UNIVERSIDAD CENTRAL "MARTA ABREU" DE LAS VILLAS
FACULTAD DE MATEMÁTICA, FÍSICA Y COMPUTACIÓN

Trabajo de diploma

Aplicación de técnicas paralelas utilizando CUDA, al proceso de simulación de poblaciones en secuencias genéticas.

Autor:

Omar Enrique López González

Tutor:

MSc. Leonardo del Toro Melgarejo

Dr. Daniel Galvéz Lío

Santa Clara, 2014

Dictamen

Hago constar que el presente trabajo fue realizado en la Universidad Central "Marta Abreu" de Las Villas como parte de la culminación de los estudios de la especialidad de Ciencias de la Computación, autorizando a que el mismo sea utilizado por la institución, para los fines que estime conveniente, tanto de forma parcial como total y que además no podrá ser presentado en eventos ni publicado sin la autorización de la Universidad.

Firma del autor

Los abajo firmantes, certificamos que el presente trabajo ha sido realizado según acuerdos de la dirección de nuestro centro y el mismo cumple con los requisitos que debe tener un trabajo de esta envergadura referido a la temática señalada.

Firma del tutor

Firma del Jefe del Seminario

Agradecimientos

Quiero expresar mis más sinceros agradecimientos:

A mi tutor Leonardo Flavio del Toro Melgarejo por guiarme durante todo el proceso de investigación que dío como resultado este trabajo.

A todos los amigos que de una forma u otra me ayudaron durante los cinco años de vida universitaria, y compartieron conmigo tanto los buenos momentos como los malos.

A mi hermana, mi mamá y mi papá por el inmenso apoyo que siempre me brindaron, incluso en momentos en los cuales nos separaba una gran lejanía.

Especialmente a Eimy por su gran ayuda en todo momento , por dedicarme incondicionalmente su atención y amor, por apoyarme y animarme a rebasar las circunstancias más difíciles.

Finalmente, a todos los que me han ayudado de cualquier manera a forjarme como un profesional.

Muchas gracias a todos!!!

Resumen

Una de las áreas de la Bioinformática que ha experimentado mayor expansión en los últimos años es la investigación de procesos evolutivos mediante la aplicación de técnicas de la Biología Molecular combinadas con los métodos matemáticos. En este trabajo se exponen los resultados obtenidos a partir de la aplicación de técnicas paralelas utilizando CUDA a la simulación de la evolución de poblaciones virales. Con el propósito de evaluar las tendencias evolutivas de las poblaciones virales, en ausencia y bajo la presión selectiva del sistema inmune, fueron implementadas dos variantes paralelas de un algoritmo de generación de secuencias mutantes en el lenguaje de programación C++. En el trabajo se implementan en paralelo las fases más costosas de los algoritmos para la obtención de nuevas variantes mutacionales y la clasificación de las nuevas secuencias a partir de su correspondiente proteína, estas fueron incorporadas en una aplicación que realiza todo el proceso. Los resultados obtenidos fueron validados utilizando una aplicación secuencial que se desarrolla y se perfecciona a partir de una existente, con la cual también se comparan los tiempos de ejecución, lográndose mejoras sustanciales de estos con el uso de CUDA.

Abstract

One of the fields that have experienced major breakthroughs in the last years in Bioinformatics is the research of evolutionary processes by means of the implementation of techniques used in Molecular Biology which are combined with mathematical methods. This papers aims at displaying the results obtained from the application of parallel techniques using CUDA to the simulation of the evolution of viral populations. In order to assess the evolutionary tendencies of the viral populations, in absence and under the selective pressure of the immune system, two parallel versions of an algorithm of mutant sequences were implemented in the programming language of C++. At the time of the investigation the most expensive phases of the algorithms are implemented in parallel in order to obtain new mutational variants and the classification of new sequences from their corresponding protein; these sequences were incorporated in an application that does the whole process. The obtained results were validated using a sequential application that performs intention, with which the times of execution are also compared, achieving significant progress of these with the use of CUDA.

Tabla de Contenidos

Lista de Figuras

Lista de Algoritmos

Introducción

Una de las áreas de la Bioinformática que ha experimentado mayor expansión en los últimos años es la investigación de procesos evolutivos mediante la aplicación de técnicas de la Biología Molecular combinadas con los métodos matemáticos. Los resultados obtenidos en las investigaciones científicas han demostrado que la simulación es un método muy útil para validar una teoría o una implementación de un programa cuando el análisis del problema es complejo. Cuando el modelo es analíticamente intratable, la simulación provee una manera potente para estudiarlo. La simulación de Monte Carlo, y en particular, el método de Monte Carlo basado en Cadenas de Markov (Markov-Chain Monte Carlo, MCMC), es el fundamento para formar una familia de algoritmos de generación de muestras aleatorias en un tiempo continuo y razonable. De hecho, debido a que el proceso de evolución molecular puede ser tratado como un proceso estocástico, la aplicación de MCMC resulta apropiada para resolver el problema de simulación de la evolución molecular de poblaciones de virus patógenos, un fenómeno confrontado diariamente por el sistema inmune del hombre. En particular, en el área de la medicina es de vital importancia la introducción de tratamientos a los pacientes con nuevos antivirales contra virus tales como el de la influenza A/H1N1 y el VIH, los cuales poseen una velocidad de evolución muy alta. Hasta el presente la compresión de los procesos de evolución molecular es uno de los retos más encumbrados de la biología molecular evolutiva. Desde el punto de vista computacional, las Unidades de Procesamiento Gráfico (GPU por sus siglas en ingles) se han popularizado y masificado en los últimos tiempos, en principio por el auge de juegos en las computadoras y la utilización en estos de cada vez más recursos gráficos que requieren de una alta capacidad de procesamiento. Los programadores utilizan un modelo de programación de la GPU en un lenguaje de alto nivel y una arquitectura de

programación paralela; estos dos elementos conforman lo que NVIDIA denomina Compute Unified Device Architecture (CUDA). Considerando los antecedentes anteriores, se propone la idea de investigar el proceso de evolución de determinadas poblaciones virales de la influenza A mediante la simulación de Monte Carlo. Para este fin se propone el uso de un modelo estocástico evolutivo que permita simular matemáticamente la aparición de nuevas variantes mutacionales en las poblaciones virales a partir de la introducción aleatoria de mutaciones en las secuencias tomadas como ancestros. La implementación del modelo evolutivo se realizará sobre una GPU NVIDIA, utilizando las facilidades que nos brinda la API CUDA.

Antecedentes

Actualmente existen varias aplicaciones que se emplean para el proceso de simulación de poblaciones genéticas basadas en el algoritmo Markov Chain Monte Carlo (MCMC), constituyendo este una herramienta poderosa para la computación bayesiana. Una descripción de este método puede ser consultada en el texto "Computational Molecular Evolution" (Yang, 2006). Algunas de estas aplicaciones han sido concebidas para entornos MatLab, empleando colecciones de herramientas específicas (J.Cai et al., 2006, p.179). Otras herramientas como Evolver (Cai et al., 2006) constituyen una colección de programas diseñados para simular la evolución de la secuencia de nucleótidos de un genoma entero. Evolver es una herramienta potente y compleja, requiriendo de días o semanas de procesamiento en un clúster de computadoras para procesar grandes genomas en distancias evolutivas interesantes.

En el Centro de Estudios Informáticos (CEI), el grupo de Bioinformática aplica y desarrolla diferentes modelos para el análisis del proceso evolutivo en secuencias genéticas de virus. Para ello se emplean varios modelos evolutivos (Tamura and Nei, 1993; Sánchez and Grau, 2009) y el método de Simulación de Monte Carlo. El empleo de estos modelos, y la estimación de los parámetros del mismo partiendo de los datos disponibles, hace posible el empleo de la simulación de Monte Carlo en la predicción de posibles variantes mutacionales de genes con el propósito de evaluar las tendencias evolutivas de las poblaciones virales, en ausencia y bajo la presión selectiva del sistema inmune (Sánchez et al., 2011).

Algunos trabajos del grupo de Bioinformática han experimentado con la implementación en paralelo de algunos modelos evolutivos sobre un clúster de computadoras (Moreira et al., 2011) usando la biblioteca MPI. Estos procesos pueden tener un alto costo computacional si el número de secuencias a procesar constituye un gran volumen de datos por lo que las técnicas de programación paralela contribuyen a tratar este tipo de problemas.

La popularidad de las unidades de procesamiento gráfico (GPU) en los computadores modernos brinda un vasto potencial de paralelismo. La tecnología CUDA (Compute Unified Device Architecture) de NIVIDIA se ha convertido en una poderosa herramienta para el desarrollo de diversas aplicaciones paralelas sobre las GPU, constituyendo actualmente un campo en

desarrollo e investigación. El empleo de esta tecnología ya se ha aplicado exitosamente en trabajos recientes que emplean el algoritmo MCMC en procesos de optimización multi-objetivo (Minh, 2010) y también para el análisis filogenético estadístico (Suchard and Rambaut, 2009). Por otro lado, el grupo de Programación e Ingeniería del Software ha empleado la tecnología CUDA (*NVIDIA CUDA C Programming Guide 5.0*, 2012) para explotar sus beneficios en trabajos previos aplicados a la bioinformática (Alejo, 2012; Alba, 2013).

Planteamiento del problema

El proceso de simulación, usando técnicas estadísticas y el algoritmo MCMC, para la predicción de posibles variantes mutacionales en determinado gen al evaluar las tendencias evolutivas de las poblaciones virales, puede resultar en un problema con una alta complejidad computacional, que se incrementa con el número de secuencias a procesar. En la actualidad no se ha desarrollado ningún algoritmo paralelo para tratar esta problemática cuya implementación contribuya a disminuir la complejidad inherente del algoritmo y acelere el tiempo de ejecución de la misma.

A partir de la problemática descrita se plantean las siguientes **preguntas investigación**:

1. ¿Cuáles son los segmentos con mayor costo computacional en el algoritmo general empleado para la simulación de poblaciones virales mutantes?.
2. ¿Cómo implementar de manera paralela los segmentos anteriores?

Para conducir el proceso investigativo expuesto previamente se plantean los siguientes objetivos.

Objetivo general

Implementar algoritmos paralelos en diferentes etapas del proceso para la simulación de variantes mutacionales usando el modelo evolutivo TN93 y el algoritmo MCMC, para evaluar tendencias evolutivas en poblaciones virales, utilizando la tecnología CUDA.

Objetivos específicos

- Describir el proceso de simulación de variantes mutacionales considerando, o descartando la presión del sistema inmune.
- Caracterizar los rasgos del modelo de programación en CUDA y su aplicación en la bioinformática.
- Desarrollar algoritmos paralelos para:
  - La simulación de nuevas secuencias genéticas mutantes, partiendo de secuencias ancestrales.
  - Clasificación de las nuevas secuencias a partir de las proteínas sintetizadas por las mismas.

- Obtener una aplicación que implemente en CUDA estos algoritmos.
- Analizar los resultados obtenidos por los algoritmos implementados, comparando sus tiempos de ejecución con el de sus versiones secuenciales.

Justificación del trabajo

La aplicación que se pretende desarrollar, para ejecutarse sobre la GPU, sería de gran importancia para el grupo de investigación Bioinformática de la UCLV, como parte de un proyecto nacional del Ministerio de Ciencia, Tecnología y Medio Ambiente (CITMA), que lleva a cabo investigaciones sobre la predicción de mutaciones y la resistencia antiviral de virus como el VIH, el virus H5N1, H1N1 y otros.

La aplicación de técnicas paralelas a los problemas de la Bio-Informática y Biología Computacional, como una tendencia a reducir los costos en la solución de problemas reales, es importante en Cuba, tanto científica como económicamente, considerado en el proyecto 01700032 del Programa Nacional Científico Técnico de Tecnologías de la Información.

Capítulo 1

Fundamentos Biológicos, Matemáticos y Computacionales

En este capítulo se realiza una descripción de las principales bases teóricas del trabajo, pues dado el hecho de que las aplicaciones procesarán secuencias de ADN genómico y que, además, a lo largo de este trabajo se aplican una gran variedad de herramientas matemáticas y computacionales incluyendo el uso de la tecnología CUDA, se hace necesario realizar un despliegue acertado de estos conocimientos antes de hacer uso de los mismos.

1.1 Introducción a los elementos biológicos

Se presentan en esta sección algunas ideas básicas de biología molecular y algunos de los resultados anteriores relacionados con este trabajo. Ellos son imprescindibles para la comprensión de las hipótesis de investigación y los capítulos siguientes.

1.1.1 Las moléculas básicas de la vida

Los principales actores de la Biología Molecular son las moléculas del ADN, ARN y las proteínas. Estas moléculas son simultáneamente los principales actores de un poderoso

sistema de comunicación, el sistema de información genética, entrelazados por el código de comunicación conocido como Código Genético. La expresión de la información almacenada en el ácido desoxirribonucleico (ADN) se produce a través de la trascripción lineal de la secuencia de nucleótidos en la secuencia de nucleótidos del ácido ribonucleico mensajero (ARNm). La secuencia de nucleótidos del ARNm es seguidamente traducida en una secuencia lineal de aminoácidos y a partir de estos se forman las proteínas, de manera que el flujo de información es ADN → ARNm → Proteína (Minh, 2010).

1.1.2 ADN y ARN

Desde el punto de vista químico, el ADN es un polímero de nucleótidos, es decir, un polinucleótido. Un polímero es un compuesto formado por muchas unidades simples conectadas entre sí. En el ADN, cada unidad simple es denominada nucleótido, donde cada nucleótido está formado por un azúcar (la desoxirribosa), una base nitrogenada (que puede ser adenina→A, timina→T, citosina→C o guanina→G) y un grupo fosfato que actúa como enlace de cada nucleótido con el siguiente. Lo que distingue a un nucleótido de otro es, entonces, la base nitrogenada, y por ello la secuencia del ADN se especifica nombrando solo la secuencia de sus bases. La disposición secuencial de estas cuatro bases a lo largo de la cadena es la que codifica la información genética: por ejemplo, una secuencia de ADN puede ser ATGCTAGATCGC...(Alba, 2013). En la Fig. 1.1 se observa la estructura del ADN.

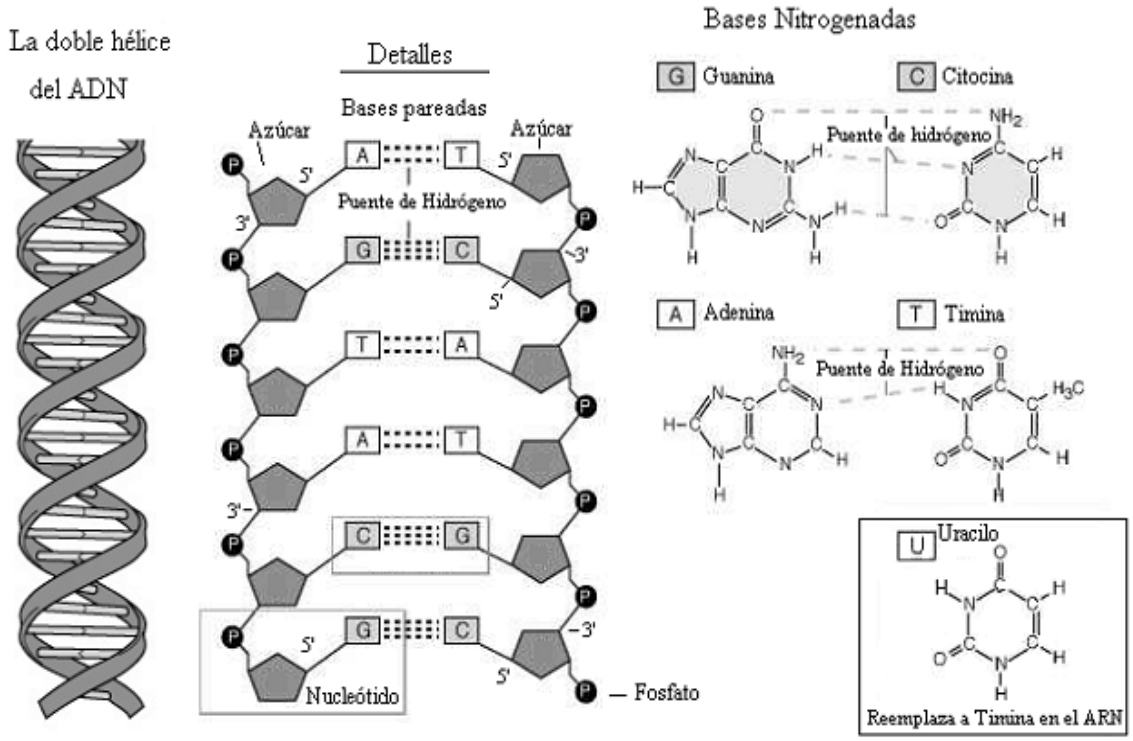

Figura 1.1: La doble hélice del ADN.

Las bases nitrogenadas se aparean en la doble hélice formando enlaces por puentes de hidrógeno de acuerdo a las siguientes reglas: G≡C, A = T, donde cada – simboliza un enlace por puente de hidrógeno y cada par contiene una purina (A o G) y una pirimidina (C o T).

El ARN al igual que el ADN está formado por una cadena de nucleótidos unidos por enlaces fosfodiester. En el proceso de transformación de ADN a ARN tenemos que la base nitrogenada T es sustituida por la base uracilo (U) y se mantienen las otras bases nucleotídicas: A, C y G como en el ADN. La función del ARN es transportar hacia el citoplasma la información genética que se encuentra almacenada en el núcleo de la célula en la secuencia de nucleótidos del ADN. Cuando se traten secuencias de ADN codificantes para proteínas, estas se pueden escribir en la práctica utilizando las bases T o U indistintamente. Existen varios tipos de ARN, cada uno con una función distinta. A los que forman parte de las subunidades de los ribosomas se les denomina ARN ribosomal (rARN); los ARN que tienen la función de transportar los aminoácidos activados, desde el citosol hasta el lugar de síntesis de proteínas en los ribosomas, se les conoce por ARN de transferencia (tARN) y los ARN que son portadores de la información genética y la transportan del genoma (molécula de ADN en el cromosoma) a los ribosomas son llamados ARN mensajero (mARN) (Minh, 2010). De los tipos de ARN mencionados anteriormente al que más se hace referencia en este trabajo es al ARNm debido a que son los portadores de la información genética.

1.1.3 Los aminoácidos y el código genético

Los aminoácidos son los pilares estructurales de las proteínas. Solo veinte aminoácidos se encuentran corrientemente presentes en las proteínas, aunque muchos otros aminoácidos desempeñan múltiples funciones en las células. La codificación de cada aminoácido se realiza a través de un grupo de tres nucleótidos comúnmente conocidos como codones. Si combinamos tres bases (tripletes) para formar un aminoácido, obtenemos un total de 64 combinaciones ($4^3 = 64$). Los científicos demostraron que hay 61 tripletes -o codones- que codifican aminoácidos, muchos de los cuales son codificados por más de un codón, por lo que se dice que el código está degenerado. Los distintos aminoácidos son codificados por un número diferente de codones (algunos por 1, otros por 2, o por 3), e incluso existen tres tripletes que no codifican para ningún aminoácido. Encontramos aminoácidos como Leucina, Serina y Arginina, cada uno de los cuales posee 6 codones que codifican para el mismo, mientras que otros como el Triptófano y la Metionina solo poseen un solo codón Fig. 1.2. El codón de los aminoácidos Metionina es usualmente utilizado en la mayoría de los seres vivos como codón de iniciación de la cadena polipeptídica. En resumen: de los 64 codones, 61 codifican aminoácidos y los tres restantes no son codificantes sino que son utilizados como señales de terminación y son llamados codones de parada (*stop codons*). De la

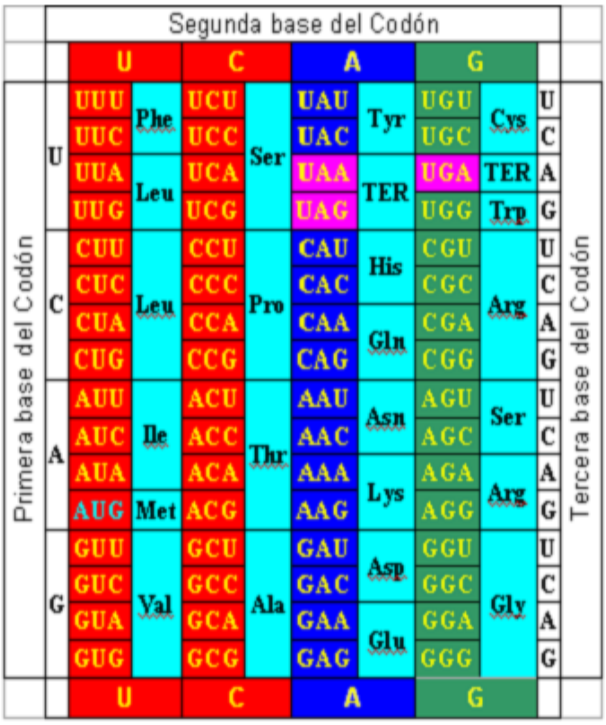

Figura 1.2: Código genético estándar. Los aminoácidos están escritos con el símbolo de tres letras. El codón utilizado con mayor frecuencia como codón de inicio de la transcripción corresponde al aminoácido Metionina: AUG. Los codones UAA, UAG y UGA son marcadores del final de los genes.

observación del Código genético se destaca que la degeneración del código implica solamente a la tercera posición del codón en la mayoría de los casos (son excepciones la Arginina, la Leucina y la Serina), de esta forma resulta que las dos primeras bases de cada codón son las determinantes principales de su especificidad. La posición tercera, esto es, el nucleótido situado en el extremo 3 del codón, tiene menor importancia y no encaja con tanta precisión (Minh, 2010).

1.2 Procedimiento evolutivo de las poblaciones virales

El estudio científico de la evolución empezó desde un artículo publicado por Darwin y Wallace en 1858. Primero, ellos postularon que la evolución ha ocurrido principalmente como resultado de la selección natural (Nei, 1975). Esta última es efectiva solo cuando existe la variación genética, y esta variedad genética es provista primeramente por la mutación de la información genética llevada en las moléculas de ADN, las cuales son transmitidas de generación a generación. En algunos virus, la información genética es llevada en las moléculas ARN, pero la característica esencial de la herencia es la misma. En el proceso de desarrollo, la información genética contenida en la secuencia de nucleótidos ADN, es

transferida primeramente a la secuencia de nucleótidos de ARNm por un proceso simple de transcripción, uno a uno, de los nucleótidos en la molécula ADN. Por el mismo proceso, el tARN y el rARN son producidos. La información transferida al ARNm determina la secuencia de aminoácidos de la proteína que será sintetizada. Los nucleótidos de ARNm son leídos secuencialmente 3 cada vez. Cada triplete o codón es traducido en un aminoácido en la cadena de proteína mediante el código genético.

Cualquiera de las mutaciones que son reconocidas como cambios morfológicos o fisiológicos tiene que ser debido a algún cambio de las moléculas ADN. Existen cuatro tipos básicos de cambios en las moléculas ADN: reemplazo de un nucleótido por el otro, eliminación de nucleótidos, adición de nucleótidos, e inversión de nucleótidos. La adición, la eliminación y la inversión pueden ocurrir con uno o más nucleótidos como una unidad. Los reemplazos de los nucleótidos pueden ser divididos en dos clases diferentes: la transición y la transversión. Transición es el reemplazo de una purina (adenina A o guanina G) por otra purina, o de una pirimidina (timina T o citosina C) por otra pirimidina. Los otros tipos de sustituciones de nucleótidos donde el cambio ocurre entre purinas y pirimidinas son llamados transversiones (Nei, 1975).

Los genes o segmentos de las moléculas de ADN que actúan como las plantillas del mARN son llamados genes estructurales. Puesto que la secuencia de aminoácidos es determinada por la secuencia de nucleótidos de un gen estructural, cualquier cambio en las secuencias de aminoácidos es causado por la mutación ocurrida en la molécula ADN. Sin embargo, en el sentido opuesto, un cambio de mutación en la molécula ADN no es necesariamente reflejado en el cambio de la secuencia de aminoácidos. Esto es porque hay degeneración en el código genético (Nei, 1975).

1.3 Simulación de la evolución molecular

La simulación de Monte Carlo ha sido utilizada comúnmente en los estudios filogenéticos para probar diferentes métodos de reconstrucción de árboles filogenéticos y, consecuentemente, sus aplicaciones para probar los modelos evolutivos pueden ser consideradas como una extensión natural de este uso. La simulación repetitiva de un proceso evolutivo dado, bajo las restricciones impuestas por el modelo que se prueba, permite estimar las distribuciones de probabilidad para los parámetros deseados. A continuación, el árbol filogenético se puede reconstruir de nuevo sin las restricciones del modelo, y los parámetros de interés derivados de este árbol pueden ser comparados con la distribución de probabilidad correspondiente derivada del árbol restringido y simulado (Yang, 2006).

1.3.1 Cadenas de Markov

El método utilizado en este trabajo para llevar a cabo la simulación de las secuencias de ADN se basa en las cadenas de Markov (CM). Según (Bertsekas and Tsitsiklis, 2000) estas cadenas definen un tipo especial de proceso estocástico discreto o continuo en el que la probabilidad de que ocurra un evento depende del evento inmediatamente anterior. En efecto, las cadenas de este tipo tienen memoria, "recuerdan" el último evento y esto condiciona las posibilidades de los eventos futuros. Primeramente debemos decir que en todo momento las CM poseen un estado que denotaremos por $\boldsymbol{x_n}$, el cual indica el estado actual en el que se encuentra la cadena en el momento $\boldsymbol{n}$ y el mismo pertenece a un conjunto de posibles estados $\boldsymbol{S}$ que se denomina espacio de estados:

$$\boldsymbol{S = \{1, 2, \ldots, m\} \mid m \in \mathcal{Z}, m > 0} \tag{1.1}$$

Las CM se describen en términos de sus probabilidades de transición $\boldsymbol{p_{ij}}$ las cuales indican la probabilidad que existe de ir del estado $\boldsymbol{x_i}$ al estado $\boldsymbol{x_j}$. Como se ha dicho anteriormente que la memoria de una cadena solo se refiere al último evento que ha ocurrido en la misma, entonces las probabilidades de transición matemáticamente nos quedan definidas de la siguiente forma:

$$\boldsymbol{P(x_{n+1} = j \mid x_n = i_n, \ldots, x_0 = i_0) = p_{ij} = P(x_{n+1} = j \mid x_n = i)} \tag{1.2}$$

$$\boldsymbol{\forall n \mid n \geq 0, \forall i, j \in S}$$

También existen autores que se refieren a las cadenas de Markov como una cadena de variables aleatorias $\boldsymbol{\{x_0, x_1, \ldots, x_n\}}$ que pueden tomar valores del conjunto de estados $\boldsymbol{S}$ y a su vez satisfacen la probabilidad condicional anterior. El presente trabajo se desarrolla completamente con modelos donde el efecto del pasado en el futuro se resume a un estado, el cual cambia en el tiempo de acuerdo con probabilidades dadas. Nuestros modelos pueden tomar un número finito de estados y los cambios de estados se producen en instantes de tiempo discretos.

La suma de todas las probabilidades de transición del estado $\boldsymbol{i}$ hacia todos los demás deben sumar $\boldsymbol{1}$, por tanto:

$$\boldsymbol{\sum_{i=1}^{m} p_{ij} = 1, \forall i \in S} \tag{1.3}$$

es importante también destacar los vectores de probabilidad asociados a una cadena, donde tenemos que $\boldsymbol{\pi_j(t)}$ denota la probabilidad de que la cadena esté en el estado $\boldsymbol{j}$ en el instante de tiempo $\boldsymbol{t}$, por tanto:

$$\boldsymbol{\pi_j(t) = P(x_t = j)} \tag{1.4}$$

y $\boldsymbol{\pi(t)}$ denota un vector fila de probabilidades del espacio de estados en el tiempo $\boldsymbol{t}$, que tiene en su i-esima componente la probabilidad de que en el instante de tiempo $\boldsymbol{t}$ nos encontremos en el estado $\boldsymbol{i}$ del conjunto de estados $\boldsymbol{S}$. El vector $\boldsymbol{\pi(0)}$ está asociado con el inicio de la cadena y todas sus componentes serán iguales a $\mathbf{0}$ excepto la componente relativa al estado inicial de la cadena cuyo valor será igual a $\mathbf{1}$.

1.3.2 Matriz de probabilidades y probabilidad de transición en n pasos

Todos los elementos de una CM pueden definirse a través de una matriz de transición de $\boldsymbol{m}$ dimensiones y el elemento de su i-esima fila y su j-esima columna es $\boldsymbol{p_{ij}}$. Esta matriz también es posible representarla mediante un grafo de transición de probabilidad donde los nodos son los estados y las aristas son las probabilidades de transición existentes entre dichos estados. Varios problemas de las CM requieren calcular la ley de probabilidad de un estado $\boldsymbol{j}$ en un tiempo futuro teniendo como condición de que nos encontramos en el estado $\boldsymbol{i}$; es decir, se desea conocer la probabilidad de encontrarnos en $\boldsymbol{j}$ después de recorrer $\boldsymbol{n}$ estados y teniendo a $\boldsymbol{i}$ como punto de partida. Esta ley de probabilidad se define como:

$$\boldsymbol{r_{ij} = P(x_n = j \mid x_0 = i)} \tag{1.5}$$

este valor de $\boldsymbol{r_{ij}}$ se puede calcular por la siguiente recursión conocida como ecuación de Chapman-Kolmogorov:

$$\left.\begin{aligned} \boldsymbol{r_{ij}(n)} &= \boldsymbol{\sum_{k=1}^{m} r_{ik}(n-1)p_{kj} \quad n > 1, \forall i, j \in S} \\ \boldsymbol{r_{ij}(1)} &= \boldsymbol{p_{ij}} \end{aligned}\right\} \tag{1.6}$$

de las ecuaciones anteriores es posible demostrar que el cálculo de las probabilidades de transición en $\boldsymbol{n}$ pasos se puede ejecutar en función de la matriz de probabilidades a partir de

la expresión siguiente:

$$\boldsymbol{\pi(t+1) = \pi(t)M} \tag{1.7}$$

generalizando la ecuación anterior obtenemos que:

$$\boldsymbol{\pi(t) = \pi(0)M^t} \tag{1.8}$$

siendo $\boldsymbol{M}$ la matriz de probabilidades de transición. Algunas CM poseen una distribución estacionaria constituida por un vector fila de probabilidades denotado por $\boldsymbol{\pi^*}$ tal que:

$$\boldsymbol{\pi^* = \pi^* M} \tag{1.9}$$

es decir, cuando se alcanza esta distribución no importa cuántas veces multipliquemos la matriz de transición al vector estacionario, lo cual es equivalente a avanzar un instante en la cadena. Las probabilidades de estancia en cada uno de los estados no se modifican en el tiempo y se mantienen constantes. Las condiciones para que una cadena posea una distribución estacionaria es que la misma sea irreducible y aperiódica. (Bertsekas and Tsitsiklis, 2000)

1.3.3 Técnica Markov Chain Monte Carlo

Los métodos de Markov Chain Monte Carlo, o simplemente métodos MCMC, frecuentemente se aplican para resolver los problemas de integración y optimización en los espacios de dimensión grande. En la estadística computacional estos métodos son muy útiles para generar las variables de una distribución objetivo, basándose en las cadenas de Markov, cuya distribución estacionaria es la distribución de probabilidad de interés. La razón central por la cual se usan ampliamente los métodos MCMC es que estos métodos son extremadamente generales y pueden ser utilizados para generar distribuciones univariables y multivariables cuando otros métodos han fallado o son difíciles de implementar (Gentle et al., 2004). El algoritmo utilizado en este trabajo para construir las CM obtenidas a partir de las mutaciones es una variante del método Metropolis el cual constituye un método MCMC. Mediante Metropolis, podemos generar muestras aleatorias de forma tal que estas se ajusten a una distribución de probabilidad deseada $\boldsymbol{p(x)}$ teniendo como restricción que es necesario conocer

una función $\boldsymbol{f(x)}$ proporcional a la función de densidad de $\boldsymbol{p}$, es decir:

$$\boldsymbol{p(x) = \frac{f(x)}{k}} \tag{1.10}$$

donde $\boldsymbol{k}$ será un factor de normalización y no es necesario conocerlo para aplicar el algoritmo, lo cual es muy conveniente debido al hecho de que generalmente el cálculo del mismo es muy complejo. Estas muestras se producen iterativamente de forma tal que en cada paso el nuevo valor solo dependerá del valor actual, por lo cual nos encontramos con un proceso estocástico de Markov. Luego que tenemos un posible nuevo candidato se verifica si se toma el mismo como nuevo estado. La probabilidad de aceptación de este se calcula realizando una comparación entre los ajustes del estado actual y el nuevo estado a la distribución $\boldsymbol{p(x)}$ que se quiere estimar, la cual se denomina distribución objetivo. Para calcular cuánto se ajusta un valor determinado a esta distribución se hace uso de la distribución $\boldsymbol{f}$. A continuación se describen formalmente los pasos en los que consiste el algoritmo:

1. Comenzamos con un valor inicial $\boldsymbol{\theta_0}$ tal que satisfaga $\boldsymbol{f(\theta_0) > 0}$.

2. Utilizamos el valor actual de $\boldsymbol{\theta_i}$ para muestrear un nuevo candidato $\boldsymbol{\theta_{i+1}}$, luego a partir de una distribución $\boldsymbol{q(\theta_i, \theta_{i+1})}$ obtenemos la probabilidad de obtener $\boldsymbol{\theta_{i+1}}$ teniendo a $\boldsymbol{\theta_i}$ como valor previo. En la literatura $\boldsymbol{q}$ es comúnmente referida como distribución propuesta o de generación de candidatos. La única restricción que debe cumplir $\boldsymbol{q}$ es que sea simétrica, por tanto $\boldsymbol{q(\theta_i, \theta_{i+1}) = q(\theta_{i+1}, \theta_i)}$.

3. Teniendo la nueva muestra propuesta $\boldsymbol{\theta_{i+1}}$ calculamos la proporción $\boldsymbol{\alpha}$ existente entre la densidad de esta y del estado actual $\boldsymbol{\theta_i}$ tomando como distribución a $\boldsymbol{p}$, entonces:

$$\boldsymbol{\alpha = \frac{p(\theta_{i+1})}{p(\theta_i)} = \frac{f(\theta_{i+1})}{f(\theta_i)}} \tag{1.11}$$

   notemos que en la fórmula 1.11 podemos hacer uso de la función de densidad $\boldsymbol{f}$ debido a que estamos hallando la proporción entre $\boldsymbol{2}$ valores y por tanto la constante de normalización $\boldsymbol{k}$ se cancela durante el procedimiento.

4. Si la transición propuesta incrementa la densidad $\boldsymbol{(\alpha > 1)}$ entonces aceptamos el valor candidato y volvemos al paso $\boldsymbol{2}$. En caso de que la densidad disminuya $\boldsymbol{(\alpha < 1)}$ aceptamos a $\boldsymbol{\theta_{i+1}}$ con probabilidad $\boldsymbol{\alpha}$; es decir, generamos un número aleatorio a partir de $\boldsymbol{U(0, 1)}$ y si es menor que $\boldsymbol{\alpha}$ se acepta el nuevo estado. Si ocurre un rechazo entonces $\boldsymbol{\theta_{i+1} = \theta_i}$ y luego retornamos al paso $\boldsymbol{2}$.

A menudo en algunas exposiciones del algoritmo anterior se resume el cálculo de $\boldsymbol{\alpha}$ mediante

la siguiente ecuación:

$$\boldsymbol{\alpha = \min \left\{ \frac{f(\theta_{i+1})}{f(\theta_i)}, 1 \right\}} \tag{1.12}$$

y luego se acepta el candidato con $\boldsymbol{\alpha}$ (probabilidad de que ocurra el movimiento). De esta forma se genera una cadena de Markov $\boldsymbol{\theta_0, \theta_1, \ldots, \theta_n}$, la que después de un número finito de iteraciones alcanza su distribución estacionaria y los valores del vector $\boldsymbol{\theta_{k+1}, \theta_{k+2}, \ldots, \theta_{k+n}}$ son muestras de $\boldsymbol{p(x)}$ (Walsh, 2004).

Específicamente en este trabajo se utiliza una variante realizada por Hastings (1970) del método expuesto anteriormente conocida como **Metropolis-Hastings**. Lo que se plantea en esta modificación es que el algoritmo utiliza una función de probabilidad de transición $\boldsymbol{q}$ arbitraria no simétrica y la probabilidad de cambio se halla aplicando la siguiente ecuación:

$$\boldsymbol{\alpha = \min \left\{ \frac{f(\theta_{i+1}) q(\theta_{i+1}, \theta_i)}{f(\theta_i) q(\theta_i, \theta_{i+1})}, 1 \right\}} \tag{1.13}$$

1.3.4 Modelos Evolutivos

Los modelos evolutivos en filogenias moleculares describen el modo y la probabilidad de que una secuencia de nucleótidos cambie a otra secuencia de nucleótidos homóloga a lo largo del tiempo. Es decir, estos modelos describen para cada uno de los sitios de la matriz la probabilidad de que se produzca el cambio de un nucleótido a otro a lo largo de las ramas de un árbol filogenético dado.

Los modelos de evolución de nucleótidos se definen matemáticamente mediante dos clases de parámetros que determinan el cambio:

1. Frecuencia de cada nucleótido. Parámetro que mide la frecuencia de cada nucleótido en la matriz de datos y puede tomar los valores siguientes:

   (a) En los modelos evolutivos más sencillos se tiene una misma frecuencia para los cuatro nucleótidos sin tener en cuenta la frecuencia de aparición de los mismos en la matriz de datos. $\boldsymbol{(\pi_A = \pi_C = \pi_G = \pi_T = 0.25)}$.

   (b) Los modelos más complejos asumen que las frecuencias asociadas a cada uno de los nucleótidos son diferentes, y son calculadas a partir de los datos.
   $\boldsymbol{(\pi_A \neq \pi_C \neq \pi_G \neq \pi_T)}$.

2. Tipos de sustituciones y sus correspondientes tasas de sustitución (***rate parameters***) las cuales se representan a partir de una matriz de tasas de sustitución. Las tasas de sustitución se representan con las tasas relativas de cambio de un nucleótido a otro para una posición de un tiempo $\boldsymbol{t_0}$ a un tiempo $\boldsymbol{t_1}$. Así, cada posición de la matriz tendrá una probabilidad asociada de cambio para cada unidad de tiempo (unidad de distancia evolutiva). Los modelos más sencillos asumen una misma tasa relativa para todas las sustituciones posibles, mientras que los más complicados asumen una tasa relativa diferente para cada tipo de sustitución. A partir de estas tasas relativas se calcula la tasa media de sustitución ($\boldsymbol{\mu}$).

1.3.5 Modelo TN93

El modelo evolutivo utilizado en este trabajo fue propuesto por Tamura y Nei (1993) y es comúnmente conocido por TN93, formando parte de la familia de modelos GTR ***(General Time-Reversible)***. Este contiene varios parámetros para lograr una simulación más realista del comportamiento de una secuencia de nucleótidos, ausmiendo que las tasas de sustitución difieren entre los nucleótidos y las frecuencias de las bases se toman como diferentes. Además, los intercambios entre 2 pirimidinas y 2 purinas se presentan en este modelo como formas de sustitución distintas; por tanto, contamos con un parámetro adicional para cada uno de estos cambios con el objetivo de lograr cierta distinción entre uno y otro; estos parámetros son $\boldsymbol{\alpha_1}$ y $\boldsymbol{\alpha_2}$, los cuales son valores reales entre $\mathbf{0}$ y $\mathbf{1}$. La matriz $\boldsymbol{Q}$ de tasas de sustitución propuesta por este modelo es la siguiente:

$$\boldsymbol{Q} = \begin{bmatrix} -(\alpha_1\pi_C + \beta\pi_R) & \alpha_1\pi_C & \beta\pi_A & \beta\pi_G \\ \alpha_1\pi_T & -(\alpha_1\pi_T + \beta\pi_R) & \beta\pi_A & \beta\pi_G \\ \beta\pi_T & \beta\pi_C & -(\alpha_2\pi_G + \beta\pi_Y) & \alpha_2\pi_G \\ \beta\pi_T & \beta\pi_C & \alpha_2\pi_A & -(\alpha_2\pi_A + \beta\pi_Y) \end{bmatrix} \tag{1.14}$$

Aquí tenemos que la columna 1 representa un cambio partiendo de una adenina a cualquiera de las otras bases y así sucesivamente. Esta matriz es utilizada para calcular la matriz de probabilidades de transición $\boldsymbol{P}$ utilizada por la técnica MCMC, esto se abordará con mayor profundidad en el capítulo 2 (Yang, 2006).

1.4 Modelo de regresión lineal simple, herramienta estadística para el análisis relacional de datos

Con frecuencia, nos encontramos en economía con modelos en los que el comportamiento de una variable, $\boldsymbol{Y}$, se puede explicar a través de una variable $\boldsymbol{X}$; lo que representamos mediante:

$$\boldsymbol{Y} = \boldsymbol{f}(\boldsymbol{X}) \tag{1.15}$$

Si consideramos que la relación $\boldsymbol{f}$, que liga $\boldsymbol{Y}$ con $\boldsymbol{X}$, es lineal, entonces 1.15 se puede escribir así:

$$\boldsymbol{Y_t} = \boldsymbol{\beta_1} + \boldsymbol{\beta_2 X_t} \tag{1.16}$$

Las relaciones del tipo anterior raramente son exactas, sino que más bien son aproximaciones en las que se han omitido muchas variables de importancia secundaria. Debemos incluir un término de perturbación aleatoria, $\boldsymbol{u_t}$, que refleja todos los factores – distintos de $\boldsymbol{X}$ – que influyen sobre la variable endógena, pero que ninguno de ellos es relevante individualmente. Con ello, la relación quedaría de la siguiente forma:

$$\boldsymbol{Y_t} = \boldsymbol{\beta_1} + \boldsymbol{\beta_2 X_t} + \boldsymbol{u_t} \tag{1.17}$$

La expresión anterior refleja una relación lineal que nos permite explicar el comportamiento de una variable $\boldsymbol{Y}$ denominada variable explicada (o dependiente), a partir de otra variable $\boldsymbol{X}$ que llamaremos variable explicativa (o independiente), recibiendo el nombre de regresión lineal simple (Spiegel et al., 2009).

1.4.1 Construcción del modelo de regresión lineal mediante la obtención de los estimadores

Supongamos ahora que disponemos de $\boldsymbol{T}$ observaciones de la variable $\boldsymbol{Y(Y_1, Y_2, \ldots, Y_T)}$ y de las correspondientes observaciones de $\boldsymbol{X(X_1, X_2, \ldots, X_T)}$. Si hacemos extensiva 1.17 a

la relación entre observaciones, tendremos el siguiente conjunto de $\boldsymbol{T}$ ecuaciones:

$$\left.\begin{aligned} \boldsymbol{Y_1} &= \boldsymbol{\beta_1} + \boldsymbol{\beta_2 X_1} + \boldsymbol{u_1} \\ \boldsymbol{Y_2} &= \boldsymbol{\beta_1} + \boldsymbol{\beta_2 X_2} + \boldsymbol{u_2} \\ &\cdots\cdots\cdots\cdots\cdots \\ \boldsymbol{Y_T} &= \boldsymbol{\beta_1} + \boldsymbol{\beta_2 X_T} + \boldsymbol{u_T} \end{aligned}\right\} \tag{1.18}$$

El sistema de ecuaciones 1.18 se puede escribir abreviadamente de la forma siguiente:

$$\boldsymbol{Y_t} = \boldsymbol{\beta_1} + \boldsymbol{\beta_2 X_t} + \boldsymbol{u_t} \quad \boldsymbol{t = 1, 2, \ldots, T} \tag{1.19}$$

El objetivo principal de la regresión es la determinación o estimación de $\boldsymbol{\beta_1}$ y $\boldsymbol{\beta_2}$ a partir de la información contenida en las observaciones de dos variables $\boldsymbol{X}$ e $\boldsymbol{Y}$ que disponemos sobre una muestra de individuos. El primer paso en un análisis de regresión es representar estos datos sobre unos ejes coordenados $\boldsymbol{x - y}$. Esta representación es el llamado *diagrama de dispersión* (Spiegel et al., 2009). Nos puede ayudar mucho en la búsqueda de un modelo que describa la relación entre las dos variables. Si la relación lineal de dependencia entre $\boldsymbol{Y}$ y $\boldsymbol{X}$ fuera exacta, las observaciones se situarían a lo largo de una recta Fig. 1.3.

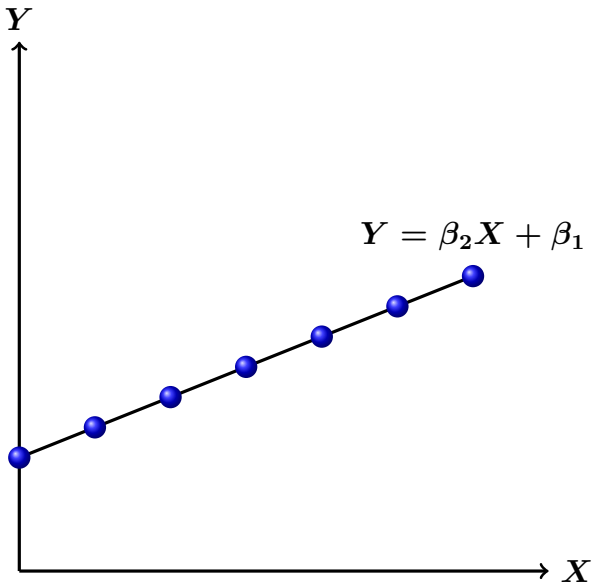

Figura 1.3: Relación de dependencia lineal exacta entre las variables $\boldsymbol{X}$ y $\boldsymbol{Y}$

Pero si la dependencia entre $\boldsymbol{Y}$ y $\boldsymbol{X}$ es estocástica, entonces, en general, las observaciones no se alinearán a lo largo de una recta, sino que formarán una nube de puntos Fig. 1.4. Si designamos mediante $\boldsymbol{\beta_1}'$ y $\boldsymbol{\beta_2}'$ las estimaciones de $\boldsymbol{\beta_1}$ y $\boldsymbol{\beta_2}$ respectivamente, la ordenada de

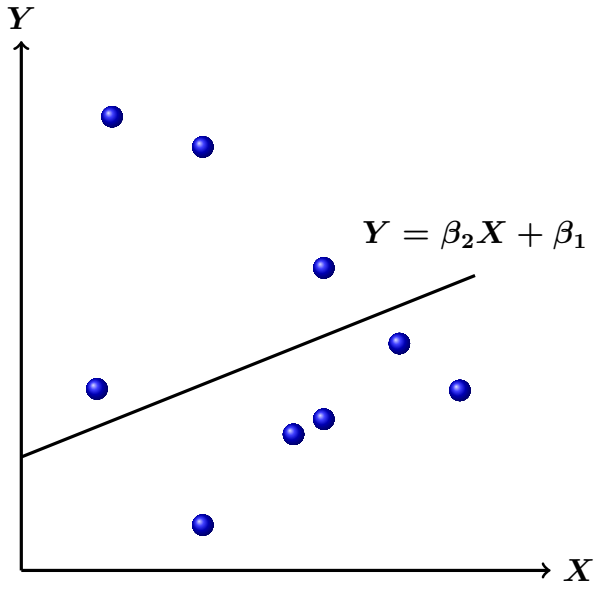

Figura 1.4: Representación de una dependencia estocástica entre $\boldsymbol{X}$ y $\boldsymbol{Y}$

la recta para el valor $\boldsymbol{X_t}$ vendrá dada por:

$$\boldsymbol{Y_t = \beta_1^{'} + \beta_2^{'} X_t} \tag{1.20}$$

El problema ahora es hallar unos estimadores $\boldsymbol{\beta_1}^{'}$ y $\boldsymbol{\beta_2}^{'}$ tales que la recta que pasa por los puntos $\boldsymbol{(X_t, Y_t^{'})}$ se ajuste lo mejor posible a los puntos $\boldsymbol{(X_t, Y_t)}$. Se denomina error o residuo a la diferencia entre el valor observado de la variable dependiente y el valor ajustado, es decir:

$$\boldsymbol{u_t^{'} = Y_t - Y_t^{'} = Y_t - \beta_1^{'} - \beta_2^{'}} \tag{1.21}$$

En el modelo de regresión lineal simple hay tres parámetros que se deben estimar: los coeficientes de la recta de regresión, $\boldsymbol{\beta_1}$ y $\boldsymbol{\beta_2}$, y la varianza de la distribución normal, $\boldsymbol{\sigma^2}$. El cálculo de estimadores para estos parámetros puede hacerse por diferentes métodos, siendo el más utilizado el método de mínimos cuadrados.

Este método consiste en buscar los valores de los parámetros $\boldsymbol{\beta_1}^{'}$ y $\boldsymbol{\beta_2}^{'}$ de manera que la suma de los cuadrados de los residuos ($\boldsymbol{S}$) sea mínima. Esta recta es la recta de regresión por mínimos cuadrados. Siendo la suma de los cuadrados la expresión:

$$\boldsymbol{S = \sum_{t=1}^{T} (Y_t - \beta_1^{'} - \beta_2^{'} X_t)^2} \tag{1.22}$$

Para minimizar $\boldsymbol{S}$, derivamos parcialmente respecto a $\boldsymbol{\beta_1}'$ y $\boldsymbol{\beta_2}'$ y obtenemos:

$$\frac{\boldsymbol{\partial S}}{\boldsymbol{\partial \beta_1'}} = -\mathbf{2}\sum_{t=1}^{T}(\boldsymbol{Y_t} - \boldsymbol{\beta_1'} - \boldsymbol{\beta_2'}\boldsymbol{X_t}) \tag{1.23}$$

$$\frac{\boldsymbol{\partial S}}{\boldsymbol{\partial \beta_2'}} = -\mathbf{2}\sum_{t=1}^{T}(\boldsymbol{Y_t} - \boldsymbol{\beta_1'} - \boldsymbol{\beta_2'}\boldsymbol{X_t})\boldsymbol{X_t} \tag{1.24}$$

ahora igualamos a cero las ecuaciones anteriores y realizando el trabajo algebraico correspondiente obtenemos el siguiente sistema de ecuaciones, conocido como *sistema de ecuaciones normales* de la recta de regresión:

$$\left.\begin{aligned}\sum_{t=1}^{T}(\boldsymbol{Y_t} - \boldsymbol{\beta_1'} - \boldsymbol{\beta_2'}\boldsymbol{X_t}) = \mathbf{0}\\ \sum_{t=1}^{T}(\boldsymbol{Y_t} - \boldsymbol{\beta_1'} - \boldsymbol{\beta_2'}\boldsymbol{X_t})\boldsymbol{X_t} = \mathbf{0}\end{aligned}\right\} \tag{1.25}$$

resoloviendo el sistema planteado anteriormente obtenemos que el estimador de $\boldsymbol{\beta_2'}$ es :

$$\boldsymbol{\beta_2'} = \frac{\sum_{t=1}^{T}(\boldsymbol{Y_t} - \overline{\boldsymbol{Y}})(\boldsymbol{X_t} - \overline{\boldsymbol{X}})}{\sum_{t=1}^{T}(\boldsymbol{X_t} - \overline{\boldsymbol{X}})^{\mathbf{2}}} \tag{1.26}$$

y

$$\boldsymbol{\beta_1'} = \overline{\boldsymbol{Y}} - \boldsymbol{\beta_2'}\overline{\boldsymbol{X}} \tag{1.27}$$

dividiendo el numerador y el denominador de 1.26 por $\boldsymbol{T}$ obtenemos que:

$$\boldsymbol{\beta_2'} = \frac{\frac{\sum_{t=1}^{T}(\boldsymbol{Y_t} - \overline{\boldsymbol{Y}})(\boldsymbol{X_t} - \overline{\boldsymbol{X}})}{\boldsymbol{T}}}{\frac{\sum_{t=1}^{T}(\boldsymbol{X_t} - \overline{\boldsymbol{X}})^{\mathbf{2}}}{\boldsymbol{T}}} = \frac{\boldsymbol{cov(X,Y)}}{\boldsymbol{var(X)}} \tag{1.28}$$

dado que la varianza de $\boldsymbol{X}$ no puede ser negativa, el signo de $\boldsymbol{\beta_2'}$ será el mismo que el de la covarianza muestral de $\boldsymbol{X}$ y $\boldsymbol{Y}$. La recta de regresión ahora se puede escribir de la manera

siguiente:

$$\boldsymbol{Y^{'} = \beta_1^{'} + \beta_2^{'} X} \tag{1.29}$$

los residuos de la recta de regresión los denotaremos por $\boldsymbol{e_i}$ y se calculan a partir de:

$$\boldsymbol{e_i = Y_i - Y_i^{'}} \tag{1.30}$$

donde $\boldsymbol{Y_i^{'}}$ es el valor estimado por la recta de regresión.

1.4.2 Los residuos en la estadística

En la regresión lineal, los residuos o residuales son utilizados para verificar la adecuación del modelo. Los residuos expresan la diferencia entre una observación y su valor ajustado. Estos pueden ser usados para evaluar la adecuación del ajuste de un modelo, con respecto a la elección de la función de varianza, la función enlace y en términos del predictor lineal (Spiegel et al., 2009). Los residuos $\boldsymbol{e_i = Y_i - Y_i^{'}}$ de un modelo de regresión lineal son variables aleatorias con distribución $\boldsymbol{e_i \sim N(0, \sigma^2(1 - h_{ii})), i = 1, 2, ..., n}$ siendo $\boldsymbol{n}$ el tamaño de la muestra de los datos. Existen las 3 variantes de residuos siguientes:

1. Residuos crudos: Es la diferencia entre el valor observado y el valor predictivo, tal y como se muestra en la ecuación (1.30).
2. Residuos estandarizados: Son aquellos que se obtienen dividiendo los residuos crudos por la estimación del error estándar y se definen como:

$$\boldsymbol{r_i = \frac{e_i}{S_R\sqrt{1 - v_{ii}}} \quad i = 1, 2, \ldots n} \tag{1.31}$$

3. Residuos estudentizados: son los que resultan de dividir los residuos raw por la estimación de su desviación estándar y se definen como :

$$\boldsymbol{t_i = \frac{e_i}{S_{R(i)}\sqrt{1 - v_{ii}}} \quad i = 1, 2, \ldots n} \tag{1.32}$$

Para chequear el modelo de regresión se deben utilizar los residuos estandarizados o los estudentizados. Típicamente, las desviaciones estándares de los residuos en una muestra varían considerablemente de un punto de datos al otro aunque los errores todos tienen igual la desviación estándar, particularmente en el análisis de regresión. Por tanto, no es conveniente comparar los residuos en diferentes puntos de datos sin primero estudentizarlos. Esta técnica es muy importante para la detección de los *outliers*, o datos atípicos, los cuales se definen

como las observaciones con residuos estandarizados grandes ($|\boldsymbol{r_i} > \mathbf{2}|$). Nos interesa ahora cómo se estudentizan los residuos de un modelo de regresión lineal simple. Para ello, partimos de una matriz del tamaño ($\boldsymbol{nx2}$) siguiente:

$$\boldsymbol{X} = \begin{bmatrix} \mathbf{1} & \boldsymbol{x_1} \\ \mathbf{2} & \boldsymbol{x_2} \\ \boldsymbol{\cdots} & \boldsymbol{\cdots} \\ \boldsymbol{n} & \boldsymbol{x_n} \end{bmatrix}$$

donde $\boldsymbol{x_i(i = 1, ..., n)}$ son los datos de la muestra. La matriz "hat" $\boldsymbol{H}$ es la proyección ortogonal sobre el espacio columna de la matriz $\boldsymbol{X}$ y se obtiene por:

$$\boldsymbol{H = X(X^T X)^{-1} X^T} \tag{1.33}$$

la varianza del i-esimo residuo es:

$$\boldsymbol{var(\hat{e}_i) = \sigma^2(1 - h_{ii})} \tag{1.34}$$

siendo $\boldsymbol{h_{ii}}$ el i-esimo elemento de la diagonal principal de la matriz $\boldsymbol{H}$; luego el residuo estudentizado correspondiente se calcula por:

$$\boldsymbol{t_i = \frac{\hat{e}_i}{\hat{\sigma}\sqrt{1 - h_{ii}}}} \tag{1.35}$$

donde $\hat{\boldsymbol{\sigma}}$ es la estimación apropiada de $\boldsymbol{\sigma}$ que viene calculado de la siguiente forma:

$$\boldsymbol{\hat{\sigma}^2 = \frac{1}{n - m} \sum_{j=1}^{n} \hat{\sigma}_j{}^2} \tag{1.36}$$

Sabiendo que $\boldsymbol{m}$ es el número de los parámetros en el modelo ($\boldsymbol{m}$ es igual a $\mathbf{2}$ para el modelo simple). Sin embargo, es deseado excluir la i-esima observación del proceso de estimación de las varianzas cuando se considera que el i-esimo caso pudiera ser un outlier. Consecuentemente, $\hat{\boldsymbol{\sigma}}$ se puede estimar con la forma:

$$\boldsymbol{\hat{\sigma}_i^2 = \frac{1}{n - m - 1} \sum_{\substack{j=1 \\ j \neq i}}^{n} \hat{\sigma}_j{}^2} \tag{1.37}$$

que se basa en todos los casos, excepto el i-esimo. Si esta última estimación es utilizada, excluyendo un caso atípico, entonces el residuo se dice que es estudentizado externamente; de lo contrario, si usamos la forma (1.36) para estimar la varianza $\boldsymbol{\sigma}$ estamos estudentizando el residuo internamente.

1.5 El modelo de programación en CUDA

El diseño del modelo de programación en CUDA está pensado de forma tal que las aplicaciones creadas pueden, de forma transparente, escalar su paralelismo para así incrementar el número de núcleos computacionales (del Toro Melgarejo et al., 2012). La programación en CUDA está basada en tres abstracciones básicas; una jerarquía de grupos de hilos, de tipos de memoria, y barreras de sincronización. La estructura que conforma la jerarquía de hilos en este modelo está formada por tres unidades básicas: mallas, bloques e hilos. Múltiples bloques conforman las mallas; un bloque está formado por un grupo de hilos, la unidad constituyente más elemental. Una malla puede ser definida como un grupo de bloques que ejecuta cierta función llamada *kernel*. La Fig. 1.5 muestra la organización de esta jerarquía.

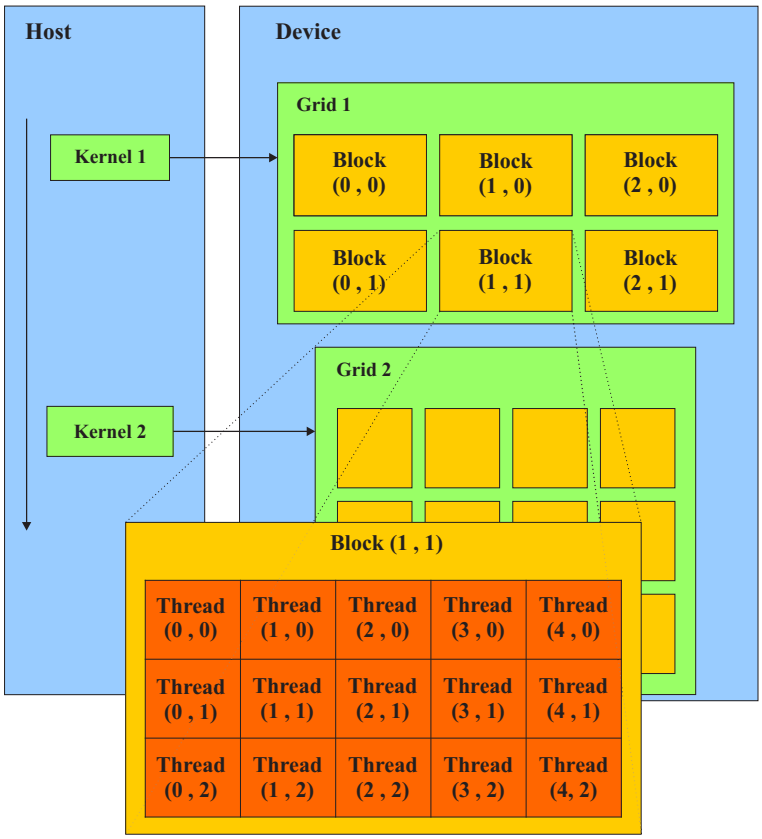

Figura 1.5: Lotes de hilos

Los hilos dentro de un bloque son ejecutados concurrentemente en un Multiprocesador de flujo (SM[1]). A su vez los bloques dentro de las mallas son distribuidos entre los diferentes multiprocesadores, lo que establece otro grado de paralelismo. Los SM son las unidades de cálculo por la cuales está constituido una moderna GPGPU[2] típica, a las cuales le son asignados bloques de hilos para su ejecución paralela. Cada SM tiene una o más unidades de recolección de instrucciones, múltiples unidades aritmético lógicas (ALU), llamadas también núcleos CUDA para la ejecución paralela, cierta cantidad de memoria paralela accesible por todos los hilos del bloque, y un grupo de registros compartidos entre los hilos. Como las diferentes ALU comparten una misma unidad de instrucciones, los hilos asignados a estas ejecutan la misma instrucción utilizando una arquitectura llamada SIMT[3] (Alba, 2013).

Las funciones que se ejecutarán en la GPU paralelamente por varios hilos son llamadas *kernel*. Mediante una de las extensiones al lenguaje C creadas por NVIDIA, los *kernels*, se definen usando la declaración específica **___global___**. Uno de los rasgos principales que define a un *kernel* es la "configuración de ejecución", toda llamada a una función ___global___ debe especificarla. Esta configuración define la dimensión de la malla de bloques y la cantidad de hilos por bloques con que será ejecutado el *kernel* en la GPU. Los hilos pertenecientes a un mismo bloque pueden trabajar en colectivo compartiendo información a través una cantidad limitada de memoria compartida, accesible solo por los hilos del bloque. El trabajo de estos hilos puede ser sincronizado mediante directivas de bloqueo lo que posibilita la coordinación entre estos; sin embargo, hilos agrupados en diferentes bloques no pueden comunicarse entre sí.

Este modelo permite a los *kernels* ejecutarse de manera eficiente en varios dispositivos sin recompilación y con diferentes capacidades paralelas en todos los bloques de una malla de forma secuencial si se tiene muy poca capacidad de paralelismo, o en paralelo si se tiene mucha capacidad de paralelismo, aunque por lo general se usa una combinación de ambos (del Toro Melgarejo et al., 2012).

Los hilos que se ejecutan en un dispositivo CUDA tienen acceso a múltiples espacios de memoria: global, local, compartida, textura y registros, tal como se muestra en la Fig. 1.6. Cada hilo tiene un espacio de memoria local privado; a su vez, cada bloque de hilos posee memoria compartida visible solo a todos los hilos del bloque y con la misma duración de vida que el bloque. Además, todos los hilos ejecutando cierto *kernel* tienen acceso a la misma memoria global. Los espacios de memoria global, constante y de textura, son persistentes a través de los diferentes lanzamientos de *kernels* en la misma aplicación (Alba, 2013).

[1]Del inglés *Streaming Multiprocessor*
[2]Del inglés *General-Purpose Computing on Graphics Processin Units*
[3]Del inglés *Single Instructions Multiple Threads*

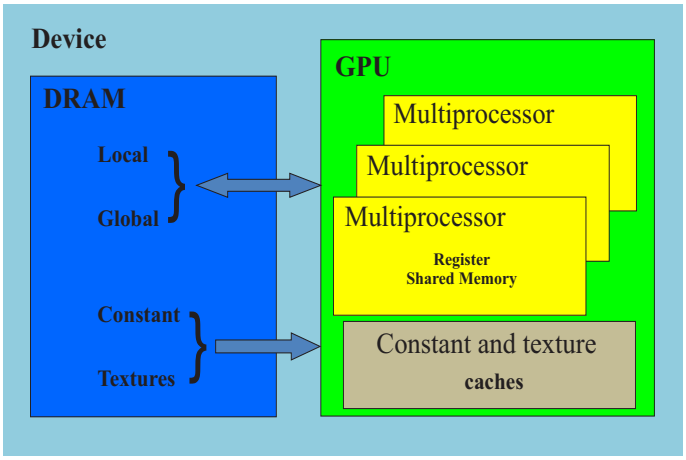

Figura 1.6: Espacios en memoria

1.6 Estructura de datos Trie

Esta sección contiene los elementos teóricos básicos necesarios para lograr una correcta comprensión del funcionamiento de la estructura de datos Trie. Se hace aquí un análisis sobre la complejidad computacional que tienen algunas de las posibles operaciones que se pueden realizar sobre la estructura, además se compara de forma concisa la misma con una tabla hash siendo esta última el otro tipo de dato abstracto que se pudo haber utilizado en el desarrollo de la aplicación.

1.6.1 Definición formal

El trie es una estructura de datos de tipo árbol que permite la recuperación de información[4]. La información almacenada en un trie es un conjunto de claves, donde una clave es una secuencia de símbolos pertenecientes a un alfabeto. Las claves son almacenadas en las hojas del árbol y los nodos internos son pasarelas para guiar la búsqueda. El árbol se estructura de forma que cada letra de la clave se sitúa en un nodo de manera que los hijos de un nodo representan las distintas posibilidades de símbolos diferentes que pueden continuar al símbolo representado por el nodo padre. Por tanto, la búsqueda en un trie se hace de forma similar a como se hacen las búsquedas en un diccionario. Por eficiencia se suelen eliminar los nodos intermedios que solo tienen un hijo; es decir, si un nodo intermedio tiene solo un hijo con cierto caracter entonces el nodo hijo será el nodo hoja que contiene directamente la clave completa (Crochemore et al., 2001).

En la Fig. 1.7 se presenta un trie en el que se encuentran almacenadas las palabras a, to, tea, ted, ten, in, and y inn; en la misma podemos apreciar la agrupación de caracteres que se

[4]Su nombre proviene de la palabra en inglés re**trie**val

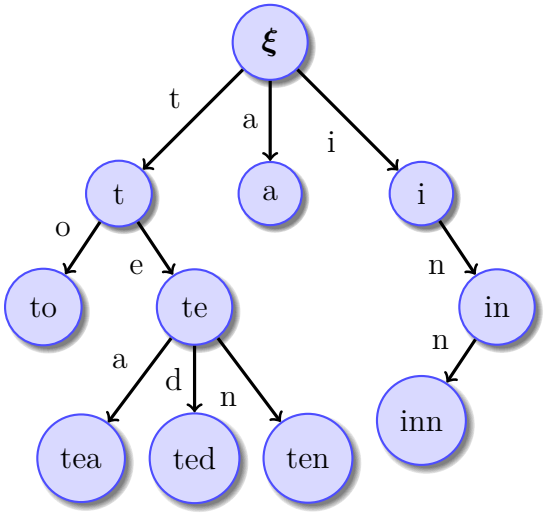

Figura 1.7: Representación gráfica de un trie

realiza en los nodos con un solo hijo.

Según (Crochemore et al., 2001) podemos decir que matemáticamente un trie es un caso especial de autómata finito determinista (S, Σ, T, s, A), que sirve para almacenar un conjunto de cadenas E en el que:

- Σ es el alfabeto[5] sobre el cual están definidas las cadenas.
- S es un conjunto de estados, cada uno de los cuales representa un prefijo de E.
- La función de transición $T : S \times \Sigma \rightarrow S$ está definida como sigue: $T(x, \sigma) = x\sigma$ si $x, x\sigma \in S$, e indefinida en otro caso.
- El estado inicial s corresponde a la cadena vacía ξ.
- El conjunto de estado de aceptación $A \subseteq S$ es igual a E.

1.6.2 Ventajas de un Trie

En un trie las búsquedas de una clave de longitud m tendrán en el peor de los casos un costo lineal $O(m)$. Esta estructura requiere muy poco espacio para almacenar gran cantidad de cadenas pequeñas, puesto que las claves no se almacenan explícitamente. Como sustitución de la estructura de datos tabla hash podemos decir que una búsqueda en implementaciones imperfectas de estas últimas tienen un coste del orden de $O(n)$ mientras que en el trie tiene un coste de $O(l)$ siendo n el total de claves almacenadas y l la longitud de la clave que se busca, esto se debe a las colisiones de claves, las cuales solo se producen en una tabla hash. En un trie no es necesario definir una función hash, además nos puede proporcionar un ordenamiento alfabético de las entradas por clave. También los contenedores que almacenan distintos valores asociados a una única clave solo son necesarios si tenemos más de un valor asociado a

[5]En este trabajo dicho alfabeto es A,C,G,T

una única clave mientras que en una tabla hash siempre se necesitan estos contenedores para las colisiones de clave.

1.7 Las bibliotecas GSL y CURAND

La *GNU Scientific Library(GSL)* es una biblioteca escrita en el lenguaje C, destinada a cálculos numéricos necesarios en matemáticas y en la ciencia de modo general, la misma se distribuye bajo la licencia GNU GPL. Esta biblioteca puede ser utilizada junto con las clases de C++, pero sin utilizar punteros a funciones miembros(Galassi et al., 2013). De las herramientas proporcionadas por la biblioteca se han utilizado en el trabajo las destinadas al trabajo con vectores, matrices, permutaciones y álgebra lineal.

La biblioteca CURAND forma parte del kit de herramientas de CUDA, su documentación es también incluida en este kit. Esta biblioteca brinda una variedad de facilidades para la generación de números pseudoaleatorios. La biblioteca consta de dos partes fundamentales, una diseñada para brindar estos servicios en la CPU, y otra en la GPU. Las funciones implementadas del lado de la GPU permiten generar números pseudoaleatorios en esta, y usarlos directamente desde la GPU sin necesidad de escribirlos a memoria global para después leerlos. Las funciones que permiten la generación de estos números lo hacen de forma concurrente, aprovechando el poder de paralelismo que brindan los GPU de NVIDIA (Alba, 2013).

1.8 Conclusiones parciales del capítulo

Hemos expuesto los conceptos básicos de la evolución molecular y el mecanismo de aplicar la técnica de simulación para la evolución de una población viral en el cual el método de Markov es usado como el núcleo para implementar nuestro algoritmo en el capitulo siguiente. Las secuencias generadas por este método son considerados como las mutaciones de las secuencias de la población inicial teniendo en cuenta que el tamaño de las poblaciones nuevas creadas por el algoritmo son diferentes dependiendo de la configuración empleada al desarrollar los diseños experimentales.

Se ha abordado también la técnica del análisis de regresión lineal como una herramienta estadística muy útil para estudiar la relación entre los datos que se usa en la implementación del algoritmo propuesto en las secciones siguientes. Finalmente, se expone el modelo de programación en CUDA a través de sus principales características, así como la estructura de datos trie, las bibliotecas CURAND y GSL, que se emplearán en el desarrollo de la aplicación.

Capítulo 2

Implementación de los algoritmos para la simulación de las poblaciones virales

Este capítulo tiene como objetivo exponer los principales componentes que constituyen la aplicación, además, se realiza con detalle la descripción algorítmica de las estrategias secuenciales y paralelas desarrolladas para lograr la simulación de las poblaciones virales. El desarrollo del mismo se realiza buscando la mayor correspondencia posible entre sus contenidos y los presentados en el capítulo precedente.

2.1 Aplicación de la técnica MCMC a la simulación de poblaciones virales.

Anteriormente se presentó a MCMC como una técnica para generar aleatoriamente muestras de una distribución objetivo, haciendo uso de las cadenas de Markov para explorar el espacio de estados. Para utilizar este algoritmo en la mutación de secuencias de ADN primeramente debemos establecer los estados posibles que conformarán la cadena de Markov. En el presente trabajo las secuencias de ADN están conformadas por 4 bases nitrogenadas y estas

se representan por los enteros $\mathbf{0, 1, 2, 3}$ en lugar de adenina(A), guanina(G), citosina(C) y timina(T) respectivamente, es decir, un gen o individuo se define como un arreglo de enteros que en cada posición tendrá uno de los valores previamente mencionados; es decir, las secuencias de ADN obtenidas de esta forma se pueden considerar vectores de $(\boldsymbol{Z_4})^n$; por lo tanto nuestro espacio de estados $\boldsymbol{S}$ para las mutaciones es:

$$\boldsymbol{S = \{0, 1, 2, 3\}}$$

Durante el procedimiento de mutación se asume el mismo proceso de sustitución en todos los sitios, además a cada uno de ellos le corresponde una cadena de Markov cumpliéndose la condición de que dichas cadenas son independientes. En la actualidad existen varios métodos para realizar las mutaciones a partir de MCMC, uno de ellos se denomina ***método de desarrollar las secuencias a lo largo del árbol filogenético***, conocido como un método fácil y muy usado en la simulación computacional (Yang, 2006). En este método primero se genera una secuencia para la raíz del árbol, de forma tal que los nucleótidos que constituyen la misma se crean de acuerdo a sus distribuciones equivalentes bajo el modelo de frecuencias $\boldsymbol{\pi_A, \pi_C, \pi_G, \pi_T}$, la secuencia obtenida se denomina ancestro, y permite desarrollar y producir descendientes en su nodos hijos.[1] Este proceso se repite para cada rama del árbol, de modo que se va generando una secuencia en un nodo, después que la secuencia del nodo padre ya se ha generado. Para simular la evolución de una secuencia a lo largo de una rama de longitud $\boldsymbol{t}$, es necesario calcular la matriz de probabilidades de transición siguiente :

$$\boldsymbol{P(t)} = \begin{bmatrix} \boldsymbol{p_{AA}(t)} & \boldsymbol{p_{AC}(t)} & \boldsymbol{p_{AG}(t)} & \boldsymbol{p_{AT}(t)} \\ \boldsymbol{p_{CA}(t)} & \boldsymbol{p_{CC}(t)} & \boldsymbol{p_{CG}(t)} & \boldsymbol{p_{CT}(t)} \\ \boldsymbol{p_{GA}(t)} & \boldsymbol{p_{GC}(t)} & \boldsymbol{p_{GG}(t)} & \boldsymbol{p_{GT}(t)} \\ \boldsymbol{p_{TA}(t)} & \boldsymbol{p_{TC}(t)} & \boldsymbol{p_{TG}(t)} & \boldsymbol{p_{TT}(t)} \end{bmatrix} \tag{2.1}$$

donde $\boldsymbol{p_{ij}(t)}$ denota la probabilidad de que ocurra un cambio a $\boldsymbol{j}$ después de $\boldsymbol{t}$ instantes de tiempo, partiendo desde $\boldsymbol{i}$.[2] Este proceso se repite hasta que se generan todos los sitios de la secuencia objetivo. La simulación de la evolución de los sitios en una secuencia se manifiesta en el evento de la mutación, en la cual el nucleótido mutado es determinado aleatoriamente (E. Gultepe, 2005). Estas mutaciones estocásticas se obtienen a partir del método MCMC, aplicándose este de la forma descrita en el capítulo anterior.

[1] En la implementación de la aplicación las secuencias no son generadas, estas son leídas como datos y a partir de las mismas se calculan cada una de las frecuencias, siendo esto análogo a lo que plantea el método.

[2] El valor de $\boldsymbol{t}$ lo podemos interpretar como el número de mutaciones que han ocurrido desde el ancestro hasta el nodo hijo.

2.1.1 Cálculo de la matriz P

Las mutaciones en cada sitio que se produzcan durante la simulación dependen directamente de la matriz de probabilidades de transición ($\boldsymbol{P}$), pues la misma posee las probabilidades de cambio que se emplearán para formar las cadenas de Markov. Esta matriz está estrechamente relacionada con el modelo evolutivo, debido a que según sea el modelo que estemos empleando obtendremos la matriz de tasas de sustitución $\boldsymbol{Q}$ y a partir de la misma podemos calcular a $\boldsymbol{P}$ mediante la fórmula:

$$\boldsymbol{P(t) = e^{Qt}} \tag{2.2}$$

en este trabajo se utiliza el modelo TN93 por lo que $\boldsymbol{Q}$ posee la estructura definida en 1.14. Según (Yang, 2006) si $\boldsymbol{Q}$ es diagonalizable se cumple que:

$$\boldsymbol{Q = UDU^{-1}} \tag{2.3}$$

donde $\boldsymbol{U}$ es una matriz no singular de dimensión 4x4, $\boldsymbol{U^{-1}}$ es su inversa y $\boldsymbol{D = diag\{\lambda_1, \lambda_2, \lambda_3, \lambda_4\}}$ es una matriz diagonal creada a partir del vector de valores propios $\boldsymbol{\lambda_i}$ de $\boldsymbol{Q}$, la ecuación (2.3) se denomina descomposición espectral de $\boldsymbol{Q}$. De esta ecuación obtenemos que:

$$\boldsymbol{Q^2 = (UDU^{-1})(UDU^{-1}) = UD^2U^{-1} = U(diag\{(\lambda_1)^2, (\lambda_2)^2, (\lambda_3)^2, (\lambda_4)^2\})U^{-1}} \tag{2.4}$$

de forma análoga obtenemos $\boldsymbol{Q^m = U(diag\{(\lambda_1)^m, (\lambda_2)^m, (\lambda_3)^m, (\lambda_4)^m\})U^{-1}}$ para cualquier entero $\boldsymbol{m}$. En general cualquier función algebraica $\boldsymbol{f}$ aplicada a la matriz $\boldsymbol{Q}$ se puede calcular como $\boldsymbol{f(Q) = U(diag\{f(\lambda_1), f(\lambda_2), f(\lambda_3), f(\lambda_4)\})U^{-1}}$ siempre y cuando $\boldsymbol{f(Q)}$ exista; por tanto utilizando la ecuación 2.4 podemos escribir a 2.2 de la siguiente forma:

$$\boldsymbol{P(t) = e^{Qt} = U(diag\{e^{\lambda_1 t}, e^{\lambda_2 t}, e^{\lambda_3 t}, e^{\lambda_4 t}\})U^{-1}} \tag{2.5}$$

los $\boldsymbol{\lambda_i (i = 1, 2, 3, 4)}$ son los valores propios de la matriz $\boldsymbol{Q}$. Las columnas de $\boldsymbol{U}$ y las filas de $\boldsymbol{U^{-1}}$ son los vectores propios derechos e izquierdos correspondientes de $\boldsymbol{Q}$ respectivamente. Para el modelo TN93 se determinan analíticamente los valores propios de $\boldsymbol{Q}$; tenemos que los mismos son:

$$\boldsymbol{\lambda_1 = 0; \quad \lambda_2 = -\beta; \quad \lambda_3 = -(\pi_R \alpha_2 + \pi_Y \beta); \quad \lambda_4 = -(\pi_Y \alpha_1 + \pi_R \beta)}$$

Las matrices de vectores propios serán:

$$\boldsymbol{U} = \begin{bmatrix} \mathbf{1} & \mathbf{1/\pi_Y} & \mathbf{0} & \boldsymbol{\pi_C/\pi_Y} \\ \mathbf{1} & \mathbf{1/\pi_Y} & \mathbf{0} & \mathbf{-\pi_T/\pi_Y} \\ \mathbf{1} & \mathbf{-1/\pi_R} & \boldsymbol{\pi_G \pi_R} & \mathbf{0} \\ \mathbf{1} & \mathbf{-1/\pi_R} & \mathbf{-\pi_A \pi_R} & \mathbf{0} \end{bmatrix} \tag{2.6}$$

$$\boldsymbol{U^{-1}} = \begin{bmatrix} \boldsymbol{\pi_T} & \boldsymbol{\pi_C} & \boldsymbol{\pi_A} & \boldsymbol{\pi_G} \\ \boldsymbol{\pi_T \pi_R} & \boldsymbol{\pi_C \pi_R} & \boldsymbol{\pi_A \pi_R} & \boldsymbol{\pi_G \pi_R} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{-1} \\ \mathbf{1} & \mathbf{-1} & \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{2.7}$$

Sustituyendo $\boldsymbol{U}$ y $\boldsymbol{U^{-1}}$ en la ecuación (2.5) obtenemos a $\boldsymbol{P(t)}$ como:

$$\begin{bmatrix} \boldsymbol{\pi_T + \frac{\pi_T \pi_R}{\pi_Y} e_2 + \frac{\pi_C}{\pi_Y} e_4} & \boldsymbol{\pi_C + \frac{\pi_C \pi_R}{\pi_Y} e_2 - \frac{\pi_C}{\pi_Y} e_4} & \boldsymbol{\pi_A(1 - e_2)} & \boldsymbol{\pi_G(1 - e_2)} \\ \boldsymbol{\pi_T + \frac{\pi_T \pi_R}{\pi_Y} e_2 - \frac{\pi_T}{\pi_Y} e_4} & \boldsymbol{\pi_C + \frac{\pi_C \pi_R}{\pi_Y} e_2 + \frac{\pi_T}{\pi_Y} e_4} & \boldsymbol{\pi_A(1 - e_2)} & \boldsymbol{\pi_G(1 - e_2)} \\ \boldsymbol{\pi_T(1 - e_2)} & \boldsymbol{\pi_C(1 - e_2)} & \boldsymbol{\pi_A + \frac{\pi_A \pi_Y}{\pi_R} e_2 + \frac{\pi_G}{\pi_R} e_3} & \boldsymbol{\pi_G + \frac{\pi_G \pi_R}{\pi_Y} e_2 - \frac{\pi_G}{\pi_R} e_3} \\ \boldsymbol{\pi_T(1 - e_2)} & \boldsymbol{\pi_C(1 - e_2)} & \boldsymbol{\pi_A + \frac{\pi_A \pi_Y}{\pi_R} e_2 - \frac{\pi_A}{\pi_R} e_3} & \boldsymbol{\pi_G + \frac{\pi_G \pi_R}{\pi_Y} e_2 + \frac{\pi_A}{\pi_R} e_3} \end{bmatrix}$$

donde $\boldsymbol{e_i = exp(\lambda_i t) = e^{\lambda_i t}}$. Cuando $\boldsymbol{t}$ incrementa de 0 a $\boldsymbol{\infty}$ los elementos $\boldsymbol{p_{ii}(t)}$ de la diagonal principal de la matriz anterior decrementan de 1 a $\boldsymbol{\pi_i}$, mientras que los elementos $\boldsymbol{p_{ij}(t)}$ incrementan de 0 a $\boldsymbol{\pi_{ij}}$; por tanto $\boldsymbol{p_{ij}(\infty) = \pi_j}$, además $\boldsymbol{(\pi_A, \pi_C, \pi_G, \pi_T)}$ es la distribución estacionaria de la cadena que se construye a partir de $\boldsymbol{P}$.

La matriz anterior se calcula para cada mutación que se vaya a generar de la secuencia ancestral. Luego que se tiene la misma podemos efectuar la técnica de **MCMC** sobre la secuencia ancestral obteniéndose las mutaciones. El pseudo-código de este proceso se muestra en el algoritmo 2.1. Según (Minh, 2010) una vez que el valor candidato $\boldsymbol{x^*}$ es aceptado, esto es equivalente a decir que ha ocurrido una mutación del nucleótido en el sitio dado. Este cambio da como consecuencia que tanto la probabilidad de distribución como la probabilidad de mutación del nucleótido se van a determinar para el nuevo estado de acuerdo a la probabilidad previamente definida del nuevo nucleótido. En el caso de que no haya cambio ninguno, o sea, que la cadena de Markov se mantenga en el estado actual con el mismo valor de nucleótido en el sitio dado, entonces no hay necesidad de cambiar las probabilidades de distribución y mutación de este último. Se puede apreciar que cuando se aplica este proceso sobre una secuencia de ADN se trabaja con cada uno de los nucleótidos, de manera tal que la nueva secuencia generada tiene la misma longitud y se comporta como una mutación de su ancestro, formándose a partir de los componentes ya modificados y evidentemente posee propiedades biológicas diferentes (Minh, 2010).

Algoritmo 2.1 Algoritmo MCMC para la generación de nuevas mutaciones

Entrada: Secuencia ancestral $\boldsymbol{A}$ de longitud $\boldsymbol{N}$.
Salida: Secuencia descendiente $\boldsymbol{D}$.

1: Elegir la secuencia ancestral $\boldsymbol{A}$;
2: **for** $\boldsymbol{i = 0}$ hasta $\boldsymbol{N - 1}$ **do**
3: $\quad \boldsymbol{x^i = A_i}$;
4: $\quad$ Muestrear $\boldsymbol{u \sim U(0, 1)}$;
5: $\quad$ Muestrear $\boldsymbol{x^* \sim P(x^*|x^i)}$;
6: $\quad$ **if** $\boldsymbol{u < min(1, \frac{f(x^i)P(x^*, x^i)}{f(x^*)P(x^i, x^*)})}$ **then**
7: $\quad\quad \boldsymbol{D_i = x^*}$;
8: $\quad$ **else**
9: $\quad\quad \boldsymbol{D_i = x^i}$;
10: $\quad$ **end if**
11: **end for**
12: Retornar la secuencia descendiente $\boldsymbol{D}$;

2.2 Descripción del proceso de evolución genética de las poblaciones virales y de los principales algoritmos utilizados para desarrollar el mismo

El proceso evolutivo comienza a partir de un conjunto de secuencias de ADN, donde cada una de ellas se considera un ancestro, que puede generar una cantidad de descendientes llamados mutaciones. La forma de seleccionar un ancestro incluido en este conjunto para

realizar el proceso se lleva a cabo aleatoriamente, mediante la distribución de probabilidad de las secuencias de la población inicial[3], de forma tal que una de ellas es seleccionada si su probabilidad es mayor que un número generado de forma aleatoria, en caso de existir más de una secuencia que cumpla esta condición se escoge la que primero se encuentre. Si no existe dentro de las probabilidades ninguna mayor que el número generado, entonces escogemos como ancestro la secuencia con mayor frecuencia de aparición. El procedimiento de mutación es complejo y puede realizarse sujeto a restricciones con el fin de obtener mutaciones que satisfagan cierta condición. A continuación veremos cómo se realiza la generación de las secuencias mutantes, en un caso sin restricciones y en el otro con la aplicación de las restricciones establecidas.

2.2.1 Generación de mutaciones sin restricciones

En este caso las secuencias mutantes se generan directamente para formar las nuevas clases[4] a partir de los ancestros, sin pasar a revisarse las condiciones propuestas; por tanto, es más sencillo y consume menos tiempo de ejecución. Las secuencias generadas de esta manera se incluyen todas en la población de los virus, y se clasifican según sean sus propiedades estructurales. Concretamente, este proceso se hace según los pasos descritos a continuación:

1. Especificar la cantidad de descendientes que pudiera tener el ancestro.
2. Escoger una ancestor aleatoriamente a partir de la distribución de frecuencia de las secuencias y luego generar una mutación del mismo aplicando la estrategia MCMC.
3. Comprobar que no existan codones de parada en la mutación obtenida, en caso de exitir realizar las operaciones necesarias para eliminar los mismos y obtener la secuencia mutante sin codones paradas.
4. Comparar la secuencia mutante con el ancestro y con el resto de las otras secuencias; en caso de que la misma sea igual que uno de ellos, se aumentará el contador de este en 1, de lo contrario se agregará la mutación a la población como una nueva clase del virus y se aumentará el número de clases en 1. En los dos casos la cantidad total de secuencias de la población se incrementa en una unidad.
5. El ancestro actual se traslada a la secuencia recién creada y el algoritmo vuelve al paso 2 en caso que no se haya acabado la cantidad de descendientes generados del primer ancestro; de lo contrario se mueve al paso siguiente.
6. Si el número de posibles ancestros no se ha rebasado, seleccionar el nuevo ancestro entre las secuencias de la población inicial disponible, de forma independiente de la selección de los otros ancestros y volvemos al paso 1; de otro modo se mueve al paso siguiente.

[3]La distribución que se emplea aquí es similar a una distribución uniforme discreta, con la diferencia de que no todas las secuencias poseen igual probabilidad pues esta se calcula mediante $\boldsymbol{p(S_i) = F(S_i)/N}$, siendo $\boldsymbol{N}$ el total de frecuencias y $\boldsymbol{F(S_i)}$ la cantidad de repeticiones de la secuencia $\boldsymbol{S_i}$.

[4]Aquí una clase hace referencia a una secuencia.

7. Terminar el algoritmo.

Este procedimiento se realiza para cada generación de virus y se repite tantas veces como el número de generaciones especificado al principio. Es importante enfatizar que cada vez que se hayan terminado de generar las mutaciones de una generación, es necesario actualizar la distribución de probabilidades de las clases antes de pasar a la nueva, pues la misma influye directamente en la selección de los ancestros que se efectuará al empezar el proceso de crear las nuevas secuencias mutantes. De este modo los ancestros seleccionados en una generación podrían incluir a las secuencias que fueron creadas en las generaciones anteriores. Al finalizar este proceso, se obtiene una población creciente de varias generaciones, incluyendo todas las secuencias mutantes, sin tener en cuenta ninguna condición (Minh, 2010).

Cuando se obtienen las mutaciones es necesario verificar la presencia de codones de terminación (codones de parada ó *stop codons*) en cada una de las secuencias obtenidas. Un codón (nucleótido triple) de terminación es aquel que no determina en el código genético aminoácido alguno. Su función es acotar el mensaje cifrado por el ADN que da lugar al ARN mensajero (mARN). En otras palabras, un codón de terminación señala a la maquinaria de la molécula que se ha alcanzado el fin del proceso de sintetizar las proteínas (Minh, 2010). En el código genético estándar, los codones de terminación se presentan en tres diferentes formas: TGA, TAG y TAA en ADN y sus correspondientes copias UGA, UAG y UAA en ARN respectivamente. El objetivo del chequeo de la generación de los codones de parada en las secuencias de mutación, es simplemente detectar el lugar donde aparecen los mismos a lo largo de las secuencias, luego es necesario quitar estos codones modificando las bases que forman parte de ellos. Una descripción completa de este proceso es la siguiente:

1. Transformar la secuencia de nucleótidos codificada por los caracteres $\boldsymbol{\{A, C, G, T\}}$ en la misma secuencia representando la misma mediante los números $\boldsymbol{\{0, 1, 2, 3\}}$[5] respectivamente.

2. Encontrar el valor númerico que representa cada uno de los codones de la secuencia, esto se hace utilizando los sucesores de cada uno de los números que componen un triplete multiplicando el primero por 4, el segundo por 16 y el tercero por 1 y luego hallando la suma de los resultados de cada una de las multiplicaciones, por ejemplo:

   $$\mathbf{TAG = 302 \rightarrow 4 * (3 + 1) + 16 * (0 + 1) + 1 * (2 + 1) = 35}$$

3. Localizar los codones de terminación TAA, TAG y TGA en la secuencia a partir de los valores calculados para cada codón, conociendo que los valores representantes de estos codones son 33, 35 y 65 respectivamente.

4. Eliminar los *stop codons* de la secuencia; en caso que existieran modificando las bases

[5]Se utiliza esta codificación en lugar de $\boldsymbol{\{1, 2, 3, 4\}}$ que resulta un poco más natural, para lograr correspondencia con la representación de los arreglos de c++ que comienzan en la posición **0**.

que forman parte de ellos de manera que la primera se mantiene igual, mientras que la segunda y la tercera se reemplazan por las otras bases diferentes generadas de forma aleatoria empleando el método de **Metropolis-Hasting**, en caso de obtener un codón de parada mediante las bases generadas entonces las tercera base del aminoácido es remplazada por T. Este proceso se repite para cada uno de los tripletes de la secuencia garantizándose así que no quede ningún *stop codons*.

La mutación de un ancestro determinado es aquella que resulta finalmente después de realizar este mecanismo de detección de los codones de parada. De esta forma garantizamos que las secuencias generadas por el modelo evolutivo mantendrán cuantitativamente la completitud de la información genética de las mismas, en el proceso de sintetizar las secuencias de aminoácidos. En el algoritmo 2.2 se presenta el pseudo-código de la versión secuencial del algoritmo correspondiente con el proceso descrito, el mismo se diferencia de la descripción anterior en que el procesamiento de la secuencia se realiza codón por codón.

Algoritmo 2.2 Algoritmo para encontrar los codones de parada

Entrada: Secuencia mutante $\boldsymbol{M}$ de longitud $\boldsymbol{n}$.
Salida: Secuencia mutante sin codones de parada.

1: Calcular el total de codones $\boldsymbol{t = n/3}$;
2: **for** $\boldsymbol{i = 1}$ hasta $\boldsymbol{t}$ **do**
3: Calcular el codón $\boldsymbol{C_i}$ de M;
4: **if** $\boldsymbol{C_i}$ = codón de parada **then**
5: $\boldsymbol{C_i^* = Mutar(C_i)}$ {Algoritmo **Metroplis-Hastings**};
6: **if** $\boldsymbol{C_i^*}$ = codón de parada **then**
7: $\boldsymbol{Modificar(C_i^*)}$ {Modifica la última posición del codón por T};
8: $\boldsymbol{C_i = C_i^*}$;
9: **else**
10: $\boldsymbol{C_i = C_i^*}$;
11: **end if**
12: Ubicar $\boldsymbol{C_i}$ dentro de $\boldsymbol{M}$;
13: **end if**
14: **end for**

2.2.2 Generación de mutaciones con restricciones

Al igual que el procedimiento descrito anteriormente las secuencias de ADN mutante se generan mediante los mismos pasos explicados con la diferencia de que entre el paso 2 y 3 se realiza un proceso de clasificación sobre la secuencia mutante denominado proceso de selección. Este último consiste en el hecho de que una nueva secuencia, generada a partir de un ancestro dado, se analizará de forma tal que la misma pudiera satisfacer o no la condición de resistencia contra la vacuna[6]. En el caso de que la mutación sea reconocida por la vacuna,

[6]La vacuna es una secuencia de ADN del virus que se esté analizando, que se utiliza para realizar el proceso de selección sobre las nuevas mutaciones

entonces será eliminada definitivamente y, por supuesto, no seguirá generando descendientes. De lo contrario, es decir, si la nueva secuencia no es reconocida por la vacuna, se agregará a la población de las secuencias, tal y como se ha descrito en el paso 3 anterior. La simulación computacional realiza la selección de las secuencias por la vacuna, mediante el análisis de regresión lineal. Este método estadístico se aplica en este caso de la forma siguiente:

- Transformar la secuencia ADN mutante a la secuencia de aminoácidos correspondiente, basado en el código genético, convirtiéndose ésta posteriormente en un vector de valores reales obtenido a partir de las energías de los aminoácidos.
- Crear un vector con las medias aritméticas de esos valores reales teniendo en cuenta el tamaño de la ventana elegida de forma aleatoria.
- Realizar el mismo proceso con la vacuna para obtener otro vector de medias.
- Aplicar el modelo de regresión lineal simple sobre estos vectores reales dado que la variable dependiente (denotada por $\boldsymbol{Y}$) corresponde a la media de la secuencia mutante, y la variable independiente (denotada por $\boldsymbol{X}$) es correspondiente a la media de la secuencia vacuna.

Al encontrar el modelo de regresión adecuado, podemos determinar los residuos del mismo, y luego estudentizarlos para buscar los *outliers*. Estos últimos nos sirven para confirmar el hecho de que la mutación se reconociera o no por la vacuna. Si no existen *outliers* decimos que esta última no reconoce la secuencia mutante, de lo contrario se dice que sí y la misma queda eliminada. Con todo lo que se ha explicado podemos describir el algoritmo en los siguientes pasos:

1. Especificar la cantidad de descendientes que pudiera tener el ancestro.
2. Escoger aleatoriamente a partir de la distribución de frecuencia de las secuencias un ancestro, luego generar una mutación del mismo aplicando la estrategia MCMC.
3. Comprobar que no existan codones de parada en la mutación obtenida, en caso de exitir realizar las operaciones necesarias para eliminar los mismos y obtener la secuencia mutante sin *stop codons*.
4. Transformar la secuencia de ADN mutante y la ancestral (o la vacuna si esta existe) en las secuencias de aminoácidos correspondientes, luego convertirlas en los vectores de energías para calcular después los vectores de medias conociendo el tamaño del desplazamiento de la ventana.
5. Aplicar la técnica del modelo de regresión lineal para encontrar los residuos. Al estudentizar estos últimos, se obtendrán los residuos estudentizados que serán usados para encontrar los posibles *outliers*. La presencia de cuatro o más *outliers* es el criterio tomado para aceptar en la población la nueva variante mutacional. En el caso de aceptarlo, se continúa con el paso siguiente. De lo contrario se elimina la variante y se vuelve al paso 2.

6. Comparar la secuencia mutante generada con el ancestro y con el resto de las otras secuencias; en caso de que la misma sea igual que uno de ellos, se aumentará el contador de este en 1, de lo contrario se agregará la mutación a la población como una nueva clase del virus y se aumentará el número de clases en 1. En los dos casos la cantidad total de secuencias de la población se incrementa en una unidad.
7. El ancestro actual se traslada a la secuencia recién creada y el algoritmo vuelve al paso 2 en caso que no se haya acabado la cantidad de descendientes generados del primer ancestro; de lo contrario se mueve al paso siguiente.
8. Si el número de posibles ancestros no se ha rebasado, seleccionar el nuevo ancestro entre las secuencias de la población inicial disponible, de forma independiente de la selección de los otros ancestros y volver al paso 1; de otro modo se mueve al paso siguiente.
9. Terminar el algoritmo.

El procedimiento anterior describe la evolución de las poblaciones con la técnica de selección, de modo que no todas las secuencias mutantes generadas se agregan a la población actual, sino que se escogen solamente las que satisfagan la condición de no reconocerse por la vacuna, o sea, las secuencias resistentes a la misma. La operación de selección es un fenómeno común en la naturaleza y en la vida real, en el cual solo los individuos resistentes pueden sobrevivir y adaptarse a los cambios frecuentes del ambiente. Es evidente que el resultado obtenido mediante este proceso de generación de mutaciones, en el que se incluye la restricción selectiva, difiere de la que no tiene en cuenta la restricción en el hecho de que el tiempo de ejecución crece y el tamaño de la población final es reducido (Minh, 2010).

Algoritmo 2.3 Clasificar una secuencia a partir de una vacuna.

Entrada: secuencia mutante **M**, secuencia vacuna **V**.
Salida: Acepta o no a **M** como posible mutación.

```
1: R = standard_resid(V,M);
2: V_R = aminoacids_means(V);
3:
4: if (student_resid(V_R, R)) then
5:     Aceptar la secuencia
6: else
7:     No aceptar la secuencia obtenida como una mutación
8: end if
```

En el algoritmo (2.3) tenemos el pseudo-código que describe los pasos mediante los cuales se clasifica una secuencia, en el mismo se utilizan varias funciones auxiliares, una de ellas es **aminoacids_means** la cual recibe una secuencia de ADN y retorna un vector de valores reales, calculados utilizando las energías de los aminoácidos que conformaban la secuencia.

También se utiliza la función **standard_resid** y su tarea es recibir **2** secuencias de ADN y obtener los residuos estandarizados[7] relativos al modelo de regresión lineal que se obtiene a partir de los vectores de medias de los aminoácidos de cada una de las secuencias, el algoritmo que obtiene estos residuos es el (2.4), luego que se tienen los mismos entonces utilizando la función **student_resid** se estudentizan y en la misma se chequea si existen o no valores *outliers*, para luego proceder a realizar la clasificación de la secuencia; el algoritmo (2.5) contiene los pasos a seguir durante este último proceso de clasificación.

[7]La teoría concerniente a estos se expuso en la subsección (1.4.2), ahí podemos encontrar también los fundamentos principales de los residuos estudentizados.

Algoritmo 2.4 Para hallar los residuos estándares de un modelo de regresión lineal

Entrada: secuencia mutante **M**, secuencia vacuna **V**.
Salida: Vector de residuos estandarizados $\boldsymbol{R}$.

1: $\boldsymbol{X}$ = **aminoacids_means**($\boldsymbol{V}$);
2: $\boldsymbol{Y}$ = **aminoacids_means**($\boldsymbol{M}$);
3: Declarar e inicializar en **0** a sumX, sumY, sumXX, sumYY, sumXY;
4:
5: **for** $\boldsymbol{i = 1}$ hasta $\boldsymbol{n - 1}$ **do**
6: sumX += $\boldsymbol{X_i}$;
7: sumY += $\boldsymbol{Y_i}$;
8: sumXX += $\boldsymbol{(X_i)^2}$;
9: sumYY += $\boldsymbol{(Y_i)^2}$;
10: sumXY += $\boldsymbol{X_i * Y_i}$;
11: **end for**
12:
13: yBar = sumY/n;
14: xBar = sumX/n;
15: Syy = sumYY-sumY*sumY/n;
16: Sxy = sumXY-sumX*sumY/n;
17:
18: slope = Sxy/Sxx;
19: intercept = yBar - xBar*slope;
20: r = Sxy***sqrt**(Syy*Sxx);
21:
22: **for** $\boldsymbol{i = 1}$ hasta $\boldsymbol{n - 1}$ **do**
23: yobs = slope*$\boldsymbol{X_i}$ + intercept;
24: $\boldsymbol{R_i}$ = $\boldsymbol{Y_i}$ - yobs;
25: **end for**
26: **return** $\boldsymbol{R_i}$;

Algoritmo 2.5 Estudentiza un conjunto de residuos estandarizados y clasifica

Entrada: Vector de residuos estandarizados $\boldsymbol{R}$, vector de medias $\boldsymbol{V}$ realtivo a la vacuna.
Salida: **true** o **false**.

1: Declarar matriz $\boldsymbol{M}$ de $\boldsymbol{n}$x**2** dimensiones;
2: **for** $\boldsymbol{i = 1}$ hasta $\boldsymbol{n - 1}$ **do**
3: $\quad \boldsymbol{M_{i,0} = 1}$;
4: $\quad \boldsymbol{M_{i,1} = X_i}$;
5: **end for**
6:
7: Declarar matriz $\boldsymbol{P}$ de **2**x**2** dimensiones;
8: $\boldsymbol{P = (M^T)M}$;
9: $\boldsymbol{P^{-1}} = \textbf{invert}(\boldsymbol{P})$;
10:
11: Declarar matriz $\boldsymbol{H}$ de $\boldsymbol{n}$x$\boldsymbol{n}$ dimensiones;
12: $\boldsymbol{H = X(P^{-1})X^T}$;
13:
14: $\boldsymbol{sum = 0}$;
15: **for** $\boldsymbol{i = 1}$ hasta $\boldsymbol{n - 1}$ **do**
16: $\quad \boldsymbol{sum += (R_i)^2}$;
17: **end for**
18: $\boldsymbol{var\_est} = \sqrt{\dfrac{\boldsymbol{sum}}{\boldsymbol{n - 2}}}$;
19:
20: $\boldsymbol{c = 0}$;
21: **for** $\boldsymbol{i = 1}$ hasta $\boldsymbol{n - 1}$ **do**
22: $\quad \boldsymbol{rs} = \dfrac{\boldsymbol{R_i}}{(\boldsymbol{var\_est})\sqrt{\boldsymbol{1 - H_{i,i}}}}$;
23:
24: $\quad$ **if** $(|\boldsymbol{rs}| > \boldsymbol{2})$ **then**
25: $\quad\quad \boldsymbol{c++}$;
26: $\quad$ **end if**
27: **end for**
28:
29: **if** $(\boldsymbol{c >= 4})$ **then**
30: $\quad$ **return true**;
31: **else**
32: $\quad$ **return false**;
33: **end if**

2.3 Implementación secuencial en C++ de la simulación de poblaciones virales

C++ es un lenguaje orientado a objetos derivado del C que fue diseñado a mediados de los años 80 por Bjarne Stroustrup. La intención de su creación fue extender el exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un

lenguaje híbrido. Posteriormente se añadieron facilidades de programación genérica, la cual se sumó a los otros dos paradigmas que ya estaban admitidos (programación estructurada y programación orientada a objetos, POO). Por esto se suele decir que el C++ es un lenguaje de programación multiparadigma y no es un lenguaje orientado a objetos puro. Más bien se trataba de añadirle "objetos" al clásico C , ya que el nuevo paradigma de programación "con objetos", se mostraba como un paso adelante en el arte de la programación. En el diseño de la Biblioteca Estándar C++, se ha usado ampliamente la dualidad de mezcla de un lenguaje tradicional con elementos de POO, lo que ha permitido un modelo muy avanzado de programación extraordinariamente flexible (programación genérica). Desde luego, C++ es un lenguaje de programación extremadamente largo y complejo; sin embargo, ha experimentado un extraordinario éxito desde su creación. De hecho, muchos sistemas operativos, compiladores e intérpretes han sido escritos en C++ (Minh, 2010). Una de las razones de su éxito es ser un lenguaje de propósito general que se adapta a múltiples situaciones. Para comprobar el éxito e importancia de los desarrollos realizados en C++ se puede visitar la página que mantiene su fundador en: www.research.att.com. En la Fig. 2.1 se puede apreciar el flujo de trabajo para desarrollar la simulación, basándonos en este esquema se construye la estructura de clases de la aplicación.

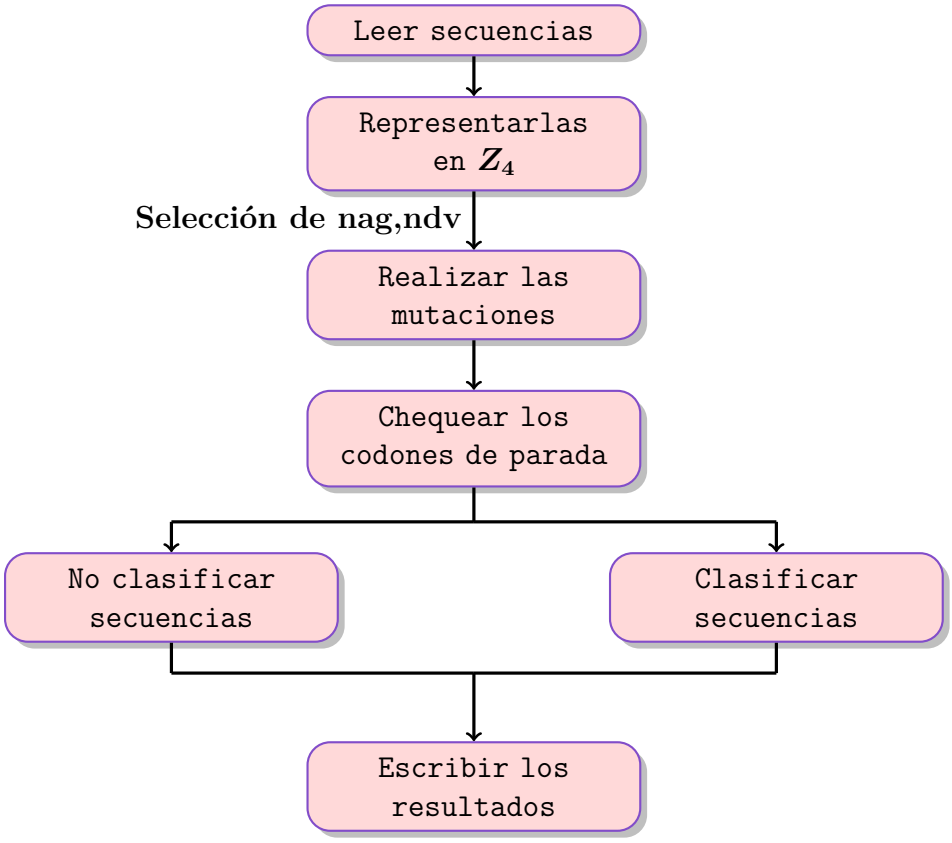

Figura 2.1: Esquema general del método propuesto para la simulación de la evolución de poblaciones virales.

2.3.1 Diseño del diagrama de clases y relaciones entre las mismas para simular la evolución de las secuencias sin selección

El algoritmo de simulación de la evolución se ha construido a partir de un grupo de clases basándonos en la técnica orientada a objetos. Cada clase se comporta como una entidad caracterizada por sus propios atributos y operaciones y realiza su función en conjunto con otras clases del diseño. En la Fig. 2.2 se muestra el modelo del diseño utilizado para la aplicación; en este se presentan las principales clases así como las relaciones existentes entre las mismas. El esquema consta de los 3 grupos[8] de clases siguientes:

1. El primer grupo está compuesto por las clases consideradas como básicas que se definen en el modelo. Para realizar las tareas simples del procesamiento de las secuencias de ADN contamos con las clases **AA_Sequence** y **DNA_Sequence**. También tenemos aquí la clase **Header**, la cual tiene como objetivo servir de encabezado común para todas las demás entidades del modelo; en la misma se encuentran almacenados los principales archivos cabeceras necesarios en el la aplicación, dentro de estos archivos tenemos la

[8]Las clases aquí son agrupadas teniendo en cuenta su funcionalidad

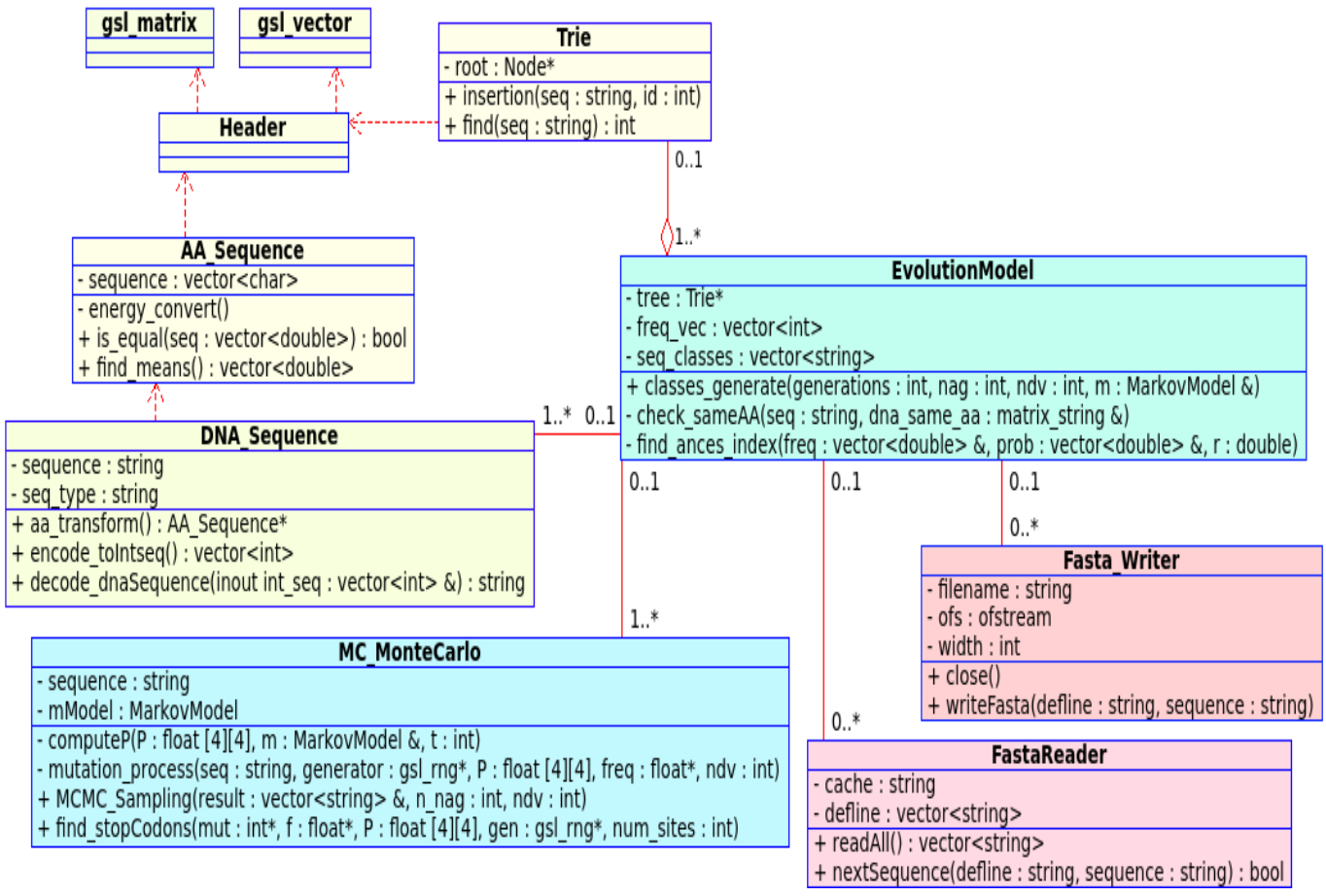

Figura 2.2: Diagrama de clases sin presión selectiva.

inclusión de los concernientes a la *Gnu Scientific Library*(GSL), que son utilizados en la aplicación para complementar los algoritmos implementados en otras clases facilitando el trabajo algebraico necesario. Para mayor información sobre esta biblioteca se puede consultar la dirección www.gnu.org/software/gsl/manual. Es fundamental también la clase **Trie** cuyo objetivo no es más que brindarnos las funcionalidades de la estructura de datos "trie" explicada en la sección 1.6.

2. El segundo grupo lo conforman aquellas clases que llevan a cabo la tarea principal de la aplicación que es obtener las mutaciones, el mismo está compuesto por: **MC_MonteCarlo** y **EvolutionModel**. La función de **EvolutionModel** es recibir del programa principal (main) los parámetros de la aplicación, entre ellos: el número de generaciones, total de ancestros por cada generación, la base de casos, entre otros. La misma procesa estos parámetros, obtiene aleatoriamente un ancestro y le envía la secuencia ancestral a **MC_MonteCarlo** para que obtenga las mutaciones; luego que ya se tienen las mutaciones esta se encarga de realizar todo el procedimiento de almacenamiento de los resultados en los archivos de salida y también actualiza el conjunto de secuencias disponibles, además del vector de frecuencias de aparición; en síntesis, su tarea es preparar lo necesario para la mutación y procesar el resultado obtenido al mutar. Una de las funcionalidades más importantes de esta clase es clasificar las mutaciones de acuerdo a las proteínas que estas representan, de forma tal que si existen 2 secuencias distintas que sintetizan una misma proteína entonces dichas mutaciones forman parte del conjunto representado por la proteína codificada. Por ejemplo, supongamos que nuestras secuencias están conformadas por un solo aminoácido; es decir, tienen solamente 3 bases, entonces podemos tener como mutaciones a UUA y UUG representando ambas al mismo aminoácido y por tanto a la misma proteína, en este caso entonces utilizando un vector se almacena una de las 2 mutaciones como representante de la proteína, la otra se inserta en la estructura a continuación de la anterior, y así ocurrirá con las demás mutaciones obtenidas que se ajusten a una misma codificación. Por otro lado, la clase **MC_MonteCarlo** es la encargada de mutar aplicando el algoritmo (2.1) una secuencia ancestral, tantas veces como el valor especificado por número de descendientes por virus; esta se encarga de calcular en cada mutación la matriz de probabilidades de transición implementando lo expuesto en la sección (2.1.1). También el procedimiento de encontrar los codones de parada de una mutación descrito en el algoritmo (2.2) se lleva a cabo en **MC_MonteCarlo**. La tarea de la misma termina cuando ya se han procesado todos los descendientes del ancestro, retornando los mismos a **EvolutionModel** para ser procesados.

3. El tercer grupo del sistema es responsable de realizar los procesos de leer y escribir datos desde y hacia los archivos. Para ello, se han usado las clases **FastaReader** y **FastaWriter** que heredan de las clases superiores **FileReader** y **FileWriter**, respecti-

vamente. La primera es usada para la lectura mientras que la segunda se utiliza para la escritura. Ambas clases son típicas en el trabajo con los archivos en formato Fasta[9], que se caracterizan por la estructura de almacenamiento de las secuencias de ADN, especificando una cabecera junto con el alineamiento completo para cada una de ellas. La herencia de las clases superiores es útil en el caso de realizar trabajos simples con los archivos como el chequeo de espacios en blanco, recuperación de las líneas y cerrado de archivos, etc. Las clases **FastaReader** y **FastaWriter** forman parte de la colección de clases de la biblioteca BOOM (*Bioinformatics Object-Oriented Modules*) que el lector podría encontrar en la página http://www.bioinformatics.org/project/?group_id=466.

2.3.2 Diseño del diagrama de clases y relaciones entre las mismas para simular la evolución de las secuencias con selección

El procedimiento de selección de secuencias que satisfagan ciertas condiciones, se agrega al modelo a la hora de chequear la resistencia de las mutaciones contra la vacuna. Para definir una secuencia como vacuna, se escoge entre los mutantes generados aquella secuencia que, al ser comparada con el ancestro en cuanto a características físico-químicas mediante el análisis de regresión lineal, no presente *outliers*. La misma es aplicada en la selección de las secuencias descendientes con el propósito de determinar cuáles de estas últimas se eliminan y cuáles son resistentes, de modo que se añaden como nuevos individuos a la población las mutaciones que presentan resistencia a la vacuna. El esquema de la Fig. 2.3 muestra el modelo evolutivo completo en el que se adiciona la restricción de selección de las secuencias de ADN mutantes. Al igual que en el caso anteriormente presentado, este modelo se mantiene igual en cuanto a la estructura del diseño algorítmico, excepto que ahora además de los 3 bloques explicados anteriormente tenemos un bloque adicional que contiene las clases relativas al análisis de regresión lineal. Este nuevo bloque se compone por las clases siguientes **Regression**, **LinRegressor** y **LinearFunc**. De estas nuevas 3 clases las 2 últimas se reimplementaron tomando como base 2 clases de la biblioteca BOOM mencionada anteriormente. La clase **Regression** tiene como funcionalidades principales encontrar el modelo de regresión lineal a partir de 2 conjuntos de valores; es decir, construir la recta de regresión y obtener los valores aproximados de $\boldsymbol{Y}$, luego la clase se encarga de encontrar los residuos del modelo, estandarizarlos y estudentizarlos para luego encontrar los *outliers* y realizar el procedimiento de casificación. Para realizar todas las tareas anteriores dicha clase utiliza a las restantes clases del bloque. Al igual que en el esquema anterior, se utilizaron coloraciones distintas para la representación gráfica de cada uno de los grupos de entidades. Esta forma de dividir trabajos, basada en el uso de clases y objetos, solamente la podría tener un lenguaje orientado a objetos. El mejor y más poderoso de los que hemos

[9]Este es un formato de fichero informático basado en texto, utilizado para representar secuencias de ADN.

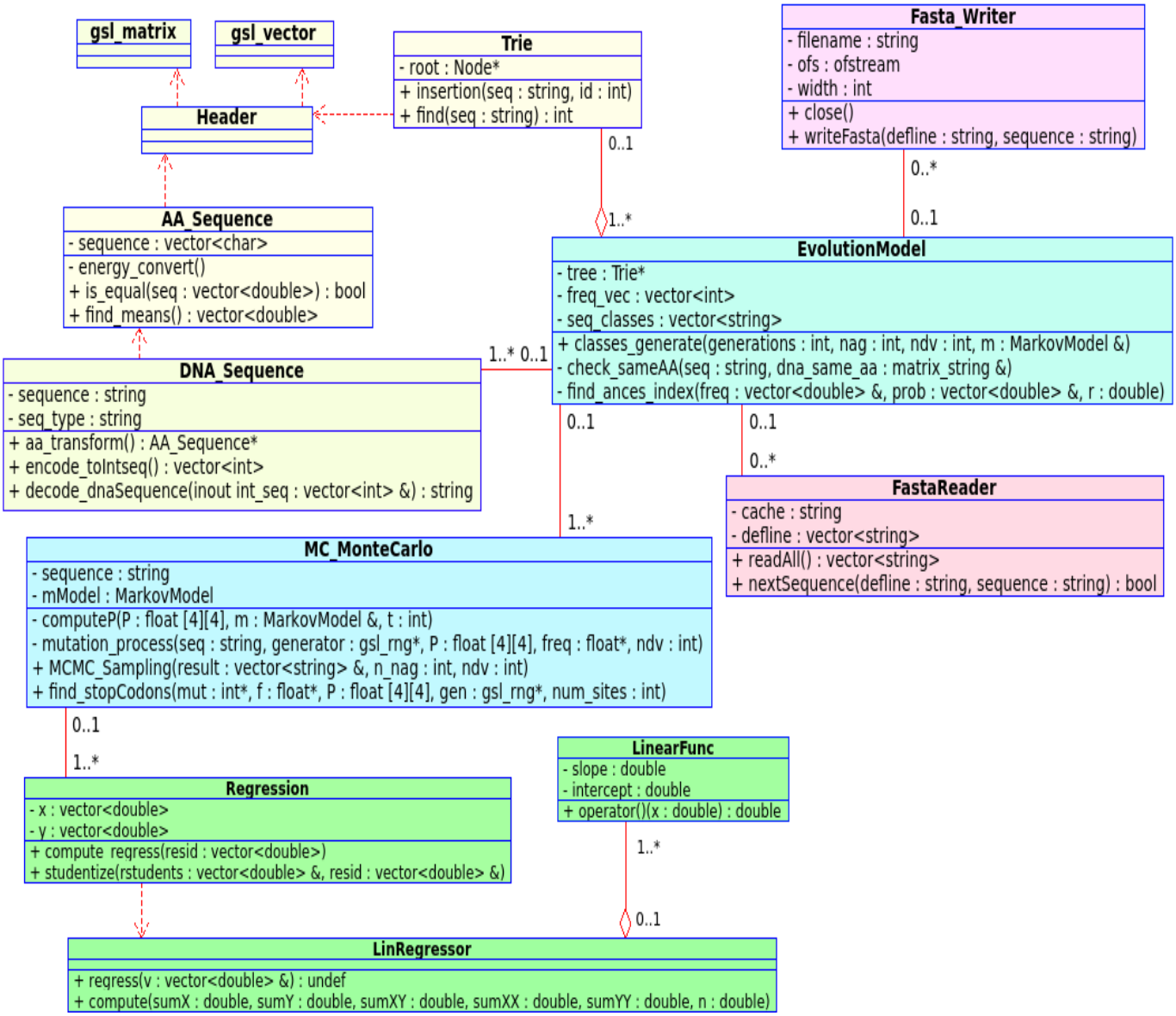

Figura 2.3: Diagrama de clases con presión selectiva.

usado, el C++, nos ha facilitado muy eficientemente la programación y en el uso de las bibliotecas estándares disponibles del lenguaje. Además de estas ventajas otra conveniencia que podemos tener al utilizar el C++ para desarrollar la aplicación es la total compatibilidad existente entre este y CUDA, siendo por lo tanto mucho menor el trabajo a realizar durante la paralelización del procedimiento pudiéndose reutilizar gran parte del código empleado en la versión secuencial.

2.4 Implementación paralela utilizando CUDA de la simulación de las poblaciones virales

Como hemos dicho con anterioridad el proceso de generar mutaciones de una secuencia genética puede ser en ocasiones muy complejo computacionalmente, y por tanto una aplicación que lleve a cabo el mismo tendría un elevado tiempo de ejecución. En la sección anterior se ha expuesto una manera de desarrollar una aplicación que lleve a cabo el mecanismo evolutivo de forma secuencial, aquí se expondrá como obtener una solución paralela del problema utilizando CUDA, con el objetivo de lograr un menor tiempo de ejecución que la versión anterior.

2.4.1 Estrategias de solución paralelas

Luego de ejecutar la versión secuencial con distintos parámetros de entradas ha sido posible analizar exhaustivamente el tiempo de ejecución de la aplicación. El principal objetivo de este análisis es detectar los momentos del proceso que consumen la mayor parte del tiempo de ejecución. Los procedimientos de mayor consumo temporal que se identificaron fueron la mutación y la clasificación de las secuencias de acuerdo a las proteínas que estas representaban. Esta última funcionalidad es la que mayor tiempo consumía y de una generación a otra aumentaba el tiempo que esta tarda en realizarse, debido a que se van añadiendo las nuevas mutaciones a la base lo cual implica un mayor número de comparaciones. En la Fig. 2.4 se muestra el flujo de trabajo general llevado a cabo de forma paralela. Las estrategias seguidas para paralelizar estas funciones se exponen a continuación.

Paralelización de las mutaciones

Las mutaciones como se ha mencionado previamente en el trabajo se producen en cada uno de los sitios de las secuencias, teniendo como base inicial de un sitio los nucleótidos que componen el ancestro que se esté mutando. Estos sitios mutan de forma independiente, es decir que las mutaciones que ocurran en uno de ellos no influyen de ninguna manera en las que pueden tener lugar en los otros. Esta independencia es una característica que facilita directamente el proceso de paralelización, pues la misma cumple con las características necesarias para la aplicación de la técnica *SIMD (Single Instructions Multiple Data)*. Dentro de la computación, *SIMD* es una técnica para conseguir paralelismo a nivel de datos. Los repertorios de este tipo consisten en instrucciones que aplican una misma operación sobre un conjunto más o menos grande de datos. Es una organización en donde una única unidad de control común despacha las instrucciones a diferentes unidades de procesamiento. Todas estas

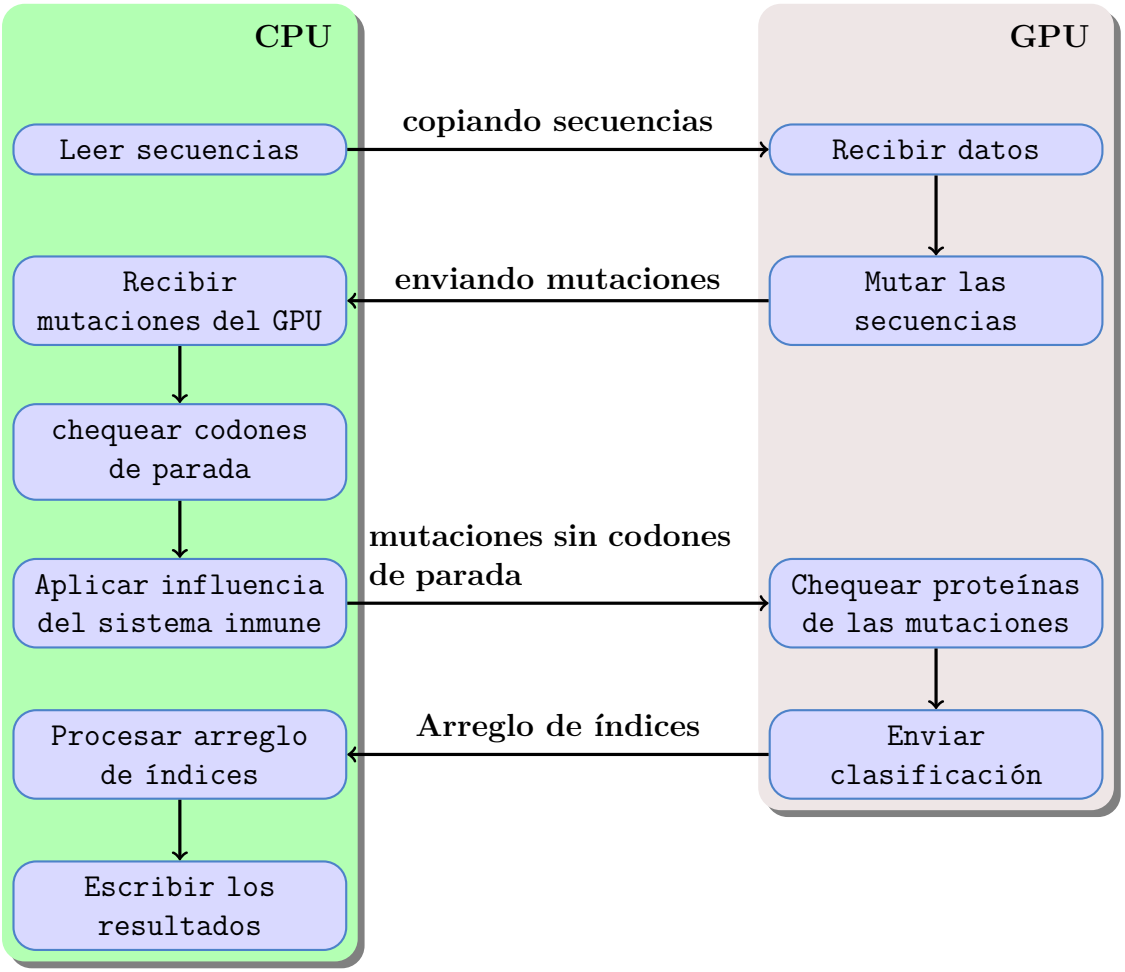

Figura 2.4: Esquema general para la paralelización de la evolución de poblaciones virales.

reciben la misma instrucción, pero operan sobre diferentes conjuntos de datos. Es decir, la misma instrucción es ejecutada de manera síncrona por todas las unidades de procesamiento (Grama and Gupta, 2003). Aquí, como tenemos que el proceso de mutar un sitio es invariable, y para llevarlo a cabo solo nos interesa la base nitrogenada que posea inicialmente el ancestro y el sitio en el que nos encontremos, entonces programando dicho procedimiento como una operación y aplicándola a más de un sitio durante una ejecución obtenemos la paralelización deseada.

En la figura 2.5 tenemos un esquema que muestra gráficamente la idea que se ha seguido para lograr el paralelismo a la hora de obtener las mutaciones.

La implementación desarrollada en CUDA de esta funcionalidad se hizo en un kernel ubicado en la clase **MC_MonteCarlo**. La secuencias se modelaron como arreglos de tipo *char* linealizados, pues al solo contar con 4 bases se desaprovecharía demasiado espacio si para un sitio se utilizara el tipo de dato *short*[10] para almacenar la codificación de dicho nucleótido; esta variante posee un menor consumo en memoria, pero el procesamiento de un sitio necesita

[10]Este tipo de dato tiene un tamaño de 16 bits siendo capaz de almacenar $\mathbf{2^{16}}$ números.

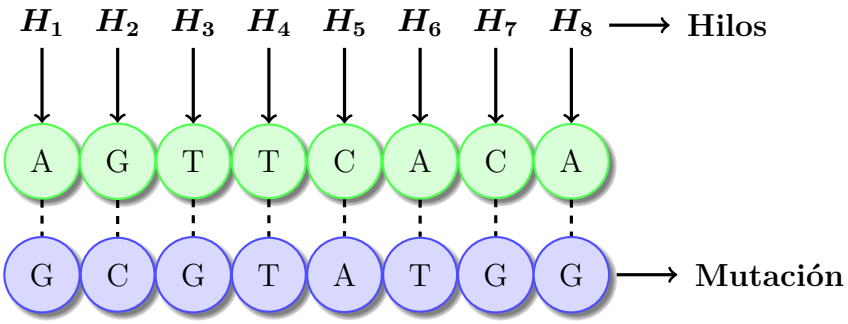

Figura 2.5: Proceso de mutación paralelizado por sitio

de más operaciones debido a que es necesario realizar conversiones de las bases a su respectiva codificación númerica antes de mutar las mismas, y después de obtener la mutación el resultado debe ser codificado como un caracter. Si analizamos el algoritmo 2.1 podemos ver que las mutaciones se llevan a cabo de forma tal que se obtiene una mutación en cada iteración del algoritmo; por tanto, si necesitamos obtener 10 descendientes, el mismo debe ejecutarse 10 ocasiones. En la implementación paralela el algoritmo recibe todos los ancestros de una generación y obtiene todas las mutaciones de estos, dicha tarea se logra en un solo paso.

El pseudo-código 2.6 contiene el procedimiento de las mutaciones paralelizado. Este algoritmo utiliza otras funciones definidas anteriormente como, por ejemplo, la función mutación, la cual se corresponde con el pseudo-código 2.1 explicado en la sección 2.1. Otros métodos como el de codificar las bases no se exponen de forma explícita, pues su tarea es bastante trivial, debido a que no es más que llevar de las bases A, C, G, T a los números 0, 1, 2, 3 y viceversa. Los valores aleatorios $\boldsymbol{u}, \boldsymbol{v}$ utilizados en el algoritmo se generan dentro del kernel a partir de una distribución uniforme, utilizando la biblioteca CURAND mencionada en 1.7. La sincronización de los hilos se logra utilizando **__syncthreads()**, la misma pertenece al conjunto de funciones propias de CUDA. Dicha sincronización es necesaria ya que el encargado en cada bloque del cálculo de la matriz $\boldsymbol{P}$ es el hilo con índice **0**, por tanto los demás hilos del bloque deben esperar a que este coloque dicha matriz en memoria compartida para luego comenzar las mutaciones. Esta forma de obtener la matriz de probabilidades de transición, a pesar de contener divergencia[11] entre los hilos de un bloque, conduce a un menor consumo temporal gracias a la rapidez con la que se producen los accesos a la memoria compartida por parte de los hilos.

[11]Significa que todos los hilos no ejecutan las mismas instrucciones, es decir algunos realizan tareas que otros nunca realizarán.

Algoritmo 2.6 Algoritmo paralelo para realizar las mutaciones

Entrada: arreglo de ancestros **A**, arreglo para almacenar las mutaciones **M**, número de ancestros **nag**, número de descendiente por cada ancestro **ndv**, número de sitios **N**, frecuencias de aparición **F**, **alpha1**, **alpha2**, **beta**.

Salida: Mutaciones almacenadas en **M**.

```
1: Declarar en memoria compartida arreglo P con 16 posiciones;
2:
3: tid ← id_hilo + id_Bloque * size_Bloque;
4: Declarar u,v;
5:
6: while (tid < N) do
7:   for (i = 0 hasta nag) do
8:     pos ← i*nag+tid;
9:     c ← A[pos];
10:    old ← convert_base(c, 0);
11:    j ← 0;
12:    Declarar r;
13:
14:    repeat
15:      if (id_hilo = 0) then
16:        computeP(P, beta, alpha1, alpha2, F, pos);
17:      end if
18:
19:      Sincronizar_Hilos;
20:      u ← U(0, 1);
21:      v ← U(0, 1);
22:      Declarar new;
23:
24:      mutation(P, old, new, u, v, F);
25:      r ← convert_base(new, 1);
26:      M[i*N*ndv+N*j + tid] = r;
27:      old ← new;
28:
29:      j++;
30:    until (j < ndv)
31:  end for
32:
33:  tid ← tid + size_Malla * size_Bloque;
34:
35: end while
```

Clasificación de las mutaciones mediante sus proteínas

Previamente, mencionabamos el proceso de clasificación que se realizaba con las mutaciones obtenidas en una generación. Este se llevaba a cabo con el objetivo de detectar cúando 2

secuencias distintas sintetizaban una misma proteína. En la versión secuencial esto se hace comparando cada secuencia obtenida con cada una de las que ya se encuentran almacenadas. En dicha tarea se tienen en cuenta tanto las mutaciones que han sido generadas como las secuencias iniciales. Si las mutaciones se están obteniendo sin presión selectiva la base de casos crece de una generación a otra mediante miles de nuevas mutaciones y además el algoritmo que logra esta funcionalidad tiene una complejidad polinomial $\boldsymbol{O(n^2)}$, por tanto la ejecución del mismo en la CPU consume una cantidad de tiempo considerable, siendo CUDA una alternativa de solución para reducir el tiempo de ejecución.

Para aplicar el paralelismo a este procedimiento de clasificación hemos utilizado un arreglo de índices ($\boldsymbol{I}$), el cual tendrá una posición por cada una de la secuencias obtenidas en una generación; es decir, que su longitud será igual a $\boldsymbol{nag * ndv}$, este es inicializado con $\boldsymbol{-100}$ en todas sus posiciones.
En los sitios de este arreglo se tendrá $\boldsymbol{-100}$ en caso de que la correspondiente secuencia no sintetice una proteína obtenida anteriormente, un valor del intervalo $\boldsymbol{[0, nag - 1]}$ si la misma produce una proteína igual que algunas de las correspondientes a los ancestros o el mismo estará en el intervalo $\boldsymbol{[nag, nag + nag * ndv - 1]}$ si la proteína se corresponde con algunas de las mutaciones que se generaron junto con ella en la actual generación.

En el algoritmo se asigna un hilo a cada una de las secuencias; es decir, de forma diferente a como se paralelizaron las mutaciones, en las que un hilo era responsable de la mutación de un sitio. Luego que el hilo sabe que mutación le corresponde entonces este realiza la comparación entre su respectiva secuencia y todos los ancestros, si la misma no es clasificada por ningún ancestro se procede a compararla con cada una de las mutaciones que se obtuvieron de forma posterior a ella, esto último se hace buscando las que son clasificadas por ella.
Suponiendo que estamos analizando la mutación $\boldsymbol{M_j}$, en caso de que la clasificación mediante los ancestros sea positiva para el ancestro $\boldsymbol{A_i}$ se asigna $\boldsymbol{i}$ a la posición $\boldsymbol{j}$ del arreglo que se tiene compartido, si la clasificación se lograra para algunas de las secuencias $\boldsymbol{M_k}$ con $\boldsymbol{k > j}$, entonces lo que hace el algoritmo es que en la posición correspondiente a la mutación $\boldsymbol{k}$ asigna $\boldsymbol{j}$. Notemos que durante el algoritmo si una posición cumple que $\boldsymbol{I_i = -100}$ cuando va a ser procesada, entonces dicho hilo pasa a procesar otra secuencia pues la actual ha sido clasificada.
Como se hace uso de un arreglo en memoria compartida y el mismo es escrito y leído por los hilos de forma simultánea, entonces este constituye una sección crítica, por tanto se hace uso de un semáforo para realizar tanto las lecturas como las escrituras de forma atómica. El algoritmo 2.7 contiene el pseudo-código que realiza el procedimiento descrito anteriormente.

Los arreglos que se muestran como parámetros de entrada se encuentran linealizados, en la implementación de la aplicación la conversión de los mismos a aminoácidos se realiza

Algoritmo 2.7 Algoritmo para clasificar las mutaciones

Entrada: aminoácidos de las mutaciones $\boldsymbol{M}$, aminoácidos de los ancestros $\boldsymbol{A}$, índices $\boldsymbol{I}$, **num_sitios**, **nag**, **ndv**

Salida: Arreglo de índices ($\boldsymbol{I}$) con valores de las clasificaciones.

```
1: codons ← num_sites/3;
2: tot_seqs ← nag * ndv;
3: tid ← id_hilo + id_Bloque * size_Bloque;
4: while (tid < tot_seqs) do
5:   if (I[tid] = -100) then
6:     flag ← false; i ← 0;
7:     while (!flag && i < nag) do
8:       j ← 0;
9:       while (j < codons) do
10:        if (A[i * codons + j] != S[tid * codons + j]) then
11:          break;
12:        end if
13:        j ← j + 1;
14:      end while
15:      if (j = codons) then
16:        flag ← true;
17:        if (I[tid] = -100) then
18:          I[tid] ← i;
19:        end if
20:      end if
21:      i ← i + 1;
22:    end while
23:    if (!flag && I[tid] = -100) then
24:      i ← tid + 1;
25:      while (i < tot_seqs) do
26:        j ← 0;
27:        while (j<codons) do
28:          if (S[tid * codons + j] != S[i * codons + j]) then
29:            break;
30:          end if
31:          j ← j + 1;
32:        end while
33:        if (j = codons && I[i] = -100) then
34:          I[i] ← nag + tid;
35:        end if
36:        i ← i + 1;
37:      end while
38:    end if
39:  end if
40:
41:  tid ← tid + size_Malla * size_Bloque;
42:
43: end while
```

dentro del algoritmo pero esto no se ha mostrado así por cuestiones relativas al diseño del documento.

No obstante, la idea central del método gira alrededor de los arreglos de aminoácidos.

2.4.2 Presión selectiva y paralelismo

El proceso de presión selectiva explicado en la sección 2.2.2 a pesar de ser costoso no se desarrolló en paralelo, debido a las características que posee el mismo. Una de las razones principales por la que se desarrolló esta funcionalidad de forma secuencial es la alta linealidad que encontramos dentro del procedimiento, como se explicó anteriormente si obtenemos una secuencia que no es reconocida por la vacuna con la cual se ha hecho el análisis de regresión lineal, entonces se añade la nueva secuencia a la población, se toma como ancestro y por tanto también como vacuna. Por lo expuesto podemos apreciar que no es correcto tomar un ancestro como vacuna común para todos sus descendientes, pues para saber con quién analizar la secuencia $\boldsymbol{S_k}$ debemos haber analizado todas las secuencias $\boldsymbol{S_0, S_1, \ldots, S_{k-1}}$, lo cual dificulta en gran medida la paralelización. Una posible solución a este inconveniente es tener un hilo por cada ancestro, de forma tal que este realize todos los chequeos necesarios entre el ancestro correspondiente y sus respectivas mutaciones, logrando de esta manera aplicar paralelismo al problema, pero como en cada generación de los experimentos realizados el número de ancestros se encuentra entre **2** y **5**, siendo este último entonces el máximo número de hilos posibles, esto implica que no se aprovechen las bondades que brindan las GPU en cuanto a capacidad de cálculo, ya que estas son capaces de lanzar miles de hilos en una ejecución.

Otras de las ventajas que ofrece una versión secuencial que realize la simulación de la influencia del sistema inmune en las mutaciones es la disponibilidad de la GSL. Se expuso en la subsección 1.4.2 que para llevar a cabo el cálculo de los residuos estudentizados de un modelo de regresión lineal, se deben utilizar elementos algebraicos como la inversa de una matriz y su traspuesta. Esta biblioteca mencionada con anterioridad en 1.7 nos brinda varios mecanismos del álgebra lineal, a partir de los cuales es posible lograr una correcta implementación de las funcionalidades deseadas. Por el contrario, en caso de desarrollar una alternativa paralela para esta tarea se tendría que implementar utilizando CUDA todas las funciones necesarias lograr una correcta clasificación, haciéndose entonces mucho más engorroso el desarrollo de la aplicación.

Por tanto, la etapa "Aplicar influencia del sistema inmune" mostrada en la Fig. 2.4, no se implementará de forma paralela utilizando CUDA. Esta característica es la que distingue a la variante que simula el proceso de selección respecto a su homologa sin selección.

2.5 Conclusiones parciales del capítulo

Se ha desarrollado una herramienta generadora de secuencias de ADN a partir de una población inicial de secuencias virales, basada en la técnica de simulación de Monte Carlo. Los algoritmos se implementan usando el paradigma de programación orientada a objetos y el lenguaje C++ es seleccionado para construir la aplicación. Las versiones paralelas se desarrollaron mediante el uso de este lenguaje en conjunto con CUDA. El estudio desarrollado permitirá simular la evolución de las poblaciones, tanto para el caso sin restricciones como para el caso en el que se aplican condiciones en el proceso de selección de las secuencias mutantes. Para ello se desarrollarán 2 aplicaciones, una de forma secuencial y otra paralela utilizando CUDA. El objetivo de realizar estas tareas es la comparación entre dos implementaciones de la simulación y el análisis de la diferencia entre ellas, las cuales serán presentadas y discutidas en el siguiente capítulo, mostrando los resultados obtenidos en los experimentos computacionales realizados usando el modelo propuesto sobre una base de datos real.

Capítulo 3

Resultados y Discusión

Los algoritmos expuestos en el capítulo anterior conducen a un procedimiento general (Fig. 2.4) para realizar la simulación del proceso evolutivo de una población de secuencias de ADN genómicas, tanto de forma secuencial como paralelamente. Para exponer las diferencias en cuanto al tiempo de ejecución que presentan los algoritmos desarrollados, se aplican los mismos a genomas de 2 subtipos del virus de la influenza humana A (H3N2 y H1N1). Los experimentos se realizaron en más de una tarjeta gráfica con el fin de mostrar la escalabilidad de la aplicación.

3.1 Características de software y hardware

La aplicación se desarrolló utilizando el *IDE*[1] ***Netbeans* 7.4**, sobre el sistema operativo ***Debian* 7.3**. Los experimentos se realizaron sobre una computadora con 3 GB de memoria RAM y un microprocesador **Intel Core 2 Quad** modelo **Q8200**. En estos se utilizaron 2 tarjetas de video cuyas especificaciones se muestran en la tabla 3.1. Como se observa, la primera de las 2 tarjetas posee una mayor capacidad de cómputo ya que contiene el cuádruple de núcleos CUDA que la segunda, por lo tanto teóricamente se espera que la aplicación demore como mínimo 4 veces más en la segunda tarjeta con respecto a la primera.

[1]Del inglés Integrate Development Environment

Procesador	Arquitectura	Memoria	Núcleos
GeForce GT 630	**GK107**	**2048 Mb**	**192**
GeForce GT 610	**GF119**	**1024 Mb**	**48**

Tabla 3.1: Especificaciones de las tarjetas gráficas.

3.2 Descripción de las bases de datos

Las bases de secuencias genómicas fueron obtenidas del Centro Nacional para la Información Biotecnológica de Estados Unidos[2], donde se encuentran disponibles de forma totalmente libre y gratuita. Los experimentos se realizaron sobre dos bases de casos. La primera de ellas contiene un alineamiento de secuencias de la variante H3N2 (archivo **FASTA_H3N2.fas**), conformada por un conjunto de 198 secuencias cada una con 1765 sitios, con la presencia de gaps[3] en 73 de ellos. El segundo alineamiento se compone de 2451 sequencias de la variante H1N1 (archivo **FASTA_USA.fas**), con 1803 sitios dentro de los cuales podemos encontrar 830 gaps. Con estas bases de casos se probaron las 2 aplicaciones desarrolladas (secuencial y paralela) con sus respectivas variantes (con selección y sin selección).

3.3 Discusión de los resultados experimentales

Las pruebas realizadas tienen como objetivos verificar las ventajas que nos proporciona CUDA en cuanto al tiempo de ejecución, mostrar si la aplicación paralela posee algún tipo de escalabilidad, además de ejemplificar como influye el número de mutaciones obtenidas por cada experimento en las ganancias, este último se logra mediante comparaciones entre 2 tablas referentes a una misma base de casos.

Los parámetros de las pruebas se tomaron de forma tal que el número de ancestros por generación (nag) varía su valor en el intervalo entero $[\mathbf{2, 5}]$, la cantidad de descendientes por ancestro (ndv) debe tomar uno de los valores $\mathbf{\{1000, 1500, 2000, 2500, 3000\}}$ y el número de generaciones(ngen) se establece en dependencia de la base de secuencias que se estén procesando (en los experimentos realizados se toman con valores 1 o 2)[4]; estos parámetros se tomaron de acuerdo a lo expuesto en (Minh, 2010).

[2]NCBI por sus siglas en inglés, disponible en web: http://www.ncbi.nlm.nih.gov/genomes/FLU/Database.

[3]Un gap es un sitio de una secuencia genómica que no se pudo determinar la base que le correspondía al mismo.

[4]Se limita a 2 este valor debido al tiempo excesivo que consume la aplicación secuencial.

En la tabla 3.2 se muestran los resultados obtenidos en las simulaciones para el alineamiento H3N2. En esta prueba se emplearon como parámetros los siguientes: una sola generación, 5 ancestros por cada generación y 1000 descendientes por cada ancestro. A partir de sus datos podemos verificar la diferencia de los tiempos de ejecución.

Variante	Experimento	Mutaciones	Tiempo	GVS
Sin selección	**secuencial**	**3751**	**2949.65s**	**–**
	paralelo 1	**4750**	**4.29s**	**687**
	paralelo 2	**4750**	**10.07s**	**292.91**
Con selección	**secuencial**	**2457**	**1265.05s**	**–**
	paralelo 1	**1715**	**19.25s**	**65**
	paralelo 2	**1700**	**29.35s**	**43.1**

Tabla 3.2: **Resultados del alineamiento H3N2 con ngen = 1, nag = 5, ndv = 1000.**

El significado de cada columna en la tabla anterior es el siguiente:

- **Variante**: Se refiere a si se utilizó o no el procedimiento de presión selectiva.
- **Experimento**: Especifica uno de los 3 tipos de experimentos siguientes.
  - Secuencial.
  - Paralelo 1: se emplea la tarjeta con 192 núcleos.
  - Paralelo 2: se utiliza la tarjeta gráfica que solo posee 48 núcleos.
- **Mutaciones**: Total de mutaciones obtenidas en cada experimento.
- **Tiempo**: Tiempo de ejecución de cada expermimento, dado en segundos.
- **GVS**: Ganancias con respecto a la versión secuencial.

En la tabla 3.3 se muestran los resultados de la segunda prueba realizada utilizando el alineamiento H3N2 donde los mismos parámetros de la prueba anteriror excepto que en este caso se simulan 2 generaciones.

Variante	Experimento	Mutaciones	Tiempo	GVS
Sin selección	**secuencial**	**5327**	**4943.70s**	**–**
	paralela 1	**9500**	**13.4s**	**368**
	paralela 2	**8581**	**29.92s**	**165.23**
Con selección	**secuencial**	**3517**	**2559.07s**	**–**
	paralela 1	**3439**	**40.91s**	**62.55**
	paralela 2	**2685**	**60.72s**	**42.14**

Tabla 3.3: **Resultados del alineamiento H3N2 con ngen = 2, nag = 5, ndv = 1000.**

En las 2 tablas se observa que el factor de escalibilidad[5] de la aplicación no es bueno. Por ejemplo para la variante con selección se obtuvo para una generación **1.51** y para 2 generaciones se tiene **1.48**.

Mediante una comparación entre las ganancias[6] de las tablas observamos que en caso de influir el sistema inmune en las mutaciones los valores de GVS se mantienen bastantes similares entre las 2 pruebas observándose solo una variación de **2.45** que puede ser considerada poco significativa, mientras que la comparación para el caso sin selección revela que la diferencia es mucho mayor siendo esta de **219**. Esto último se debe principalmente a la diferencia en el total de mutaciones obtenidas por cada uno de los experimentos de las diferentes variantes lo cual se muestra a continuación.

En la tabla 3.2 tenemos que la diferencia entre las mutaciones obtenidas en paralelo-1 y el secuencial es de **999**, mientras que durante la segunda generación el experimento paralelo-1 obtiene **4173** secuencias más que el secuencial, lo cual influye notablemente en el tiempo de ejecución y por tanto en el GVS obtenido.

La tabla 3.4 muestra los resultados de las simulaciones realizadas para el alineamiento H1N1. En esta prueba se emplearon como parámetros los siguientes: una sola generación, 4 ancestros por cada generación y 2000 descendientes por cada ancestro. Lo más importante es el factor GVS obtenido por los experimentos paralelos bajo la presión selectiva.

Variante	Experimento	Mutaciones	Tiempo	GVS
Sin selección	**secuencial**	**6002**	**3682.25s**	**–**
	paralela-1	**8000**	**4.95s**	**743, 88**
	paralela-2	**8000**	**11.25s**	**327, 31**
Con selección	**secuencial**	**76**	**13.29s**	**–**
	paralela-1	**836**	**10.77s**	**1, 23**
	paralela-2	**300**	**15.62s**	**0.85**

Tabla 3.4: **Resultados del alineamiento H1N1 con ngen = 1, nag = 4, ndv = 2000.**

La tabla 3.5 está conformada por los resultados de la segunda prueba realizada sobre el alineamiento H1N1. Los parámetros de esta prueba son iguales a los de la anterior con la diferencia de que se realizan 2 generaciones del procedimiento. Los resultados mostrados en esta última tabla son similares a los expuestos en la tabla 3.4 en cuanto al factor GVS. Podemos apreciar que para las variantes sin selección tenemos que el menor número de mutaciones obtenido entre los distintos experimentos es de **8539**.

[5]Se refiere a si la aplicación aprovecha las diferencias entre las tarjetas.
[6]Se hace referencia a las ganancias del experimento paralelo 1.

Variante	Experimento	Mutaciones	Tiempo	GVS
Sin selección	**secuencial**	**12003**	**14445.17s**	**–**
	paralela-1	**14048**	**14.64s**	**986, 69**
	paralela-2	**8539**	**34, 61s**	**417, 4**
Con selección	**secuencial**	**147**	**27.54s**	**–**
	paralela-1	**1196**	**20.87s**	**1.31**
	paralela-2	**608**	**34.29s**	**0.8**

Tabla 3.5: **Resultados del alineamiento H1N1 con ngen = 2, nag = 4, ndv = 2000.**

A partir de las 2 tablas anteriores es posible verificar la importancia del número de mutaciones en los resultados de las pruebas realizadas.

Primeramente, tenemos que cuando no se realiza el proceso de selección el valor de GVS obtenido es elevado siendo de **743, 88** para una generación y de **986, 69** para dos generaciones. La causa de que se obtengan ganancias tan elevadas es que se espera que en cada generación se generen **8000** secuencias debido a que tenemos **4** ancestros y **2000** descendientes por cada uno, como parámetros de las pruebas. A pesar de que se puedan desechar algunas mutaciones, porque ya se han obtenido con anterioridad, se tiene un número elevado de estas para ser procesadas en cada generación y como se había mencionado previamente un alto número de mutaciones obtenidas se traduce en grandes ganancias.

Por otro lado en dichas tablas apreciamos que para los experimentos de la variante con selección no se obtienen ganancias significativas, incluso el experimento paralelo 2 posee un mayor tiempo de ejecución que el secuencial, estas bajas ganancias se deben al total de mutaciones obtenidas por el experimento secuencial (76 mutaciones según se observa en la tabla 3.4), por tanto consume muy poco tiempo durante el procedimiento, mientras que los experimentos paralelos generan[7] **10** veces más secuencias que el secuencial y para la tabla 3.4 ocurre lo mismo manteniéndose así las bajas ganancias de una generación a otra. Lo expuesto, reafirma nuevamente la influencia directa de las mutaciones en las ganancias.

3.4 Conclusiones parciales del capítulo

Se han expuesto de forma explícita los resultados de cada una de las pruebas realizadas, tanto para la simulación del proceso evolutivo de las poblaciones virales en ausencia de restricciones, como para el caso de la simulación bajo presión selectiva. Tenemos que la aplicación secuencial consume más tiempo que la paralela en ambos casos, excepto en algunos

[7] Aquí se hace referencia a las mutaciones obtenidas por la versión paralela-1

casos donde incide la cantidad de mutaciones, esto se muestra al compararse las ganancias obtenidos por los experimentos paralelos sobre sus respectivas versiones secuenciales. El número de mutaciones obtenidas en cada una de las pruebas se aprecia que varía por la propia naturaleza estocástica de los procedimientos empleados.

Conclusiones

Las ideas que a continuación se expondrán resultan una síntesis de los aspectos fundamentales trabajados en la investigación.

- La descripción de las etapas que involucra el proceso de evolución de las poblaciones virales, según el modelo TN93, permitieron concebir diferentes algoritmos paralelos para el desarrollo de las mismas.
- La paralelización con CUDA de los algoritmos seleccionados en el proceso de evolutivo de secuencias genéticas sin la influencia del sistema inmune, mejora significativamente el tiempo de ejecución de la aplicación a la que fueron integrados, respecto a la versión secuencial existente.
- Los algoritmos propuestos fueron implementados según las características de la programación en CUDA, e integrados en una aplicación que realiza todas las fases del proceso de reconstrucción ancestral. Su ejecución sobre GPUs que son de más fácil acceso y menor costo frente a los *clusters*, lo hace un proceso más viable.
- Para la simulación de la evolución de poblaciones virales con la presencia de la presión selectiva del sistema inmune, existen bases de casos en las cuales simular utilizando paralelismo no presenta mejoras significativas con respecto a la versión secuencial.

Recomendaciones

- Agregar a la aplicación otros modelos evolutivos.
- Validar las mutaciones obtenidas por la aplicación a partir de la búsqueda de posibles genes homólogos con bases de casos existentes utilizando el programa BLAST.
- Investigar posibles técnicas de optimización aplicables a la solución propuesta en la paralelización de los diferentes algoritmos abordados, con el fin de mejorar la escalabilidad de la aplicación en las diferentes tarjetas gráficas.

Referencias Bibliográficas

Alba, R. F. (2013). Reconstrucción de secuencias genéticas utilizando cuda, *Universidad Central "Marta Abreu" de Las Villas* .

Alejo, A. C. (2012). Programación pararlela usando cuda: aplicación en la bioinfórmatica, *Universidad Central "Marta Abreu" de Las Villas* .

Bertsekas, D. P. & Tsitsiklis, J. N. (2000). *Introduction to Probability*, MIT, Cambridge, Massachusetts.

Cai, J. J., Smith, D. K., Xia, X. & Yuen, K.-y. (2006). Publication date: 06 feb 2007 evolutionary bioinformatics 2006: 2 179-182, *Evolutionary Bioinformatics* **2**: 179–182.

Crochemore, M., Hancart, C. & Lecroq, T. (2001). *Algorithms on Strings*, Cambridge University Press.

del Toro Melgarejo, L. F., Lío, D. G., , Brito, D. I. T. & Alejo, A. C. (2012). Acercamiento a la programación paralela utilizando cuda, *Technical report*, UCLV. ISBN 978-959-250-841-5.

E. Gultepe, M. (2005). Monte carlo simulation and statistical analysis of genetic information coding., *Science Direct* .

Galassi, M., Davies, J., Theiler, J., Gough, B., Alken, P. & Rossi, F. (2013). *GNU Scientific Library*. Reference Manual.

Gentle, J. E., Härdle, W. & Mori, Y. (2004). *Handbook of computational statistics*, Springer.

Grama, A. & Gupta, A. (2003). *Introduction to parallel computing*, Addison-Wesley.

J.Cai, J., K.Smith, D., Xia, X. & Yuen, K.-Y. (2006). *MBEToolbox 2.0: An enhanced version of a MATLAB toolbox for Molecular Biology and Evolution*, Evol Bioinform Online.

Minh, T. (2010). Simulación de la evolución de poblaciones del virus de la influenza a/h1n1, *Universidad Central "Marta Abreu" de Las Villas* .

Moreira, J. E., Sánchez, R., del Toro Melgarejo, L., del Carmen Chávez, M. & Grau, R. (2011). Parallel implementation of basic algorithms used in the phylogenetic analysis of sequences of the influenza a virus. h1n1, *Poster in CICI 2011, International Convention, Informatics 2011* . ISBN: 978-959-7213-01-7.

Nei, M. (1975). *Molecular population genetics and evolution*, North-Holland publishing company - Amsterdam, Oxford.

NVIDIA CUDA C Programming Guide 5.0 (2012). https://developer.nvidia.com.

Sánchez, R. & Grau, R. (2009). An algebraic hypothesis about the primeval genetic code architecture, *Mathematical Bioscienses* **221**.

Sánchez, R., Grau, R., del T. Melgarejo, L., Broche, J. E. M., del C. Chávez Cárdenas, M. & Min, T. N. T. (2011). Stocasthic simulation of influenza virus population, *Poster in CICI 2011, International Convention, Informatics 2011* . ISBN: 978-959-7213-01-7.

Spiegel, M. R., Schiller, J. & Srinivasan, R. A. (2009). *Probability and Statistics*, McGrawHill.

Suchard, M. A. & Rambaut, A. (2009). Many-core algorithms for statistical phylogenetics, *Journal Bioinformatics* **25**: 1370–1376.

Tamura, K. & Nei, M. (1993). Estimation of the number of nucleotide substitutions in the control region of mitochondrial dna in humans and chimpanzees, *Journal Bioinformatics* **10**: 512–526.

Walsh, B. (2004). *Markov Chain Monte Carlo and Gibbs Sampling*. EEB 581.

Yang, Z. (2006). *Computational Molecular Evolution*, Oxford University Press.