioinformatics*, vol. 26, nº 6, pp. 841–842, mar. 2010, doi: 10.1093/bioinformatics/btq033.

[2] R. M. Layer y A. R. Quinlan, "A parallel algorithm for N-way interval set intersection", *Proc. IEEE Inst. Electr. Electron. Eng.*, vol. 105, nº 3, pp. 542–551, mar. 2017, doi: 10.1109/JPROC.2015.2461494.

[3] S. Nurk *et al.*, "The complete sequence of a human genome", *Science*, vol. 376, nº 6588, pp. 44–53, abr. 2022, doi: 10.1126/science.abj6987.

[4] Bentley y Ottmann, "Algorithms for Reporting and Counting Geometric Intersections", *IEEE Trans. Comput.*, vol. C–28, nº 9, pp. 643–647, sep. 1979, doi: 10.1109/TC.1979.1675432.

[5] M. I. Shamos y D. Hoey, "Geometric intersection problems", en *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, Houston, TX, USA, oct. 1976, pp. 208–215. doi: 10.1109/SFCS.1976.16.

[6] J. Barbay y C. Kenyon, "Adaptive Intersection and t-Threshold Problems", sep. 2001, doi: 10.1145/545381.545432.

[7] J. P. Castillo, "Estudio e implementación de algoritmo adaptativo de intersección de intervalos con aplicación a *joins* en bases de datos temporales." julio de 2021.

[8] D. Karolchik *et al.*, "The UCSC Table Browser data retrieval tool", *Nucleic Acids Res.*, vol. 32, nº Database issue, pp. D493-496, ene. 2004, doi: 10.1093/nar/gkh103.

ANEXOS

1.1. GRÁFICAS DE ALGORITMO *SWEEP*

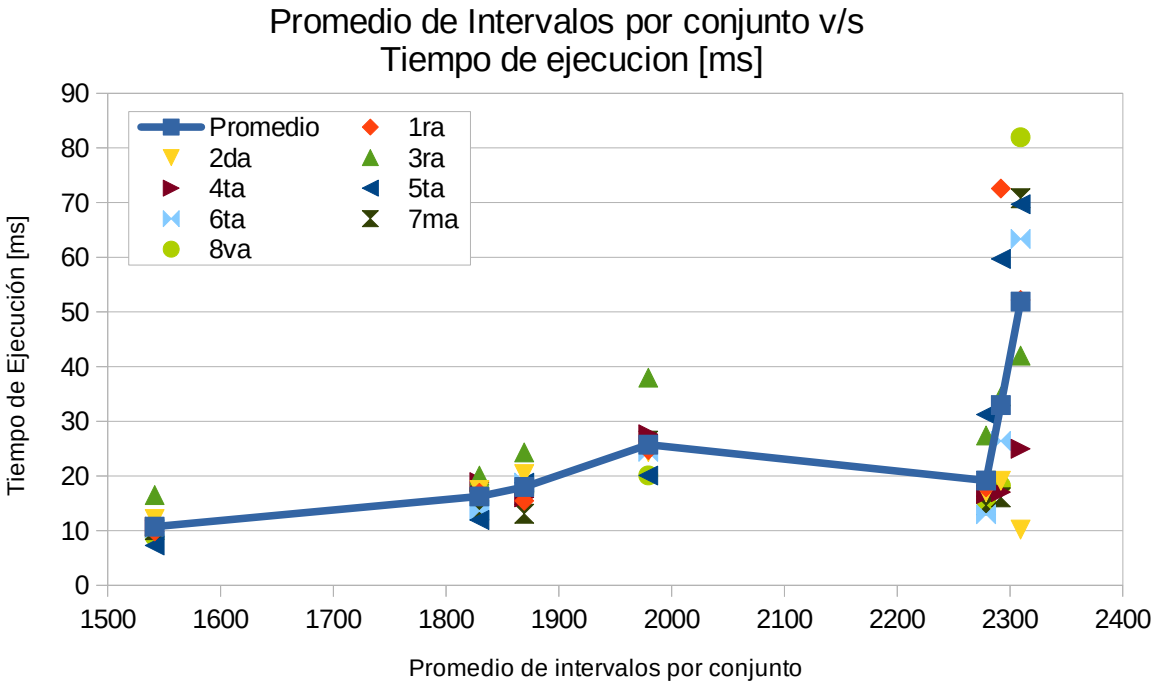

Figura 9: Gráfica de promedio de intervalos vs tiempo de ejecución algoritmo sweep.
Fuente: Origen propio.

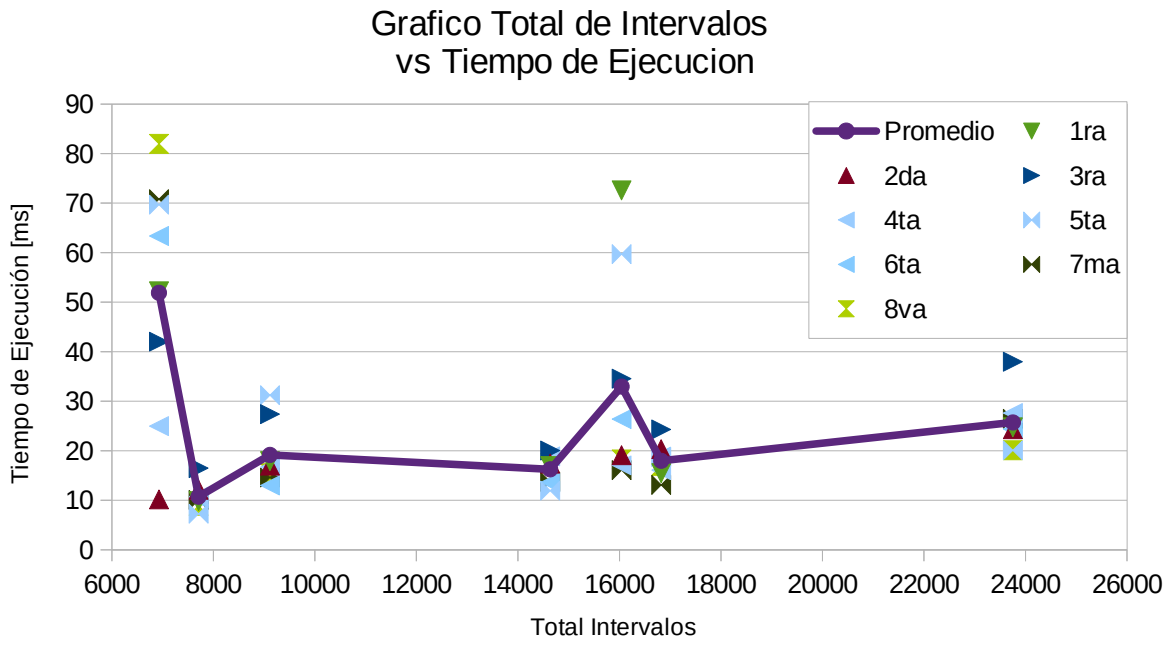

Figura 10: Gráfica de Total de intervalos vs Tiempo de Ejecución algoritmo sweep.
Fuente: Origen Propio

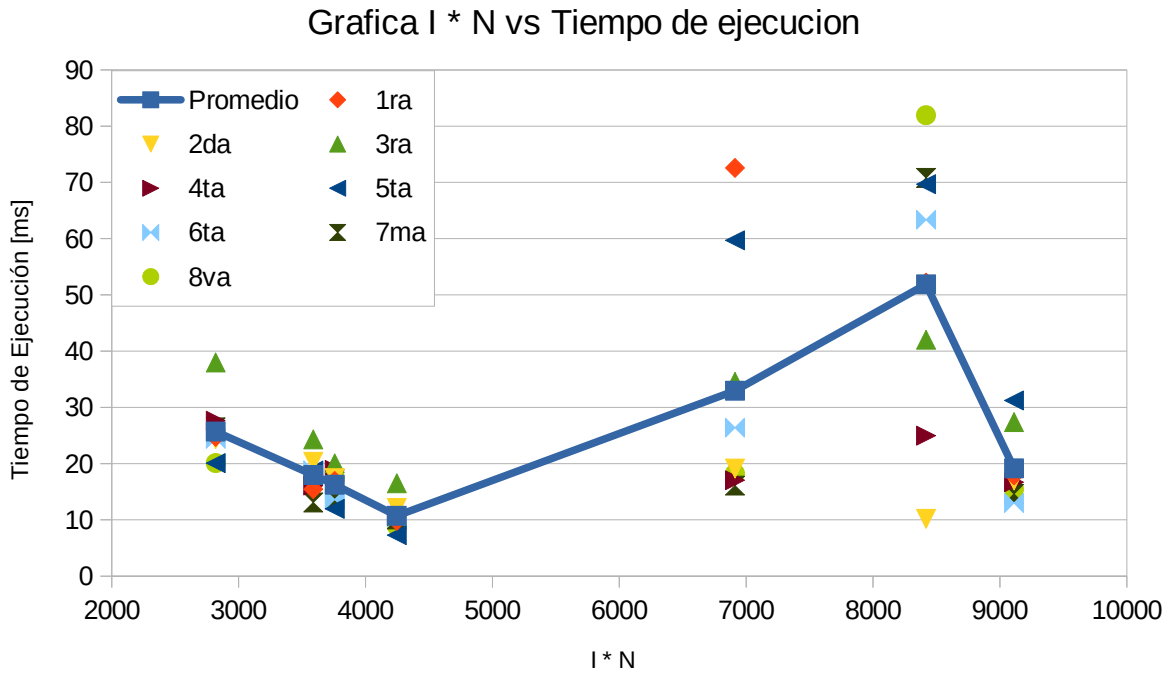

*Figura 11: Gráfica de Producto I*N vs Tiempo de Ejecución algoritmo sweep.*
Fuente: Origen Propio

1.2. GRÁFICAS DE ALGORITMO DE BÚSQUEDA

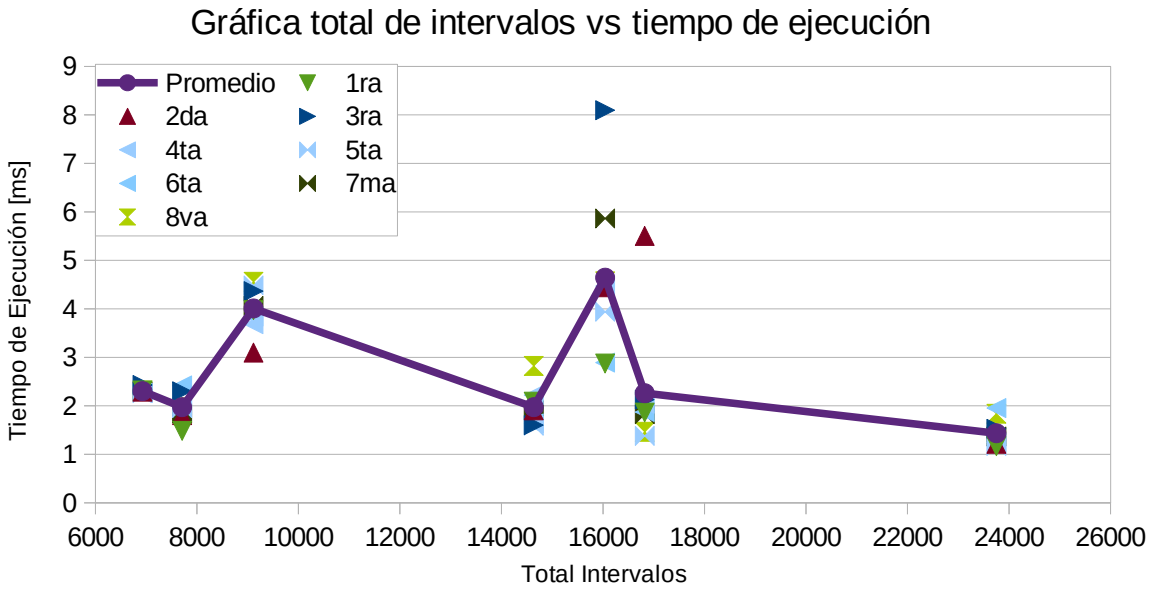

Figura 12: Gráfica total de intervalos vs tiempo de ejecución algoritmo de búsqueda.
Fuente: Origen propio.

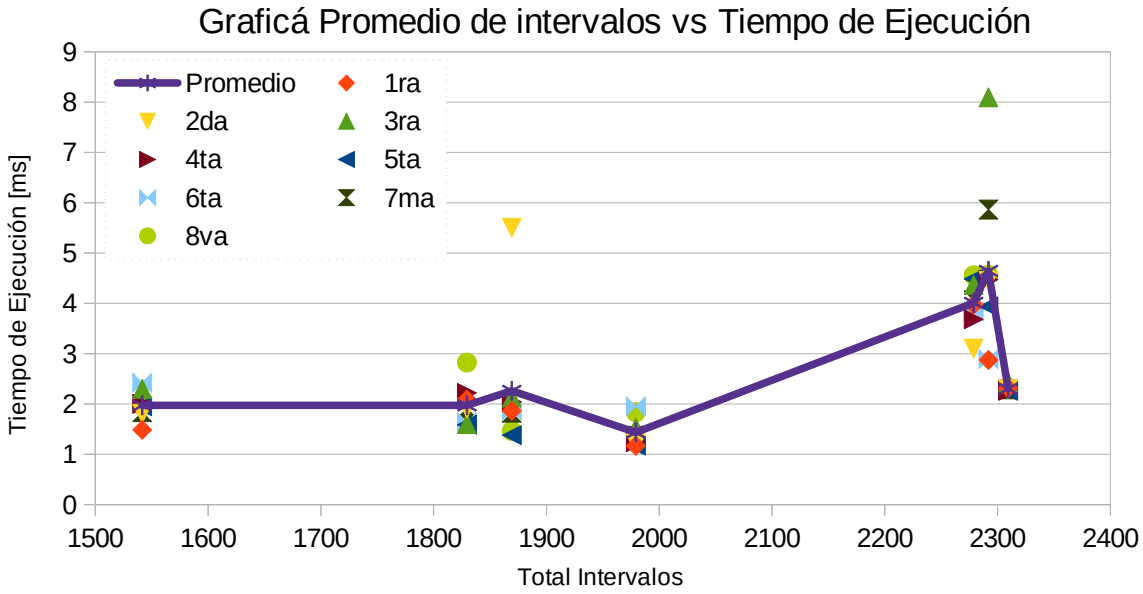

Figura 13: Gráfica total de intervalos vs tiempo de ejecución algoritmo de búsqueda.
Fuente: Origen propio.

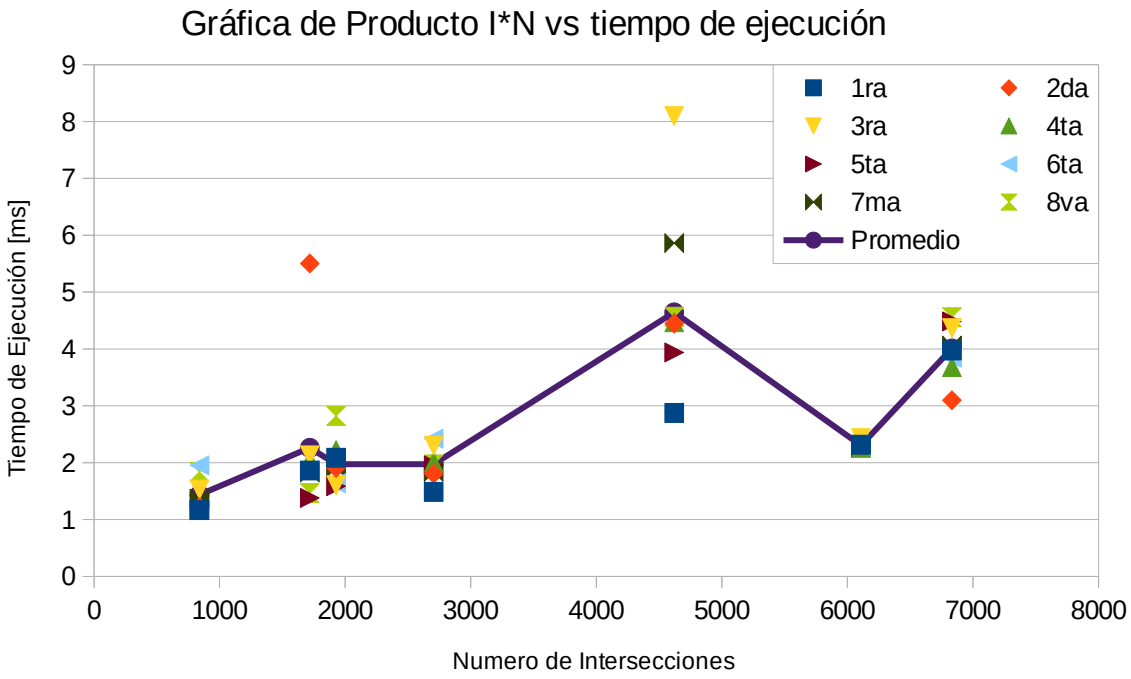

*Figura 14: Gráfica producto I*N vs tiempo de ejecución algoritmo de búsqueda.*
Fuente: Origen propio.