Generación de recomendaciones para terapias antivirales del VIH

Andrés Peralta Godoy y Ezequiel S. Velez

4 de agosto de 2010

Agradecimientos

A Daniel Gutson por su dirección en este trabajo y mentoría en C++.
A la Dra. Laura Alonso Alemany por su dirección
Al Dr. Roberto Daniel Rabinovich por encomendar y confiar en el desarrollo de su idea.
A los Dres. Pedro D'Argenio, Javier Blanco y Carlos Areces por conformar el tribunal evaluador.
A nuestras familias y amigos.

Agradecimientos especiales:

Al Ing. Gustavo Domingo Yagüez por su colaboración en C++.
Al Ing. Rafael Garabato por su instrucción en AutoTools.
Al Ing. Fabio Bustos por su colaboración en la revisión de la tesis.
A la Dra. María Pilar Adamo por colaborar en el armado de la exposición en la S.A.V.
A la Dra. Mercedes Cabrini por facilitarnos material referido a las terapias.
A los Colaboradores de Fu.De.P.A.N.

Síntesis

El presente trabajo expone el análisis, diseño y confección de un sistema de ayuda a la toma de decisiones en el tratamiento antirretroviral de pacientes VIH-positivos. El sistema desarrollado es altamente configurable, adaptándose a las características individuales de cada paciente, como por ejemplo la secuencia del virus del paciente o sus circunstancias particulares, por lo cual se trata de una implementación dentro del paradigma de medicina personalizada.

Este sistema intenta complementar las herramientas usadas por los profesionales médicos, proporcionando una exploración sistemática y exhaustiva de las diferentes opciones de terapias antirretrovirales aplicables a un paciente determinado. Cada terapia se evalúa de acuerdo a los criterios especificados por el profesional.

El sistema implementa diversos algoritmos para encontrar y evaluar las diferentes terapias aplicables a un paciente. El mismo es inheremente extensible, implementado de forma altamente modular y orientado a plugins. Este sistema sienta las bases para otros proyectos a futuro.

Este trabajo fue desarrollado en conjunto con Fu.De.P.A.N, ideado por el Dr. Daniel Rabinovich, y liberado bajo licencia GPL v3.0.

Índice general

Capítulo 1

Introducción

1.1. Introducción y Motivación

La bioinformática es una disciplina dedicada al análisis de elementos biológicos utilizando la informática como herramienta principal para generar simulaciones, probar teorías, o realizar cálculos mas complejos. En particular unas de las áreas estudiadas son la virología e infectología las cuales estudian la acción de un virus o infección en un organismo y cómo pueden ser aplicados ciertos fármacos para su tratamiento. Para este trabajo se abocará al tratamiento de pacientes VIH positivos proporcionando un sistema informático que simule la acción de antirretrovirales ante las distintas formas que posee el virus en cuestión. En la actualidad es de escasa existencia este tipo de software para fines de libre utilización, siendo los mas avanzados y complejos propiedad de grandes centros de investigación extranjeros y compañías farmacéuticas. Como resultado de varios meses de trabajo se obtuvo **A.S.O** (Antiviral Sequence Optimizer), un sistema altamente parametrizable y muy abarcativo para el estudio de terapias antirretrovirales para personas infectadas con VIH y que se podría convertir en una herramienta de soporte para los profesionales del área. Este sistema sigue el esquema de Medicina Personalizada y se desarrolla sobre la base de un trabajo previo *"Heterogeneidad en la distancia genética a resistencia en secuencias del HIV en pacientes vírgenes de tratamiento"* desarrollado por Gutson Daniel(FUDEPAN), Dario Dilernia(CNRS), Sanchez Eduardo(Fu.De.P.A.N), Gomez Carrillo Manuel(CNRS), Malbrán Francisco (Fu.De.P.A.N)y Rabinovich Roberto D.(CNRS).[18]

1.2. Usuarios del Sistema

Este sistema esta orientado principalmente, a profesionales del área de tratamiento de pacientes con VIH y de forma secundaria a desarrolladores informáticos con conocimientos en el área de la bioinformática y la virología. Podemos entonces, definir tres tipos de usuarios a quienes se destina el sistema:

1. Usuario Final o Aplicativo (UF): es el virólogo o profesional médico que utiliza el sistema para elaborar las terapias y aplicarlas en un paciente. Se limita al uso general de interacción con el sistema.

2. Usuario desarrollador de Plugins (UDP): definimos a este como el usuario que posea conocimientos en bioinformática, biología y programación y que se limita a desarrollar el funcionamiento del plugin.

3. Usuario desarrollador del Framework (UDF): este es un usuario mas avanzado cuya función es la de extender el framework. Se requieren conocimientos mucho más profundos en programación y técnicas computacionales.

1.3. Estructura del Documento

A continuación se listan los capítulos que siguen a esta introducción y un breve texto descriptivo para cada uno de ellos.

- Capítulo 2: Breve introducción de los conceptos utilizados a lo largo del trabajo, como ser conceptos biológicos y bioinformáticos, así como un panorama de la enfermedad del VIH.
- Capítulo 3: Presentación del problema, descripción de los métodos actuales para determinar terapias y problemas asociados.
- Capítulo 4: Conceptos preliminares de diseño que se utilizaron para el desarrollo del sistema.
- Capítulo 5: Diseño del sistema, decisiones de diseño y evolución del diseño desde los inicios del proceso de desarrollo hasta su terminación.
- Capítulo 6: Comprende a las clases y algoritmos utilizados para representar a los antirretrovirales.
- Capítulo 7: Algoritmos de generación para el espacio de búsqueda y fundamentos de las distintas alternativas.
- Capítulo 8: Algoritmos para combinar Antivirales y fundamentos de las distintas alternativas.
- Capítulo 9: Generación de terapias y resultado de la ejecución.
- Capítulo 10: Plugin, funcionalidad e interacción con el framework.
- Capítulo 11: Base de datos, formato y posibilidades de ampliación.
- Capítulo 12: Interfaz gráfica, descripción y funcionalidad.
- Capítulo 13: Conclusión del trabajo y trabajos futuros.
- Capítulo 14: Apéndice con definiciones y otros recursos.
- Capítulo 15: Bibliografía utilizada.

Capítulo 2

Conceptos Preliminares

2.1. Biología

Este concepto inicial es nuestro punto de partida para ingresar al tema que vamos a introducir. Definiciones varias caracterizan a la biología como el estudio del origen y evolución de los seres vivos así como sus propiedades. Entre estas características que se estudian son su morfogenia, reproducción, patologías,etc. Esta ciencia trata de establecer leyes generales que rigen la vida orgánica y sus fundamentos. Esta disciplina abarca muchas áreas de estudio, en nuestro caso nos interesaremos en el campo de la biología molecular.

2.1.1. Taxonomía Biológica

Comenzaremos definiendo la clasificación de los seres biológicos a través de un estudio de taxonomía el cual los clasifica según una jerarquía dada por su parentesco e historia evolutiva. La Fig. 2.1 muestra la clasificación taxonómica de las especies.

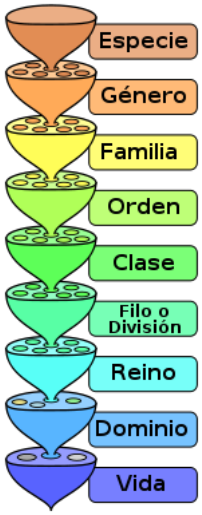

Figura 2.1: Taxonomía de las especies.

2.1.2. Organismo

Es un conjunto de átomos y moléculas que forman una estructura muy organizada y compleja. Cada organismo en particular posee una serie de características biológicas que nombramos a

continuación:

- Organización: como es la composición del mismo, si unicelular o pluricelular.
- Homeostasis: regulación de su sistema interno para mantener un estado constante.
- Irritabilidad: reacción de los estímulos externos.
- Metabolismo: un organismo consume energía para transformarla en nutrientes (anabolismo) y liberan energía al descomponer la materia orgánica (catabolismo).
- Desarrollo: los organismo cambian de tamaño al consumir nutrientes.
- Reproducción: capacidad de producir copias de si mismo de forma sexual o asexual.
- Adaptación: como se adapta a un cierto ambiente a medida que evoluciona.

Composición de los Organismos

Hasta ahora vimos cuales son las propiedades de los organismos en general, en este punto haremos una breve explicación de como esta conformado un organismo. Los organismos son sistemas físicos soportados por reacciones químicas complejas, organizadas de manera que promueven la reproducción y en alguna medida la sustentabilidad y la supervivencia del mismo. La materia viva está constituida por unos 60 elementos, casi todos los elementos estables de la Tierra, exceptuando los gases nobles. Estos elementos se llaman bioelementos o elementos biogénicos. Se pueden clasificar en dos tipos: primarios y secundarios.

- Primarios: son los indispensables para formar la materia orgánica. Son el hidrógeno, el carbono ,el oxigeno, el nitrógeno, el fósforo y el azufre.
- Secundarios: son lo elementos restantes.

2.1.3. Moléculas orgánicas

Tenemos dos tipos de moléculas orgánicas básicas:

Nucleótidos

Son moléculas de pequeña masa que conforman estructuras más complejas (ej. el ADN y el ARN). Hay seis posibles nucleótidos:

1. Adenina (A) compone el ADN y el ARN.
2. Guanina (G) compone el ADN y el ARN.
3. Timina (T) compone el ADN.
4. Citosina (C) compone el ADN y el ARN.
5. Uracilo (U) compone el ARN.
6. Flavina (F) no componen ni el ADN ni el ARN.

Aminoácidos

Son moléculas que conforman a las proteínas y son esenciales para la vida. Presentamos una tabla con los aminoácidos y sus abreviaciones de tres y una letra.

Nombre	Ab. 3 letras	Ab. 1 Letra
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartic acid	Asp	D
Cytesine	Cys	C
Glutamic acid	Glu	E
Glutamine	Gln	Q
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

2.1.4. Macromoléculas

Son estructuras biológicas compuestas de un pequeño numero de moléculas fundamentales. Tenemos cuatro categorías de macromoléculas:

Ácidos Nucleicos

Son conformadas por secuencias de nucleótidos que los organismos utilizan para almacenar información. Dentro del ácido nucleico, un codón es una secuencia particular de tres nucleótidos que codifica un aminoácido particular, mientras que una secuencia de aminoácidos forma una proteína. Haremos hincapié en dos de los ácidos nucleótidos mas importantes.

1. **ADN (Ácido desoxirribonucleico)** contiene la información genética usada para el desarrollo y el funcionamiento de los organismos vivos conocidos y de algunos virus, siendo el responsable de su transmisión hereditaria. (Fig. 2.2)
2. **ARN (Ácido ribonucleico)** esta presente tanto en las células procariotas como en las eucariotas, y es el único material genético de ciertos virus (virus ARN). (Fig. 2.2)

Proteínas

Están formadas por secuencias de aminoácidos que debido a sus características químicas se pliegan de una manera especifica y así realizan una función particular. Existen distintas clases como las enzimas, proteínas estructurales, reguladoras, etc.

Glúcidos:

Son el combustible básico de toda célula.

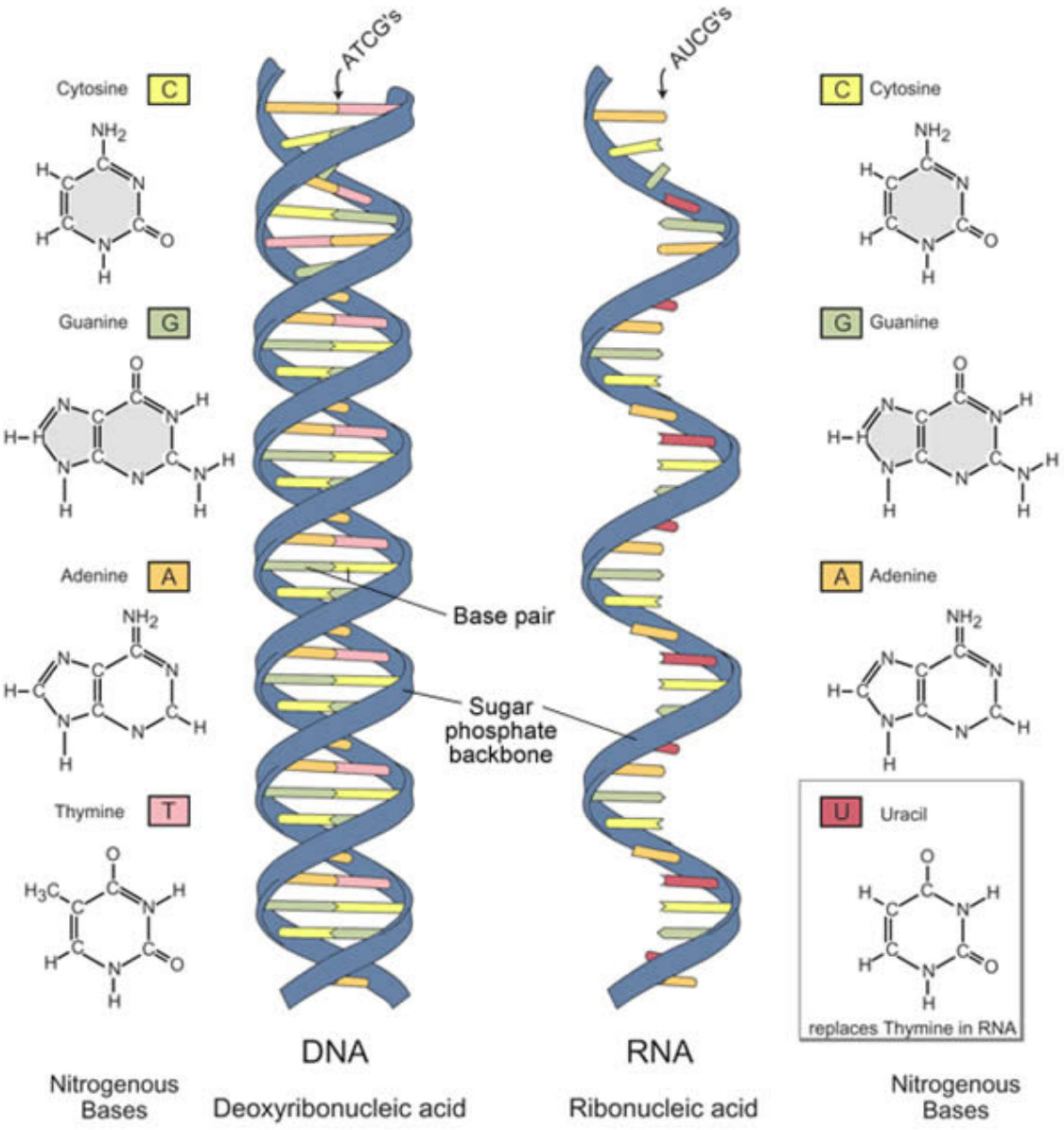

Figura 2.2: ADN y ARN.

Lípidos:

Constituye la barrera interior de una célula evitando que sustancias puedan entrar y salir de ella.

2.1.5. Células

Son las unidades básicas que conforman el organismo. Dentro de estas ocurren toda las funciones vitales ademas de contener información para transmitir a sus herederas. Se clasifican en dos tipos:

1. Procariotas: carecen de membrana nuclear.
2. Eucariotas: tiene un núcleo bien definido que contiene al ADN.

Estructura de una célula

A modo gráfico presentamos la composición de las células eucariotas y procariotas.Ver Fig. 2.3.

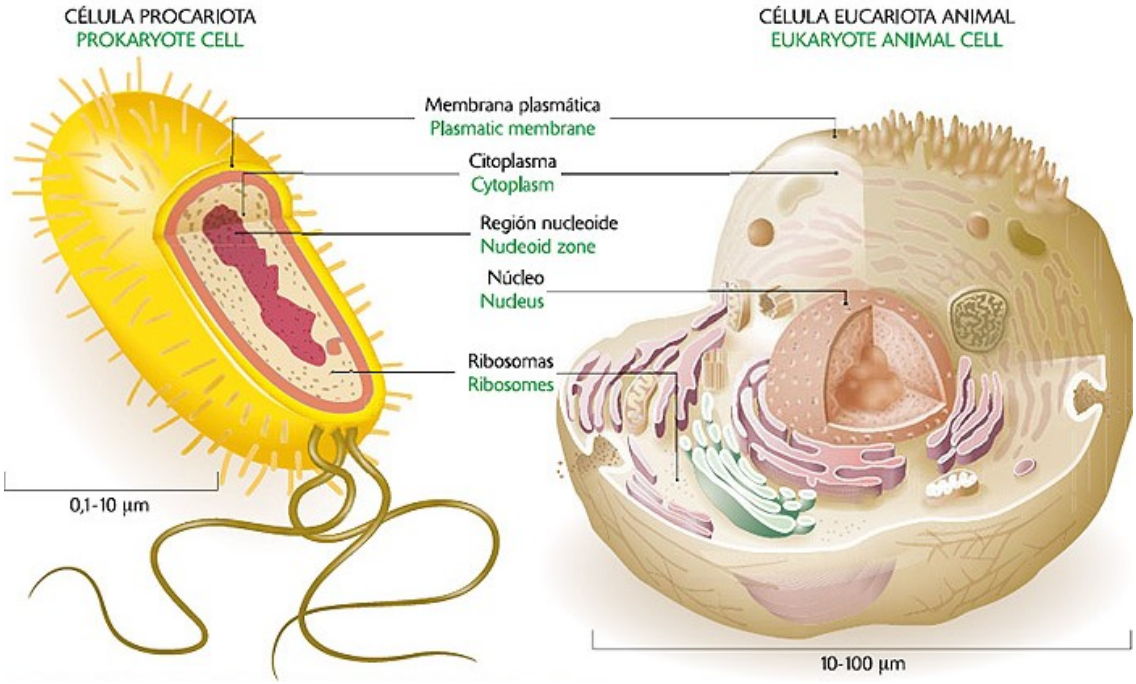

Figura 2.3: Células Procariota y Eucariota.

2.1.6. Virus

Es una entidad infecciosa microscópica que puede multiplicarse dentro de las células de otro organismo.

2.1.7. Las Células CD4

Las células CD4 (también conocidas como células T, células T ayudantes o células T4) pertenecen a un grupo de glóbulos blancos llamados linfocitos. Estas células se distinguen por dos cosas, no sólo son el blanco de ataque del VIH, sino que también tienen la responsabilidad de coordinar la manera en que el sistema inmunológico responde a las infecciones que causan enfermedades. Cuando el recuento de CD4 (número de células en un milímetro cúbico de sangre o mililitro de sangre) desciende por debajo de 200, se considera que el sistema inmunológico está debilitado y se corre un riesgo mayor de padecer una infección oportunista relacionada al SIDA, como la neumonía por Pneumocystis.

2.2. El Virus del VIH

En esta sección haremos una descripción de uno de los virus mas letales y polémicos que afectan la salud de millones de personas y cobran vidas de otras tantas. Esta introducción acompaña al tema que trata nuestro trabajo de tesis, la obtención de terapias para pacientes con VIH.

2.2.1. Historia y descubrimiento

El Virus de Inmunodeficiencia Humana ataca al sistema inmunológico, en particular a las células CD4 o T4 de las cuales necesita para reproducirse, dejando al organismo bajo en defensas ante otras enfermedades. Fue descubierto en 1983 por el equipo francés de Luc Montagner [2] como la causa del SIDA. Una vez introducido en el organismo el virus debe encontrar células susceptibles para poder replicarse: El proceso de replicación requiere una serie de pasos:

1. Absorción : El receptor celular (CD4) es reconocido por un antireceptor en la proteína $gp120$ expuesta en la envoltura viral. Una segunda molécula celular conocida como correceptor debe ser también reconocida.

2. Fusión: en este paso se produce la fusión entre la envoltura viral y la membrana plasmática de la célula permitiendo la penetración de la capside (estructura protéica) viral que contiene el genoma y enzimas necesarias para replicación.

3. Transcripción Reversa: esta produce un ADN utilizando como molde el ARN viral, posteriormente se sintetiza una doble cadena de ADN-VIH que migra hacia el núcleo de la célula infectada.

4. Integración: en esta fase el ADN-VIH se integra al ADN celular. Utiliza la integrasa para realizar esta transformación.

5. Transcripción: desde este momento el ADN comienza a producir ARN-VIH. Al ser este un proceso natural de la célula, ningún sistema de defensa es alertado. Es decir, en esta transcripción el ADN es utilizado como molde para generar ARNv que va a formar parte de la progenie viral. Además produce un ARN especial denominado ARNm(mensajero) que es necesario para sintetizar el resto de las proteínas y enzimas que el virus requiere.

6. Ensamblaje: El ARNv y enzimas necesarias para la replicación se rodean de proteínas de la capside.

7. Brotación: la cápside viral adquiere una envoltura arrastrando parte de la membrana plasmática en la cual se han incluido glicoproteínas (proteína compuesta con varios hidratos de carbono) de origen viral.

8. Maduración: gracias a la acción de la proteasa, se produce la maduración final del nuevo virus que se vuelve así plenamente infectivo. A partir de este punto el nuevo virus está preparado para atacar otras células CD4.

2.2.2. Su transmisión

Existen tres vías principales de infección:

1. Sexual: producido por el acto sexual sin protección.

2. Parenteral: la transmisión por sangre producto de la utilización de jeringas infectadas que se utilizan para la aplicación de drogas intravenosas o por transfusiones de sangre sin control respectivo.

3. Vertical: es la infección de madre a hijo que se produce en las ultimas semanas de embarazo, en el parto o en la lactancia.

2.2.3. La enfermedad del SIDA

Una vez que se ha confirmado que la persona es portadora del VIH, es probable que en un determinado momento desarrolle SIDA (Síndrome de Inmunodeficiencia Adquirida). Es difícil predecir si esto finalmente sucederá y cuándo, pues depende de la capacidad de respuesta del sistema inmune de cada individuo y de la eficacia de los tratamientos. Vale aclarar que una persona con VIH pasa a desarrollar un cuadro de SIDA cuando su nivel de células CD4 se encuentra por debajo de 200 células por mililitro de sangre.

2.2.4. Datos estadísticos

Según los datos recopilados por ONUSIDA (`http://www.unaids.org`) en nuestro país el 0,6 % (aproximadamente 240.000 personas) de la población padece VIH-SIDA siendo aproximadamente unos 40 millones de infectados en todo el mundo.

2.2.5. Tratamientos y avances en la medicina

Actualmente el único tratamiento posible para diezmar el avance del VIH es la aplicación de antirretrovirales que tienen como finalidad interferir en algunos de los procesos (antes mencionados) que el virus realiza. Lamentablemente estas terapias no son efectivas a largo plazo, ya que dada la aplicación de un fármaco, este deja de ser efectivo al cabo de unos meses debido a que el virus muta para poder evitar la acción del mismo. En el capítulo siguiente se explica con mayor detalle como son los tratamientos basados en terapias antirretrovirales en nuestro país y el mundo.

2.3. Los Antirretrovirales

Como se mencionó anteriormente, los fármacos utilizados para el tratamiento del virus del VIH son los denominados antirretrovirales. Existe una variedad de éstos provenientes de distintos laboratorios, cada uno con sus características, precios y forma de acción ante el virus. Clasificándolos por su rango de acción, hay dos clases de antirretrovirales, los Protease Inhibitors (PI) y Reverse Transcriptase Inhibitors. Dentro de la segunda clase hay dos subtipos que son los Nucleotide Reverse Transcriptase Inhibitors (NRTI) y los Non-Nucleotide Reverse Transcriptase Inhibitors (NNRTI). Existen otras dos clases mas recientes que son los Fusion or Entry Inhibitors y los Integrase Inhibitors. Cuando un antirretroviral se le suministra a un paciente, produce un efecto en el virus diezmándolo e inhibiéndolo durante un cierto tiempo. Vale aclarar que la mayoría de los antirretrovirales son altamente tóxicos con lo cual su dosis tiene que ser bien aplicada, así como tener en cuenta cuál es la dosis mínima indispensable. A continuación exhibimos una tabla con los antirretrovirales aprobados por la FDA (Food and Drugs Administration `http://www.fda.gov/MedicalDevices/default.htm`) .

Múltiples Clases Combinadas (Multi-class combinations):

Combinación	Comercial	Aprobación
EFV + TDF + FTC	Atripia	12-Jul-06
d4T + 3TC + NVP	-	Tentative only*
AZT + 3TC+ NVP	-	Tentative only*

Inhibidores de Transcriptasa Reversa Nucleósidos (NRTIs):

Abreviación	Genérico	Comercial	Aprobación
3TC	lamivudine	Epivir	17-Nov-95
ABC	abacavir	Ziagen	17-Dec-98
AZT o ZDV	zidovudine	Retrovir	19-Mar-87
d4T	stavudine	Zerit	24-Jun-94
ddI	didanosine	Videx EC	31-Oct-00
FTC	emtricitabine	Emtriva	02-Jul-03
TDF	tenofovir	Viread	26-Oct-01

Inhibidores de Transcriptasa Reversa No Nucleósidos (NNRTIs):

Abreviación	Genérico	Comercial	Aprobación
DLV	delavirdine	-	04-Apr-97
EFV	efavirenz	Sustiva	17-Sep-98
ETR	etravirine	Intelence	18-Jan-08
NVP	nevirapine	Viramune	21-Jun-96

NRTIs Combinados (Combined NRTIs):

Combinación	Comercial	Aprobación
ABC + 3TC	Epzicom (US)	02-Aug-04
ABC + AZT + 3TC	Trizivir	14-Nov-00
AZT + 3TC	Combivir	27-Sep-97
TDF + FTC	Truvada	02-Aug-04
d4T + 3TC	-	Tentative only*

Inhibidores de Proteasa (Protease Inhibitors) (PIs):

Abreviación	Genérico	Comercial	Aprobación
APV	amprenavir	Agenerase	15-Apr-99
FOS-APV	fosamprenavir	Lexiva (US)	20-Oct-03
ATV	atazanavir	Reyataz	20-Jun-03
DRV	darunavir	Prezista	23-Jun-06
IDV	indinavir	Crixivan	13-Mar-96
LPV/RTV	lopinavir + ritonavir	Kaletra	15-Sep-00
NFV	nelfinavir	Viracept	14-Mar-97
RTV	ritonavir	Norvir	01-Mar-96
SQV	saquinavir	Invirase	06-Dec-95
TPV	tipranavir	Aptivus	22-Jun-05

Inhibidores de Fusión o Entrada (Fusion or Entry Inhibitors):

Abreviación	Genérico	Comercial	Aprobación
T-20	enfuvirtide	Fuzeon	13-Mar-03
MVC	maraviroc	Celsentri	18-Sep-07

Inhibidores de Integrasa (Integrase Inhibitors):

Abreviación	Genérico	Comercial	Aprobación
RAL	raltegravir	Isentress	12-Oct-07

2.3.1. Fallo Virológico

El fallo virológico predispone al fallo inmunológico y eventualmente a la progresión de la inmunodeficiencia. El fallo virológico aislado, en ninguna circunstancia, es una causa para discontinuar el tratamiento antirretroviral. Pero en este trabajo lo usamos para hacer referencia a la falta de efectividad en el cóctel de antirretrovirales y su posterior cambio.

2.4. Medicina Personalizada

Es un modelo de aplicación médica que utiliza la información obtenida de un paciente para elaborar y optimizar terapias o tratamientos. También se puede definir a la Medicina Personalizada como un conjunto de productos y servicios que utilizan la ciencia de la genómica y la proteómica para elaborar enfoques de prevención y cuidado personal. Más información en `http://www.pwc.com/personalizedmedicine` y [17].

2.5. La Bioinformática

La bioinformática es la aplicación de las ciencias de la computación a la biología molecular. Esta ciencia comprende gran cantidad de métodos, algoritmos y técnicas estadísticas. Esto ha permitido generar modelos para la aplicación de proteínas, análisis de ADN y generación de modelos gráficos de estas estructuras entre tantas otra aplicaciones.

2.5.1. Alcance

La bioinformática comprende la serie de metodologías formales para tratar de una gran cantidad problemas de tipo biológico. Entre ellos:

1. Análisis de Secuencias: codificar y analizar secuencias de ADN de distintos organismos.[21]
2. Anotación de Genomas: es el proceso de marcado de los genes y otras características del ADN. [22]
3. Biología Evolutiva computacional: estudio ancestral de las especies. [23]
4. Análisis de expresiones genéticas: es el estudio del proceso en el cual los organismos transforma información codificada en proteínas necesarias para su desarrollo.[24]
5. Análisis de Regulación: se analizan los eventos que comienzan con una señal extracelular (como por ejemplo una hormona) y que conducen al incremente o decremento de una o más proteínas.[25]
6. Análisis de expresiones de proteínas: poder analizas las proteínas presentes en una muestra biológica.[26]
7. Análisis de mutaciones del cáncer: poder predecir en base a secuenciaciones de células las mutaciones de algunas de estas y que provocarían el cáncer.[27]
8. Predicción de estructuras de proteínas: es la traducción de proteínas en base a su conformación de aminoácidos.[28]
9. Modelado de sistemas biológicos: simulación de sistemas y subsistemas biológicos junto con su posibles comportamientos.[29]

2.5.2. Trabajos

Existe una gran cantidad de trabajos relacionados con este área y que han sido de vital importancia para llevar a cabo grandes experimentos y puesta a prueba ciertas teorías. Hacemos referencias a algunos de estos:

- Basic Local Alignment Search Tool (BLAST): es un algoritmo para la comparación de estructuras biológicas primarias como proteínas, nucleótidos, aminoácidos o ADN. Posee una gran base de datos de secuencias ya obtenidas. `http://blast.ncbi.nlm.nih.gov/Blast.cgi`

- pheno2geno: sistema de predicción fenotípica. Este software toma secuencias de ARN del virus del VIH y elabora arboles de decisión basado en probabilidades. `http://www.geno2pheno.org/`
- NetCTL Server: predicción de epitopes (un antígeno) en secuencias de proteínas. `http://www.cbs.dtu.dk/services/NetCTL/`

2.5.3. Estructuras y Algoritmos Bioinformáticos

Dada la presentación anterior de las estructuras biológicas básicas damos paso al diseño formal de estas así como algunos algoritmos que son de uso corriente. Recordemos que este trabajo comprende el nivel de estructuras primarias, es decir, las cadenas de nucleótidos y aminoácidos que conforman el ADN y ARN.

Representación de nucleótidos, aminoácidos y proteínas

Representamos a las cadenas de nucleótidos y de aminoácidos que se utilizarán a lo largo del trabajo como cadenas de caracteres. Denotamos a las mismas como expresiones regulares.

- $nuc\_arn = a|u|c|g|\_$
- $nuc\_adn = a|t|c|g|\_$
- $aminoacido = Ala|Arg|Asn...$

De la misma forma representamos a los genes de ADN, ARN y proteínas.

- $gen\_arn = (nuc\_arn)^+$
- $gen\_adn = (nuc\_adn)^+$
- $proteina = aminoacido(aminoacido)^+$

Traducción de ADN a ARN

Existe una forma algorítmica de traducir una secuencia ADN a una secuencia de ARN. Consiste analizar cada nucleótido de la cadena de ARN cambiando cada t por u.

```
arn_gen = traslate(adn_gen), donde:

arn_gen translate(adn_gen)
{
  forall nuc_adn in adn_gen do
  {
    if nuc_adn is t then replace(nuc_adn, u)
  }
}
```

Traducción de ARN a proteínas

Podemos traducir la secuencia de ARN (compuesta de nucleótidos) a una secuencia de proteínas (compuesta de aminoácidos). Cada aminoácido es correspondido por tres nucleótidos. Denominamos a esta cadena de tres caracteres como *triplete* o codón. El algoritmo toma cada triplete y lo traduce a un aminoácido correspondido por la tabla del código genético (Sección 14.1).

```
protein = translate(arn_gen), donde:

protein translate(arn_gen)
{
  i = 0
  do
  {
    triplet = substr(arn_gen, i, 3)
    if triplet != STOP then
    {
      aminoacid = genetic_code(triplet)
      protein += aminoacid
      i += 3
    }
  }loop until triplet != STOP or i>= length(arn_gen)
}
```

Capítulo 3

La problemática en los tratamientos contra el VIH

3.1. Introducción

Actualmente el desarrollo de las terapias VIH está basado en una serie de normas internacionales (`http://www.aidsinfo.nih.gov/guidelines/`) propuestas por Department of Health and Human Services (DHHS) E.U.(`http://www.hhs.gov`) y llevadas a cabo por profesionales del área de infectología, virología y bioquímica. Vale la pena mencionar que las normas de tratamiento son desarrolladas de forma empírica y basándose en experiencias propias de los profesionales que las formularon. Dado un paciente infectado pueden existir varias terapias para tratarlo. Está en el criterio del profesional el poder elegir la mejor de estas terapias. El problema radica, en elegir la o las mejores de acuerdo a las características del paciente, para lograrlo hay que analizar todo un conjunto de variables asociadas, cuya complejidad excede por mucho a las metodologías de análisis utilizadas hasta el momento, tomando en cuenta ademas, que se puede introducir una gran cantidad de errores por la aplicación del método. A continuación daremos una breve descripción de las metodologías empleadas en la actualidad:

3.1.1. El proceso de tratamiento en Estados Unidos

Una vez diagnosticada la enfermedad, se hace un análisis de CD4. De acuerdo a la cantidad de CD4 por micro litro, se decide comenzar o no el tratamiento. Generalmente se empieza a tratar pacientes con menos de 350 CD4 por micro litro. Es mandatorio realizar el estudio de resistencias y la secuenciación antes de comenzar el tratamiento. En base a los lineamientos de la DHHS y al historial clínico del paciente se decide la mejor terapia.

En Estados Unidos el tratamiento mas corriente en la actualidad consta de: Comenzar con $3TC(Lamivudine)$ o $FTC + Tenofovir + Efavirenz$. Al igual que Argentina pero se remplaza el AZT por el $Tenofovir$ que es otro análogo de nucleótido pero algo menos toxico. En Estados Unidos existe un fármaco que contiene las tres drogas denominado $Tripla$, alternativamente se aplican $3TC + Tenofovir + Atazanavir/r$(un inhibidor de la proteasa activado por Ritonavir).

3.1.2. El proceso de tratamiento en Argentina

En nuestro país una vez diagnosticada la enfermedad, se efectúa el análisis de CD4.Tanto en Argentina como en Estados Unidos hay consenso de tratar a los pacientes con menos de 350 CD4 por micro litro. No se realiza la secuencia al inicio del tratamiento (al menos no en forma regular o mandatoria).

El tratamiento inicial mas frecuente consiste en: Comenzar con $3TC + AZT + Efavirenz$ (los dos primeros son análogos de nucleótidos y el tercero es un inhibidor de la transcriptasa no análogo

de nucleótido). La alternativa es usar en lugar de $Efavirenz$, un inhibidor de la proteasa, como ser Saquinavir/r (potenciado Ritonavir).

3.2. Análisis del problema

Este trabajo busca obtener las mejores terapias para tratar un paciente y a su vez garantizar que las terapias cumplan lineamientos básicos:

- Prolongar la vida y mejorar su calidad en el largo plazo.
- Reducir los niveles del virus por debajo del límite de detección con las pruebas actuales (menos de 50 copias del ARN del VIH), o al menor número posible durante el mayor tiempo posible.
- Minimizar la toxicidad de los medicamentos a la vez que se manejan los efectos secundarios y la interacción entre los distintos medicamentos.

Ver [19] y [20] para más información.

Esto se basa en resolver la problemática de las resistencias cruzadas, es decir, qué antirretrovirales aplicar y en qué orden para poder prolongar al máximo la efectividad de la terapia. Para dar una solución a este problema desarrollamos un sistema informático que devuelve una lista con las n mejores terapias para tratar un paciente, ordenadas de más recomendable a menos recomendable.

Para encontrar las terapias adecuadas se toman en cuenta los antirretrovirales disponibles y la secuencia viral del paciente y los criterios especificados por el profesional. Estos criterios de evaluación deben ser considerados de alguna forma en el proceso de cálculo, pudiendo ser estos, la toxicidad, embarazo, coinfección, interacciones, etc. Es decir, estamos ante un claro ejemplo de Medicina Personalizada ya que se hace énfasis en el uso sistemático de información sobre un paciente para seleccionar u optimizar terapias para dicho individuo.

3.2.1. Los problemas relacionados

El Análisis inicial desemboca en una serie de problemas asociados los cuales no están dentro del alcance de este trabajo. Estos pueden ser clasificados en:

Computacionales

- Referidos a eficiencia de calculo: si bien siempre se seleccionaron los algoritmos y estructuras mas adecuadas para el desarrollo del sistema, consideramos que pueden optimizarse muchos de los métodos aquí establecidos impactando principalmente en el tiempo de calculo y en segunda medida en los recursos requeridos por sistema .
- Extensiones fáciles de desarrollar: se planteo en principio proveer un manejo de las extensiones o plugins de forma simple y clara para los usuarios finales del sistema.Se optó por proveer plugins a través de librerías dinámicas del sistema.

Biológicos

- Manejo de probabilidades asociadas a la efectividad del antirretroviral y la mutación del virus: este trabajo abarca un enfoque de tipo cualitativo-algorítmico, por lo cual no se considero el tratamiento de probabilidades y los tiempos de acción efectiva de un antirretroviral. De cualquier modo, esta posibilidad de extensión existe a través del plugin aunque su soporte por medio del sistema es limitado.
- Capacidad de análisis en paralelo de una población viral, en lugar de solo la cepa más predominante.

Terapéuticos

- Periodo de tiempo en el cual se debe modificar el tratamiento: la posibilidad de que el sistema determine cuando una terapia deja de ser efectiva para el paciente.

3.2.2. Requerimientos Funcionales

En base al estudio del problema que se planteó con anterioridad concluimos que el sistema debe proveer el siguiente funcionamiento básico:

1. Cargar una secuencia nucleotídica de ARN.
2. Cargar una base de datos de antirretrovirales.
3. Incorporar los criterios del profesional a través de un punto extensible, en este caso un plugin.
4. A partir de la secuencia inicial seleccionar los antirretrovirales que aplican.
5. Combinar los antirretrovirales de acuerdo a los requerimientos del usuario.
6. Generar una lista de mutantes como consecuencia de aplicar la combinación.
7. Procesar cada mutante.
8. Generar una terapia parcial a partir de la mutante y asignar un puntaje acorde a los criterios introducidos por el profesional.
9. Establecer un Ranking de terapias de acuerdo a sus puntajes.
10. Para cada mutante se repite el procedimiento desde el paso 4, hasta que no hayan más antirretrovirales que aplicar o se especifique su terminación.
11. Se devuelve un listado ordenado de terapias junto con información asociada a las mismas.

Una vez que obtuvimos los lineamientos generales del sistema planteamos una serie de requerimientos funcionales. Cada uno cubre alguno de los requisitos planteados por el usuario o derivan de estos.

A lo largo del desarrollo del sistema muchos requerimientos se modificaron, surgieron nuevos y algunos se descartaron. Para tener una visión más detallada de la evolución de los mismos a lo largo de todo el proceso de desarrollo se aconseja leer el Documento de Requerimientos Funcionales anexado al trabajo.

A continuación se listan los requerimientos funcionales finales del sistema.

- **Nombre del Requerimiento:** Codificación en Lenguaje C++ (RF1).
  Propósito: Realizar la programación del sistema en lenguaje C++ con orientación a objetos. Esto fue requerido por parte de Fu.De.PAN siendo este el lenguaje estándar para la mayoría de sus proyectos por su performance, reusabilidad y extensión.

- **Nombre del Requerimiento:** Carga de secuencia ARN del virus, base de datos de antirretrovirales y parámetros (RF2).
  Propósito: Cargar un archivo en formato FASTA, establecer los parámetros requeridos por el plugin.
  Input: Archivo en formato FASTA, archivo de la base de datos, Valores de los parámetros requeridos por el plugin.
  Procesamiento: cargar el archivo y convertirlo a una estructura de datos adecuada para su manipulación. Un proceso similar los parámetros del plugin.
  Output: Estructura de datos asociada al archivo FASTA y estructura de datos con los

parámetros.
Descripción: El usuario carga el archivo haciendo click en el botón Examinar. Si el formato no es correcto se muestra un mensaje de error. Se configuran los parámetros. Si los parámetros están en formato inadecuado o son inconsistentes se muestra un mensaje de error. Se hace click en Procesar.

- **Nombre del Requerimiento:** Verificar qué antirretroviral Aplica (RF3).
  Propósito: Verificar los antirretrovirales que tienen efectos sobre el virus para descartar los innecesarios.
  Input: Cadena de Aminoácidos, Base de Datos de antirretrovirales.
  Procesamiento: Compara cada antirretroviral con las secuencias a las cuales son aplicables y los que no corresponden a esta, son descartados.
  Output: Conjunto de antirretrovirales que aplican a este virus inicial (cadena de proteínas).

- **Nombre del Requerimiento:** Administrador de Plugins (RF4).
  Propósito: Poder utilizar una serie de componentes o plugins para extender la funcionalidad del sistema.
  Input: Plugin
  Procesamiento: Carga del plugin.
  Output: Mensaje de Error o mostrar parámetros requeridos por el plugin en la GUI.
  Descripción: Se selecciona la opción Examinar para buscar el plugin y se da Aceptar. Si el plugin es un formato incorrecto se muestra un mensaje de error. Si el plugin es inconsistente con la Base de Datos de antirretrovirales se muestra un mensaje de error. En caso contrario se muestran las opciones de configuración del plugin

  1. **Nombre del Sub Requerimiento:** Chequear la consistencia del plugin con la Base de Datos (RF4.1).
     Propósito: Poder comprobar que el plugin sea consistente con los datos para evitar cálculos erróneos.
     Input: Plugin.
     Procesamiento: el plugin tiene acceso a una versión de solo lectura de la BD y hace una pre evaluación de estos.
     Output: Mensaje de éxito o fracaso.

  2. **Nombre del Sub Requerimiento:** configurar lista de parámetros requeridos (RF4.2).
     Propósito: Permitirle al plugin acceder a datos en tiempo de ejecución.
     Input: Plugin.
     Procesamiento:El Administrador solicita al plugin aquellos parámetros necesarios para hacer los cálculos. Estos son mostrados al usuario a través de la GUI, inicializados por este y retornados al plugin.
     Output: Mensaje de éxito o fracaso en caso de errores en los valores establecidos por el usuario.

  3. **Nombre del Sub Requerimiento:** Adaptar el sistema de acuerdo a las modificaciones planteadas por el plugin (RF4.3).
     Propósito: Permitirle al implementador de plugin establecer los algoritmo mas eficiente y adaptaciones necesarias para mejorar el desempeño del sistema y adecuarlo a sus requerimientos.
     Input: Plugin.
     Procesamiento: adaptar cada funcionalidad del plugin sobre el sistema
     Output: Mensaje de éxito o fracaso.

- **Nombre del Requerimiento:** Funcionalidades del Plugin (RF5).
  Propósito: definir las capacidades del plugin y sus efectos sobre el sistema.

  1. **Nombre del Sub Requerimiento:** Algoritmo de generación (RF5.1).
     Propósito: Permitirle al usuario establecer el algoritmo más adecuado para construir el espacio de búsqueda.

  2. **Nombre del Sub Requerimiento:** Función de puntuación de terapias (RF5.2).
     Propósito: Proveer una función de puntuación (scoring), la cual recibía una terapia y retornaba un valor.

  3. **Nombre del Sub Requerimiento:** Función de poda (RF5.3).
     Propósito: proveer una función de poda del espacio de búsqueda la cual determinaba si continuar o no con una terapia.

  4. **Nombre del Sub Requerimiento:** Validación de base de Datos. (RF5.4).
     Propósito: proveer una función que valide la base de datos con la que va a trabajar.

  5. **Nombre del Sub Requerimiento:** Configuración de Matriz de distancia. (RF5.5).
     Propósito: permitir establecer las distancias entre el nucleótido original y el nucleótido mutado.
  6. **Nombre del Sub Requerimiento:** Injerencia del plugin sobre los datos procesados, durante generación del espacio de búsqueda (RF5.6).
     Propósito: Darle mayor libertad al implementador del plugin para manejar tanta información como requiera durante la generación de terapias.

  7. **Nombre del Sub Requerimiento:** Injerencia del plugin sobre la elección de algoritmos a aplicar en cada paso de las terapias generadas (RF5.7).
     Propósito: El plugin podría elegir algoritmos de scoring, poda y política combinatoria diferentes por cada paso de la generación de una terapia.

- **Nombre del Requerimiento:** Generador de árbol de terapias (RF6).
  Propósito: Construir un árbol cuya raíz sea la secuencia inicial del virus. Además, construye cada nivel con las mutantes obtenidas luego de aplicar un conjunto de antirretrovirales. Una terapia debe crecer hasta que no queden antirretrovirales por aplicar, se haya encontrado una cepa resistente a todos o el plugin a través de la función de poda decida no continuarla.
  Input: Cadena de Aminoácidos (secuencia viral inicial), conjunto de antirretrovirales que aplican a la secuencia, función de poda introducida por el plugin y algoritmo de generación propuesto por este.
  Procesamiento: dada una cepa, analizamos los antirretrovirales que son efectivos contra esta, los agrupamos y aplicamos. Se crean nodos nuevos por cada mutante del virus, seleccionamos aquella de menor distancia evolutiva (menor distancia con la cepa del nodo padre). Repetimos este proceso de acuerdo a lo establecido por el algoritmo de generación hasta que no se puedan continuar más terapias.
  Output: Árbol de terapias.

- **Nombre del Requerimiento:** Ranking de Terapias (RF7).
  Propósito: establecer el conjunto de terapias optimas para tratar el paciente.
  Input: árbol de terapias, función de Scoring introducida por el plugin.
  Procesamiento: Se recorre el árbol utilizando un algoritmo establecido por el plugin y

se selecciona las aplicaciones de antirretrovirales más eficientes, de acuerdo a la función de scoring.
Output: Listado ordenado de Terapias.

1. **Nombre del Sub Requerimiento:** Ranking de Terapias (RF7.1).
   Propósito: Notando la utilidad de este tipo de objetos, se opto por que este sea un rankeador genérico de objetos. **Input:** objetos, función de comparación entre objetos y cantidad de elementos (top).
   Procesamiento: a medida que se recibe un objeto se lo compara con los que están actualmente en el ranking. Luego se introduce en el ranking o se lo descarta. **Output:** Listado ordenado objetos.

- **Nombre del Requerimiento:** Muestra de los Resultados (RF8).
  Propósito: Mostrar en forma adecuada y clara los resultados calculados.
  Input: Listado de Terapias.
  Procesamiento: Ordenar los resultados de las terapias por importancia. Presentarlos en forma de columnas o como salida de un archivo. **Output:** Listado de terapias.

- **Nombre del Requerimiento:** Librería para plugin - libplugin (RF9).
  Propósito: Proveer un conjunto de librerías, con todo lo necesario para desarrollar un plugin en C++ y un constructor genérico de librerías dinámicas.
  Propósito: Proporcionarle al usuario facilidades para desarrollar plugins.
  Input: Archivo fuente (extensión *.cpp*).
  Procesamiento: compilar el archivo fuente.
  Output: Librería dinámica de C++ (extensión *.so*).

  1. **Nombre del Sub Requerimiento:** Proveer una serie de políticas estándares de generación para el espacio de búsqueda (RF9.1).
     Propósito: Permitirle al implementador de plugin contar con distintas políticas básicos de generación.
     Descripción: proveer algoritmos de generación en profundidad, en anchura y el que solo considera las N mejores opciones en cada paso.

  2. **Nombre del Sub Requerimiento:** Proveer una serie de algoritmos estándares para combinación de antirretrovirales (RF9.2).
     Propósito: Proveer al implementador de plugin un conjunto de algoritmos de combinación pre diseñados y que estos puedan combinarse para formar algoritmos más complejos.
     Descripción: proveer algoritmos para combinación de antirretrovirales: combinación newtoneana, listador simple, secuencial de un grupo de combinadores, paralelo de un grupo de combinadores y combinador Vacío.

- **Nombre del Requerimiento:** Plugin implementado sobre librerías dinámicas de C++ (RF10). **Propósito:** Proveer independencia entre el sistema y el plugin sin necesidad de re compilar todo el código por causa de modificaciones en una de las partes. Posibilidad de acceder a las funciones del plugin en tiempo de ejecución.

- **Nombre del Requerimiento:** Combinador de Antirretrovirales (RF11).
  Propósito: Se busca emular el hecho de atacar al virus con cócteles de antirretrovirales.
  Input: conjunto de antirretrovirales que aplican al virus.

Procesamiento: generar todas las posibles combinaciones de tamaño entre N y M. Con N y M establecidas por el usuario.
Output: Conjunto de conjuntos de antirretrovirales.

- **Nombre del Requerimiento:** Manejo del flujo de datos (RF12).
  Propósito: Debido a el volumen de información, la complejidad de calculo y el tiempo que tarda en generar resultados, el sistema debe confeccionar un ranking y retornar las terapias parciales, a medida que estas se generan. De esta forma, si se interrumpe la ejecución del sistema, el usuario cuenta con las N mejores terapias producidas hasta ese momento. De la misma forma, se debe liberar las terapias que ya fueron procesadas, para mejorar el uso de memoria en tiempo de ejecución.
- **Nombre del Requerimiento:** Múltiples algoritmos combinatiorios(RF13).
  Propósito: El sistema debe poder convivir con múltiples algoritmos para combinación de antirretrovirales de forma simultanea en distintos lugares del espacio de búsqueda.

Capítulo 4

Conceptos de Diseño

En este capítulo se hace una breve introducción a las herramientas de diseño empleadas en el desarrollo del sistema. Comenzando con una introducción a los conceptos de análisis y diseño orientado a objetos, continuando con patrones de diseño y finalmente una breve introducción a UML. No debería saltearse este capitulo si se desea leer el siguiente, ya que se introducen conceptos importantes de diseño.

4.1. Análisis y Diseño Orientado a Objetos

El análisis y diseño orientado a objetos es un enfoque de la ingeniería del software que modela un sistema como un grupo de objetos que interactúan entre sí. Cada objeto representa alguna entidad de interés en el sistema que se está modelando, y es caracterizado por una clase, su estado, y su comportamiento. Varios modelos pueden ser creados para mostrar la estructura estática, el comportamiento dinámico, y el despliegue en tiempo de ejecución de estos objetos. El análisis orientado a objetos (OOA) aplica técnicas de modelado de objetos para analizar los requerimientos funcionales del sistema. El diseño orientado a objetos (OOD) elabora el modelo de análisis para producir especificaciones de implementación. OOA se focaliza en que debe hacer el sistema, OOD en cambio en cómo lo debe hacer. Para más información se aconseja visitar `http://www.beroux.com/english/articles/oop_uml_and_rup.php`

4.1.1. Sistema Orientado a Objetos

Un sistema orientado a objetos está compuesto de objetos. El comportamiento de este sistema resulta de la colaboración de estos objetos donde la interacción involucra el envió de mensajes entre ellos.

4.1.2. Análisis Orientado a Objetos

El análisis orientado a objetos examina el dominio del problema, con el objetivo de producir un modelo conceptual de la información que existe en el área que esta siendo analizada. El Análisis de los modelos no tienen en cuenta las dificultades de aplicación que puedan existir, tales como la concurrencia, distribución, persistencia, o cómo el sistema se va a construir. Las limitantes de implantación son tratados durante el diseño orientado a objetos (OOD). El análisis se realiza antes que el diseño. Las fuentes para el análisis puede ser una declaración por escrito de necesidades, un documento formal, entrevistas con las partes interesadas, etc. Un sistema puede ser dividido en varios dominios, que representan diferentes negocios, tecnológicos u otras áreas de interés, cada uno de los cuales se analizan por separado. El resultado de un análisis orientado a objetos es una descripción de lo que el sistema está funcionalmente obligadas a hacer, en forma de un

modelo conceptual. Que típicamente se presenta como un conjunto de casos de uso, uno o más diagramas UML de clases, y una serie de diagramas de interacción. También puede incluir algún tipo de interfaz de usuario. El objetivo de análisis orientado a objetos es desarrollar un modelo que describa los programas informáticos, ya que trabaja para satisfacer un conjunto de requisitos definidos para el cliente.

4.1.3. Diseño Orientado a Objetos

El diseño orientado a objetos transforma el modelo conceptual producido en el análisis orientado a objetos para tener en cuenta las limitaciones impuestas por la arquitectura elegida y cualquier otra limitación de falta de funcionalidad, las limitaciones tecnológicas o medioambientales. Como ser, rendimiento de transacciones, tiempo de respuesta, de ejecución, de tiempo, entorno de desarrollo, o lenguaje de programación. Los conceptos del modelo de análisis se asignan a clases de implementación y las interfaces. El resultado es un modelo del dominio de solución, una descripción detallada de cómo el sistema se va a construir.

4.2. Patrones y Antipatrones de Diseño

4.2.1. Qué son los patrones y antipatrones de diseño

Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. Un patrón describe un problema que ocurre varias veces en un sistema, y la base de la solución a ese problema. Los antipatrones de diseño, en cambio, son patrones que invariablemente conducen a una mala solución de un problema. Tanto los patrones, como los antipatrones de diseño son el resultado del consenso de los profesionales en el área y brindan herramientas a los diseñadores de sistemas para no escoger malos caminos, valiéndose de documentación disponible en lugar de simplemente la intuición.

Patrones

Un patrón de diseño es una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que se debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reusable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias. Cada patrón de diseño se focaliza sobre un problema o issue particular de diseño (DOO).

En general, un patrón tiene cuatro elementos esenciales:

1. **El nombre del patrón** es un manejador que se usa para describir un problema de diseño, su solución, y consecuencias.

2. **El problema** describe cuando aplicar el patrón, explica el problema y su contexto. El también describe problemas de diseño especifico tales como C̈omo representar un algoritmo como un objeto*:* Ademas, describe la estructura de clases y objetos que son sintomáticas de un diseño inflexible. A veces, el problema puede incluir una lista de condiciones que deben ser reunidas antes de que tenga sentido aplicar el patrón.

3. **La solución** describe los elementos que integran el diseño, sus relaciones, responsabilidades, y colaboración. La solución no describe un diseño particular concreto o implementación, porque un patrón puede ser aplicado en muchas situaciones diferentes. De hecho, el patrón provee una descripción abstracta de un problema de diseño y como una disposición general de los elementos lo resuelve.

4. **Las consecuencias** son los resultados y compromisos de aplicar el patrón. Estas son fundamentales para evaluar alternativas de diseño y para la comprensión de los costos y beneficios de aplicar el patrón. Las consecuencias de un patrón incluye su impacto sobre la flexibilidad del sistema, expansión, o portabilidad.

4.2.2. Patrones utilizados en el sistema

A lo largo del proceso de diseño se encontraron problemas que coincidían con patrones. Estos patrones se explican brevemente a continuación:

- Observer: Define una dependencia uno a muchos entre objetos, de forma que si un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente. En la Fig. 4.1 se ve el modelo estándar del patrón.

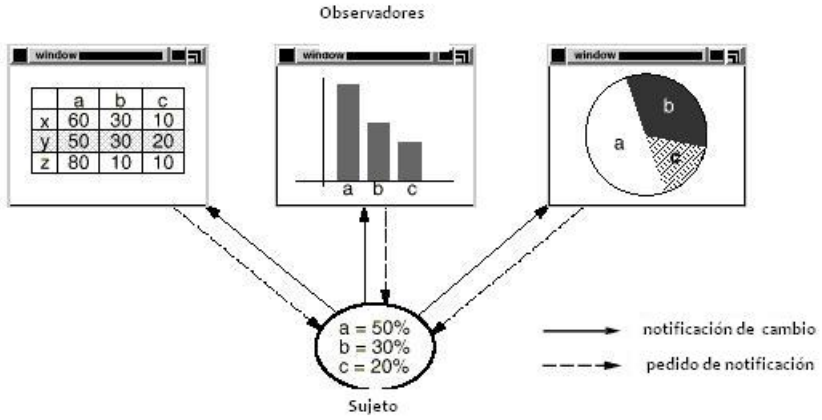

Figura 4.1: Patrón Observer.

Aplicabilidad se usa el patrón *Observer* cuando:

- Una abstracción tiene dos aspectos, uno dependiente del otro. Encapsular estos aspectos en objetos por separado le permite variarlo y reutilizarlos de manera independiente.
- Un cambio de un objeto requiere cambiar a los demás, y no sé sabe cuantos objetos hay que cambiar.
- Un objeto debe ser capaz de notificar a otros objetos sin hacer suposiciones acerca de que son estos objetos. En otras palabras, no se quiere que los objetos estén muy acoplados.

- Adapter: Convierte la interfaz de una clase en otra interfaz que el cliente espera. Los Adapters permiten que trabajen juntas clases que de otra forma no podrían, porque tienen interfaces incompatibles.

  Como se ve en la Fig. 4.2, una clase *Adapter* usa herencia multiple para adaptar una interface a otra. En este caso adapta la interfaz de $SpecificRequest()$ a $Request()$.

  Aplicabilidad se usa el patrón *Adapter* cuando:

  - Se desea utilizar una clase existente, y su interfaz no coincide con la que se necesita.
  - Se desea crear una clase reutilizable que coopera con clases no relacionadas o imprevistas, es decir, clases que no tienen necesariamente interfaces compatibles.
  - (*object adapter only*) es necesario utilizar varias subclases ya existentes, pero es muy difícil de adaptar sus interfaces por medio de cada subclase. Un adaptador de objeto puede adaptar la interfaz de su clase padre.

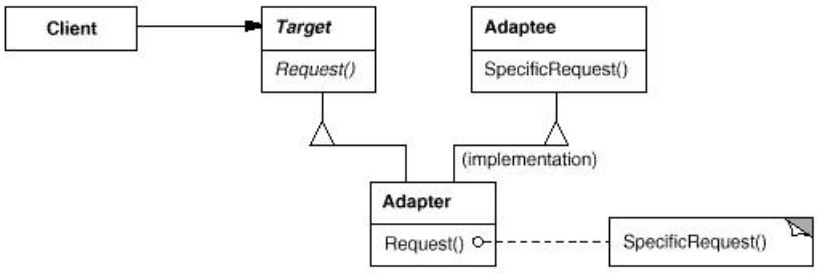

Figura 4.2: Patrón Adapter.

- Method Factory: Define una interfaz para crear un objeto, pero le permite a las subclases decidir cuales clases instanciar. $FactoryMethod$ permite aplazar la creación de instancias de una clase a las subclases.

  En la Fig. 4.3 se puede ver como se tiene por un lado una interfaz *Product* que puede tener varias implementaciones y una interfaz $Create$ encargada de producir elementos de tipo *Product*. Ejemplo: supongamos que tenemos una interfaz *Auto* (*Product*) y dos implementaciones de ella *AutoA* y *AutoB*, el $FactoryMethod$ es un método simple que dependiendo del objeto que le pida me va a construir un objeto *AutoA* o *AutoB*.

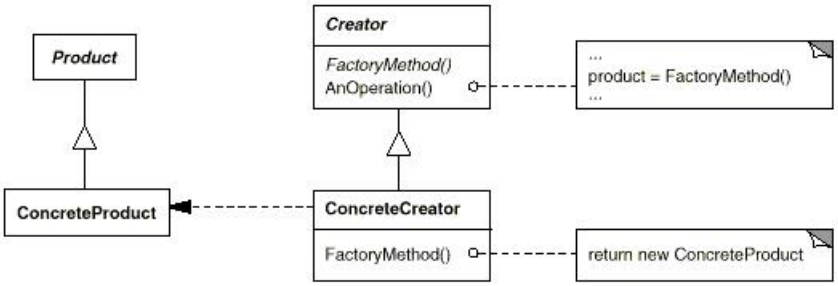

Figura 4.3: Patrón Method Factory.

Aplicabilidad Usar el $MethodFactory$ cuando:

- la clase no puede anticipar la clase de objetos que debe crear.
- la clase quiere sus subclases para especificar los objetos que crea.
- clases delegan la responsabilidad a una de varias subclases ayudantes, y desea saber a cual de las subclases se delego.

- Iterator: Proporciona una forma de acceder a los elementos de un objeto agregado de forma secuencial, sin exponer su representación subyacente. Como se ve en la Fig. 4.4, tenemos un repositorio *List* y un iterador que permite acceder al repositorio sin tener conocimiento sobre el mismo.

  Aplicabilidad Usar el patrón Iterator para:

  - acceder al contenido de un objeto agregado sin exponer su representación interna.
  - apoyar múltiples recorridos de los objetos agregados.
  - proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas (es decir, para apoyar iteración polimórfica).

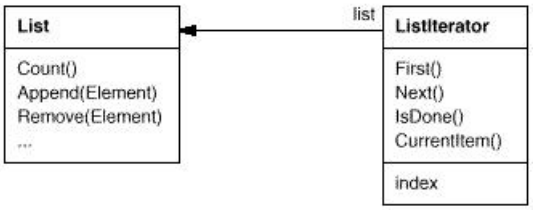

Figura 4.4: Patrón Iterator.

- Template Method: Define el esqueleto de un algoritmo en una operación, derivando algunos pasos a subclases. *TemplateMethod* permite a subclases redefinir ciertos pasos de un algoritmo sin cambiar la estructura de este. En la Fig. 4.5, tenemos un *TemplateMethod*(), que contiene las partes no variables del algoritmo y interfaces para implementar los fragmentos variables del algoritmo(*PrimitiveOperationN*()).

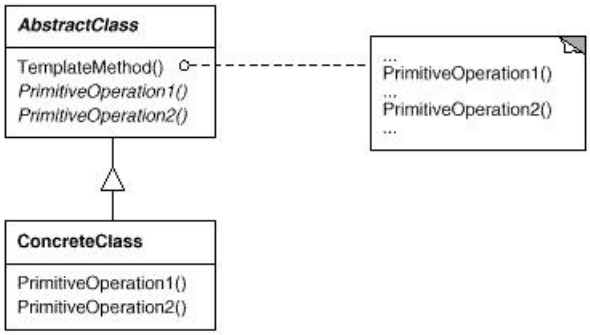

Figura 4.5: Patrón Template Method.

Aplicabilidad Usar *TemplateMethod*:

- para implementar las partes invariables de un algoritmo de forma definitiva y dejar a subclases implementar el comportamiento que puede variar.
- cuando las clases de igual comportamiento deberían ser construidas y localizadas en una clase común para evitar la duplicación de código.
- para controlar las extensiones de las subclases.

Para mayor información ver [4] y [7]

4.3. Principios de Diseño (DOO)

Dentro de la programación orientada a objetos, existen diferentes principios que ayudan en la elección del modelado de clases, y cómo estas clases deben interactuar.

4.3.1. SOLID

SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) es un acrónimo memotécnico introducido por Robert C. Martin a comienzos del 2000 el cual se basa en 5 patrones básicos de diseño y programación orientada a objetos. Cuando los principios se aplican conjuntamente hacen más probable que un programador cree un sistema de fácil mantenimiento y extensible en el tiempo [6].

Principio de Simple Responsabilidad (SRP) En la programación orientada a objetos, el Principio de Simple Responsabilidad, establece que cada objeto debe tener una única responsabilidad, y que la responsabilidad debe ser completamente encapsulado por la clase. Todos sus servicios deben estar estrechamente alineados con esa responsabilidad.

Principio de Apertura/Cierre (OCP) El principio de apertura/cierre establece que ëntidades de software (clases, módulos, funciones, etc) debe quedar abierto a la extensión, pero cerrado para su modificación; es decir, una entidad puede permitir que su comportamiento pueda ser modificado sin alterar su código fuente.

Principio de sustitución de Liskov (LSP) El principio de sustitución de Liskov es una definición particular de una relación de subtipificación, llamado subtipificación de comportamiento, que fue introducido inicialmente por Barbara Liskov. Establece que "los tipos derivados deben poder ser sustitutos completos de sus tipos base".

Principio de Segregación de Interfaz (ISP) El principio de interfaz de la segregación dice que una vez que una interfaz se ha puesto demasiado "gorda"tiene que ser dividida en partes más pequeñas y más interfaces específicas para que cualquier cliente de la interfaz pueda sólo saber acerca de los métodos que les pertenecen. En pocas palabras, ningún cliente debería ser obligado a depender de métodos que no utiliza.

Principio de Inversión de Dependencia (DIP) El principio de inversión de dependencia promueve que hay que depender de abstracciones, no de concreciones. Dicho de otra forma, los módulos de más alto nivel no deben depender de los de más bajo nivel sino que ambos deben depender de abstracciones. Y a su vez, las abstracciones no deben depender de los detalles sino al contrario.

4.4. Tipos de Diseño

Hay distintas paradigmas de diseño, para el desarrollo de este trabajo tuvimos en cuenta los siguientes:

- El diseño orientado a datos (Data-driven design) es el resultado de adaptar métodos de diseño de tipos abstractos de datos a la programación orientada a objetos. Las clases se definen en torno a los datos de las estructuras que deben mantenerse.

- El diseño orientado a responsabilidad (Responsibility-driven design) define clases alrededor de los términos de un contrato, es decir, una clase debe ser definida en torno a la responsabilidad y la información que comparte [5].

Finalmente se optó por el diseño orientado a responsabilidades, este punto se explica detalladamente en el capitulo 5.

Capítulo 5

Diseño del Sistema

5.1. Introducción

Desde el comienzo se encaró una metodología de desarrollo organizada y escalada para abarcar cada uno de los requerimientos funcionales planteados. El diseño fue indispensable para permitirnos distinguir cada parte del sistema, sus funcionalidades y las tecnologías asociadas, haciendo que el planteo de la solución sea mas claro y el proceso de codificación más natural. Este capitulo explica las distintas etapas de diseño. Comenzando por el Diseño de Arquitectura donde se da una descripción detallada del sistema obtenido a través del análisis de sus partes. Continuando con el diseño de Alto nivel explicado a través del Diagrama de Paquetes. Además, se explica como los requerimientos funcionales influenciaron en las decisiones tomadas durante el proceso de diseño.

Vale aclarar que estas especificaciones se encuentran también disponibles en el documento de diseño estándar que anexamos al presente trabajo.

5.2. Decisiones de Diseño

En esta sección damos una explicación detallada de las decisiones de diseño, cómo derivan de los requerimientos funcionales, de los principios de diseño (SOLID) y del uso de patrones de diseño.

Forma de modelado del diseño Al momento de elegir la forma de modelado se optó el diseño orientado a responsabilidades (Responsibility-driven design), debido a era más fácil encarar el problema desde el punto de vista de las responsabilidades, en lugar de hacerlo a partir de los datos involucrados (debido a su cantidad y complejidad). Además, permite cumplir con los principios de SOLID de manera simple.

Flujo de datos basado en observadores Del requerimiento RF12 se desprende la necesidad de manejar los datos de forma diferente. O sea, no esperar a que un bloque de datos sea generado completamente para procesarlo, si no hacerlo a medida que se genera cada dato. Para lograrlo se pensó las siguientes alternativas.

- Hacer que cada módulo acceda a los datos y los procese cuando los necesite. Esta opción concuerda con el patrón *Visitor*.
- Hacer que los módulos avisen cuando generan un nuevo dato y los módulos dependientes de estos actualicen su estado con el dato nuevo. Esta otra opción concuerda con patrón *Observer*.

Por la naturaleza del problema y dado que es más probable que el módulo encargado de generar los datos haga esperar al modulo que los requiere, se opto por la segunda opción. Se adapto el diseño del sistema para que todo el flujo de datos, desde la entrada de parámetros hasta la salida del las terapias contenidas en el *Ranker*, se base en el patrón *Observador* (Cadena de observadores).

Incorporación de un plugin Las pautas de tratamiento del VIH varían mucho de un profesional a otro y son muy pocos los acuerdos a nivel mundial sobre este tema. Por lo cual un sistema que se apegue a un grupo de pautas fijas resultaría en un trabajo limitado. La decisión de incorporar un plugin al sistema surge como respuesta a la necesidad de adaptación de este a distintos usuarios.

Por otro lado, el administrador de plugins hace un extenso uso del patrón *Adapter* en su labor de adaptar las interfaces del plugin a las del sistema.

Vale la pena denotar que el uso de un plugin y del administrador de plugins, como nexo entre este y el sistema, es un claro ejemplo de OCP.

Plantilla *Ranker* Originariamente se pensó el objeto *Ranker* como un contenedor de terapias que solo contiene las N mejores de acuerdo a algún valor. Sin embargo, se notó que un objeto con esta capacidad podía ser de mucha utilidad a lo largo de todo el proyecto, inclusive en otros. Se optó por crear una plantilla *Ranker* basándose en el patrón $TemplateMethod$. Vale la pena mencionar que esta librería fue incorporada a Mili (Minimalistic Libraries de C++).

Plantilla $VariantSet$ Al momento de diseñar como el plugin pediría los parámetros necesarios al usuario, nos encontramos con el inconveniente de lidiar con los tipos variables de estos. Para solucionarlo se pensaron varias posibilidades, entre ellas: hacer uso de *boost* :: *any*, *boost* :: *variant*, implementar un contenedor capaz de almacenar cualquier tipos de datos o incurrir en punteros nulos.(Boost: es un conjunto de bibliotecas de software libre y revisión por pares preparadas para extender las capacidades del lenguaje de programación C++ `http://www.boost.org`).

Se decidió descartar el uso de *any* y *variant* de boost, ya que incorporar boost al sistema implicaría un aumento importante en la complejidad de este. O sea, no valía el esfuerzo incorporarla al sistema sólo para el uso de estas variables. Se optó por construir un contenedor, que a partir de herramientas del lenguaje, almacene cualquier tipo de datos. Para construirlo se utilizó el patrón $TemplateMethod$, así los métodos de inserción y remoción no dependerán de un tipo especifico. Esta plantilla también fue incorporada a Mili, para su uso en otros proyectos.

5.3. Diseño de Arquitectura

Esta etapa de diseño plantea al sistema como un conjunto de componentes y conectores que denotan las tecnologías y partes involucradas en el mismo.

5.3.1. Fundamentación de la Arquitectura

Este diseño divide el sistema en cuatro grandes partes: el plugin, las librerías externas, la base de datos y el núcleo del sistema. De esta forma, se separa el núcleo de todo aquello que resulta ajeno al proceso de generación. Logrando un sistema compacto (apegado a OCP), robusto y modular que garantiza poco acoplamiento. Es decir, el conjunto de elementos de calculo y valores que pueden variar de acuerdo a un usuario particular, se enmarcan dentro del *Plugin* y *AntiviralDataBase*. Permitiendo que el sistema propiamente dicho permanezca invariable ante futuras modificaciones.

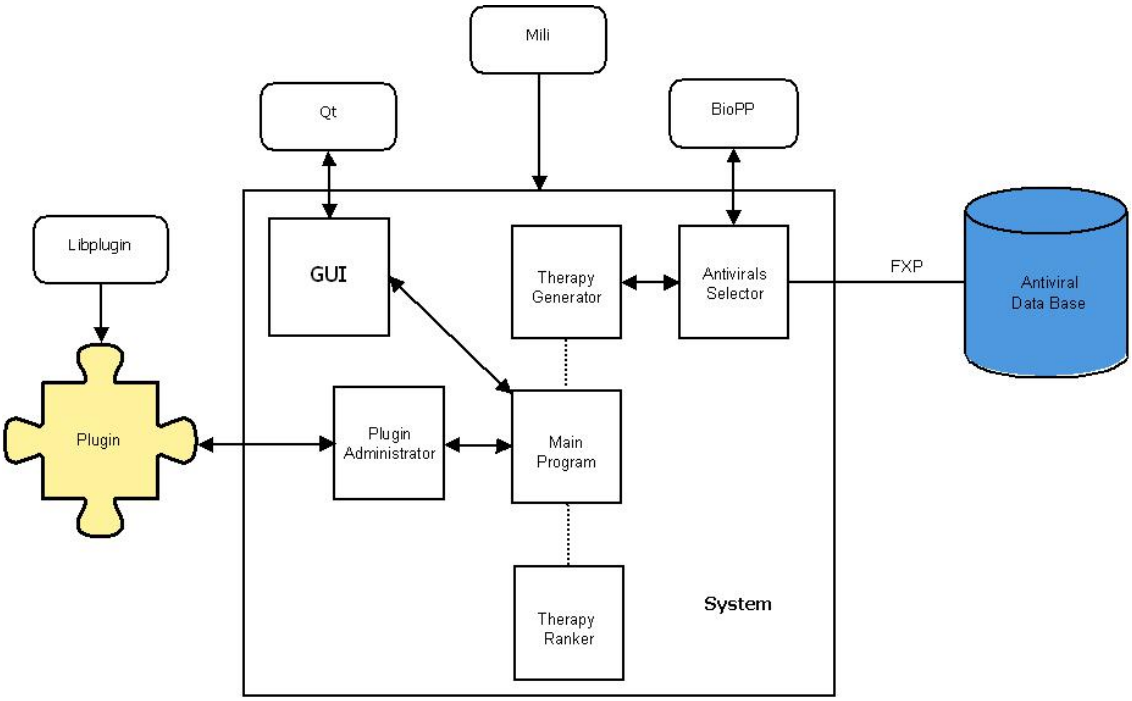

Figura 5.1: Arquitectura del sistema.

5.3.2. Catálogo de Componentes

- **System:** Aplicación principal que enmarca los siguientes componentes.
  - **Main Program:** componente principal del sistema, que coordina las distintas partes para la ejecución. Sus funciones son comunicarse con la interfaz gráfica GUI para pedir y mostrar datos, cargar la base de datos $AntiviralDataBase$, solicitar la carga de un plugin a través del $PluginAdministrator$, ejecutar la generación del espacio de búsqueda $TherapyGenerator$ y la posterior selección de las mejores terapias $TherapyRanker$.
  - **GUI:**Interfaz gráfica, su función es hacer más simple la interacción entre el usuario y el sistema.
  - **Plugin Administrator:** componente encargado de cargar y descargar los plugins, haciendo los respectivos chequeos de consistencia con el sistema.
  - **Therapy Generator:** se encargado de generar el espacio de búsqueda de terapias, tomando en cuenta las modificaciones incorporadas a través del $PluginAdministrator$.
  - **Combinatory Engine:** motor de combinación de antirretrovirales. Su función es generar combinaciones antirretrovirales de acuerdo a los requerimientos del plugin. Esta presente en varios componentes ($Plugin$, $TherapyGenerator$, $libplugin$). En el cap. 7 se hace una descripción detallada del mismo.
  - **Antivirals Selector:** componente encargado establecer los conjuntos de antivirales aplicables a una secuencia nucleotídica.
  - **Ranker:** componente encargado de generar una lista ordenada, con las N terapias de mayor puntaje, ponderadas de acuerdo a los parámetros establecidos por el usuario.
- **Libplugin:** Conjunto de herramientas y librerías (SDK) para la construcción de plugins.
- **Plugin:** modulo independiente que se acopla al sistema para aumentar o modificar su funcionalidad. El cual provee algoritmos, políticas y datos en tiempo de ejecución.

- **Qt:** Framework de desarrollo para interfaces gráficas.(`http://nokia.qt.com`)
- **BIOpp:** Biblioteca para manipulación de cadenas de aminoácidos/proteínas.(`http://biopp.googlecode.com`)
- **Mili:** Biblioteca de soporte para simplificar el uso del lenguaje de implementación (C++).(`http://mili.googlecode.com`)
- **Antiviral Data Base:** Almacenamiento de datos asociados a antirretrovirales. En este caso es indiferente el tipo de Base de Datos utilizada, solo interesa el acceso a su contenido. En particular se almacenan los datos de antirretrovirales en un archivo XML. Este componente se comunica con el sistema a través de FXP. En el capitulo 11 se hace una descripción detallada del mismo.

5.3.3. Descripción del Funcionamiento

Como se ve en la Fig. 5.1 el sistema necesita de elementos externos. Primero, toma la base de datos (*AntiviralDataBase*) que contiene los antivirales. Luego, incorpora las funcionalidades del *Plugin* proporcionado por el usuario por medio del *PluginAdministrator*. A continuación, se solicitan los parámetros requeridos por el *Plugin* y la secuencia nucleotídica del virus a través de la *GUI*. El generador de terapias *TherapyGenerator* toma la secuencia y pregunta a *AntiviralSelector* el subconjunto de antivirales que son efectivos contra el virus. Luego Genera las terapias y a medida que son generadas *TherapyRanker* las somete a un ranking de acuerdo a su valoración. Finalmente, se entrega al usuario las N mejores terapias a través de la *GUI*.

5.4. Diagrama de Paquetes

El diagrama de paquetes muestra una perspectiva más detallada del sistema y provee una forma fácil y modular de introducirse al diseño de alto nivel.

A continuación se detalla cada paquete describiendo las clases que lo componen.

5.4.1. Miscellaneous

El paquete Miscellaneous contiene clases que se usan en varios paquetes.

- **Observer** provee una interfaz usada a lo largo de todo el sistema y responde al requerimiento funcional RF12. Donde se propone la utilización de un diseño basado en Observers para mejorar la eficiencia del sistema.
- **Ranker** otro de los elementos necesarios para organizar las terapias es el Ranker. Utilizamos esta clase genérica que permite la reutilización y que fue agregada a la librería Mili.

5.4.2. DataBase

Este paquete se desprende del RF2 establecido en el capitulo 3.

- **DataBase** la clase DataBase engloba el concepto de la base de datos de antirretrovirales y todas las funcionalidades asociadas, como por ejemplo, poder cargar un archivo XML, desabilitar elementos, etc.

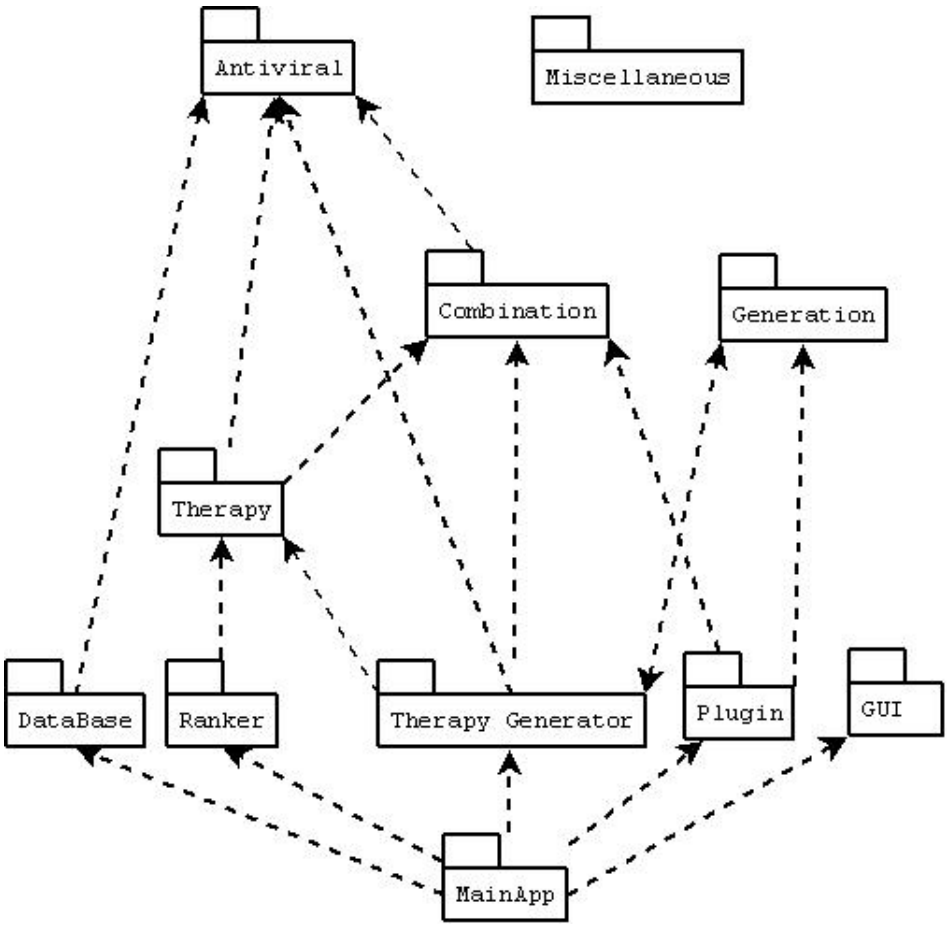

Figura 5.2: Diagrama de Paquetes.

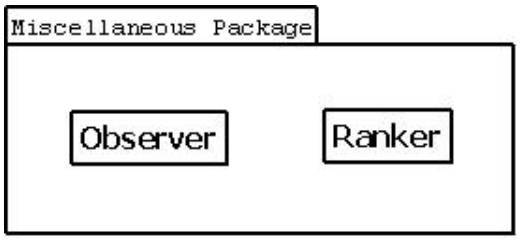

Figura 5.3: Paquete Miscellaneous.

Figura 5.4: Paquete Data Base.

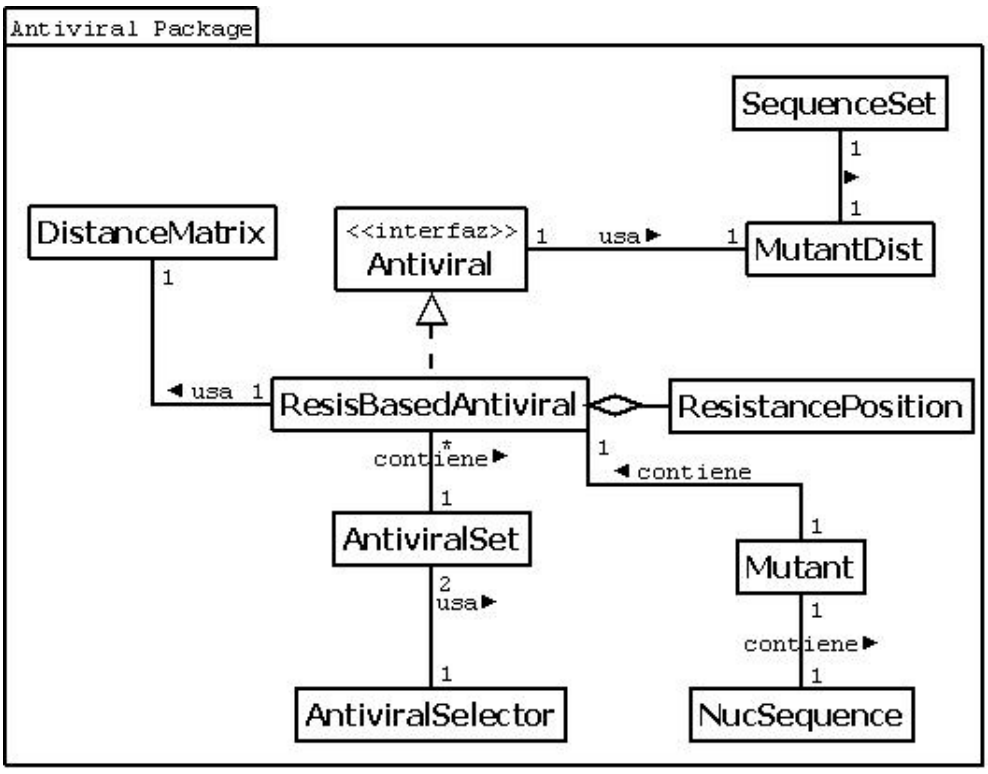

Figura 5.5: Paquete Antiviral.

5.4.3. Antivirals

La representación de antirretrovirales es central en este trabajo ya que son los elementos con los cuales el sistema realizará sus cálculos. En el capítulo 6 se hace una descripción detallada de los antirretrovirales y los algoritmos asociados.

- **NucSequence** esta clase proviene de BIOpp y representa a una secuencia de Nucleótidos con varios métodos asociados para su manejo. Se representa como una cadena de cuatro posibles caracteres (A C G T).
- **SequenceSet** con este tipo de dato definimos a un conjunto de secuencias nucleotídicas.
- **MutantDist** cuando un antirretroviral aplica lo hace con una distancia genética determinada. Consideramos para este diseño el tomar la menor de las distancias. La clase MutantDist encierra al conjunto de mutantes y al valor mínimo que se consideró.
- **DistanceMatrix** la matriz de distancias genéticas define al valor entre dos nucleótidos. Esta implentación deriva de BIOpp.
- **Antiviral** clase que provee la interfaz de Antiviral. Esta define un conjunto de funcionalidades básicas de un antirretroviral y utiliza alguna implementacion determinada de este.
- **ResistancePosition** esta es una clase que define a una posición de resistencia en particular.
- **ResisBasedAntiviral** es la implementacion de un Antirretroviral basado en posiciones de resistencia. Puede ocurrir que la interfaz utilice otra representación para los antirretrovirales. Ver capitulo 6.
- **AntiviralSet** describe un conjunto de Antirretrovirales y operaciones para este.
- **Mutant** es un tipo de dato que contiene información de una Mutante representada como
- **AntiviralSelector** uno de los mecanismos que utiliza el sistema es el de poder seleccionar un conjunto de antirretrovirales que aplican a una secuencia. Esta clase provee esta funcionalidad colocando en un AntiviralSet los antirretrovirales que cumplen con esto. Esto responde al requerimiento funcional RF3.

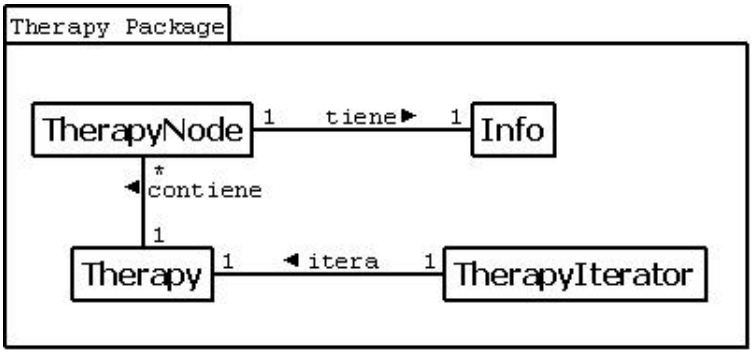

Figura 5.6: Paquete Therapy.

5.4.4. Therapy

Este paquete contiene las clases asociadas al concepto de terapias y responde a los requerimientos RF5.6 y RF5.7.

- **Info** contiene información relacionada a un *TherapyNode*, por ejemplo porque se asigno un determinado puntaje, efectos adversos de la combinación de antirretrovirales o particularidades de la mutante. Esta información se usa posteriormente para fundamentar y brindar información adicional sobre las terapias seleccionadas por el ranking. El implementador de plugin puede incorporar a esta clase toda la información que crea necesaria (RF5.6)

- **TherapyNode** representa un nodo del árbol de generación, contiene la información necesaria para el calculo de mutantes, combinación de antirretrovirales, secuencia nucleotidica, algoritmo combinatorio a utilizar (RF5.7) y información adicional asociada al nodo (contenida en un objeto $Info$). Representa un punto de inflexión en el tratamiento del paciente, en el cual se modifica la combinación de antirretrovirales a aplicar y la mutante que esta combinación genera.

- **Therapy** es una sucesión de *TherapyNode*s. La cual representa una posible terapia de antirretrovirales, junto con las mutaciones que esta genera.

- **TherapyIterator** es una aplicación del pattern *Iterator*, en este caso para iterar sobre una terapia.

5.4.5. Ranker

Este paquete contiene las clases asociadas al rankeo de terapias y responde a los requerimientos RF7 y RF12.

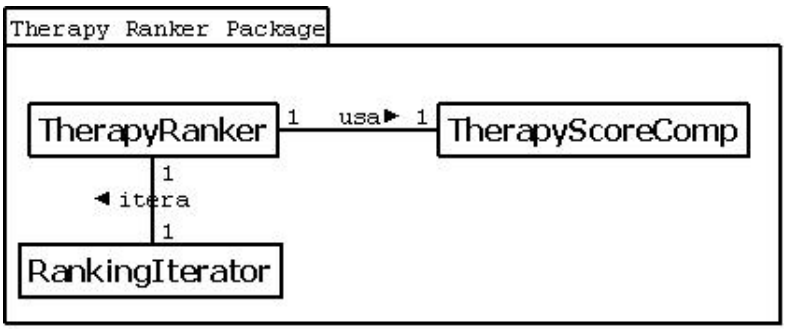

Figura 5.7: Paquete Ranker.

- **TherapyRanker** hereda de un *Ranker* genérico y implementa la interfaz *Observer*. Su función es observar las terapias a medida que se generan (RF12) y rankearlas de acuerdo a su puntaje (RF7).

- **TherapyScoreComp** es usada por el ranker de terapias para determinar cuales terapias entran en el ranking y cuales no. Su función es tomar dos terapias y decidir cual de las dos es mejor.

- **RankingIterator** es una aplicación del pattern *Iterator* en este caso para iterar sobre el ranking de terapias.

5.4.6. Generation

Este paquete contiene todas las clases asociadas a políticas de generación para el árbol de terapias. Responde a los requerimientos RF9.1 y RF5.1.

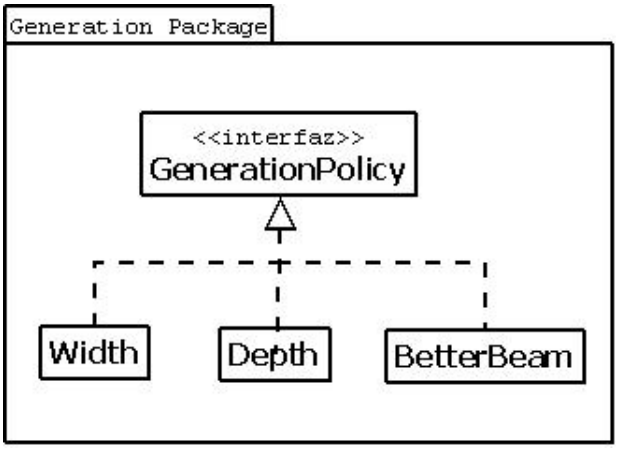

Figura 5.8: Paquete Generation.

- **GenerationPolicy**provee la interfaz para desarrollar políticas de generación. Las políticas de generación surgen como respuesta al RF5.1. La idea detrás de una política de generación es simplificar el algoritmo de generación evitando que el implementador de plugin deba construir objetos complejos y tener un algoritmo, diferente, por tipo de generación. En cambio, se opto por un algoritmo de generación único pero influenciable mediante políticas. (Este punto se explica mejor en el cap. 7)

  Las siguientes clases surgen en respuesta a RF9.1. El funcionamiento de las políticas mencionadas en las últimas tres clases se explican detalladamente en el capitulo 7.

- **Width** hereda de *GenerationPolicy* y implementa una política similar a BFS (Breadth-first search), se adapta el algoritmo para generar nodos en lugar de buscarlos.

- **Depth** hereda de *GenerationPolicy* y implementa una política similar a DFS (Depth-first search), se adapta el algoritmo para generar nodos en lugar de buscarlos.

- **BetterBeam**también hereda de *GenerationPolicy* y implementa la política Better Beam, o sea solo toma en consideración los mejores N nodos en cada paso.

5.4.7. Combination

Este paquete contiene el grupo de clases asociadas a la combinación de antivirales y responden a los requerimientos RF9.2, RF11 y RF12.

- **CombinatoryPolicy** provee la interfaz para desarrollar las políticas combinatorias.

- **CombinationStatus** sirve para indicar al usuario de *CombinatoryPolicy* el estado en que termino la combinación. Particularmente se usa para indicar el fallo virológico cuando una combinación de antirretrovirales deja de ser efectiva por diversos motivos.

- **CombinationObserver** provee la interfaz para observar al algoritmo combinador. En un principio se planteo generar todas las combinaciones y luego iterarlas usando el pattern *Iterator*. Con la aparición del RF12 para que el sistema procese información a medida que la genera, se opto por una adaptación del patter *Observer*.

  Las siguientes clases implementan la interfaz *CombinatoryPolicy* y se explican más en detalle en el cap. 8.

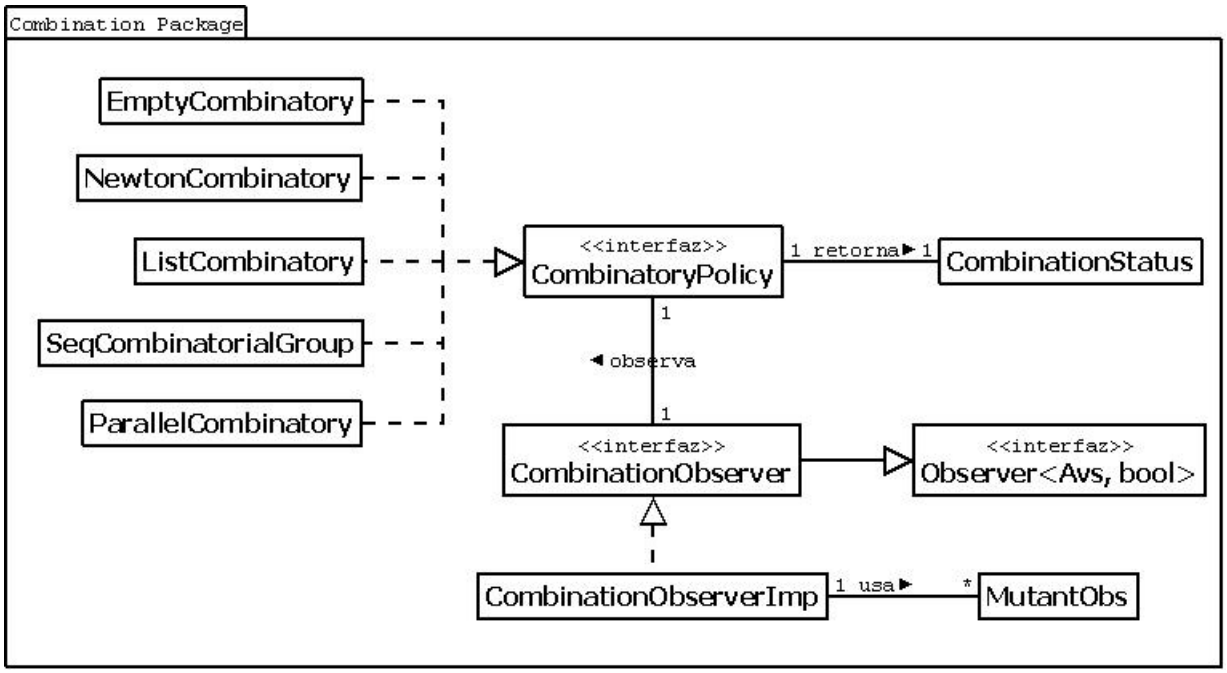

Figura 5.9: Paquete Combination.

- **EmptyCombinatory** implementa un algoritmo que no genera combinaciones y retorna un *Status* exitoso, sería el equivalente a SKIP.

- **NewtonCombinatory** implementa el algoritmo de combinación al estilo newtoneana que retorna como resultado tantas combinaciones como lo indica el coeficiente binomial (todos los subconjuntos de tamaño K de un conjunto de tamaño N).

- **ListCombinatory** implementa un listador simple itera sobre el conjunto de antirretrovirales y los retorna uno a uno como combinaciones.

- **SeqCombinatorialGroup** implementa una política de combinación compuesta, que toma un grupo de algoritmos de combinación y las hace combinar de forma secuencial, de manera que un único observador reciba las combinaciones.

- **ParallelCombinatory** implementa una política compuesta, la cual toma un grupo de algoritmos combinadores y une sus combinaciones a medida que estas se generan. Se entrega al observador la unión de las combinaciones generadas por cada algoritmo que contiene.

5.4.8. Plugin

Este paquete contiene todas las clases asociadas al concepto de Plugin y de su interacción con el sistema.

- **UserParameter** provee la interfaz para implementar los parámetros que necesite el plugin para funcionar.

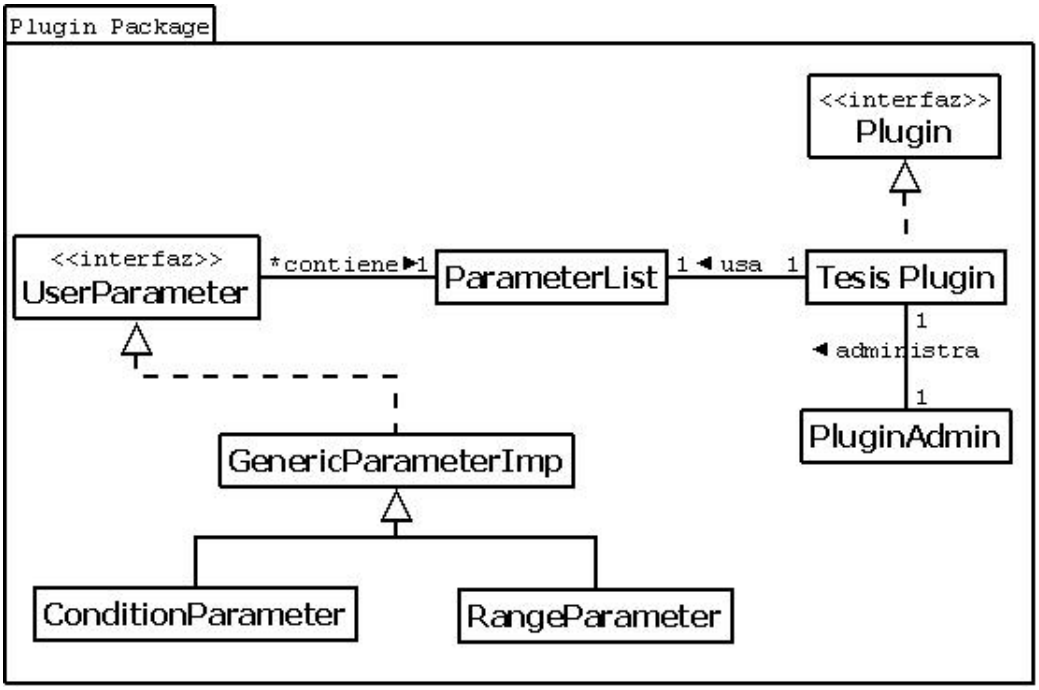

Figura 5.10: Paquete Plugin.

- **GenericParameterImp** implementa la interfaz $UserParameter$ para manipular un argumento genérico (float, string, int, etc.).

  Además, el implementador de plugin puede proveer su propia implementación de $UserParameter$ con los chequeos que crea necesarios (RF4.2).

  Por ejemplo.

- **ConditionParameter** generaliza la clase $GenericParameterImp$ para chequear una condición.
- **RangeParameter** generaliza la clase $GenericParameterImp$ para chequear que el que el valor este contenido dentro de un rango.
- **ParameterList** es básicamente un contenedor que provee un conjunto de métodos para manipularlo.
- **Plugin** provee la interfaz para implementar un plugin, con las requerimientos mínimos que usa el sistema (RF5).
- **TesisPlugin** implementa la interfaz *Plugin* y representa el plugin que se distribuye junto con este trabajo el cual se explica en detalle en el cap 10.
- **PluginAdmin** es el encargado de mediar entre el plugin y el sistema, se encarga de chequear la consistencia con el sistema, incorporar el plugin al sistema y removerlo en caso de ser necesario (RF4).

5.4.9. Therapy Generator

Este paquete contiene el grupo de clases directamente relacionadas a la generación de terapias (RF6).

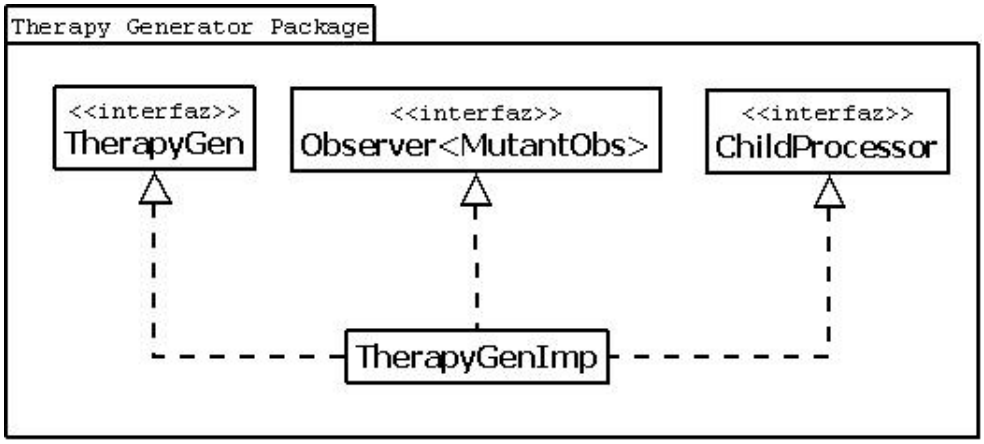

Figura 5.11: Paquete Therapy Generator.

- **TherapyGen** provee la interfaz básica de generación de terapias con la que interactua el *MainApp*.
- **ChildProcessor** provee una interfaz en la cual se establece como procesar un nodo del árbol y como descartarlo.
- **Observer(MutantObs)** provee la interfaz para observar el generador de mutantes(observador de combinaciones).
- **TherapyGenImp** esta clase, una de las más complejas del sistema, implementa las interfaces *TherapyGen*, *ChildProcessor* y *Obsever*(*MutantObs*). Su propósito es generar el árbol de terapias a partir de la secuencia inicial y los antirretrovirales de la base de datos.Además, cumple con el RF12 generando el árbol a medida que se observan las mutantes generadas. El funcionamiento del generador de terapias se explica en detalle en el capítulo 7.2.

5.4.10. MainApp

- **MainApp** la aplicación principal encargada de obtener y configurar los parámetros de entrada del sistema. Luego invoca a las funciones encargadas de calcular las terapias.

5.4.11. GUI

Para que el uso del sistema fuera más orientado a los usuarios finales, se adicionó una interfáz gráfica. La misma está implementada en Qt el cual provee sus propias clases y funcionalidades. Listaremos las más importantes.

- **MainWindow** clase que define la aplicación principal. Provee los métodos para crear ventanas y manejarlas en el sistema operativo.
- **QMenu** define a un menú con opciones y funcionalidades.
- **QString** representación de Qt de una cadena de caracteres.
- **QMessageBox** ventana con mensaje de texto para notificar errores, advertencias o información.
- **QButton** clase para representar un Botón junto a los eventos que provee.
- **QLineEdit** linea de texto editable.

- **QLabel** clase para representar un etiqueta.
- **QTab** solapas seleccionables.
- **QPrinter** clase que provee las funcionalidades de una impresora.
- **QTextDocument** clase para definir un documento de texto.

Capítulo 6

Diseño de clases asociadas a Antirretrovirales

6.1. Introducción

En la elaboración de este trabajo, uno de los mecanismo más importantes está en la aplicación de los antirretrovirales a una secuencia de ARN del virus para la confección de las terapias. Como habíamos mencionado con anterioridad, cuando un antirretroviral es aplicado, el virus intenta mutar produciendo que el antirretroviral deje de ser efectivo. Es este capítulo presentaremos los algoritmos y otros aspectos de diseño que hacen a la simulación de la aplicación de estos sobre una secuencia. Esta funcionalidad es esencial para el armado del árbol de terapias que forma parte del *TherapyGenerator* y del *AntiviralSelector* (de la Fig. 5.1).

A cada aminoácido hay una serie de tripletes (tres nucleótidos) que corresponden. Este mapeo entre tripletes y aminoácidos se denomina *Código Genético*. Esta conversión se utiliza a lo largo de los cálculos tomando los tripletes y retornando los aminoácidos que le corresponde. La funcionalidad de la conversión está definida en la librera *BIOpp*. Se puede ver que a un mismo aminoácido le pueden corresponder varios tripletes. En el apéndice se incluye una tabla con el mapa genético.

Por otro lado, recordemos que una secuencia es una cadena de caracteres, en particular una cadena de nucleótidos o de aminoácidos, ya que como se menciono en el capitulo de conceptos preliminares, hay una correspondencia entre estas dos estructuras dada por el mapa del Código Genético. De ahora en mas, secuencia, secuencia ARN, secuencia nucleotídica o de aminoácidos serán términos equivalentes.

- Secuencia de Nucleótidos:
  $CCCATTAGCCCTATTGAGACTGTACCAGTAAAATTAAAGCCAGGAAT...$
- Secuencia de Aminoácidos:
  $PISPIETVPVKLKPGMDGPKVKQWPLTEEKIKA...$

Otra forma de representar la secuencia es utilizando el formato estándar FASTA. (Ver apéndice)

6.2. Representación de un Antirretroviral

Un Antirretroviral esta representado principalmente por un listado de posiciones de resistencia, es decir, en que moléculas del virus el antirretroviral va a ser efectivo. En nuestro diseño estas posiciones serán representadas como un par:

$< pos, aminos >$, donde:
$pos \geq 0$

$$aminos = aminoacid^{+}$$

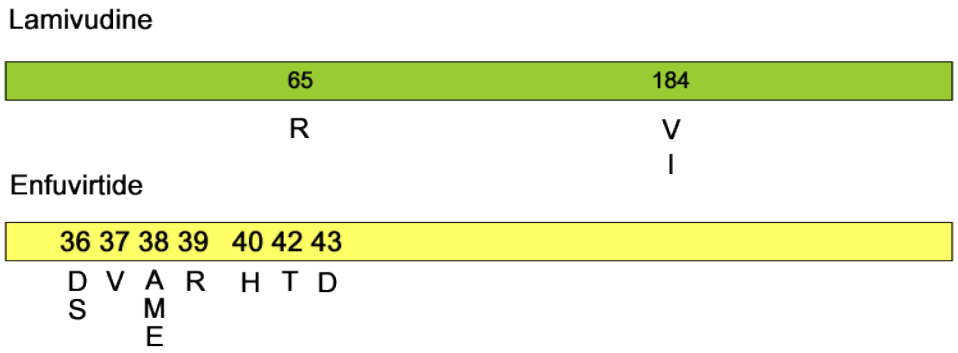

Figura 6.1: Ejemplo de dos antirretrovirales.

Es decir, la posición de resistencia describe un valor entero en el cual los aminoácidos actúan sobre un virus. Ademas se incluye otro tipo de información como un identificador, nombre, clase y tipo de antirretroviral.

6.3. Aplicación de un Antirretroviral

A lo largo de esta sección se mencionó que un antirretroviral (o un conjunto de ellos) se aplica a una secuencia nucleotídica, para eso se necesita definir un algoritmo de aplicación ya que esta depende de como se considere el antirretroviral. Esta funcionalidad es utilizada por el *AntiviralSelector* que debe obtener los antirretrovirales que aplican.

6.3.1. Algoritmo de aplicación de una sola posición

Este algoritmo toma una sola posición de resistencia del antirretroviral y la compara con la misma posición en la secuencia de ARN, verificando si aplica o no. Es decir, se asegura que ninguno de los aminoácidos de la lista *aminos* que esta en la posición *pos* correspondan con alguno de los que están en la posición $sequence[pos]$. Ademas, luego de verificar que esa posición aplica, retorna la distancia mínima, decir, de entre todos los aminoácidos buscar cual se encuentra a menor distancia del que se esta comparando en la secuencia. Si la distancia es 0 quiere decir que no se puede aplicar, ya que es el mismo.

```
<TripletList, Value> min_dist_to_resistance(sequence, resistances)
{
    TripletVal tval, min_list;
    TripletList tlist;
    Triplet pos_triplet = get_triplet(pos,sequence)

    for all (aminoacids in resistances) do
    {
        for all (triplets in GeneticCode(aminoacid)) do
        {
            insert(triplet_list, triplet)
            minimize(min_dist, distance(pos_triplet))
        }
    }
    if (min_dist != 0)
    {
        for all (triplets in triplet_list) do
        {
            if distance(triplet) == min_dist
            {
                insert(TripletList, triplet)
            }
        }
     }
     Value = min_dist
     return min_list
}
```

Algunas aclaraciones:

- La función retorna un par que contiene la lista de tripletes producida y el valor mínimo.
- Primero toma todos los aminoácidos en cada posición de resistencia.
- Obtiene los tripletes correspondientes de acuerdo al mapa genético y los coloca en una lista.
- Luego elimina de esa lista a todos los que no tienen distancia mínima.

6.3.2. Algoritmo de aplicación de un Antirretroviral

Ahora para saber si un antirretroviral aplica basta con utilizar el algoritmo anterior en todas las posiciones. Volvemos a retornar la menor distancia de entre todas las posiciones.

```
float antiviral_applies(sequence, antiviral)
{
    float min_dist = 3.0
    <TripletList, Value> T

    for all (resistances in antiviral) do
    {
        T = min_dist_to_resistance(sequence,resistance)
        min_dist = minimize(min_dist, resistance)
    }
    return min_dist
}
```

6.3.3. Ejemplo

Para aplicar un antirretroviral a una secuencia hay que garantizar que: En cada posición de la secuencia que corresponde a la posición de resistencia no coincida el aminoácido de la secuencia con ninguno de los aminoácidos de la posición de resistencia del antirretroviral. Damos un ejemplo:

- Caso en que aplica (Fig. 6.2):

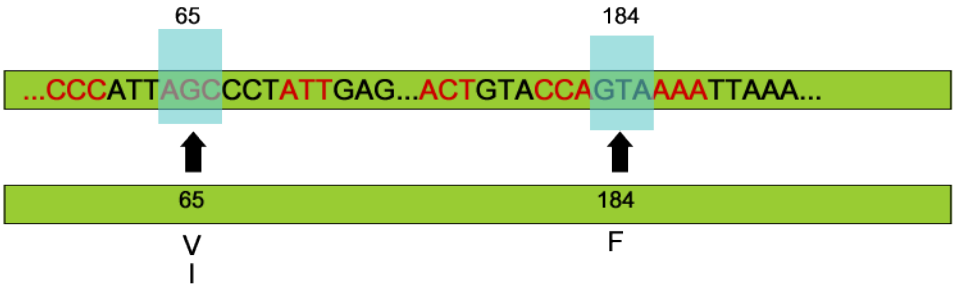

Figura 6.2: Ejemplo de aplicación de un antirretroviral.

Analicemos cada posición de resistencia para ver que ninguna coincide, además calculamos la distancia mínima entre cada de ellas con respecto a las de la secuencia.

Amino del AV	Mapa Gen	Triplete	Distancia	Mínima
V	GTT	ACG	3	
V	GTC	ACG	3	
V	GTA	ACG	3	
V	GTG	ACG	2	GTG
I	ATT	ACG	2	ATT
I	ATC	ACG	2	ATC
I	ATA	ACG	2	ATA
F	TTT	GTA	2	TTT
F	TTC	GTA	2	TTC

En este caso son 6 los tripletes con distancia mínima distinta de cero, con lo cual el antirretroviral aplica a esa secuencia.

6.4. Selector de Antirretrovirales

Como venimos mencionando en esta sección, el componente *AntiviralSelector* es fundamental en el resto de las operaciones internas del sistema. En este paso se consideran sólo los antirretrovirales con distancia mínima. Describiremos el algoritmo que conforma este elemento.

```
void select_antivirals(antiviral_set, sequence, applicable_avs)
{
    float min_dist(3.0f)

    for all (av in antiviral_set) do
    {
        float distance = applies(sequence, av)
        if (distance != 0.0)
        {
            float temp_dist = minimize(min_dist, distance)
            if (temp_dist < min_dist)
            {
                applicable_avs.clear()
                insert(applicable_avs, av)
                min_dist = temp_dist
            }
            else if (temp_dist == min_dist)
            {
                insert(applicable_avs, av)
            }
        }
    }
}
```

Algunas aclaraciones:

1. *antiviral_set* contiene todos los antirretrovirales disponibles.

2. *applicable_avs* es un conjunto con los antirretrovirales que aplican.

3. El algoritmo toma cada antirretroviral del conjunto y obtiene su distancia mínima.

4. Luego, si la distancia es distinta de 0 va colocando en el conjunto resultante los de mínima distancia.

6.5. Mutación de una Secuencia

Una vez que obtenemos el listado de antirretrovirales que aplican a una secuencia procedemos a aplicarlos, es decir, la secuencia sufre una mutación. Esta se representa cambiando el aminoácido en la secuencia (en esa posición de resistencia) por el de menor distancia (o las menores). O sea, por cada aminoácido de una posición de resistencia hay una secuencia mutante. Formalizando esto:

Por cada antirretroviral con n posiciones de resistencia, en donde cada posición de resistencia tiene a lo sumo m aminoácidos, producirá $n * m$ mutantes. De igual forma no todas las mutantes son consideradas, ya que como habíamos aclarado antes, siempre se toman los aminoácidos que se encuentran a menor distancia, es decir, $m * n$ es una cota máxima para las mutantes.

6.5.1. Algoritmo de Mutación

```
void mutate_sequence(sequence, mutations, antiviral)
{
    float min_dist = applies(sequence)
    if (min_dist > 0) then
    {
        for all (resistances in antiviral) do
        {
            <TripletList,Value> tv
            tv = min_dist_to_resistance(sequence,resistance)

            for all(triplets in TripletList)
            {
                a_mutation = sequence
                if (min_dist == Value) then
                {
                    set_triplet(pos, triplet, mutation)
                    insert(mutations, a_mutation)
                }
            }
        }
    }
}
```

Algunas aclaraciones:

1. *mutations* es un listado de las mutantes producidas.
2. Primero se verifica que el antirretroviral aplique.
3. Luego para cada posición de resistencia se obtiene su valor mínimo.
4. Se cambia para cada triplete en la lista de tripletes (producida) en la secuencia inicial.

6.5.2. Ejemplos

Tomando en consideración el ejemplo anterior, podemos ver que; como existen 6 tripletes con distancia mínima se producirán 6 mutantes de la secuencia original. La Fig. 6.3 muestra gráficamente como quedarían las mutantes.

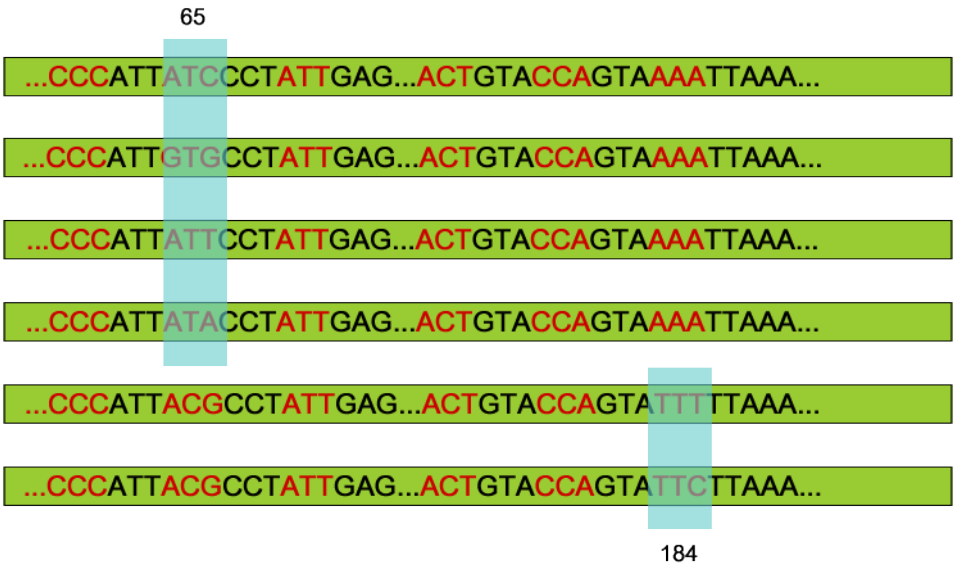

Figura 6.3: Mutantes resultantes.

Capítulo 7

Algoritmo de generación de terapias

7.1. Introducción

En este capitulo se hace una descripción detallada del algoritmo de generación de terapias, tanto de su parte constante como sus partes variables. Primero se explica la parte invarible del algoritmo, luego las partes variables de este y las distintas alternativas provistas por el SDK y finalmente se hace una breve introducción a la selección de la política de generación a través del plugin.

La elección de la política se hace a través del plugin y se proveen un conjunto de políticas convencionales por medio del SDK (Software Developement Kit). El objetivo es permitirle al usuario (implementador plugin) adaptar la generación de terapias a sus requerimientos.

7.2. Algoritmo de Generación (parte invariable)

Basándose en la necesidad de un algoritmo altamente parametrizable que pueda modificar su comportamiento de una forma sencilla, sin violar OCP. Se uso el patrón *Templatemethod* para separar el algoritmo en dos partes una parte fija (invariable) y una parte variable. Para implementarlo se desarrollaron los conceptos de algoritmo de generación para terapias (parte fija) y políticas de generación (parte variable). implementó la idea de algoritmo genérico y políticas de generación donde el sistema provee un algoritmo, que en etapas determinadas, consulta a una política cual es el siguiente paso a seguir.

```
1 void generate_therapies(init_sequence)
2 {
3     root = create_node(init_sequence)
4     process_node(root)
5 }
```

El algoritmo toma la secuencia inicial introducida por el usuario, crea un nodo y llama a una sub rutina recursiva *process_node*. Esta tiene la función de procesar el nodo para generar el siguiente nivel de mutantes.

```
Score process_node(node)
{
    valid_antivirals = select_antivirals(get_sequence(node))

    if (valid_antivirals is not empty)
    {
        combine(valid_antivirals)
        post_combinations_callback(process_node)
    }
    delete node
    return score
}
```

Funcionamiento: toma un nodo del árbol, determina el conjunto de los antirretrovirales que son efectivos contra esa secuencia (mutante). Si el conjunto de antirretrovirales aplicables no es vació se generan todas las combinaciones según el Algoritmo Combinatorio (Explicado en detalle en el capítulo 8). Una vez generadas las combinaciones se pregunta a la política a través del método *post_combination_callback* que hacer con los nodos generados en la etapa de combinación. Finalmente, al terminar de procesar el nodo se lo elimina y se retorna su puntaje.

Esta sub rutina es llamada por el algoritmo combinatorio cada vez que una combinación de antirretrovirales genera una nueva mutante.

```
void update(mutant)
{
    node = create_node(mutant)
    therapy_status = continue_therapy(node)

    case(therapy_status)
    {
        Last :
            add_child(parent, node)
            therapy = get_therapy(node)
            ranker(therapy)
            delete node

        Continue :
            add_child(parent,node)
            therapy = get_therapy(node)
            ranker(therapy)

            you_must = node_creation_callback(node)
            if (you_must == ProcessChild)
                process_node(node)

        Discard :
            delete node
    }
}
```

Funcionamiento: toma una nueva mutante, genera un nodo a partir de esta y pregunta que debe hacer con la futura terapia parcial.

- Last (someterla al ranking y descartarla): Agrega el nodo a la terapia, la coloca en el ranking y elimina el nodo.
- Continue (continuarla): Agrega el nodo a la terapia, la coloca en el ranking, a través de la política decide que hacer con el nodo generado *node_creation_callback*.
  - ProcessChild (procesarlo): llama a *process_node*.
  - ContinueWithSibling (continuar con el siguiente): continua con la siguiente secuencia y deja este sin procesar.
- Discard (descartarla): elimina el nodo.

Vale la pena notar las etapas del calculo, en la que una política iba a influenciar el algoritmo de generación. Se identificaron dos etapas que permiten hacer el algoritmo versátil:

- Después de generar un nodo *node_creation_callback*, dando la posibilidad de procesarlo o continuar con el siguiente.
- Después de generar todos los nodos hijos *post_combinations_callback*, de esta forma la política pueda indicarle al algoritmo que hacer con los nodos que le quedaron pendientes de procesar (ContinueWithSibling).

7.3. Políticas de Generación (parte variable)

La idea detrás de las políticas es disminuir el acoplamiento entre el sistema y el plugin tanto como sea posible. Una política es un objeto simple que mediante un conjunto de mensajes pre establecidos indica al algoritmo genérico que hacer en partes determinadas del calculo. Para esto las políticas implementan la interfaz *GenerationPolicy* que consta de dos primitivas:

- *node_creation_callback*: indica que hacer con un nodo una vez generado.
- *post_combinations_callback*: indica que hacer luego de generar todos los nodos hijos del nodo actual.

7.3.1. Las Políticas Provistas

A continuación daremos una descripción de las políticas provistos en el SDK junto con las ventajas y desventajas de cada uno. Notar que son algoritmos de búsqueda sobre árboles y que fueron adaptados para usarlos como algoritmos de generación.

B.F.S (Breadth First search)

Este clásico algoritmo comienza con el nodo raíz y explora todos los nodos vecinos. Luego cada uno de estos, exploran sus nodos vecinos, y así sucesivamente, hasta explorar todos los nodos.

Para hacer que el algoritmo genérico se comporte como BFS los callback se implementaron así:

```
1 CallbackAction node_creation_callback(node)
2 {
3     insert_into(pending, node)
4     return ContinueWithSibling
5 }
```

Se guarda el nodo en la lista de pendientes y se retorna *ContinueWithSibling* para indicar que se debe procesar el siguiente.

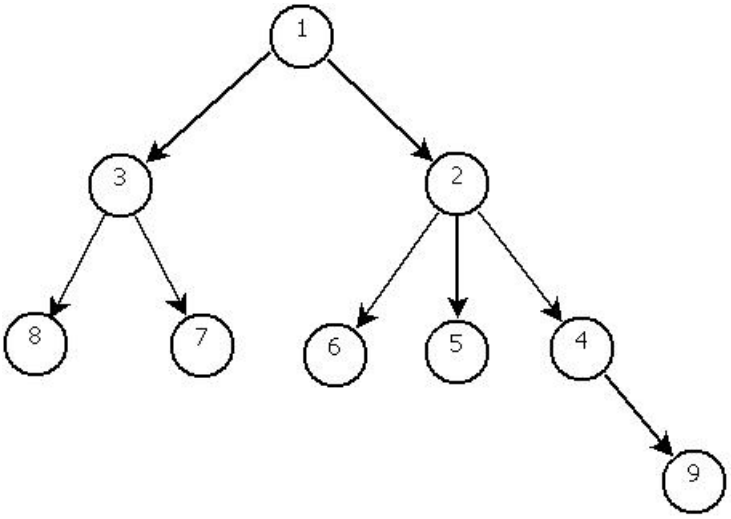

Figura 7.1: Ejemplo de orden en que se generan los nodos para BFS.

```
void post_combinations_callback(processor)
{
    for all node in pending do
    {
        processor(node)
    }
}
```

Después de realizar las combinaciones se procesan todos los nodos de la lista de pendientes.

D.F.S (Depth First Search)

Otro clásico algoritmo, en este caso se comienza con el nodo raíz y explora, tan lejos como pueda, a lo largo de cada rama antes de retroceder. Dado que el algoritmo llega al final de una rama de forma mas rápida, se podrán obtener las terapias de manera más rápida gracias al diseño basado en observadores.

Para hacer que el algoritmo genérico se comporte como DFS los callback se implementaron así:

```
CallbackAction node_creation_callback(node)
{
    return ProcessChild
}

virtual void post_combinations_callback(processor)
{}
```

Notar que cada vez que genero un nodo lo proceso inmediatamente, por lo que luego realizar las combinaciones no hay nodos que procesar.

Better Beam

Es un caso particular del algoritmo Best First Search el cual recorre un árbol pero teniendo en cuenta un criterio, en este caso la decisión de tomar los N mejores nodos. Cuando nos referimos

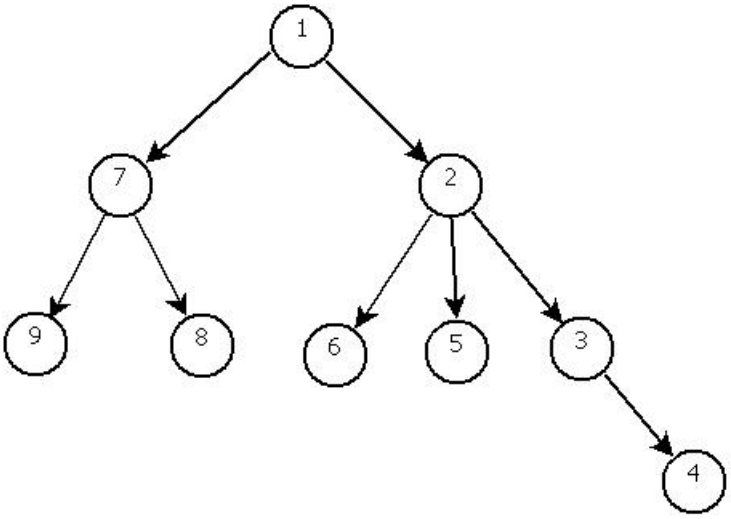

Figura 7.2: Ejemplo de orden en que se generan los nodos para DFS.

a m̈ejores; nos referimos a aquellos con puntaje más alto según la función de scoring provista por el plugin. Este enfoque reduce el tiempo de computo notablemente.

La implementación es similar a la BFS, solo que en lugar de guardar los nodos en un repositorio común (*pending*), se los guarda en un objeto *Ranker* que solo conserva los N mejores.

7.4. Especificando la Política al Plugin

Como se aclaro anteriormente, el plugin es quien da al sistema la política a usar, lo cual da al implementador de plugin la posibilidad de definir su propia política de generación. Esto se logra proveyendo en la interfaz *plugin* el método *generation_policy*() que provee al sistema de un objeto política. Dentro de este método el implementador puede optar por retornar su propia política o optar por una de las políticas pre diseñada:

- $BetterBeam(N)$, donde N es un entero mayor a 1.
- *Depth* recorrido en profundidad.
- *Breadth* recorrido en ancho.

Capítulo 8

Combinación de Antirretrovirales

8.1. Introducción

En este capítulo se describe como surge la necesidad de combinar antirretrovirales. Ademas se introducen las distinta primitivas combinatorias provistas por el SDK y finalmente se explica como se asocia el concepto de fallo virológico con la modificación de las primitivas de combinación.

8.2. ¿Por qué usar combinaciones de antirretrovirales?

Los antirretrovirales individualmente no suprimen la infección por VIH a largo plazo, por lo cual actualmente se usan combinaciones de estos. Las combinaciones de antirretrovirales actúan incrementando el número de obstáculos para la mutación viral, manteniendo bajo el número de copias virales. Las combinaciones de antirretrovirales pueden ser de sinergismo positivo o negativo. Esto limita el número de combinaciones disponibles. Por ejemplo, la combinación de *ddI* y *AZT* es de sinergismo negativo, ya que tomados juntos, cada fármaco inhibe la acción del otro. Otros factores que limitan las combinaciones son la aparición de efectos colaterales severos. La necesidad de un horario de administración complicado, dificulta la adherencia apropiada al tratamiento. La infección por el virus de hepatitis B (VHB) puede dificultar que las personas con VIH tomen medicación antirretroviral (ARV). Debido a que el VHB puede ocasionar graves daños hepáticos si no se trata ya que muchos ARVs son metabolizados por el hígado.

8.3. Funcionalidad

Siguiendo con la idea de un sistema altamente parametrizable. Se optó por proporcionar las primitivas combinatorias al sistema, a través del *Plugin*. Además, a diferencia de las políticas de generación (cap. 7) que se establecen al comienzo de la ejecución, la primitiva combinatoria debía poder cambiar en cada nodo de la terapia (RF13). Así, en caso de un fallo virológico la terapia podía tomar el rumbo que el usuario determine a través de un cambio en la primitiva de combinación.

Una primitiva combinatoria representa una forma de combinar antirretrovirales. Es decir, debe generar subconjuntos de un conjunto de antirretrovirales. Además, al igual que en todo el sistema se usa el patrón *Observer*, las primitivas deben notificar al *observador de combinaciones* cada nueva combinación. Finalmente, al terminar debe retornar un objeto *Status* indicando si hubo o no fallo virológico.
Desde el punto de vista del sistema, el *observador de combinaciones* recibe una combinación, genera las mutantes resultantes de aplicar esa combinación y las entrega al siguiente observador.

Todos las primitivas deben implementan la interfaz *GenerationPolicy* la cual consta de dos métodos importantes:

- *void set_observer*(*observer*) Establece el observador de combinaciones, el cual va a recibir las combinaciones a medida que se generan.

- *void combine*(*antivirals*, *status*) Este método realiza las combinaciones a partir del conjunto de antirretrovirales. Ademas, toma como argumento un objeto *Status* para establecer el estado luego de combinar.

8.4. Lista de primitivas combinatorias

Dada la complejidad de generar algoritmos combinatorios, se proveyó junto con el sistema de un conjunto acotado de algoritmos (SDK). Este conjunto no implementa algoritmos complicados, pesados y difíciles de mantener. En su lugar se opto por un grupo de algoritmos simples y ensamblables, de esta forma el usuario puede desarrollar primitivas tan complejos como desee usando unidades simples y fáciles de manipular.

Las primitivas se pueden dividir en dos clases, simples y compuestas. Las primitivas simples son algoritmos combinadores propiamente dichos. En cambio, las compuestas permiten combinar a las simples, pero no producen combinaciones por si solas.

A continuación se listan y explican las primitivas simples provistas en el SDK.

8.4.1. Combinador vació (EmptyCombinatory)

Esta primitiva cumple la función de *skip*, no retorna ninguna combinación y el *Status* no indica fallo virológico.

8.4.2. Combinador Newtoniano (CombinatoryNewton)

En este caso, se implementa el clásico algoritmo que retorna todos los subconjuntos de tamaño K de un conjunto de tamaño N. El *Status* retorna fallo virológico ($Fail$) en caso que el conjunto de antirretrovirales sea vació ($N = 0$) o si $K¿N$

```
1 void choose(antivirals, k)
2 {
3     combination = empty_combination(k)
4     comb_builder(combination, k, antivirals)
5 }
```

choose toma como argumento el conjunto de antivirales de tamaño N y el tamaño de los subconjuntos K. Ademas, internamente llama al método recursivo *comb_builder* con una combinación vacía.

```
void comb_builder(combination, pos, antivirals)
{
    if (pos == 0)
    {
        notify(combination)
    }
    else
    {
        for all antiviral in antivirals do
        {
            combination[pos-1] = antiviral
            comb_builder(combination,pos-1,antivirals-{antiviral})
        }
    }
}
```

Esta sub rutina toma un antirretroviral del conjunto, lo introduce en la posición *pos* de la combinación y se llama recursivamente con esta combinación, la posición anterior (libre) y el conjunto de antirretrovirales menos el antirretroviral que ya esta contenido en la combinación. Así, sucesivamente hasta llenar la combinación con k elementos. Cuando llega a la posición 0 (caso base) notifica al observador la nueva combinación.

8.4.3. Listado Simple (ListCombinatory)

Este algoritmo retorna uno a uno los antivirales del conjunto que se le da como argumento. Es decir, retorna combinaciones de un único antirretroviral. El *Status* retorna *Fail* cuando el conjunto de antirretrovirales de entrada es vació.

```
void combine(antivirals, status)
{
    if (antivirals != empty)
    {
        for all antiviral in antivirals do
        {
            update(antiviral)
        }
        status = Success
    }
    else
    {
        status = Fail
    }
}
```

El funcionamiento es simple, verifica que el conjunto de antirretrovirales no sea vació. Luego por cada antirretroviral del conjunto llama a la sub rutina *update* del observador para notificar la nueva combinación.

A continuación se explican las primitivas compuestas provistas por el SDK.

8.4.4. Combinador Secuencial (SeqCombinatorialGroup)

Este algoritmo funciona como nexo entre diferentes algoritmos. Básicamente, toma un conjunto de combinadores y los hace combinar en secuencia, uno detrás de otro. Podría pensarse como una

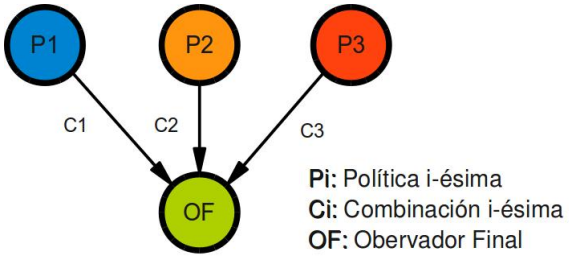

Figura 8.1: Combinador Secuencial.

cola (FIFO) de combinadores, el primero en ser introducido es el primero en combinar. Esta primitiva provee el método *add_policy*(*combine_alg*, *antivirals*), donde *combine_alg* es el algoritmo combinador y *antivirals* es el conjunto de antirretrovirales con los cuales se le permite combinar.

```
1  void combine(available_av, status)
2  {
3      status = Fail
4      aux_status = Fail
5
6      for all combinator in set_of_combinators do
7      {
8          intersec = set_intersection(available_av,get_av(combinator))
9
10         combine(intersec, aux_status)
11         if (aux_status == Success)
12         {
13             status = aux_status
14         }
15     }
16 }
```

El algoritmo toma un *combinador* del conjunto, realiza la intersección entre los antirretrovirales disponibles y aquellos que acepta el combinador. Luego, combina la intersección de antirretrovirales y guarda el status en *aux_status*. Si el *aux_status* es *Success* pisamos en valor de *status*, en caso contrario se deja *Fail*. De esta forma, si todos los algoritmos combinatorios fallan, se considera que el secuencial fallo.

8.4.5. Combinador Paralelo (ParallelCombinatory)

Este algoritmo toma un grupo de primitivas combinatorias y las hace correr en "paralelo". De manera que, cada combinación del primero se une con cada combinación del siguiente y así sucesivamente hasta que la última combinación es entrega al observador.

Esta primitiva tiene internamente una lista de N observadores donde los primeros $N-1$ se denominan observadores auxiliares (*AuxObserver*) y el último observador final (*FinalObserver*). Hay exactamente un observador auxiliar por cada primitiva combinatoria contenida en la primitiva paralela, la función de estos observadores es actuar como nexo entre primitivas y unir las combinaciones intermedias. El observador final se encarga de trasmitir la combinación resultante al observador de la primitiva paralela.

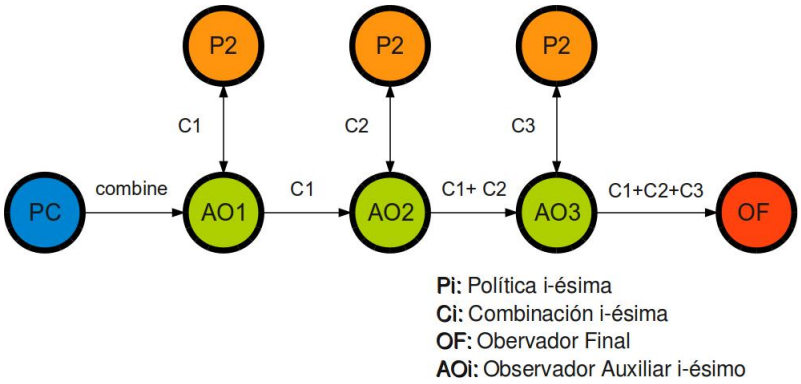

Figura 8.2: Combinador Paralelo.

```
void add_policy(policy, reference_av, is_mandatory)
{
    aux_observer = new AuxObserver(policy,reference_av,is_mandatory)
    enqueue(aux_observer)
}
```

add_policy se encarga de incorporar primitivas combinatorias al combinador paralelo. Toma como argumento la primitiva combinatoria (*policy*), el conjunto de antirretrovirales con los que se le permite combinar a la política (*reference_av*) y un flag que indica si la primitiva es obligatoria o no (*is_mandatory*). En caso de ser obligatoria, si no genera combinaciones se considera fallo virológico. Si no es obligatoria, y no genera combinaciones es ignorada y se continua con la siguiente en la cola.

```
void combine(available_av, status)
{
    combine_obs(first_obs, empty_comb, available_av,status)
}
```

El método *combine* de la primitiva paralela implementa la interfaz *CombinatoryPolicy*. Su única función es llamar al método *combine_obs* del primer observador en la lista, con una combinación vacía, los antirretrovirales disponibles y el status.

```
void combine_obs(observer, before_comb, available_av, status)
{
    intersec = set_intersection(reference_av, available_av)

    combine(intersec, status)

    if (status == Fail)
    {
        if (!is_mandatory)
        {
            combine_obs(next_obs, before_comb, available_av,status)
        }
        else
        {
            notify_fail(status)
        }
    }
}
```

Este método (*combine_obs*) toma el observador sobre el que va a combinar, la combinación del combinador anterior, los antirretrovirales disponibles para combinar y el status. Primero, calcula la intersección entre el conjunto de antirretrovirales de referencia y el conjunto disponible para combinar (*intersec*). Luego, ejecuta la primitiva combinatoria con *intersec*. Si el status resultante de la combinación es *Fail* hay dos opciones:

- la primitiva no es obligatoria, en cuyo caso se llama al combinador del siguiente observador auxiliar *next_obs* con la combinación anterior $before\_comb$.
- la primitiva es obligatoria, en este caso se notifica un fallo virológico.

```
void update_obs(combination)
{
    union = set_union(combination, before_combination)
    combine_obs(next_observer, union, available_av, status)
}
```

Este método es llamado cada vez que la primitiva, asociada a este observador auxiliar, genera una combinación (*combination*). Básicamente, se une la primitiva actual con la anterior (*union*) y se llama al combinador del siquiente observador.

```
void combine_obs(observer, combination, available_av,status)
{
    update(parallel_observer, combination)
}
```

Esta versión de *combine_obs* pertenece al observador final y implementa la misma interfaz. La función de este método es enviar la combinación resultante de la unión de todas las combinaciones al observador de la primitiva paralela.

Ejemplo de uso de la primitiva paralela:

```
1
2  subset_1 = filter(available_antivirals, cPI)
3  subset_2 = filter(available_antivirals, cNRTI)
4  subset_3 = filter(available_antivirals, cNNRTI)
5
6  list_comb_1 = new ListCombinatory("list␣cPi")
7  list_comb_2 = new ListCombinatory("list␣cNRTI")
8  list_comb_3 = new ListCombinatory("list␣cNNRTI")
9
10 example_parallel = new ParallelCombinatory("parallel")
11 example_parallel.add_policy(list_comb_1, subset_1, true)
12 example_parallel.add_policy(list_comb_2, subset_2, true)
13 example_parallel.add_policy(list_comb_3, subset_3, false)
14
15 example_parallel.combine(apply_antivirals, status)
```

En el ejemplo se hace uso del método $filter$, una de las herramientas provistas por el SDK para la construcción de plugins. $filter$ genera un subconjunto ($subset\_N$) de un conjunto mayor ($available\_antivirals$) con todos los elementos que cumplen cierto parámetro (cPI, $cNRTI$ y $cNNRTI$). Luego se crean tres primitivas de listado simple ($list\_comb_N$), una para cada subconjunto de antirretrovirales. A continuación, se crea la primitiva paralela ($example\_parallel$) y se le incorporan las primitivas secuenciales a través de add_policy. Finalmente, obtenemos una primitiva que realiza combinaciones de tamaño tres de los cuales uno es de tipo cPI, otro de tipo $cNRTI$ y el último de tipo $cNNRTI$.

8.5. Cambio de la primitiva combinatoria en caso de Fallo Virológico

Como respuesta a los requerimientos RF5.7 y RF13 se hicieron modificaciones en el sistema y en la interfaz de plugin para que se pudiera cambiar la primitiva combinatoria en cada nodo de una terapia. Esto es posible gracias al método $get\_new\_combination\_policy(status)$ presente como interfaz en cada nodo. Así, si se recibe un status $Fail$ (fallo virológico) por parte de una primitiva, el sistema pregunta al plugin por una nueva. En caso que el plugin entregue una nueva primitiva para ese nodo el sistema hace combinar la nueva primitiva. En caso contrario se da por finalizada la terapia.

Capítulo 9

Resultado de la ejecución

El resultado del programa es sin duda el elemento clave de este sistema con el cual el usuario final dictaminará la aplicación de una terapia o analizará alguna de sus teorías. Teniendo presente esta premisa, es necesaria una salida clara y bien informada de lo obtenido. Una vez ejecutado el sistema se retornará un en de texto plano formato *.trp* que contiene el listado de las terapias dentro del ranking, de a acuerdo a las especificaciones del usuario junto con información asociada a las mismas y a su elección. Otro de los aspectos a tener en cuenta respecto a este trabajo es que el sistema ayuda a determinar *cómo* comenzar una terapia pero no *cuando*. Es necesario volver a aclarar que a partir de este punto está en el criterio y conocimiento del usuario su aplicación clínica.

9.1. Información contenida en las terapias

Una terapia en particular dispone de la siguiente información:

- Fecha de realización del cálculo.
- Observaciones relacionadas a ese ranking.
- Una secuencia de conjuntos de antirretrovirales aplicados a cada paso.
- Listado de mutantes generadas a cada paso (opcional).
- Puntaje asociado utilizado para elaborar el ranking de la terapia.
- Información adicional establecida por el plugin (opcional).

9.2. Ranking de una terapia

Se presenta un listado terapias ordenado de mejor puntaje en adelante. Como ejemplo presentamos un archivo de salida de lo obtenido.

```
Mon Jun 28 11:05:47 2010

------------------------------------------------------------
Terapia de prueba
------------------------------------------------------------

Therapy: 1
Score: 0.2

Step 1: Saquinavir + Zidovudine |
Step 2: Saquinavir + Tenofovir  |

Therapy: 2
Score: 0.3

Step 1: Saquinavir + Nevirapine |
Step 2: Saquinavir + Zidovudine |
Step 3: Saquinavir + Tenofovir  |
```

Este es un ejemplo de una salida del sistema que muestra un ranking con dos terapias posibles junto con información agregada al encabezado.

Podemos ver el resultado como un árbol en donde cada camino desde la raíz a una hoja es una terapia. Los nodos $M1$ y $M2$ (Fig. 9.1) son las mutantes de la secuencia inicial, mientras que $M11$ y $M12$ son las mutantes de $M1$.

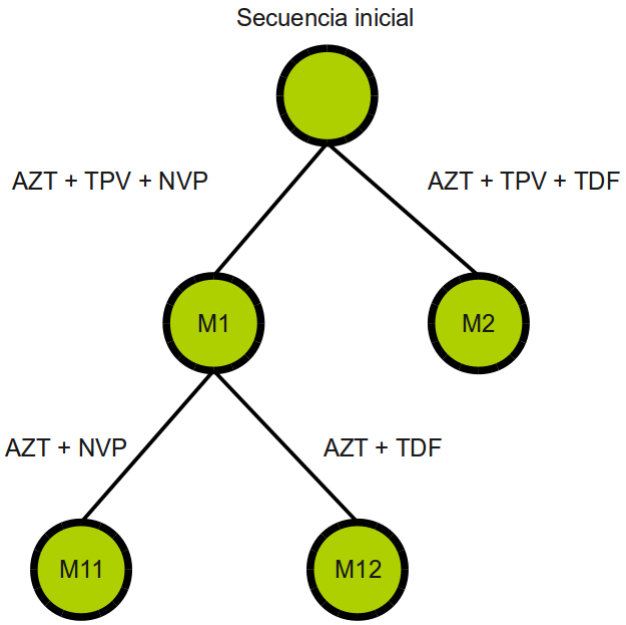

Figura 9.1: Árbol de terapias.

Capítulo 10

Plugin

10.1. Introducción

Uno de los requerimientos fundamentales de este sistema es la posibilidad de extensión y modificación de funcionalidad por parte de sus usuarios. La idea general radica en que el usuario de este software pueda agregar información o cálculos específicos, como ser en las funciones de scoring,la generación del nodo, función de poda, etc. Para esto proveemos un mecanismo de integración de plugins implementados con librerías dinámicas del sistema, dando libertad para explorar teorías o cálculos que se requieran.

10.2. Plugin

Este ha sido uno de los requerimientos principales desde el primer planteo del sistema, ya que representa uno de los componentes principales que permiten extender la funcionalidad sin modificar el framework (OCP). Fue esa la razón por la cual se hizo distinción entre los tipos de usuarios mencionados en el capítulo 1. Apuntamos a que el sistema pueda ser altamente parametrizable a través de este mecanismo de plugins.

10.3. Nodo de Terapia

Anteriormente nos habíamos referido a las terapias como ramas en un árbol en donde cada nodo representa la aplicación de una nueva combinación de antirretrovirales y la secuencia mutante del virus, causada por esta. Cada elemento del árbol es denominado como Nodo de Terapia ($TheapyNode$) y que consta de lo siguiente:

1. Un identificador único.
2. Una referencia a su nodo padre (si es que posee).
3. Una secuencia nucleótidica.
4. El conjunto de antirretrovirales que produjeron la secuencia.
5. Algoritmo de combinación.
6. Información extra que se quiera agregar a cada paso.

10.4. Alcance del Plugin

La integración del plugin al sistema permite modificar funcionalidades u obtener información especifica, siendo este capaz de poder realizar las siguientes extensiones o cambios:

- Elegir la política de generación entre las disponibles o proponer una propia: el capítulo 7 se presentaron cuales son las políticas proporcionadas por el sistema a través del SDK. El usuario que implementa el plugin también puede usar su propia política de generación implementando la interfaz *GenerationPolicy*.
- Definir una función de scoring especifica, que asigne un puntaje a una terapia: esta función tiene una gran responsabilidad a la hora de construir las terapias, ya que un scoring adecuado proporcionaría mejores resultados. Esta función debe satisfacer la interfaz *Score scoring*() y puede ser tan compleja como se desee, además debe ser especificada en el plugin.
- Cambiar la implementación interna de ciertos tipos de datos como $Info$, *TherapyNode* y *Antiviral*. Permitiendo incorporar información adicional para cálculo y fundamentación de la terapia ante el usuario.
- Definir una función de poda en el espacio de búsqueda de terapias. Esta se encarga de eliminar del árbol de terapias todas aquellas, que por diversos motivos sean inviables desde el punto de vista de los criterios definidos en el plugin. En el sistema implementa la interfaz *TherapyStatus continue_therapy*() y establece el estado de una terapia parcial (*TherapyStatus*).
- Seleccionar cuales serán los antirretrovirales a utilizar. Esta opción permite seleccionar, de la Base de Datos, un subconjunto de antirretrovirales antes de hacer el cálculo. Descartando aquellos que por ejemplo no están disponibles en la zona o se consideran innecesarios.
- Elegir la primitiva combinatoria (presentado en el capítulo 8) a aplicar en cada paso de la terapia. Estas primitivas se establecen en cada *TherapyNode* de la terapia al momento de generar el nodo.
- Capacidad de interactuar con el usuario en tiempo de ejecución: el plugin puede solicitar al usuario una cantidad variable de parámetros para realizar sus cálculos internos. Por ejemplo, recuento de CD4, edad del paciente, etc. Es decir, el sistema proporciona los mecanismos para que las funcionalidades del plugin se comuniquen con el usuario en tiempo de ejecución.
- Definir los valores de la matriz de distancia de nucleótidos: la distancia entre un nucleótido y su mutante está dado por una matriz que define cada distancia. El plugin es capaz de establecer cual es su valor.

Por otro lado el plugin no permitirá:

- Modificar registros de la base de datos: dado que la base de datos es reutilizada para otros cálculos, no es conveniente permitir que se eliminen ó alteren registros.
- Establecer el orden del Ranking: de forma predeterminada el ranking esta ordenado de mayor puntaje a menor puntaje en base a la función de scoring, es decir, de la mejor en adelante. No se posibilita que el plugin decida otro orden basado en algún otro criterio.
- Elegir la forma en que un antirretroviral aplica: el sistema trabaja con definición de aplicación de los antirretrovirales de acuerdo de sus posiciones de resistencia, pero podría ocurrir que trabaje con probabilidades u otros enfoques. Por el momento el sistema no soporta otra forma para considerar antirretrovirales.
- Cambiar la forma en que funciona el Administrador de Plugins: El *PluginAdministrator* es el encargado de integrar el *Plugin* al sistema, un plugin no puede modificar la forma en que se integrara al sistema ya que podría ser causal de un funcionamiento erróneo.

10.5. Administrador de Plugin

La entidad a través de la cual el sistema se comunica con un plugin es el *PluginAdministrator*, este es parte del framework y establece en que partes del sistema tendrá injerencia el plugin. provee las siguientes funcionalidades:

- Cargar un plugin: introducir al sistema todas las funcionalidades provistas por el plugin.
- Descargar un plugin: remover del sistema los elementos del plugin, permitiendo introducir uno diferente.
- Chequear consistencia: del plugin con el sistema y del plugin con la base de datos.
- Obtener la lista de parámetros: lista de argumentos necesarios para que el plugin haga los cálculos y que debe establecer el usuario.

Es decir, el plugin por sí solo no provee una interacción con el framework, si no que implementa las funcionalidades. El *PluginAdministrator* maneja esas implementaciones y las introduce al sistema, haciendo un chequeo de consistencia.

10.6. SDK Libpluginaso

Las herramientas básicas para construir un plugin están provistas por la librería *libplguinaso*. El SDK tiene como objetivo simplificar la creación de plugins a los usuarios con conocimientos básicos de computación.

El usuario que desarrolle plugins debe definir cada una de las primitivas en la interfaz *Plugin* y $TherapyNode$, ó usar como base de desarrollo el plugin Base explicado más adelante.

10.6.1. Implementación

Se optó por utilizar librerías dinámicas de C++ para su implementación, ya que si bien se había propuesto confeccionar el plugin en lenguaje Pyhton, decidimos que para el alcance de este trabajo es más viable poder implementar los plugins en el mismo lenguaje del framework. Sin embargo sugerimos que en próximas versiones del sistema se incluya la posibilidad de manejar plugins con este otro lenguaje ya que es común su uso por parte de especialistas en el área de la biología.

10.6.2. Compilación de un plugin

Los recursos necesario para la construcción del plugin se encuentran en el directorio *libplugin*. Una vez confeccionado el plugin con las funcionalidades deseadas se compila con el $Makefile$ que se encuentra en ese mismo directorio. Esto crea un archivo con extensión *.so* que será utilizado en el framework.

10.6.3. Base Plugin

Para este trabajo incluimos otro plugin con funcionalidades por default, que implementa parte de las interfaces necesarias a un nivel sumamente básico. Tiene como objetivo proveer una base a partir de la cual generar plugins más complejos.

- Nombre: `Default Plugin`
- Función de Scoring: sin implementación.
- Función de poda: *Continue*. Siempre continuar la terapia.

- Política de generación: `Width`
- Política de Combinatoria: `Empty`. Sin combinatoria.

10.7. Thesis Plugin

Se provee un plugin en este trabajo, con muchos de los lineamientos más generales del tratamiento contra el VIH. *ThesisPlugin* fue pensado a modo de ejemplo, su función es mostrar como se usan muchas de las herramientas del SDK. Vale aclarar que tiene cierta complejidad y cuenta con las siguientes funcionalidades:

- Nombre: `Thesis Plugin`
- Función de Scoring: Esta función retorna la suma de las distancias mínimas entre mutantes.

```
score scoring(node)
{
    if(parent(node) != NULL)
        partialTherapyScore = scoring(parent(node)) + value(node);
    else
        partialTherapyScore = value(node);
}
```

  Se computa al scoring sumando los valores que posee el nodo.

- Función de Poda: Esta función retorna siempre *Continue*, por lo cual ninguna terapia se descarta.
- Política de generación: Se decidió usar una política exhaustiva como `Depth`, la cual permite activar ciertas optimizaciones en el manejo de memoria.
- Política de Combinatoria: Se decidió inicialmente combinar con tres antirretrovirales, en particular comenzar con dos de tipo $NNRTI$ y uno PI o con dos de tipo $NRTI$ y uno PI . Continuar así hasta que se llegue a un fallo virológico y entonces descartar el antirretroviral para el cual ocurrió el fallo y seguir con combinaciones de tamaño dos, repetir este proceso en cada fallo virológico, hasta que no queden más.
- Matriz de Distancia: Se necesita tomar una matriz con distancia entre nucleótidos igual a 1.

```
set_default_matrix(DistanceMatrix Matrix)
{
    for all (columns in Matrix) do
    {
        for all (rows in Matrix) do
        {
            if(row==column) then
                Matrix[row][column] = 0
            else
                Matrix[row][column] = 1
        }
    }
}
```

  Es Decir, la matriz quedaría definida como sigue:

	A	C	G	T
A	0	1	1	1
C	1	0	1	1
G	1	1	0	1
T	1	1	1	0

Capítulo 11

Base de Datos

11.1. Introducción

Otro de los componentes externos al sistema es la base de datos de antirretrovirales que son necesarios para la generación de las terapias. Se tomo la base de datos de forma independiente con lo cual esta puede ser manipulada o reemplazada sin acceder al código del software y que da libertad al usuario de poder configurarla con la información necesaria (OCP). Inicialmente y a modo de ejemplo utilizaremos una base de datos genérica con los 18 antirretrovirales provistos por el I.A.S. (*www.iasociety.org*) [13] que se compone de los antirretrovirales utilizados en la actualidad y que se listan en la sección 2,3.

11.2. Base de datos genérica

La base de datos que incluimos en este sistema provee información básica de los antirretrovirales a utilizar como su nombre oficial, un id para indexación y un listado de posiciones de resistencia (ver capítulo 6). El formato elegido para su representación es XML. También anexamos a la información la clase y el tipo de antirretroviral, donde la clase puede ser:

- Protease Inhibitor (PI).
- Nucleoside and Nucleotide Analogue Reverse Transcriptase Inhibitors (NRTI).
- Nonnucleoside Analogue Reverse Transcriptase Inhibitors (NNRTI).

A su vez, como mencionamos anteriormente las dos ultimas clases se clasifican dentro del mismo tipo de antirretroviral, los Inhibidores de Transcriptasa reversa, con lo cual definimos también ese tipo, pudiendo ser:

- Protease Inhibitor.
- Analogue Reverse Transcriptase Inhibitors.

11.2.1. Ejemplo de un registro en la base de datos

Como ejemplo mostramos la forma en que se denota un antirretroviral en la base de datos:

```
<antiviral id="Abacabir"  num = "0" type = "tRT" class = "cNRTI">
  <resistance pos="65" aminos="R"/>
  <resistance pos="74" aminos="V"/>
  <resistance pos="115" aminos="F"/>
  <resistance pos="184" aminos="V"/>
</antiviral>
```

11.3. Agregar información a la base de datos

La información incluida en la base de datos es la mínima indispensable para poder hacer los cálculos requeridos por el sistema, como ser la aplicación de un antirretroviral a una secuencia. Con lo cual, las posiciones de resistencia son los datos principales para poder ejecutarse en caso de que elijamos plantear aplicaciones por resistencias. ¿ Pero,qué sucede si queremos considerar otras variables como atributos extra que posea el antirretroviral, por ejemplo, su toxicidad, costo, entre otras, y que son cruciales para la elaboración de una terapia?. Permitiremos anexar esta información a la base de datos (addendum) y desde el plugin especificar que esa variable sea tenida en cuenta para la confección de la terapia.

11.3.1. Addendum

Se definió con este nombre a la extensión de los datos de un antirretroviral en la base de datos. De forma muy simple se puede colocar un nuevo conjunto de atributos con su valor. Para eso se agrego la siguiente linea luego de la especificación de las resistencias:

```
...
<addendum attrib="price" value="34"/>
<addendum attrib="toxicity" value="0.34"/>
<addendum attrib="pregnancy" value="1"/>
...
```

Para garantizar la completitud y correctitud de los datos se debe corroborar que las nuevas variables estén presentes en cada una de las definiciones de los antirretrovirales y que estas sean correctas, por ejemplo, que el valor de un aminoácido sea realmente un aminoácido. La razón es que el sistema no pre-procesa la base de datos en búsqueda de inconsistencias, para eso podemos estableces un criterio de los datos y que se puede definir en el plugin. Es decir, que los datos sean correctos dependerá del contexto del cálculo que se desee realizar.

11.4. FXP

Para el parseo del archivo XML se utilizó FXP (Fudepan XML Parser), esta librería está pensada para trasladar información de un árbol XML a una estructura específica del sistema, en nuestro caso, la representación del antirretroviral.

Capítulo 12

Interfaz de Usuario

12.1. Introducción

Teniendo en cuenta a quienes está destinado el uso del sistema se planteó la necesidad de incluir una interfaz gráfica que facilite su uso y comprensión. La importancia de proveer una interfaz gráfica hace en muchos casos a la popularidad del uso de un sistema, teniendo en cuenta que los usuarios principales no están familiarizados con consolas de comandos de texto. En este caso incluimos una modesta interfaz gráfica que simplifica la carga de los parámetros que *ASO* utiliza y presenta los resultados obtenidos.

12.2. Implementación de la interfaz gráfica

Para su implementación utilizamos Qt (`nokia.qt.com`), un framework multiplataforma que provee una gran cantidad de Widgets (elementos de interfaz gráfica) para construir GUI's de forma intuitiva y simple. La interfaz permite cargar los archivos necesarios y configurar los parámetros del plugin, así como visualizar los resultados obtenidos para luego analizarlos e imprimirlos. Su versatilidad permite que la interfaz cree Widgets de acuerdo a los parámetros necesarios que utilice el plugin. Esto es crucial ya que el papel del plugin en el sistema es fundamental para su funcionamiento. También se tiene en cuenta que estos parámetros poseen valores por defecto para facilitar la configuración del sistema.

12.2.1. Dependencias de la interfaz

Para poder utilizar la GUI sera necesario tener:

- Sistema operativo Linux o Windows.
- Qt versión 4.6 o superior.

12.2.2. Demostración y uso de la interfaz

A continuación exhibimos algunas imágenes de la interfaz.

1. Compilación: En el directorio *qt_gui* se encuentra el archivo Makefile que permite la compilación del sistema.
2. Ventana Principal: En esta primera parte se seleccionan los archivos correspondientes a la secuencia inicial y la base de datos ademas se debe indicar el nombre del archivo de salida. Opcionalmente se pueden agregar comentarios que serán anexados al archivo de las terapias (Fig. 12.1).

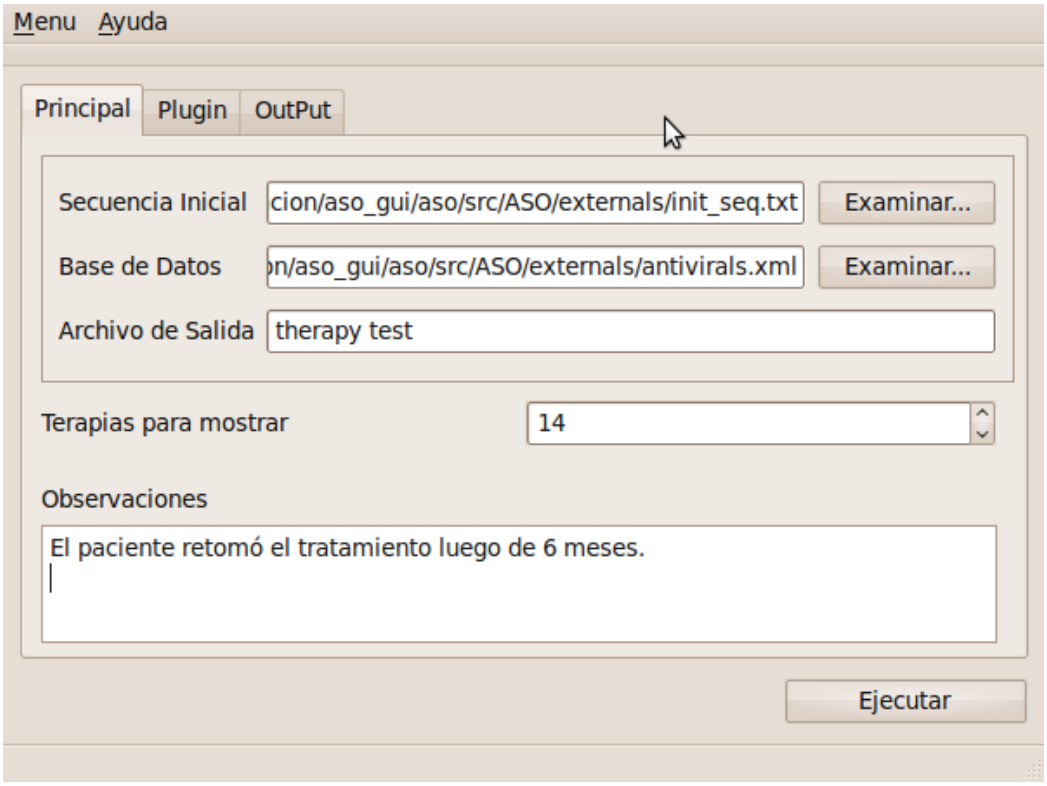

Figura 12.1: Pantalla principal.

3. Plugin: Se debe seleccionar un plugin con extensión *.so* el cual debe ser construido previamente. Una vez cargado el plugin se muestran los parámetros configurables que este posee (Fig. 12.2).

4. Output: Una vez configurados los parámetros requeridos y ejecutado el programa, en la solapa de OutPut podremos ver la salida con el ranking de las terapias pudiendo este imprimirse. Los archivos de salida se nombran con extensión *.trp* y son colocados en el directorio *output* (Fig 12.3).

12.3. Interfaz de Consola

También se provee una interfaz de consola que permite la utilización del sistema a usuarios que no simpaticen con las interfaces gráficas o que pretendan ahorrar rendimiento del sistema. La sintaxis de la interfaz de consola se define como:

```
./aso SEQUENCE_FILE DATABASE_FILE PLUGIN_FILE RANK_VALUE OUT_FILE
```

Donde:

- $SEQUENCE\_FILE$: Ruta del archivo de secuencia inicial. Debe ser de extensión *.txt*.
- $DATABASE\_FILE$: Ruta del archivo de la base de datos. Debe ser de extensión *.xml*.
- $PLUGIN\_FILE$: Ruta del archivo con extensión *.so* que corresponde al plugin.
- $RANK\_VALUE$: Cantidad de terapias a tener considerar en el ranking. Debe ser un valor mayor a 0.

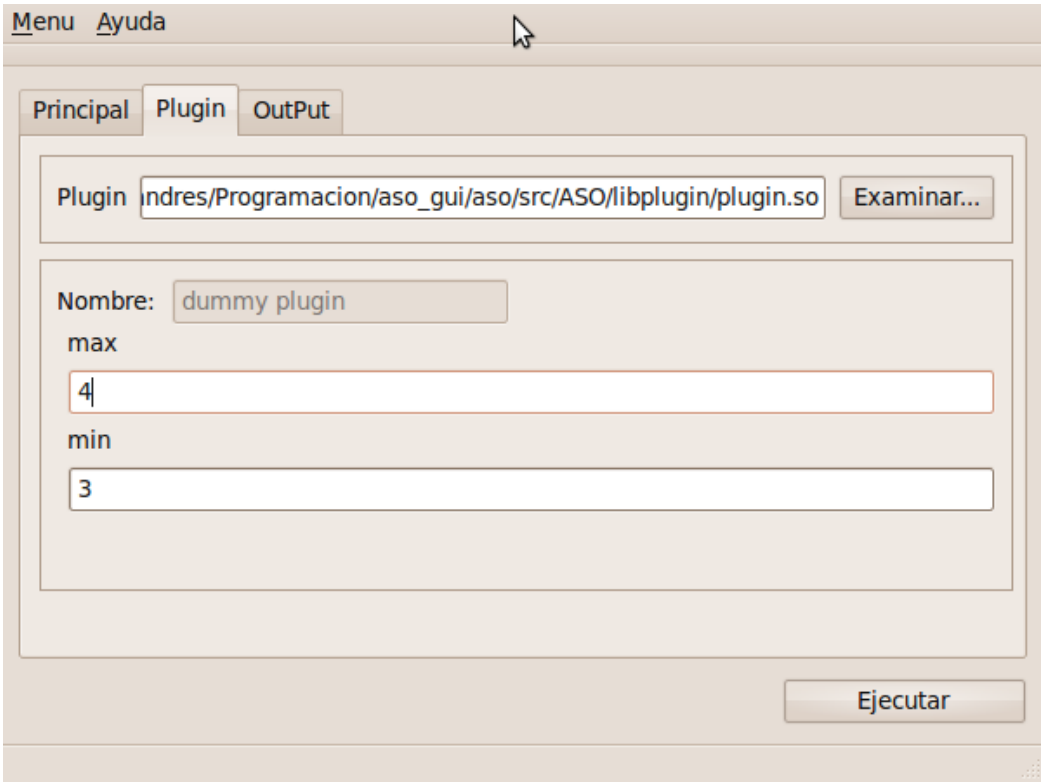

Figura 12.2: Pantalla de configuración de Plugin.

- *OUT_FILE*: Nombre del archivo de salida.

Luego el sistema pide si se desean colocar algunas observaciones al archivo de salida y a continuación solicita completar los parámetros del plugin. Los resultados son retornados en el mismo formato (un archivo con extensión *.trp*) en el directorio *outputs*.

12.4. Interfaz híbrida

En un momento se propuso implementar una interfaz gráfica con Zenity (`http://live.gnome.org/Zenity`), una herramienta que permite mostrar elementos gráficos de $GTK+$ utilizando scripts de bash. Esto permitía que algunos elementos del sistema pudieran ser configurables a través de cuadros de texto. El problema radicó en que zenity tiene grandes limitaciones de uso en cuanto a variedad de opciones o configuraciones ya que los elementos vienen preestablecidos.

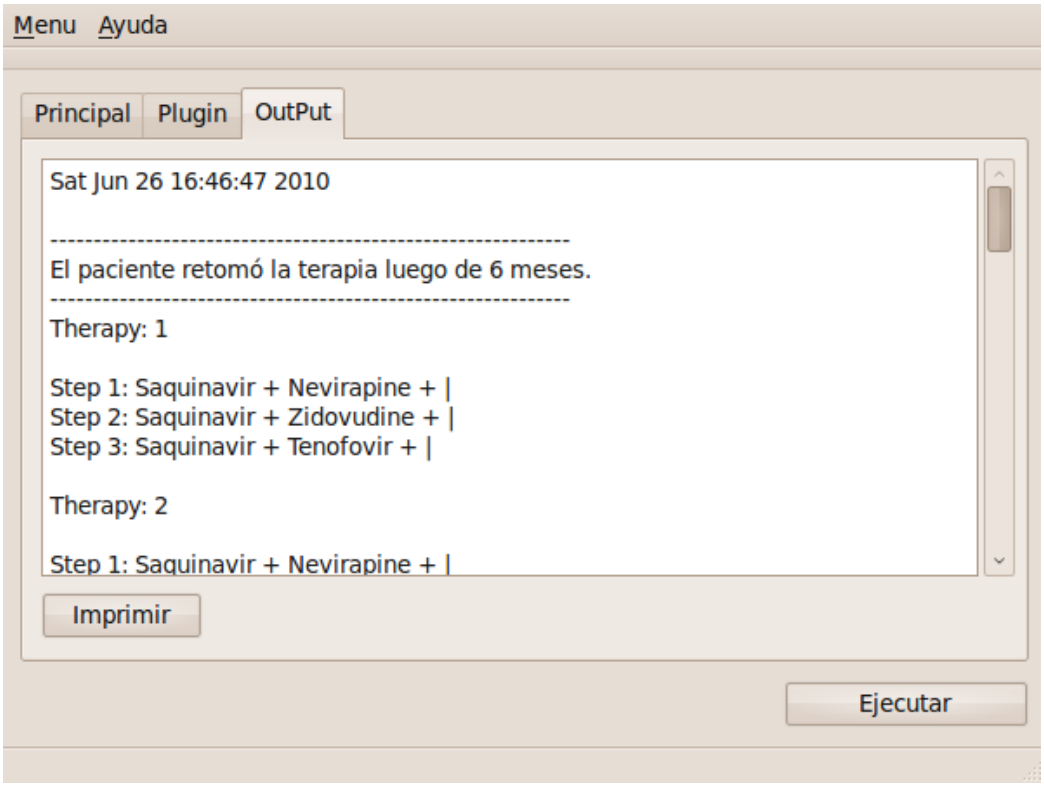

Figura 12.3: Pantalla de output.

Capítulo 13

Conclusiones del Trabajo

13.1. Introducción

Este último capitulo está destinado a las conclusiones, aprendizajes y resultados obtenidos en el desarrollo de este trabajo. Las mismas están divididas en dos aspectos, logros y aportes.

13.2. Logros

Consideramos que este trabajo ha aportado en gran parte al área de la bioinformática como a la comunidad de software libre, pudiendo ser de utilidad en el uso clínico día a día de forma que se puedan explorar las posibles terapias antirretrovirales. Esta precisión está supeditada a las variables que se quieran tener en consideración así como la información previamente obtenida para garantizar un resultado más preciso.

Como resultado obtuvimos un sistema de uso libre y accesible a los profesionales del área de la virología, sirviendo este como una herramienta de formación, investigación o para aplicación clínica.
Ademas, con este trabajo se muestra que es factible construir un sistema bioinfomático con recursos limitados y pudiendo ser utilizado en computadores con capacidad de calculo estándar. También da a lugar a gran cantidad de trabajo futuro relacionado con las posibles extensiones del mismo.
Este proyecto de trabajo final fue presentado previamente en dos oportunidades:

1. XXIX Reunión Científica Anual de la Sociedad Argentina de Virología (S.A.V): participamos como expositores en este evento presentando al plantel de virólogos nuestra idea. Pudimos recibir un importante aporte que nos fue de gran utilidad, considerando que estábamos tratando con los posibles usuarios finales del sistema.

2. Grupo de Investigación de Procesamiento del Lenguaje Natural (Fa.M.A.F): es esa oportunidad expusimos el proyecto ante otros especialistas de las ciencias de la computación que contribuyeron en el mejoramiento de los métodos y algoritmos utilizados.

13.3. Aportes

La tarea del desarrollo de este sistema ha permitido que hayamos podido colaborar y aportar a otros proyectos relacionados.

- BIOpp: como mencionamos con anterioridad, es un conjunto de librerías de biología molecular que proveen funcionalidades para el manejo de estructuras como cadenas de nucleótidos y

aminoácidos. Hemos agregado algunas funcionalidades que fueron requisitorias para nuestro sistema.

- Mili: librería minimalista que provee funciones útiles para C++ y todo código reutilizable en otros proyectos de Fu.De.P.A.N. Colaboramos implementando funciones y estructuras que fueron necesarias a lo largo del proyecto.
- FXP (FuDePAN XML Parser): se corrigieron algunos bugs de esta librería.

13.4. Trabajo Futuro

La versatilidad del framework planteada en un principio permite continuar este trabajo adaptándolo a diferentes usos. Podemos listar una serie de trabajos futuros que hemos propuesto:

- Construir plugins específicos de acuerdo a pautas medicas ya establecidas: los plugins son cruciales para el cálculo. Se debe poder proveer un conjunto de funciones estándar que se puedan reutilizar y poder así confeccionar plugins tan complejos como se requieran.
- Mejorar el uso en sistema Windows: se debe elaborar una mejor interfaz e instaladores para el sistema operativo Windows.
- Permitir utilizar plugins en lenguaje *python*: para elaborar plugins de forma más simple proponemos realizarlos en lenguaje pyhton ya que su utilización resultaría más simple que la utilización de librerías dinámicas del sistema.
- Metodología AOP para plugins: implementar plugins basados en metodologías orientada a aspectos que describen de forma más natural su programación.
- Proveer manejo de probabilidades: como mencionamos a lo largo del trabajo, éste comprende un sistema cualitativo y no cuantitativo, es decir, en ningún momento se contemplan cuales son las mutaciones más probables si no que tomamos la de menor distancia.
- Manejo de tiempos evolutivo del virus: se podrían contemplar los tiempos en los cuales el virus muta.
- Proveer mas políticas de generación de terapias y de combinatorias: además de las provistas en el sistema se podrían implementar algunas más eficientes o que posean otra forma de calculo.
- Visualización de los resultados: sería útil poder proveer una forma más gráfica de listar los resultados, como por ejemplo la visualización del árbol de terapias.
- Paralelización del cálculo: existen algunos casos en los que el cálculo de terapias requiere mucho tiempo, y por ende, consumo de recursos del sistema. Sería optimo poder efectuar estos cálculos en paralelo utilizando un sistema multi-core o cluster.

Anhelamos que el aporte científico a través de este sistema contribuya al avance en el tratamiento del virus del VIH garantizando una mejor calidad de vida a quienes lo padecen, así como un mejor panorama de análisis para los profesionales de la virología, posibilitando que se puedan desarrollar nuevos estudios en base a esta herramienta. También esperamos que este trabajo pueda ser utilizado para la posible elaboración de un ensayo clínico y que Fu.De.P.A.N tramitará y así mismo de posibles trabajos futuros que puedan extender las posibilidades del sistema.

13.5. Repositorio del sistema

Se puede acceder al código fuente de este proyecto y a su documentación visitando la dirección: `http://aso.googlecode.com`

Capítulo 14

Apéndice

14.1. Unified Modeling Language (UML)

Los Diagramas de Estructura Estática de UML se utilizan para representar tanto Modelos Conceptuales como Diagramas de Clases de Diseño, convirtiéndola en una herramienta útil a la hora de diseñar el sistema en cualquiera de sus niveles. Este lenguaje gráfico provee notación estándar para la documentación de elementos tales como: bases de datos, clases, conexiones, etc y pudiendo especificar métodos, funciones e interacciones entre estos [3]. UML provee una extensa cantidad de elementos para desarrollo en la ingeniería del software, pero solo describiremos los utilizados en este trabajo. Para más información visitar `http://www.clikear.com/manuales/uml/diagramasestructuraestatica.aspx`.

14.1.1. Notas

Una nota sirve para añadir cualquier tipo de comentario a un diagrama o a un elemento de un diagrama. Es un modo de indicar información en un formato libre, cuando la notación del diagrama en cuestión no nos permite expresar dicha información de manera adecuada. Una nota se representa como un rectángulo con una esquina doblada con texto en su interior. Puede aparecer en un diagrama tanto sola como unida a un elemento por medio de una línea discontinua. Puede contener restricciones, comentarios, el cuerpo de un procedimiento, etc.

Figura 14.1: Ejemplo de notas.

14.1.2. Dependencias

La relación de dependencia entre dos elementos de un diagrama significa que un cambio en el elemento destino puede implicar un cambio en el elemento origen (por tanto, si cambia el elemento destino habría que revisar el elemento origen). Una dependencia se representa por medio de una línea de trazo discontinuo entre los dos elementos con una flecha en su extremo. El elemento dependiente es el origen de la flecha y el elemento del que depende es el destino (junto a él está la flecha).

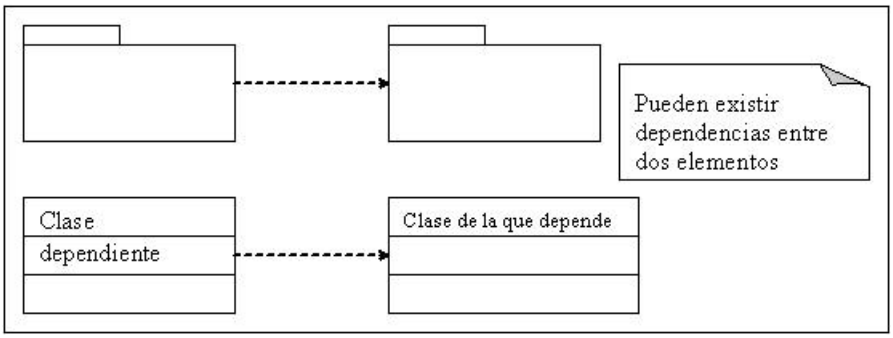

Figura 14.2: Ejemplo de dependencias.

14.1.3. Clases, atributos y métodos

Una clase se representa mediante una caja subdividida en tres partes: En la superior se muestra el nombre de la clase, en la media los atributos y en la inferior los métodos. Una clase puede representarse de forma esquemática, con los atributos y operaciones suprimidos, siendo entonces tan solo un rectángulo con el nombre de la clase. En la Fig. 4.8 se ve cómo una misma clase puede representarse a distinto nivel de detalle según interese, y según la fase en la que se esté.

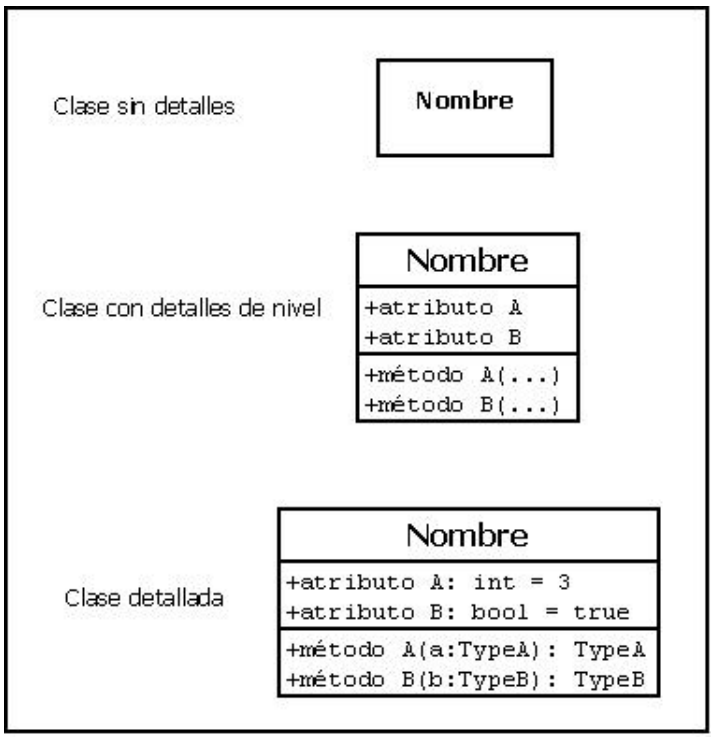

Figura 14.3: Ejemplo de clases.

14.1.4. Relaciones entre clases

Asociaciones

Las asociaciones entre dos clases se representan mediante una línea que las une. La línea puede tener una serie de elementos gráficos que expresan características particulares de la asociación. A

continuación se verán los más importantes.

Nombre de la asociación y dirección El nombre de la asociación es opcional y se muestra como un texto que está próximo a la línea. Se puede añadir un pequeño triángulo negro sólido que indique la dirección en la cual leer el nombre de la asociación. En el ejemplo de la Fig 4.9 se puede leer la asociación como "Director manda sobre Empleado".

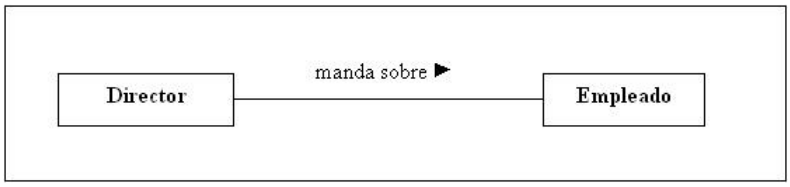

Figura 14.4: Ejemplo de la asociación y dirección.

Los nombres de las asociaciones normalmente se incluyen en los modelos para aumentar la legibilidad. Sin embargo, en ocasiones pueden hacer demasiado abundante la información que se presenta, con el consiguiente riesgo de saturación. En ese caso se puede suprimir el nombre de las asociaciones consideradas como suficientemente conocidas. En las asociaciones de tipo agregación y de herencia no se suele poner el nombre.

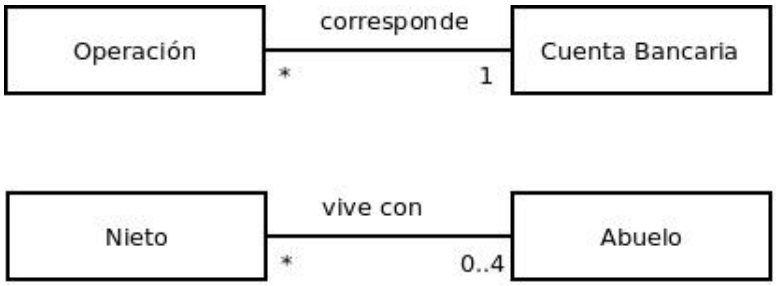

Figura 14.5: Ejemplo de Multiplicidad.

Multiplicidad La multiplicidad es una restricción que se pone a una asociación, que limita el número de instancias de una clase que pueden tener esa asociación con una instancia de la otra clase. Puede expresarse de las siguientes formas:

- Con un número fijo: 1.
- Con un intervalo de valores: 2..5.
- Con un rango en el cual uno de los extremos es un asterisco. Significa que es un intervalo abierto. Por ejemplo, 2..* significa 2 o más.
- Con una combinación de elementos como los anteriores separados por comas: 1, 3..5, 7, 15..*.
- Con un asterisco: * . En este caso indica que puede tomar cualquier valor (cero o más).

Roles Para indicar el papel que juega una clase en una asociación se puede especificar un nombre de rol.

Se representa en el extremo de la asociación junto a la clase que desempeña dicho rol.

Agregación El símbolo de agregación es un diamante colocado en el extremo en el que está la clase que representa el "todo".

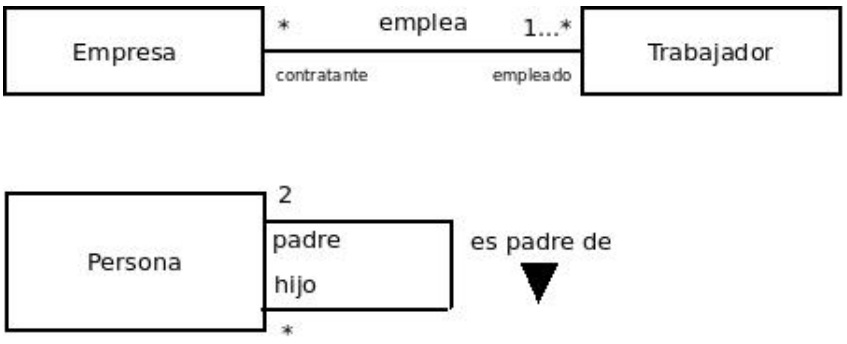

Figura 14.6: Ejemplo Roles.

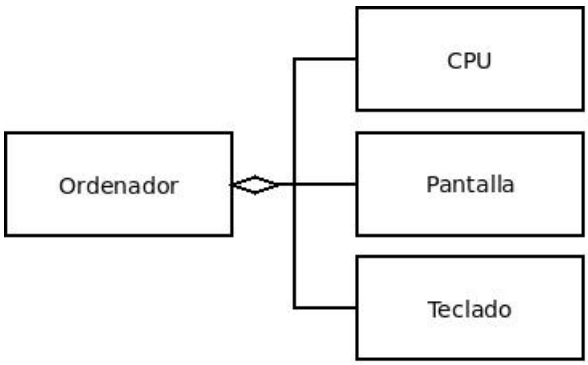

Figura 14.7: Ejemplo Agregación.

14.1.5. Herencia y polimorfismo

La relación de herencia se representa mediante un triángulo en el extremo de la relación que corresponde a la clase base o clase más general.

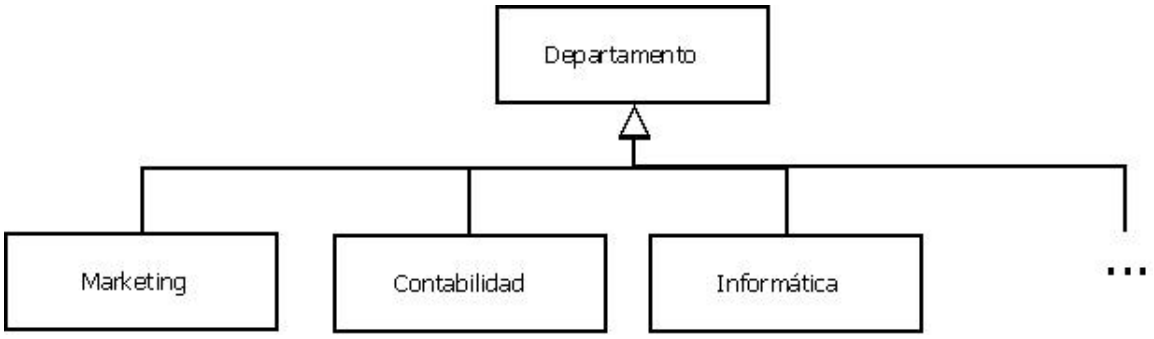

Figura 14.8: Ejemplo Herencia.

Si se tiene una relación de herencia con varias clases derivadas, pero en un diagrama concreto no se quieren poner todas, esto se representa mediante puntos suspensivos. En el ejemplo de la Fig. 4.13, sólo aparecen en el diagrama 3 tipos de departamentos, pero con los puntos suspensivos se indica que en el modelo completo (el formado por todos los diagramas) la clase "Departamento" tiene subclases adicionales, como podrían ser "Recursos Humanos" y "Producción".

14.2. Código Genético

Tabla del código genético.

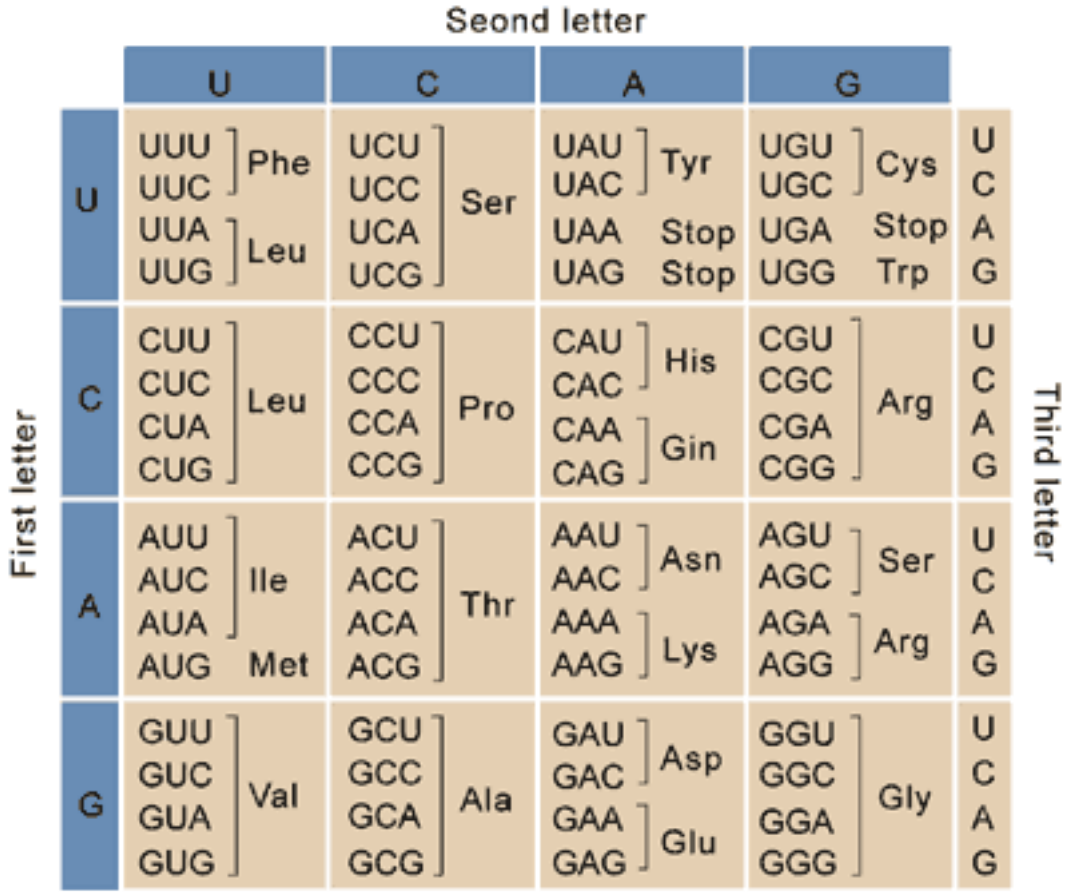

Figura 14.9: Código genético.

14.3. Formato FASTA

Es un formato de texto estándar utilizado en la bioinformática y que representa secuencias de aminoácidos con su identificación de una única letra. Ademas se puede colocar una cabecera con información adicional o colocar varias secuencias en un mismo archivo. Mas información sobre este formato en `http://www.ncbi.nlm.nih.gov/blast/fasta.shtml`.

14.4. Pseudo Código

A lo largo del trabajo se definieron algoritmos utilizando un pseudo código similar a C. Este no pertenece a ningún estándar pero nos pareció conveniente utilizarlo.

14.4.1. Gramática

Utilizamos notación formal para denotar la gramática.

```
<sentence> :: = <sentence>..<sentence>
      | while <sentence> do { <sentence> }
      | for all <element> in <set of elements> do { <sentence> }
      | if <sentence> { <sentence> } else { <sentence> }
      | <type> <element>
      | <element> = <element>
      | type <namefunction>(<element>,..,<element>) {<sentence>}
      | return <element>
      | case (<element>) {<element>: <sentence>...<element>: <sentence>}

<type> :: = C++ types

<element> :: = C++ instance
```

Es decir este pseudo código trata de imitar al de C++ eliminando algunos elementos que no son necesarios para el caso, como iteradores, estructuras, clases, puntos y coma, entre otras cosas.

14.5. Métricas obtenidas

A modo de resumen enumeramos muchas de las métricas obtenidas producto de este trabajo:

14.5.1. Código

Métricas relacionadas con el código: estas fueron obtenidas a partir de las siguientes herramientas:

1. Gcov: herramienta para calcular el cubrimiento de código. `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`
2. LCov: herramienta complementaria a Gcov que elabora un archivo *html* más legible. `http://linux.die.net/man/1/lcov`
3. CCCC: software para calcular la cantidad de lineas de código y otras métricas. `http://cccc.sourceforge.net/`
4. Doxygen: herramienta de documentación automática de código. `http://www.stack.nl/~dimitri/doxygen/`
5. Astyle: pequeña herramienta que permite definir el estilo del código de forma automática.

Las métricas obtenidas:

- Líneas de Código (LOC): 2800 (aprox)
- Líneas de Comentario (COM): 1500 (aprox)
- Proporción (LOM/COM): 2
- Complejidad ciclomática: 19 (Moderate Risk)
- Cobertura de Código por líneas: 91,9 %
- Cobertura de Código por funciones: 81,1 %

Es importante notar que el diseño ha resultado efectivo ya que el problema, que a pesar de ser complejo, pudo ser resuelto en escasas lineas de código.

14.5.2. Otras Métricas

Algunos números interesantes:

- Cantidad de textos leídos: 30 aproximadamente.
- Cantidad de horas de trabajo: 660 hs aproximadamente (durante 10 meses).
- Cantidad de mails enviados y recibidos: 500 aproximadamente.
- Cantidad de años figurativos: 10 años mas viejos.

14.6. Otros Documentos

Incluimos a este trabajo los documentos de Especificación de Requerimientos y el Documento de Diseño.

Bibliografía

[1] Pankaj Jalote, *An integrated approach to software engineering*, (New York 2003), Springer-Verlag New York, Inc.

[2] Barré-Sinoussi F, Chermann JC, Rey F, Nugeyre MT, Chamaret S, Gruest J, Dauguet C, Axler-Blin C, Vézinet-Brun F, Rouzioux C, Rozenbaum W, Montagnier L (1983). *Isolation of a T-lymphotropic retrovirus from a patient at risk for acquired immune deficiency syndrome (AIDS)*. Science 220 (4599): 868–871 (1983).

[3] Jason Gorman, *Object-Oriented Analysis & Design: Chapter 3*,(London 2005).

[4] Erich Gamma and John Vlissides and Ralph Johnson and Richard Helm, *Design Patterns CD: Elements of Reusable Object-Oriented Software, (CD-ROM)* (Boston, MA, USA 1998), Addison-Wesley Longman Publishing Co., Inc.

[5] In Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications *Object-oriented design: a responsibility-driven approach.* (New Orleans, Louisiana, United States, October 02 - 06, 1989). OOPSLA '89. ACM Press, New York, NY, 71-75.

[6] Robert C. Martin ("Uncle Bob") *Principles Of OOD* , Last verified 2009-01-14. (Note the "first five principles", though the acronym is not used in this article.) Dates back to at least 2003.

[7] William J. Brown, Raphael C. Malveau, Hays W. "Skip"McCormick (III), Thomas J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*

[8] Grady Booch, *Object Oriented Analysis and Design*, (1994) The Benjamin / Cummings Publishing Company, Inc.

[9] Bjarne Stroustrup, *The C++ programming language 2nd ed*, (1991), Addison-Wesley.

[10] Jasmin Blanchette and Mark Summerfield, *C++ GUI Programming with Qt 4* (2006) Prentice Hall.

[11] JN Leonard and DV Schaffer, *Antiviral RNAi therapy: emerging approaches for hitting a moving target*, Department of Chemical Engineering and the Helen Wills Neuroscience Institute, University of California, Berkeley, CA, USA (2006).

[12] Niko Beerenwinkel, Thomas Lengauer, Martin Daumer, Rolf Kaiser, Hauke Walter, Klaus Korn, Daniel Hoffmann and Joachim Selbig, Methods for optimizing antiviral combination therapies, (2003), Vol. 19 Suppl. 1 2003, pages i16–i25.

[13] Victoria A. Johnson, MD, Françoise Brun-Vézinet, MD, PhD, Bonaventura Clotet, MD, PhD, Huldrych F. Günthard, MD, Daniel R. Kuritzkes, MD, Deenan Pillay, MD, PhD, Jonathan M. Schapiro, MD, and Douglas D. Richman, MD, *Update of the Drug Resistance Mutations in HIV-1* (2008).

[14] Barbara Liskov, *Keynote address - data abstraction and hierarchy*, (1987 Cambridge) MIT Laboratory for Computer Science, Cambridge, Ma.

[15] Robert Cecil Martin,*Agile Software Development, Principles, Patterns, and Practices.*, (2002) Prentice Hall.

[16] K. Lieberherr,I. Holland, A. Riel, (1988 Boston), Object-Oriented Programming:An Objective Sense of Style, Northeastern University, College of Computer Science.

[17] Willard, H.W., and Ginsburg, G.S., (eds),*Genomic and Personalized Medicine*,(2009), Academic Press.

[18] Gutson, Daniel(FUDEPAN); Dario, Dilerna(CNRS); Sanchez, Eduardo(FUDEPAN); Malbran, Francisco(FUDEPAN); Gomez Carrillo, ManuelL(CNRS); Rabinovich, Roberto D(CNRS), *Heterogeneidad en la distancia genética a resistencias en secuencias del HIV en pacientes vírgenes de tratamiento.*, Fu.De.PAN (2009)

[19] Information, Inspiration and Advocacy for People Living With HIV/AIDS, *Strategies for HIV Therapy* (2008), `http://www.thebody.com/content/art5735.html`

[20] Dybul M, Fauci AS, Bartlett JG, Kaplan JE, Pau AK; Panel on Clinical Practices for Treatment of HIV, *Guidelines for using antiretroviral agents among HIV-infected adults and adolescents*, Ann Intern Med. 2002

[21] Li, Wen-Hsiung (2006). *Molecular Evolution, Sinauer Associates Inc.*

[22] Federico Abascal Sebastián de Erice, *Análisis de genomas. Métodos para la predicción y anotación de la función de las proteínas*, Centro Nacional de Biotecnología (Madrid 2003).

[23] United Nations Environment Programme - UNEP, et al. (1992). *The diversity of life. Global Biodiversity Strategy: Guidelines for action to save, study and use Earth's biotic wealth sustainably and equitably.* World Resources Institute (por la edición en web).

[24] Wirta, V. (2006). *Mining the transcriptome – methods and applications.* Royal Institute of Technology, School of Biotechnology (Estocolmo).

[25] King, M. W. (2008).*Control of Gene Expression.* The Medical Biochemistry Page.

[26] Poetz, O., et al. (2005). *Protein microarrays: catching the proteome.* Mechanisms of Ageing and Development 126 (1). Págs. 161-170.

[27] Zhao, X., et al. (2004). *An Integrated View of Copy Number and Allelic Alterations in the Cancer Genome Using Single Nucleotide Polymorphism Arrays.* Cancer Research 64. Págs. 3060-3071.

[28] Marshall Nirembergs *The genetic code*, Nobel Lecture, (December 12, 1968)

[29] Kitano, H. (2002). *Systems Biology: A Brief Overview. Science 295 (5560).* ISSN 0036-8075, Págs. 1662-1664.