



UNIVERSIDAD VERACRUZANA

FACULTAD DE FÍSICA E INTELIGENCIA ARTIFICIAL
MAESTRÍA EN INTELIGENCIA ARTIFICIAL

**Aplicaciones de la Inteligencia Artificial
al análisis de Biosecuencias**

TESIS

Que para obtener el grado de:

Maestro en Inteligencia Artificial

Presenta:

Julio César Sandria Reynoso

Director de tesis:

Dr. Miguel Angel Jiménez Montaña

Resumen

En esta tesis se presentan métodos para analizar secuencias biológicas (biosecuencias) así como otros tipos de secuencias. Las biosecuencias son las secuencias de bases nucleótidas y aminoácidos formadoras del ADN y las proteínas respectivamente. La enorme cantidad de secuencias de ADN y proteínas en bases de datos públicas en todo el mundo hace propicio usar diversas técnicas de inteligencia artificial para su adecuada manipulación y análisis, como son la minería de datos, el aprendizaje de máquina, el reconocimiento de patrones y el agrupamiento. Aunque existen ya una gran cantidad de programas para analizar secuencias basados en diferentes algoritmos, la principal contribución de este trabajo es el desarrollo de una herramienta de software para sistemas Windows y Linux: WinGramm 2 y LinGramm 2, basada en el algoritmo *Grammar*, que calcula la complejidad gramatical de una secuencia. Grammar es un algoritmo de compactación que encuentra redundancia en secuencias, mediante el descubrimiento de patrones, a través de un proceso de inferencia de gramáticas libres del contexto. Se ejemplifica el uso de esta herramienta aplicándola al análisis de secuencias de ADN, proteínas y cartas en cadena.

Agradecimientos

A mi esposa, por su amor y comprensión.

A mis hijos Carolina y Rafael, que le dieron un nuevo sentido a mi vida.

A mi madre, por haberme guiado correctamente hasta que tomé mi propio camino.

Al Dr. Manuel Martínez Morales, porque sin su ayuda no hubiera terminado la maestría.

Al Dr. Jiménez Montaña, Luis Nava, Rusalky y Antero, por las recomendaciones, comentarios, correcciones, sugerencias y mejoras para el programa WinGramm 2.

A mis profesores de la Maestría en Inteligencia Artificial, que me hicieron conocer y trabajar en la Ciencia de la Computación.



Contenido

Lista de Figuras.....	vii
Lista de Tablas.....	viii
1 Introducción.....	1
1.1 Motivación.....	1
1.2 Objetivo de este trabajo.....	1
1.3 Contribuciones.....	2
1.4 Estructura de la tesis.....	2
1.5 Versiones del software.....	2
2 Antecedentes.....	3
2.1 Lenguajes, autómatas y gramáticas.....	3
2.1.1 Conceptos preliminares.....	3
2.1.2 Autómatas finitos.....	4
2.1.3 Clasificación de lenguajes.....	5
2.1.4 Gramáticas.....	5
2.1.5 Gramáticas libres del contexto.....	7
2.2 Inferencia Gramatical.....	7
2.3 Medidas de información.....	9
2.3.1 Entropía.....	9
2.3.2 Complejidad gramatical.....	11
2.3.3 Redundancia algorítmica.....	13
2.3.4 Secuencias subrogadas.....	13
2.3.5 Distancia algorítmica.....	14
2.4 Biosecuencias: Secuencias biológicas.....	14
2.4.1 ADN, Proteínas y Genes.....	14
2.4.2 Bases de datos.....	17
2.4.3 GenBank.....	18
2.4.4 Árboles filogenéticos.....	21
2.4.5 Contenido informacional.....	23
2.4.6 Análisis de secuencias de ADN.....	24
3 Desarrollo de software científico y de IA.....	30
3.1 Ingeniería del Software.....	30
3.1.1 El ciclo de vida clásico.....	30
3.1.2 Construcción de prototipos.....	31
3.1.3 El modelo en espiral.....	33

3.1.4	El modelo de ensamblaje de componentes	34
3.2	Software científico y de inteligencia artificial	35
3.2.1	Exactitud	35
3.2.2	Validación y verificación	36
3.3	Portabilidad	36
3.4	WinGramm 2	37
3.4.1	Código Orientado a Objetos	38
3.4.2	Mejoras	41
3.4.3	Actualizaciones en Internet	42
3.5	LinGramm 2	42
4	Aplicaciones	44
4.1	Análisis de secuencias de cartas en cadena	44
4.1.1	Problema	44
4.1.2	Material y métodos	45
4.1.3	Resultados	46
4.2	Análisis de secuencias de proteínas	47
4.2.1	Problema	47
4.3	Análisis de secuencias de ADN	48
4.3.1	Aplicación 1. Descubriendo secuencias simples de ADN	48
4.3.2	Aplicación 2. Descubriendo similitud	51
5	Conclusiones	53
	Referencias bibliográficas	55
	Apéndice A. Descripción de WinGramm 2.1	59
A.1	Características	59
A.2	Ventanas principal y Project	60
A.3	Cálculo de la complejidad gramatical	61
A.4	Ejemplo de gramática generada por WinGramm 2	66
A.5	Cálculo de la distancia algorítmica, distancia relativa y factor de similitud – matrices de distancias	68
A.5	Construcción de árboles filogenéticos con UPGMA	70

Lista de Figuras

Figura 2-1. Modelo general del proceso de inferencia	8
Figura 2-2. Algoritmo Grammar paso a paso	12
Figura 2-3. ADN	15
Figura 2-4. Replicación del ADN	17
Figura 2-5. Base de datos de bases de datos	18
Figura 2-6. Página principal de GenBank.....	19
Figura 2-7. Resultado de la búsqueda en GenBank de la secuencia completa de ADN mitocondrial humano	19
Figura 2-8. GenBank Sumario de la secuencia completa de ADN mitocondrial humano ...	20
Figura 2-9. Parte de la secuencia de ADN mitocondrial humano en formato FASTA	20
Figura 2-10. Árbol filogenético para cinco especies de primates.....	23
Figura 2-11. Gráfica de frecuencia de K , de 500 secuencias aleatorias y 500 del gen HUMTPA.....	25
Figura 2-12. Árbol filogenético de 10 secuencias aleatorias (ACGT *) y 10 secuencias del gen HUMTPA (w^*)	27
Figura 2-13. Árbol filogenético de genomas mitocondriales completos de 20 mamíferos ..	29
Figura 2-14. Tres posibles grupos entre primates, ferungulados y roedores	29
Figura 3-1. Ciclo de vida clásico.	31
Figura 3-2. Creación de prototipos.	32
Figura 3-3. El modelo en espiral.....	33
Figura 3-4. El modelo de ensamblaje de componentes.	34
Figura 3-5. Verificación y validación del software científico	36
Figura 3-6. WinGramm 2.1 en un ambiente Windows XP.....	38
Figura 3-7. Clase TSequence	39
Figura 3-8. Clase Tgrammar	39
Figura 3-9. Código del método GrammaticalComplexity1Click.....	40
Figura 3-10. LinGramm 2.1 en un ambiente Linux Red Hat 7.3.....	43
Figura 4-1. Ejemplo de carta cadena	44
Figura 4-2. Árbol filogenético para las cartas en cadena.....	46
Figura 4-3. Diferencias entre las cartas más parecidas	46
Figura 4-4. Árbol generado a partir de la matriz de Miyazawa-Jernigan	47
Figura 4-5. Alfabetos reducidos para el árbol Miyazawa-Jernigan	47
Figura 4-6. Árboles filogenéticos de las secuencias de mioglobina con los alfabetos reducidos de Miyazawa-Jernigan.....	48
Figura 4-7. Parte del sumario del gen HUMTPA obtenido en GenBank	49
Figura 4-8. Parte del gen HUMTPA en formato FASTA.....	49
Figura 4-9. Fraccionando secuencia del gen HUMTPA.....	49

Lista de Tablas

Tabla 2-1. Jerarquía de lenguajes formales de Chomsky.	5
Tabla 2-2. (a) Distancia genética entre todos los pares de cuatro grupos y (b) entre los grupos después de que los grupos 1 y 2 han sido agrupados.	22
Tabla 2-3. Matriz de distancias para cinco especies de primates	22
Tabla 2-4. 10 secuencias aleatorias ($l=50$) de menor y mayor complejidad gramatical (K)	24
Tabla 2-5. 10 secuencias (fracciones, $l=50$) del gen HUMTPA de menor y mayor complejidad gramatical (K)	25
Tabla 2-6. Matriz de distancias algorítmicas de 10 secuencias aleatorias (ACGT *) y 10 secuencias del gen HUMTPA (w^*)	26
Tabla 2-7. Matriz de distancias de genomas mitocondriales completos de 20 mamíferos...	28
Tabla 3-1. Desempeño de WinGramm 1 vs. WinGramm 2	42
Tabla 4-1. Matriz de distancia algorítmica de las cartas en cadena.....	45
Tabla 4-2. Subsecuencias ($w=128$, $o=64$) del gen HUMTPA con mayor Redundancia Algorítmica R , con sus correspondientes repeticiones y frecuencias.	50
Tabla 4-3. Comparación de resultados de WinGramm 2 con Milosavljevic (1999)	51
Tabla 4-4. Distancia algorítmica de 10 fragmentos del segmento 22,001-36,000 del gen HUMTPA.....	52
Tabla 4-5. Complejidad grammatical de 10 secuencias del segmento 22001-26000, cada secuencia está concatenada con el corpus Alu.....	52

1 Introducción

En este capítulo se explica la motivación de este trabajo, el objetivo planteado, qué contribución aporta al área de las ciencias de la computación, la biología molecular y bioinformática, así como la forma en que se estructuró la tesis.

1.1 Motivación

El interés por este trabajo inicia de la participación del autor como becario en el proyecto de investigación *Modelaje y caracterización de patrones en las ciencias biológicas y del comportamiento por métodos computacionales* financiado por el Consejo Nacional de Ciencia y Tecnología (CONACyT), Ref. 25977.

En dicha colaboración se usó el programa Grammar para descubrir patrones en alumnos que presentaron examen de admisión para ingresar a la Universidad Veracruzana en 1998 (Zamora y Sandria, 2000).

El programa Grammar es una implementación para DOS (Disk Operating System) del algoritmo *Grammar* para el cálculo de la *complejidad gramatical* propuesto por Ebeling y Jiménez Montaña (1980), que es un algoritmo de compactación y reconocimiento de patrones mediante inferencia gramatical (gramáticas libres del contexto). Una segunda implementación se hizo para plataforma Windows: WinGramm 1, que es un paquete de herramientas que incluye la funcionalidad del programa Grammar y muchas funciones más.

Debido a que el uso de WinGramm 1 con muchas secuencias es tardado, poco práctico y limitado, a que su mantenimiento y actualización es muy difícil por la forma en que está programado (y a la falta de documentación técnica), se presentó necesario desarrollar la nueva versión del programa: WinGramm 2, que mejorara su desempeño, fuera más práctico, fácil y rápido de usar, más automatizado, con más funcionalidades, de fácil mantenimiento y multiplataforma (para Windows y Linux).

1.2 Objetivo de este trabajo

El objetivo de esta tesis fue desarrollar una herramienta de software basada en la complejidad gramatical, que facilitara el análisis de secuencias, principalmente biológicas a investigadores de las áreas de biología molecular, bioinformática y áreas relacionadas.

La aplicación de la inteligencia artificial se da implícitamente en el algoritmo para el cálculo de la complejidad gramatical, Grammar, ya que éste se basa en la teoría de autómatas para inferir gramáticas libres del contexto y de allí descubrir patrones en las secuencias analizadas.

1.3 Contribuciones

La principal contribución que aporta este trabajo es una herramienta de software que permite analizar secuencias biológicas y de otros tipos, de forma fácil y bastante automatizada, en computadoras personales con sistemas operativos Windows o Linux.

1.4 Estructura de la tesis

El documento está estructurado en cinco capítulos del siguiente modo:

1. **Introducción.** En este capítulo se explica la motivación de este trabajo, el objetivo planteado, qué contribución aporta al área de las ciencias de la computación, la biología molecular y bioinformática, así como la forma en que se estructuró la tesis.
2. **Antecedentes.** En este capítulo se hace una revisión sobre lenguajes, autómatas, gramáticas y el proceso de inferencia gramatical, se muestran las diferentes medidas de información que calcula WinGramm 2, se muestran los conceptos fundamentales sobre las secuencias biológicas a analizar y se ejemplifica el uso de la complejidad gramatical en el análisis de biosecuencias.
3. **Desarrollo de software científico** En este capítulo se describen conceptos de ingeniería de software, la metodología de desarrollo de software seguida para la realización de WinGramm 2, se hace énfasis en el uso adecuado de técnicas de programación, tanto para software en general como científico y de inteligencia artificial (IA), la problemática de la portabilidad del software y se describen de manera general las características de WinGramm 2.
4. **Aplicaciones.** En este capítulo se muestran ejemplos de análisis de secuencias, iniciando con un conjunto de cartas en cadena (secuencias de caracteres que forman palabras), secuencias de proteínas y de ADN.
5. **Conclusiones.**

1.5 Versiones del software

WinGramm 2 es la versión para sistemas Windows y LinGramm 2 es la versión para sistemas Linux. Debido a que el desarrollo del software se hizo principalmente en sistemas Windows, se estará usando de manera general el nombre WinGramm 2. Cualquier cosa en particular que se mencione sobre WinGramm 2 se aplica también a LinGramm 2, a menos que se indique explícitamente lo contrario.

Con este trabajo se libera la versión 2.1.1 de WinGramm.

2 Antecedentes

En este capítulo se hace una revisión sobre lenguajes, autómatas, gramáticas y el proceso de inferencia gramatical, se muestran las diferentes medidas de información que calcula WinGramm 2, se muestran los conceptos fundamentales sobre las secuencias biológicas a analizar y se ejemplifica el uso de la complejidad gramatical en el análisis de biosecuencias.

2.1 Lenguajes, autómatas y gramáticas

2.1.1 Conceptos preliminares

Un **símbolo** es un objeto o una entidad abstracta, como una letra o un dígito, base de nuestra principal forma de comunicación: el *lenguaje*.

Un **alfabeto** es un conjunto finito de símbolos. Por ejemplo, el alfabeto romano: $\{a, b, \dots, z\}$, el alfabeto binario: $\{0, 1\}$. Cualquier objeto puede ser un símbolo en un alfabeto, pero por simplicidad, se suele usar solamente letras, números u otros caracteres comunes.

Una **cadena** es una secuencia finita de símbolos tomados de un alfabeto. Por ejemplo *aeiou* es una cadena sobre el alfabeto $\{a, b, \dots, z\}$. Una **cadena vacía** es una cadena que no contiene símbolos, y se denota como *e*. Los datos que una computadora manipula están modelados matemáticamente por *cadena de símbolos*.

La **longitud** de una cadena está dada por el número de símbolos que ésta contiene. Por ejemplo, la longitud de la cadena *aabb* es 4. Se denota la longitud de una cadena *w* como $|w|$. Así, $|w|=1$, $|1001|=4$ y $|e|=0$.

La **concatenación** de las cadenas *x* y *y*, escrita $x \circ y$ o simplemente *xy*, es la cadena *x* seguida de la cadena *y*. Una cadena *v* es una **subcadena** de *w* si y sólo si existen las cadenas *x* y *y* tales que $w=xvy$. Si $w=xv$ para alguna *x*, entonces *v* es un **sufijo** de *w*. Si $w=vy$ para alguna *y*, entonces *v* es un **prefijo** de *w*. Una cadena puede tener varias ocurrencias de una misma subcadena; por ejemplo, *ababab* tiene tres ocurrencias de *ab* y dos de *abab*. Para cada cadena *w* y cada número natural *i*, la cadena w^i se define como

$$\begin{aligned} w^0 &= e, \text{ la cadena vacía;} \\ w^{i+1} &= w^i w, \text{ para cada } i \geq 0; \end{aligned}$$

de esta manera $w^1 = w$ y $(pa)^2 = papa$.

El **inverso** de una cadena *w*, denotado por w^R , es la cadena “deletreada hacia atrás”, por ejemplo: $\text{inverso}^R = \text{osrevni}$.

Un **lenguaje** es un conjunto de cadenas (incluyendo la cadena vacía) sobre un alfabeto fijo Σ , y se denota como Σ^* . Por ejemplo, si el alfabeto $\Sigma = \{a\}$, entonces, el lenguaje $\Sigma^* = \{e, a, aa, aaa, \dots\}$. Si $\Sigma = \{0,1\}$, entonces $\Sigma^* = \{e, 0, 1, 00, 01, 10, 11, 000, \dots\}$. Así, Σ^* , \emptyset (conjunto vacío) y Σ son lenguajes. Aunque un lenguaje es simplemente una clase especial de conjunto, se puede especificar un lenguaje finito listando todas sus cadenas. Por ejemplo, $\{abc, lmn, f, z\}$ es un lenguaje sobre $\{a, b, \dots, z\}$. Sin embargo, la mayoría de los lenguajes de interés son infinitos, de manera que listar todas sus cadenas es imposible. Por ejemplo: $\{0, 01, 011, 0111, \dots\}$, $\{w \in \{0,1\}^*: w \text{ tiene un igual número de 0's y 1's}\}$, y $\{w \in \Sigma^*: w = w^R\}$. Por lo que se hace necesario especificar lenguajes infinitos con el esquema

$$L = \{w \in \Sigma^*: w \text{ tiene la propiedad } P\}$$

Un punto central en la teoría de computación es la representación de lenguajes mediante especificaciones finitas. Cualquier lenguaje finito se puede representar por enumeración de todas sus cadenas, pero la situación cambia cuando se deben considerar lenguajes infinitos.

En la representación de lenguajes infinitos deben considerarse *expresiones*. Una **expresión** es una cadena de símbolos que describe cómo se puede construir un lenguaje usando las operaciones descritas anteriormente.

Sea $L = \{w \in \{0,1\}^*: w \text{ tiene dos o tres ocurrencias de 1, la primera y segunda no son consecutivas}\}$. Este lenguaje puede describirse usando solamente conjuntos simples y los símbolos \cup , $*$ (y concatenación) como

$$L = \{0\}^* \{1\} \{0\}^* \{0\} \{1\} \{0\}^* ((\{1\} \{0\}^*) \cup \emptyset^*)$$

Los únicos símbolos usados en esta representación son las llaves $\{ \}$ y $\}$, los paréntesis $(\)$, \emptyset , 0, 1, $*$ y \cup . De hecho, se pueden omitir las llaves y escribir simplemente

$$L = 0^* 1 0^* 0 1 0^* (1 0^* \cup \emptyset^*)$$

Una expresión como ésta, que describe un lenguaje exclusivamente por medio de símbolos simples y \emptyset combinados, con la ayuda de paréntesis y usando los símbolos \cup y $*$ es llamada **expresión regular**.

2.1.2 Autómatas finitos

Un **autómata finito** es un modelo matemático de un sistema, con entradas discretas y salidas discretas. El sistema puede estar en cualquiera de un número finito de configuraciones o “estados” (Hopcroft y Hullman, 1979). La teoría de autómatas finitos es fundamental en el diseño de algoritmos y programas de computadoras (Lewis y Papadimitriou, 1981). Por ejemplo, la fase de análisis léxico de un compilador frecuentemente está basado en la simulación de un autómata finito. El problema de encontrar una ocurrencia de una cadena dentro de otra se puede resolver eficientemente por métodos de la teoría de autómatas finitos. Un autómata finito es un reconocedor de un lenguaje. La definición matemática formal del un autómata finito es la siguiente:

Definición. Un *autómata finito* es un quintuple $M = (K, \Sigma, \delta, s, F)$, donde

K es un conjunto finito de *estados*,
 Σ es un *alfabeto de entrada* finito,
 $s \in K$ es el *estado inicial*,
 $F \subseteq K$ es el conjunto de *estados finales*, y
 δ es la *función de transición* que transforma $K \times \Sigma$ en K .

2.1.3 Clasificación de lenguajes

En 1956, Noam Chomsky estableció su teoría sobre lenguajes formales que se conoce como Jerarquía de Chomsky, la cual comprende 4 tipos de lenguajes con sus correspondientes gramáticas y máquinas asociadas (Tabla 2-1).

Lenguaje	Gramática	Máquina	Ejemplo
Lenguaje recursivamente enumerable (tipo-0)	Gramática no restringida	Máquina de Turing	Cualquier función computable
Lenguaje sensible al contexto (tipo-1)	Gramática sensible al contexto	Autómata lineal acotado	$a^n b^n c^n$
Lenguaje libre del contexto (tipo-2)	Gramática libre del contexto	Autómata de pila no determinístico	$a^n b^n$
Lenguaje regular (tipo-3)	Gramática regular	Aceptor de estados finitos determinístico o no determinístico	A

Tabla 2-1. Jerarquía de lenguajes formales de Chomsky.

Estos lenguajes forman una jerarquía estricta, donde los lenguajes recursivamente enumerables son los más generales, así:

$$\begin{array}{ccccccc}
 \text{lenguajes} & & \text{lenguajes libres} & & \text{lenguajes sensibles} & & \text{lenguajes recursivamente} \\
 \text{regulares} & \subseteq & \text{de contexto} & \subseteq & \text{al contexto} & \subseteq & \text{enumerables}
 \end{array}$$

2.1.4 Gramáticas

Una **gramática** es un formalismo que *genera* un lenguaje, a su vez, dicho lenguaje puede ser *reconocido* por un autómata. Las gramáticas proporcionan el conjunto de reglas que permiten formar sentencias correctas.

Para el uso de gramáticas es conveniente establecer algunas reglas:

- Las letras mayúsculas A, B, C, D, E y S representan variables.
- Las letras minúsculas a, b, c, d, e y los dígitos representan terminales.
- Las letras griegas minúsculas α, β y γ denotan cadenas de variables y terminales.

Una **gramática regular** o *gramática de estado finito* genera un *lenguaje regular*, que está hasta abajo de la jerarquía de Chomsky. Los lenguajes regulares tienen muchos usos en las ciencias de la computación, pero no son lo suficientemente poderosos para representar la sintaxis de la mayoría de los lenguajes de programación.

Definición. Una gramática regular G es un cuarteto $\langle VN, VT, P, S \rangle$, donde

VN es un alfabeto finito de símbolos no terminales (o variables);
 VT es un alfabeto finito de símbolos terminales, tal que VT y VN son disjuntos;
 P es un conjunto finito de producciones o reglas de la gramática G de la forma $\alpha \rightarrow \beta$, y cada producción satisface:

$|\alpha| = 1$ donde α pertenece a VN .

$|\beta|$ es de la forma aB o a , con B en VN y a en VT .

S es un elemento no terminal distinguible de VN , llamado símbolo inicial.

Ejemplo. La siguiente es una gramática regular:

$S \rightarrow bA$

$S \rightarrow aB$

$A \rightarrow abaS$

$B \rightarrow babS$

$S \rightarrow e$

que también puede escribirse usando el símbolo $|$ para agrupar producciones:

$S \rightarrow bA \mid aB \mid e$

$A \rightarrow abaS$

$B \rightarrow babS$

donde

$VN = \{S, A, B\};$

$VT = \{a, b\};$

$P = \{S \rightarrow bA, S \rightarrow aB, A \rightarrow abaS, B \rightarrow babS, S \rightarrow e\}$ donde S, A y $B \in VN$;

S es el símbolo inicial.

El operador infijo \rightarrow de las reglas de la gramática especifica que donde hay una ocurrencia de un símbolos no terminal, éste se puede reemplazar por el lado derecho de su regla correspondiente.

Una **derivación**, denotada por el operador infijo \Rightarrow , implica la reescritura de una cadena usando las reglas de la gramática. Una posible derivación de esta gramática, donde se muestran subrayados los símbolos a reemplazar por el lado derecho de la regla correspondiente es

$$S \Rightarrow b\underline{A} \Rightarrow baba\underline{S} \Rightarrow babaab\underline{B} \Rightarrow babaabab\underline{S} \Rightarrow babaabab$$

que especifica el lenguaje

$$L(G) = \{baba \cup abab\}^*$$

Cabe hacer notar que una gramática regular siempre tiene en el lado derecho de cada regla de producción al menos un símbolo terminal y cuando tiene un símbolo no terminal en el lado derecho está al final de la cadena.

Un **lenguaje regular** es generado por una *gramática regular* y es reconocido por un *autómata finito*.

2.1.5 Gramáticas libres del contexto

Una **gramática libre del contexto** genera un *lenguaje libre del contexto*. Los lenguajes libres del contexto, como los lenguajes regulares son de gran importancia práctica, sobre todo en la definición de lenguajes de programación, en la formalización del concepto de análisis de gramática, simplificación de la traducción de lenguajes de programación y en otras aplicaciones de procesamiento de cadenas.

Definición. Una gramática libre del contexto G es un cuarteto $\langle VN, VT, P, S \rangle$, donde
 VN es un alfabeto finito de símbolos no terminales;
 VT es un alfabeto finito de símbolos terminales, tal que VT y VN son disjuntos;
 P es un conjunto finito de producciones de la forma $A \rightarrow \beta$, donde A es una variable de VN y β es una cadena de símbolos de $(VN \cup VT)^*$.
 S es el símbolo inicial.

Ejemplo. La siguiente es una gramática libre del contexto:

$$\begin{aligned} S &\rightarrow AA \\ A &\rightarrow AAA \\ A &\rightarrow a \\ A &\rightarrow bA \\ A &\rightarrow Ab \end{aligned}$$

y una posible derivación es

$$\begin{aligned} S &\Rightarrow \underline{AA} \Rightarrow \underline{AAAA} \Rightarrow a\underline{AAA} \Rightarrow ab\underline{AAA} \Rightarrow aba\underline{AA} \\ &\Rightarrow aba\underline{AbA} \Rightarrow abaab\underline{A} \Rightarrow abaabb\underline{A} \Rightarrow abaabba \end{aligned}$$

que especifica el lenguaje

$$L = \{w \in \{a, b\}^* : w \text{ tiene un número par de } a\text{'s}\}$$

Cabe hacer notar que a diferencia de una gramática regular, una gramática libre del contexto puede tener o no un símbolo terminal en el lado derecho de cada regla.

Un **lenguaje libre del contexto** es generado por una *gramática libre del contexto* y es reconocido por un *autómata de pila*. Este tipo de autómatas se crean agregando una memoria limitada en forma de pila a un autómata finito.

2.2 Inferencia Gramatical

Las gramáticas se emplean para describir la sintaxis de lenguajes o las relaciones estructurales de patrones o datos; también pueden usarse para caracterizar una fuente sintáctica que genere todas las sentencias (finitas o infinitas) en un lenguaje, o los patrones (o datos) pertenecientes a una clase particular (o un conjunto). Para modelar un lenguaje o para describir una clase de patrones o estructuras de datos en estudio con mayor realismo, se espera que la gramática usada pueda ser inferida directamente de un conjunto de

sentencias o patrones muestra. Este problema de aprender una gramática basado en un conjunto de sentencias muestra es llamado **inferencia gramatical** (Fu y Taylor, 1975).

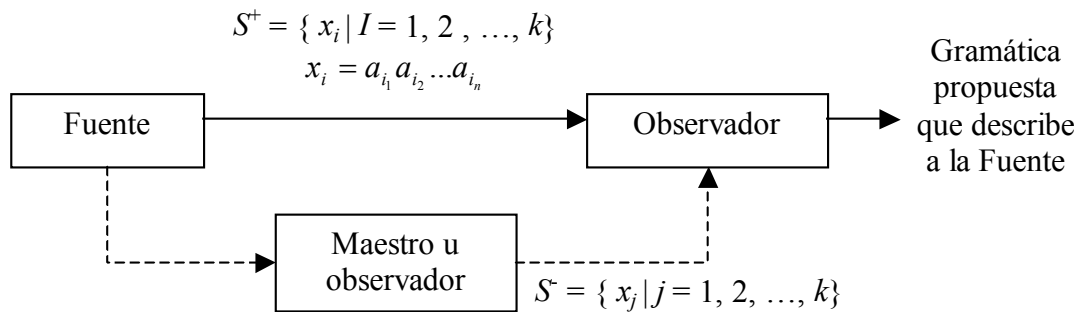


Figura 2-1. Modelo general del proceso de inferencia

La Figura 2-1 muestra la naturaleza general del problema de inferencia gramatical. Una **Fuente** genera sentencias o cadenas de la forma $x_i = a_{i_1} a_{i_2} \dots a_{i_n}$, donde los símbolos a_i son elementos del conjunto de símbolos terminales VT . Se asume que estas sentencias poseen algunas características estructurales únicas que están caracterizadas por una gramática G , que puede ser usada para modelar la fuente. Todas las sentencias que pueden ser generadas por la fuente están contenidas en el conjunto $L(G)$, el lenguaje generado por G , mientras que todas las sentencias que no pueden ser generadas están contenidas en el conjunto complemento $\overline{L(G)}$. A un observador se le da un conjunto finito S^+ de sentencias que son de $L(G)$ y posiblemente otro conjunto finito S^- de sentencias de $\overline{L(G)}$. Usando esta información el observador debe inferir las reglas sintácticas de la gramática desconocida G . El conjunto S^+ se puede obtener observando la salida de la fuente. El conjunto S^- no es tan fácil de obtener. Este conjunto se puede definir únicamente por un maestro externo con información extra sobre las propiedades de G . El maestro puede usar esta información para definir aquellas sentencias que pertenecen a S^- .

Si se tiene que S es una muestra finita de algún lenguaje y que G es una gramática que define el lenguaje $L(G)$, una pregunta importante que debe contestarse en esta situación es: “¿Es S una muestra del lenguaje $L(G)$?” Si G es un lenguaje regular o libre del contexto, es posible usar la definición de G para definir un autómata que pruebe cada elemento de S^+ para ver si está en $L(G)$ y cada elemento de S^- para ver si no está en $L(G)$. Este procedimiento de prueba se basa en la siguiente propiedad de estos lenguajes: Sea x una cadena de S , existen solamente un número finito de modos de que una cadena de longitud $|x|$ pueda ser generada por G , y cada una de estas cadenas de longitud $|x|$ pueda ser enumerada en una manera directa. El autómata esencialmente trata de generar la cadena x usando las reglas de reescritura de G . Si una derivación de x se encuentra usando estas reglas de reescritura, x es aceptada como una cadena de $L(G)$; de lo contrario, x es rechazada como una cadena de $L(G)$. Puesto que S tiene un número finito de cadenas siempre es posible, usando una prueba exhaustiva, decir si S es o no una muestra del lenguaje $L(G)$.

Fu y Taylor (1975, 1975b) hicieron una revisión sobre inferencia gramatical, presentando algoritmos de inferencia para gramáticas regulares y libres del contexto. También, presentan técnicas para inferir gramáticas de árboles y gramáticas estocásticas. Más recientemente Parekh y Honavar (2000) detallaron varios enfoques para aprender gramáticas regulares y gramáticas regulares estocásticas. Revisan enfoques para aprender gramáticas libres del contexto y mencionan enfoques para inferir gramáticas libres del contexto estocásticas.

Ebelin y Jiménez-Montaña (1980) proponen un algoritmo para inferir una gramática libre del contexto para macromoléculas biológicas, que permite medir la complejidad de la gramática obtenida: *complejidad gramatical*.

2.3 Medidas de información

2.3.1 Entropía

Shanon (1948) definió la *entropía* como una cantidad que pudiera medir, de algún modo, cuánta información y a qué tasa es producida por algunos procesos. Tiene la forma:

$$H = -K \sum_{i=1}^n p_i \log p_i$$

Cantidades de la forma $H = -\sum p_i \log p_i$ juegan un papel central en la *Teoría de la Información* como medidas de información, selección e incertidumbre. La forma de H es similar a la entropía definida en ciertas formulaciones de mecánica estadística.

En el análisis de biosecuencias, se tiene que una proteína o ADN es una secuencia de símbolos que contiene información sobre una especie animal o vegetal.

Las secuencias de símbolos se componen de letras de un alfabeto de λ letras (por ejemplo, para $\lambda = 4$, $\{A, C, G, T\}$ es el alfabeto DNA; para $\lambda = 2$, $\{0,1\}$ es el alfabeto binario). Las subcadenas de n letras son llamadas palabras- n . Si cada palabra i puede ser esperada en cualquier sitio arbitrario con una probabilidad bien definida, entonces las entropías de las palabras- n están dadas por

$$H_n = -\sum_i p_i^{(n)} \log_2 p_i^{(n)}$$

La sumatoria debe llevarse a cabo sobre todas las palabras con $p_i > 0$. El máximo número de palabras es λ^n , de modo que hay un incremento dramático del número de posibles palabras con respecto a n que hacen difícil estimar entropías de alto orden. Las entropías H_n miden la cantidad promedio de información contenida en una palabra de longitud n . Definiendo la *auto-información* de una palabra de longitud n como,

$$I_n = -\log_2 p_i^{(n)}$$

entonces,

$$H_n = \langle I_n \rangle$$

es el valor esperado de I_n . Las entropías condicionales,

$$h_n = H_{n+1} - H_n$$

dan la nueva información de el $(n + 1)$ ésimo símbolo dados los n símbolos precedentes. La *entropía de la fuente* (entropía de Shanon)

$$h = \lim_{n \rightarrow \infty} h_n = \lim_{n \rightarrow \infty} \frac{H_n}{n}$$

cuantifica el contenido de información por símbolo, y el decaimiento de las correlaciones medidas de h_n con la secuencia. H_n y h_n son buenos candidatos para detectar estructura en secuencias simbólicas ya que responden a cualquier desviación de independencia estadística. En una secuencia aleatoria con probabilidades equidistribuidas, $p^{(n)} = 1/\lambda^n$ se conserva para las probabilidades de palabras- n . Por lo tanto,

$$H_n = n \cdot \log_2 \lambda$$

Para secuencias binarias $\lambda = 2$, y $H_n = n$ bits. H_n exhibe una escala lineal para procesos equidistribuidos no aleatorios,

$$H_n = n \cdot H_1$$

siendo el coeficiente

$$H_1 = -\sum_{i=1}^{\lambda} p_i \log_2 p_i$$

la entropía ordinaria de la función de incertidumbre. Para secuencias de longitud $n = 1$, $H_n = n \cdot H_1$ se reduce a $H_{\max} = \log_2 \lambda$, que es el máximo valor posible de la entropía.

Para un proceso de Markov de orden k ,

$$h_n = h \quad \forall n \geq k$$

se mantiene; por lo tanto, las desviaciones de h_n de su límite indican dependencia estadística sobre la escala de palabras- $(n+1)$. El valor esperado de la complejidad algorítmica (Kolmogorov) es asintóticamente igual a la entropía de la fuente h .

Shannon define la redundancia informacional R_S como

$$R_S = 1 - \frac{H_1}{H_{\max}}$$

Si $p_i = 1/\lambda \forall i$, entonces $H_1 = H_{\max}$, y $R_S = 0$; esto es, cuando la secuencia es equiprobable, la redundancia del mensaje es cero. Alternativamente, suponiendo $p_j = 1$ para cualquier j y que $p_i = 0 \forall i \neq j$, entonces $H_1 = 0$ y $R_S = 1$.

La mayoría de las veces, las entropías H_n son estimadas de las frecuencias normalizadas de ocurrencias:

$$H_n^{obs} = -\sum_i \frac{k_i}{N} \log_2 \frac{k_i}{N}, \text{ y } I^{obs} = -\log_2 \frac{k_i}{N}$$

que son llamadas “entropías observadas” (respectivamente, auto-información observada). Aquí N denota el número total de palabras en la secuencia, y k_i es el número de ocurrencias de una cierta palabra i . En contraste con las cantidades informacionales definidas antes, que están referidas a un ensamble de secuencias, las cantidades observadas se refieren a *secuencias individuales*. En lo que sigue todas las entropías serán *entropías observadas* (con el superíndice “obs” suprimido por conveniencia).

2.3.2 Complejidad gramatical

La complejidad gramatical proporciona una medida del grado de aperiodicidad de una secuencia y también un criterio de optimización para evaluar categorizaciones de amino ácidos (Jiménez-Montaña, 1984).

El algoritmo para calcular la complejidad gramatical (*Grammar*) sobre una secuencia dada (ver Figura 2-2) funciona del siguiente modo:

1. Se buscan todas las subcadenas de longitud dos dentro de la secuencia para encontrar la subcadena que más se repite. Todas las ocurrencias de esta subcadena se sustituyen por un símbolo no terminal (una categoría sintáctica), si el número de ocurrencias es mayor que dos. El proceso se repite para la secuencia transformada, hasta que no haya mas subcadenas de longitud dos que se repitan mas de dos veces.
2. Después, se busca la subcadena de mayor longitud y se sustituye por un símbolo no terminal, si la longitud de la subcadena es mayor que dos. El proceso se repite hasta que no haya subcadenas de longitud mayor a dos que se repitan dos veces.

Sea G una gramática libre del contexto con alfabeto $V = V_T \cup V_N$, que genera solamente la palabra w . Esta gramática es un “programa” o “descripción” de la palabra w . El algoritmo anterior no debe entenderse para comprimir una secuencia *per se*, sino que fue diseñado para estimar la *complejidad gramatical* de la secuencia. Esta cantidad se define como sigue:

La complejidad de una regla de producción $A \rightarrow q$ se define por una estimación de la complejidad de la palabra en el lado derecho: $q = a_1^{v_1} \dots a_m^{v_m}$

$$K(A \rightarrow q) = \sum_{j=1}^m \{[\log_2 v_j] + 1\}$$

en donde $a_j \in V_T \cup V_N$, para cada $j = 1, \dots, m$; y $[x]$ denota la parte entera de un número real.

La complejidad $K(G)$ de una gramática G se obtiene sumando las complejidades de las reglas individuales. Finalmente, la estimación de la complejidad de la secuencia original es:

$$K(G(w)) = \min\{K(G) \mid G \rightarrow w\}$$

<p>a) Paso 1: Secuencia inicial (palabra w): 10010100100110011122 20000111101100000222</p> <p>Subsecuencias: Frec. Pares 7 10 8 00 7 01 5 11 2 22 Regla: [3] --> 00</p>	<p>b) Paso 2: Secuencia: 1 [3] 101 [3] 1 [3] 11 [3] 11122 2 [3] [3] 1111011 [3] [3] 0222</p> <p>Subsecuencias: Frec. Pares 5 1 [3] 5 [3] 1 2 10 2 01 5 11 2 22 2 [3] [3] Regla: [4] --> 1 [3]</p>	<p>c) Paso 3: Secuencia: [4] 10 [4] [4] 1 [4] 111222 [3] [3] 111101 [4] [3] 0222</p> <p>Subsecuencias: Frec. Pares 3 [4] 1 2 10 2 1 [4] 3 11 2 22 Regla: [5] --> [4] 1</p>
<p>d) Paso 4: Secuencia: [5] 0 [4] [5] [5] 11222 [3] [3] 111101 [4] [3] 0222</p> <p>Subsecuencias: Frec. Pares 3 11 2 22 Regla: [6] --> 11</p>	<p>e) Paso 5: Secuencia: [5] 0 [4] [5] [5] [6] 222 [3] [3] [6] [6] 01 [4] [3] 0222</p> <p>Subsecuencias: Frec. Subsecuencias 1 222 Regla: [7] --> 222</p>	<p>f) Gramática obtenida [S] --> [5] 0 [4] [5] [5] [6] [7] [3] [3] [6] [6] 0 1 [4] [3] 0 [7]</p> <p>[3] --> 00 [4] --> 1 [3] [5] --> [4] 1 [6] --> 11 [7] --> 222</p>
<p>g) Cálculo de la complejidad gramatical sobre la gramática obtenida: $K(G) = K([S]) + K([3]) + K([4]) + K([5]) + K([6]) + K([7]) = 17 + 2 + 2 + 2 + 2 + 2 = 27$</p>		

Figura 2-2. Algoritmo Grammar paso a paso

2.3.3 Redundancia algorítmica

Una definición algorítmica de redundancia es

$$R_0 = 1 - \frac{C_m}{\langle C_0 \rangle}$$

donde C_m es la complejidad del mensaje original. C_0 es la complejidad de una secuencia de símbolos equiprobable revuelta aleatoriamente de la misma longitud donde $p_i = 1/\lambda$. El subíndice “0” es usado para indicar que esta es la complejidad de un conjunto de datos subrogado revuelto aleatoriamente. Nótese que la distribución de la secuencia original no es necesariamente $p_i = 1/\lambda$. $\langle C_0 \rangle$ denota el valor medio de la complejidad encontrada promediando valores de varios subrogados construidos independientemente. La complejidad algorítmica da el valor más alto para secuencias aleatorias. $\langle C_0 \rangle$ es, por lo tanto, una estimación empírica de C_{max} y R_0 es la generalización sensible a la secuencia de la redundancia de Shanon, R_S . En este trabajo estimamos las complejidades con la complejidad gramatical libre del contexto (K). Por lo tanto R_0 la calculamos en función de K :

$$R_0 = 1 - \frac{K_m}{\langle K_0 \rangle}$$

2.3.4 Secuencias subrogadas

Para validar los resultados, se construyen ensambles subrogados consistiendo de un número finito de secuencias que son de la misma longitud que la secuencia original y que comparten con ésta ciertas propiedades estadísticas, por ejemplo, frecuencia de letras con la secuencia original. Se consideran tres tipos de ensambles de secuencias subrogadas. En la revoltura aleatoria estándar, la secuencia de datos original es revuelta. Esto conserva la frecuencia de las letras pero destruye todas las correlaciones. En la revoltura de pares aleatoria, la secuencia de datos original es revuelta de modo que la frecuencia de letras, la frecuencia de pares y por lo tanto, las correlaciones de pares son conservadas. En la revoltura triple aleatoria y revolturas de alto orden, la secuencia de datos es revuelta de modo que las frecuencias de tripletes (o frecuencias de grupos mayores) son conservadas.

Tomando como medida significativa la medida- S , definida por la diferencia entre el valor original y el valor subrogado medio de una medición, dividido por la desviación estándar de los valores subrogados:

$$S = \frac{|M_{orig} - \langle M_{subs} \rangle|}{\sigma_{subs}}$$

Así, la medida- S es el número de desviaciones estándar que la cantidad de interés difiere de el valor medio de sus substitutos.

2.3.5 Distancia algorítmica

La *distancia algorítmica* es una métrica para el espacio de secuencias binarias en términos de la complejidad de Kolmogorov. Esta mide la menor cantidad de información envuelta en transformar una secuencia s en una secuencia t . Li et al. (2001) introdujeron una versión normalizada de esta cantidad y la aplicaron al problema de comparar dos genomas. Contrastando con las tradicionales distancias de secuencias que requieren un alineamiento, la distancia algorítmica trabaja con secuencias no alineadas (Jiménez-Montaña, Feistel and Diez-Martínez, 2003). Dadas dos secuencias x y y , su distancia (Li et al., 2001) se define como

$$d(x, y) = 1 - \frac{K(x) - K(x | y)}{K(xy)} \quad 0 \leq d(x, y) \leq 1$$

donde $K(x | y)$ es la complejidad condicional de Kolmogorov de x dada y , y $K(x) = K(x | \varepsilon)$, donde ε es la secuencia vacía, es la complejidad incondicional de Kolmogorov de x . $K(x | y)$ mide la aleatoriedad de x dada y .

El numerador $K(x) - K(x | y)$ es la cantidad de información en y acerca de x . Dentro de un factor aditivo logarítmico, es cierto que

$$K(x) - K(x | y) = K(y) - K(y | x)$$

donde $K(y) - K(y | x)$ es la cantidad de información en x acerca de y . El denominador $K(xy)$ es la cantidad de información en la cadena x concatenada con y . Esto sirve como un factor de normalización, para obtener una medida independiente de la longitud. La expresión

$$R(x, y) = \frac{K(x) - K(x | y)}{K(xy)}$$

mide la no relación de x y y (Jiménez-Montaña, Feistel and Diez-Martínez, 2003).

2.4 Biosecuencias: Secuencias biológicas

2.4.1 ADN, Proteínas y Genes

El **Ácido Desoxirribonucleico (ADN)** es el material genético de todos los organismos celulares. El ADN lleva la información necesaria para dirigir la síntesis de proteínas y la

replicación. La síntesis de proteínas es la producción de las proteínas que necesita la célula para realizar sus actividades y desarrollarse. La replicación es el conjunto de reacciones por medio de las cuales el ADN se copia a sí mismo cada vez que una célula se reproduce y transmite a la descendencia la información de síntesis de proteínas que contiene. En casi todos los organismos celulares el ADN está organizado en forma de cromosomas situados en el núcleo de la célula¹.

Cada molécula de ADN está constituida por dos cadenas o bandas formadas por un elevado número de nucleótidos (compuestos químicos). Estas cadenas forman una especie de escalera retorcida tan conocida como doble hélice (Figura 2-3). Cada nucleótido está formado por tres unidades: una molécula de azúcar llamada desoxirribosa, un grupo fosfato y uno de cuatro posibles compuestos nitrogenados llamados bases: adenina (abreviada como A), guanina (G), timina (T) y citosina (C). La molécula de desoxirribosa ocupa el centro del nucleótido y está flanqueada por un grupo fosfato a un lado y una base al otro. El grupo fosfato está a su vez unido a la desoxirribosa del nucleótido adyacente de la cadena. Estas subunidades enlazadas desoxirribosa-fosfato forman los lados de la escalera; las bases están enfrentadas por parejas, mirando hacia el interior, y forman los travesaños.

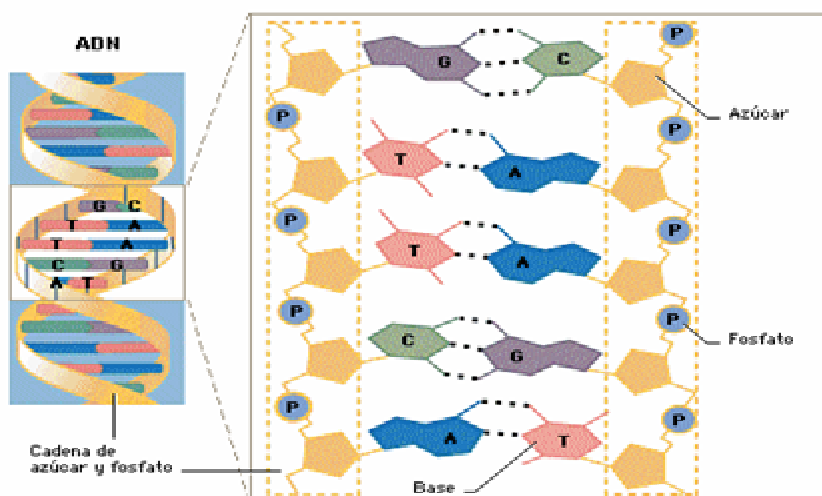


Figura 2-3. ADN

Los nucleótidos de cada una de las dos cadenas que forman el ADN establecen una asociación específica con los correspondientes de la otra cadena. Debido a la afinidad química entre las bases, los nucleótidos que contienen adenina se acoplan siempre con los que contienen timina, y los que contienen citosina con los que contienen guanina. Las bases complementarias se unen entre sí por enlaces químicos débiles llamados puentes de hidrógeno.

El ADN incorpora las instrucciones de producción de proteínas. Una **proteína** es un compuesto formado por moléculas pequeñas de aminoácidos, que determinan su estructura

¹ Enciclopedia Microsoft Encarta 2000, Microsoft Corporation.

y función. La secuencia de aminoácidos está a su vez determinada por la secuencia de bases de los nucleótidos del ADN. Cada secuencia de tres bases, llamada triplete, constituye una palabra del código genético o codón, que especifica un aminoácido determinado. Así, el triplete GAC (guanina, adenina, citosina) es el codón correspondiente al aminoácido leucina, mientras que el CAG (citosina, adenina, guanina) corresponde al aminoácido valina. Por tanto, una proteína formada por 100 aminoácidos queda codificada por un segmento de 300 nucleótidos de ADN. De las dos cadenas de polinucleótidos que forman una molécula de ADN, sólo una, llamada paralela, contiene la información necesaria para producción de una secuencia de aminoácidos determinada. La otra, llamada antiparalela, ayuda a la replicación.

Un **gen** es una secuencia de nucleótidos de ADN que especifica el orden de aminoácidos de una proteína por medio de una molécula intermediaria de ARNm (Ácido Ribonucleico mensajero). Un **genoma**, en virus y bacterias, es la totalidad de los genes que porta el individuo, mientras que en los organismos eucariotas, este término se refiere a los genes radicados en el núcleo de la célula; se tiene en cuenta una dotación haploide, es decir un único juego de cromosomas.

La sustitución de un nucleótido de ADN por otro que contiene una base distinta hace que todas las células o virus descendientes contengan esa misma secuencia de bases alterada. Como resultado de la sustitución, también puede cambiar la secuencia de aminoácidos de la proteína resultante. Esta alteración de una molécula de ADN se llama mutación. Casi todas las mutaciones son resultado de errores durante el proceso de replicación. La exposición de una célula o un virus a las radiaciones o a determinados compuestos químicos aumenta la probabilidad de mutaciones.

En casi todos los organismos celulares, la replicación de las moléculas de ADN tiene lugar en el núcleo, justo antes de la división celular. Empieza con la separación de las dos cadenas de polinucleótidos, cada una de las cuales actúa a continuación como plantilla para el montaje de una nueva cadena complementaria (Figura 2-4). A medida que la cadena original se abre, cada uno de los nucleótidos de las dos cadenas resultantes atrae a otro nucleótido complementario previamente formado por la célula. Los nucleótidos se unen entre sí mediante puentes de hidrógeno para formar los travesaños de una nueva molécula de ADN. A medida que los nucleótidos complementarios van encajando en su lugar, una enzima llamada ADN polimerasa los une enlazando el grupo fosfato de uno con la molécula de azúcar del siguiente, para así construir la hebra lateral de la nueva molécula de ADN. Este proceso continúa hasta que se ha formado una nueva cadena de polinucleótidos a lo largo de la antigua; se reconstruye así una nueva molécula con estructura de doble hélice.

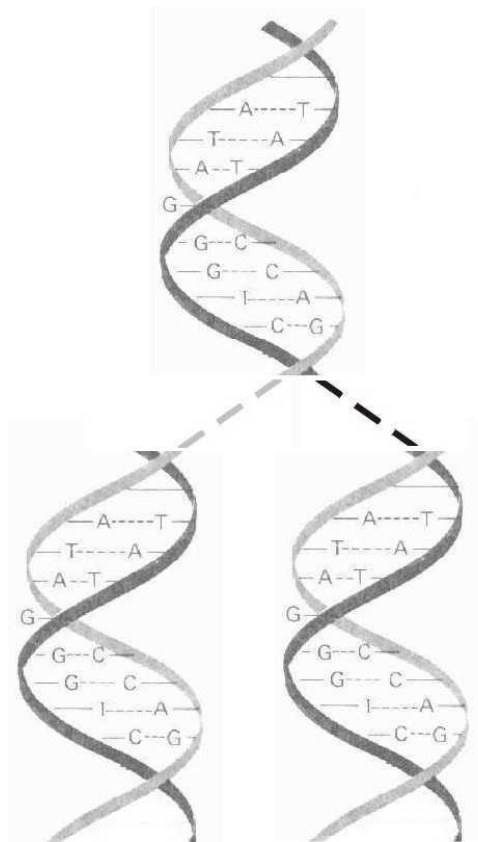


Figura 2-4. Replicación del ADN

Aunque la replicación del ADN es muy precisa, no es perfecta. Muy rara vez se producen errores, y el ADN nuevo contiene uno o más nucleótidos cambiados. Un error de este tipo, que recibe el nombre de mutación, puede tener lugar en cualquier zona del ADN. Si esto se produce en la secuencia de nucleótidos que codifica un polipéptido particular, éste puede presentar un aminoácido cambiado en la cadena polipeptídica. Esta modificación puede alterar seriamente las propiedades de la proteína resultante. Por ejemplo, los polipéptidos que distinguen la hemoglobina normal de la hemoglobina de las células falciformes difieren sólo en un aminoácido. Cuando se produce una mutación durante la formación de los gametos, ésta se transmitirá a las siguientes generaciones.

2.4.2 Bases de datos

Los recursos disponibles en Internet cambian más rápidamente que casi cualquier cosa en el mundo del procesamiento de la información. Esto ocurre con las herramientas dedicadas disponibles para el análisis de secuencias biológicas. Nuevas herramientas se hacen disponibles, mientras que otras se vuelven obsoletas. Existen muchas organizaciones que ponen a disposición del público la información que han recolectado en bases de datos (Ouellette, 1998). Hay tantas bases de datos y herramientas para su análisis en Internet, que es necesario crear bases de datos de bases de datos y herramientas, que permitan hacer

búsquedas sobre varias bases de datos alrededor del mundo; un ejemplo es DBGET Database Links (Figura 2-5), <http://www.genome.ad.jp/dbget/dbget.links.html>.

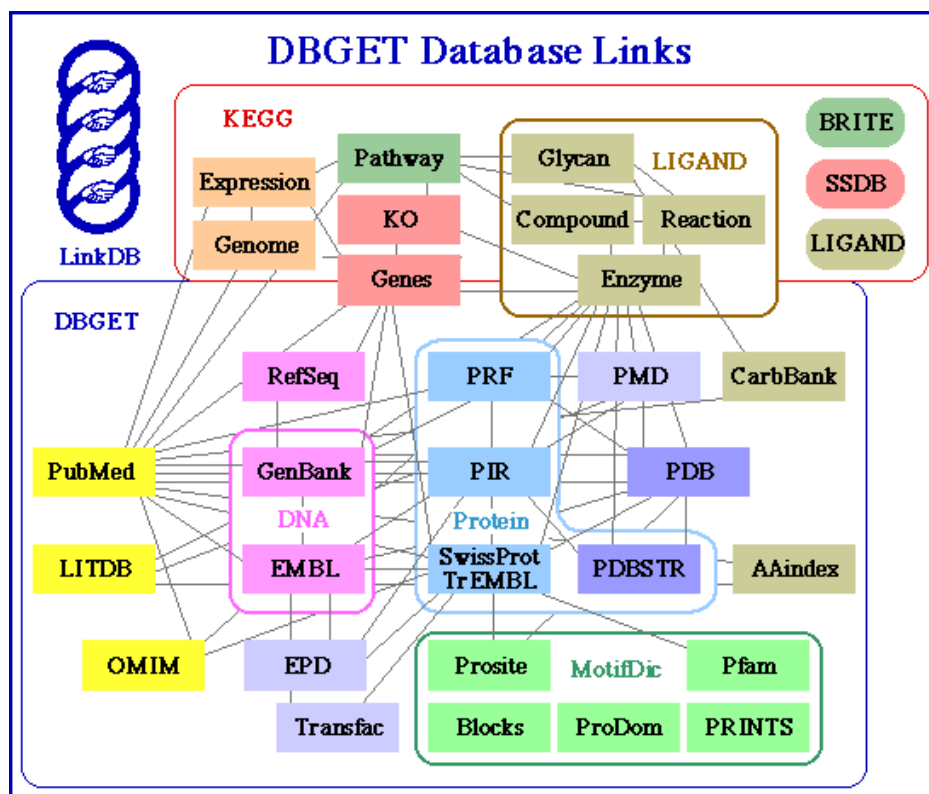


Figura 2-5. Base de datos de bases de datos

2.4.3 GenBank

GenBank es la base de datos de secuencias genéticas del National Institutes of Health, del National Center for Biotechnology Information (NCBI), USA, una colección de todas las secuencias de nucleótidos y proteínas disponibles públicamente. A enero de 2003 había aproximadamente 28,507,990,166 bases en 22,318,883 registros de secuencias. GenBank es parte del International Nucleotide Sequence Database Collaboration, que comprende el DNA DataBank de Japón (DDBJ), el European Molecular Biology Laboratory (EMBL), y el GenBank en el NCBI. Estas tres organizaciones intercambian datos diariamente.

Para obtener una secuencia biológica de GenBank, se debe entrar a la dirección Internet <http://www.ncbi.nlm.nih.gov/Genbank/index.html> desde un navegador como Internet Explorer o Netscape Navigator (Figura 2-6). La información allí es pública, y cualquier persona puede obtenerla, siempre y cuando sepa lo que está buscando.

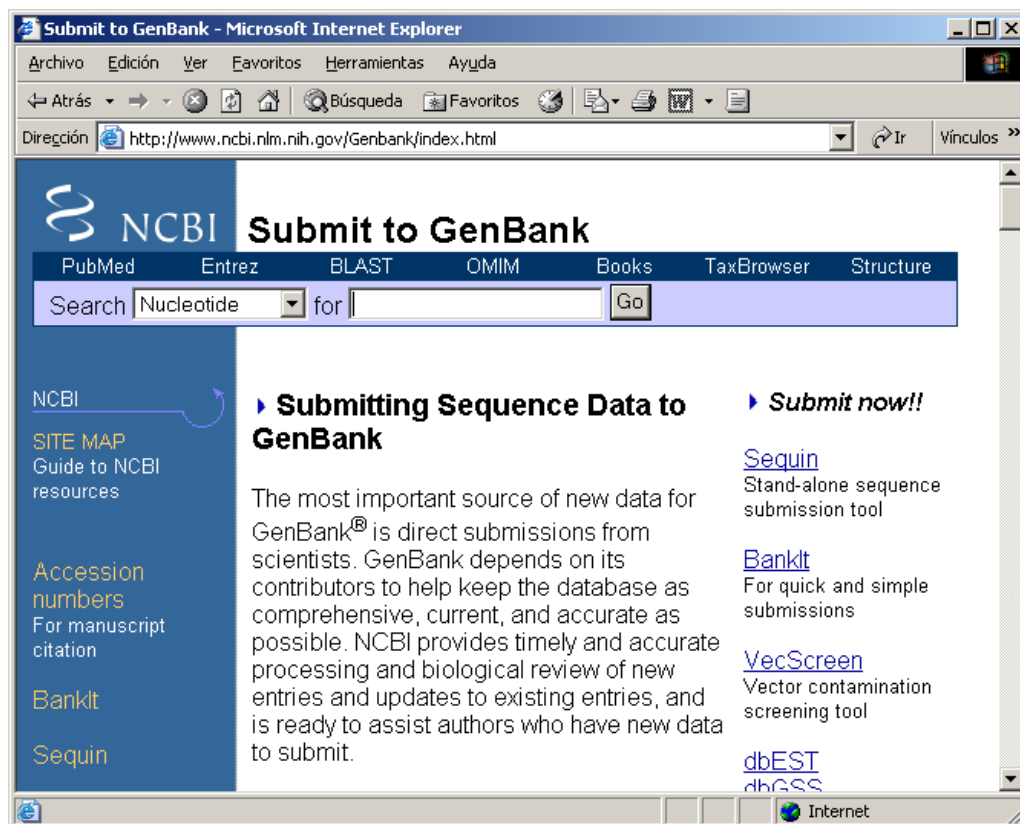


Figura 2-6. Página principal de GenBank

Por ejemplo, para obtener la secuencia completa del ADN mitocondrial humano, sabiendo que su número de acceso es el D38112, de la lista desplegable **Search** se selecciona la opción **Nucleotide**, en el campo **for** se escribe D38112 y se oprime el botón **Go**. GenBank responde con una página como la mostrada en la Figura 2-7.

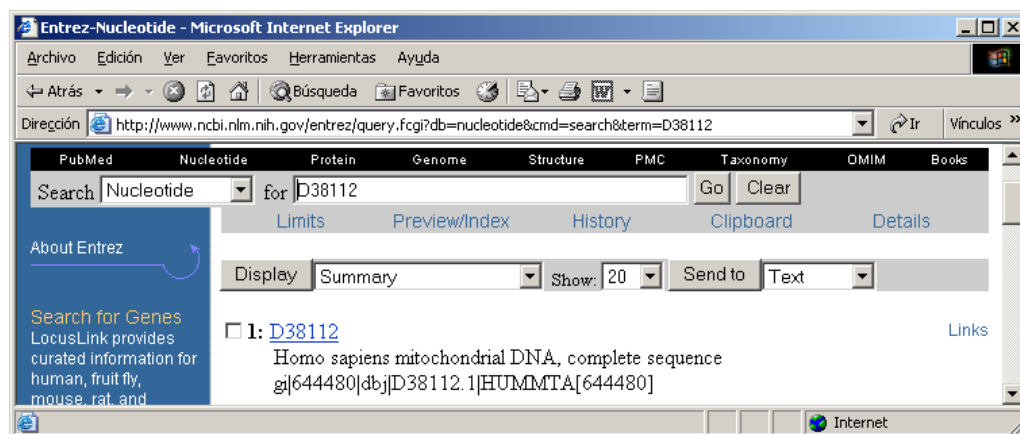


Figura 2-7. Resultado de la búsqueda en GenBank de la secuencia completa de ADN mitocondrial humano

Oprimiendo clic en el vínculo [D38112](#) se obtiene un sumario con la información referente a la secuencia buscada, como se muestra en la Figura 2-8.

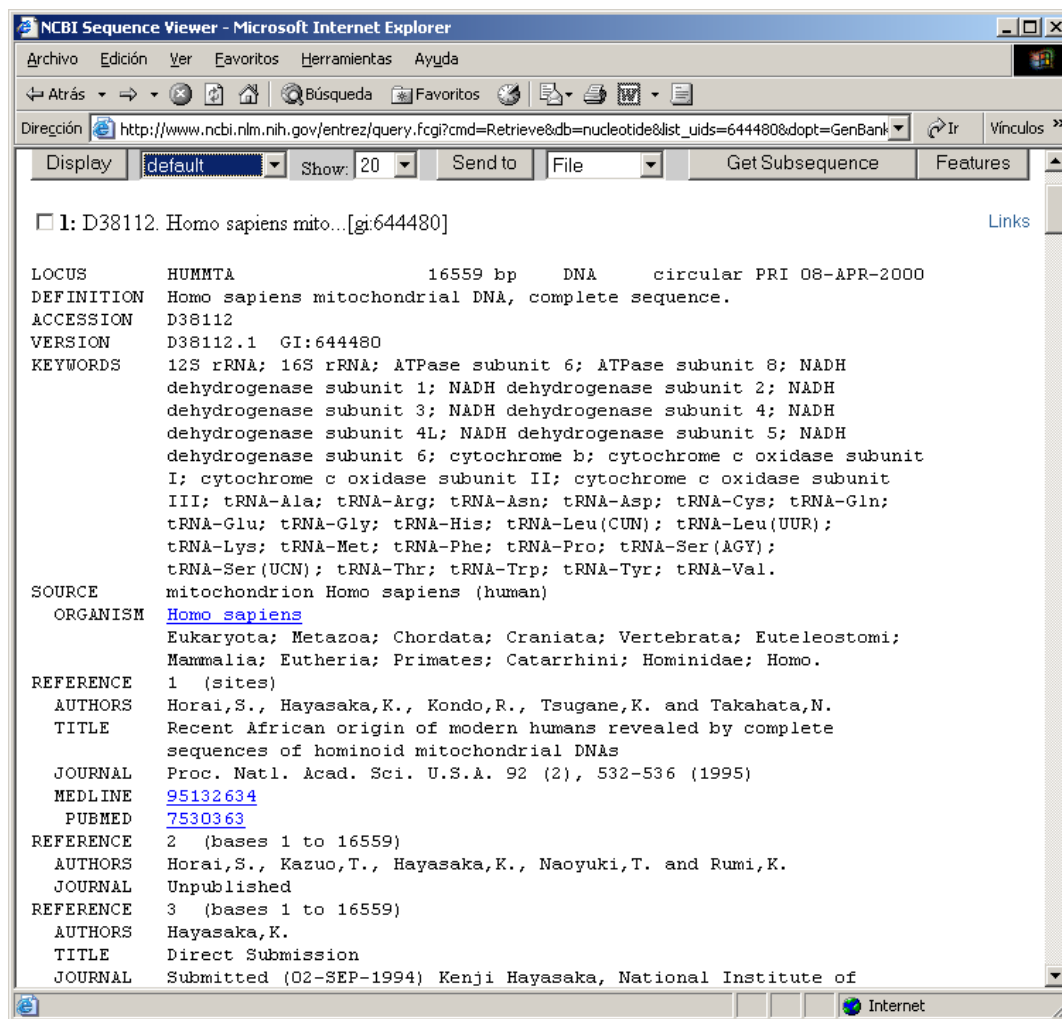


Figura 2-8. GenBank Sumario de la secuencia completa de ADN mitocondrial humano

Una vez encontrada la secuencia deseada se puede mostrar en cualquiera de varios formatos para analizarla con la herramienta preferida. Por ejemplo, para utilizarla con WinGramm 2 se obtiene en formato FASTA (Figura 2-9), sin la línea descriptiva.

```
>gi|644480|dbj|D38112.1|HUMMTA Homo sapiens mitochondrial DNA, complete sequence
GTTTATGTAGCTTACCTCCTCAAAGCAATACACTGAAAATGTTTAGACGGGCTCACATCACCCTATAAAC
AAATAGGTTTGGTCCTAGCCTTTCTATTAGCTCTTAGTAAGATTACACATGCAAGCATCCCCGTTCCAGT
GAGTTCACCTCTAAATCACCACGATCAAAAGGGACAAGCATCAAGCAGCAACAATGCAGCTCAAAACG
CTTAGCCTAGCCACACCCCAACGGAACAGCAGTGATAAACCTTTAGCAATAAACGAAAGTTAACTAA
...
AGCACTTAAACACATCTCTGCCAAACCCCAAAACAAAGAACCTTAACACCAGCCTAACCCAGATTTCAAA
TTTTATCTTTTGGGGTATGCACTTTTAAACAGTCACCCCACTAACACATATTTTCCCTCCCACTC
CCATACTACTAATCTCATCAATAACACCCCGCCATCCTACCCAGCACACACACACCCGTGCTAACCC
ATACCCGAACCAACCAACCCCAAGACACCCCCACA
```

Figura 2-9. Parte de la secuencia de ADN mitocondrial humano en formato FASTA

2.4.4 Árboles filogenéticos

Un árbol filogenético de una taxa (grupo de especies relacionadas biológicamente) dada, es un árbol que representa el curso evolutivo de tales especies.

Existen muchos métodos para construir árboles que puedan ser usados para datos moleculares. Cada método tiene sus ventajas y desventajas. De acuerdo a los tipos de datos usados, los métodos se dividen en dos categorías:

1. **Métodos basados en distancia.** En estos métodos se calcula una distancia evolutiva para todos los pares de unidades taxonómicas operacionales (OTUs; especies o poblaciones) o secuencias de ADN (o aminoácidos), y el árbol filogenético se construye considerando las relaciones entre estas distancias. Una vez que los valores de distancia se han obtenido, hay varios métodos para obtener el árbol.
2. **Métodos basados en caracteres discretos.** En estos métodos se usan datos con estados de caracteres discretos tales como estados de nucleótidos en secuencias de ADN, y el árbol se construye considerando las relaciones evolutivas de OTUs o secuencias de ADN en cada posición de carácter o nucleótido.

Para ambas categorías existen varios métodos diferentes para construir árboles que están basados en diferentes principios. Los principales son los siguientes (para más detalle ver Miyamoto and Cracraft, 1991):

1. Datos de distancia
 - a. Método de distancia aritmética (encadenamiento). UPGMA (Unweighted Pair Group Method with Arithmetic Means).
 - b. Método de distancia transformada (TD).
 - c. Método de Fitch y Margoliash (FM).
 - d. Método de evolución mínima (ME).
 - e. Método de distancia de Wagner (DW).
 - f. Método Neighborliness (ST).
 - g. Método Neighbor-Joining (NJ).
2. Datos de caracteres discretos
 - a. Método de máxima parsimonia (MP).
 - b. Método de parsimonia evolutiva (EP).
 - c. Método de máxima probabilidad (ML).

Método UPGMA

Dado que el método UPGMA es de los más usados, es el primero que se implementó en WinGramm 2, por lo que se describe a continuación.

Asumir que se tiene una cantidad de unidades taxonómicas operacionales (OTUs), indicadas por 1, 2, 3, ..., que pueden ser poblaciones, especies u otros grupos biológicos.

Asumir que las distancias genéticas entre estos grupos ya se han medido, de modo que se cuenta con una matriz de distancias genéticas, donde D_{ij} es la distancia genética entre los grupos i y j . Como ejemplo (Hendrick, 2000), se tienen cuatro OTUs y las distancias genéticas son como las dadas en la Tabla 2-2a. Ahora el objetivo es organizar estas OTUs en un modo que refleje relaciones biológicas usando estos valores de distancia genética.

(a)	1	2	3	(b)	(12)	3
2	D_{12}	—	—	3	$D_{(12)3}$	—
3	D_{13}	D_{23}	—	4	$D_{(12)4}$	D_{34}
4	D_{14}	D_{24}	D_{34}			

Tabla 2-2. (a) Distancia genética entre todos los pares de cuatro grupos y (b) entre los grupos después de que los grupos 1 y 2 han sido agrupados.

Usando el método UPGMA para determinar esta organización, se inicia agrupando los dos grupos que tienen la menor distancia genética, combinándolos en una nueva OTU. Entonces se calculan las nuevas distancias genéticas entre estas nuevas unidades y las otras restantes unidades tomando el promedio aritmético de sus distancias. Por ejemplo, asumir que hay cuatro OTUs diferentes y que las poblaciones 1 y 2 tienen las menores distancias genéticas para todas las comparaciones. Estas dos pueden ser combinadas en una nueva unidad (12). Las distancias genéticas entre estas unidades y las unidades 3 y 4 son entonces

$$D_{(12)3} = \frac{1}{2} (D_{13} + D_{23})$$

y

$$D_{(12)4} = \frac{1}{2} (D_{14} + D_{24})$$

Hay ahora una nueva matriz de valores de distancia usando las tres restantes OTUs, (12), 3 y 4; esta matriz se muestra en la Tabla 2-2b. Asumir ahora que D_{34} es la menor distancia genética, de modo que el grupo restante 3 y 4 es formado. El último paso en este simple ejemplo es calcular la distancia promedio entre los dos grupos, que es

$$D_{(12)(34)} = \frac{1}{4}(D_{13} + D_{14} + D_{23} + D_{24}) = \frac{1}{2} (D_{(12)3} + D_{(12)4})$$

En la Tabla 2-3 se presenta una matriz de distancias que consiste en el número estimado de sustituciones de nucleótidos para una secuencia de pares-base de ADN mitocondrial de cinco especies de primates (Hendrick, 2000). Usando el método UPGMA, sobre esta matriz se obtiene el árbol filogenético de la Figura 2-10, generado con WinGramm 2.1.

	(1) Humano	(2) Chimpancé	(3) Gorila	(4) Orangután
(2) Chimpancé	0.095	—	—	—
(3) Gorila	0.113	0.118	—	—
(4) Orangután	0.183	0.201	0.195	—
(5) Gibón	0.212	0.225	0.225	0.222

Tabla 2-3. Matriz de distancias para cinco especies de primates

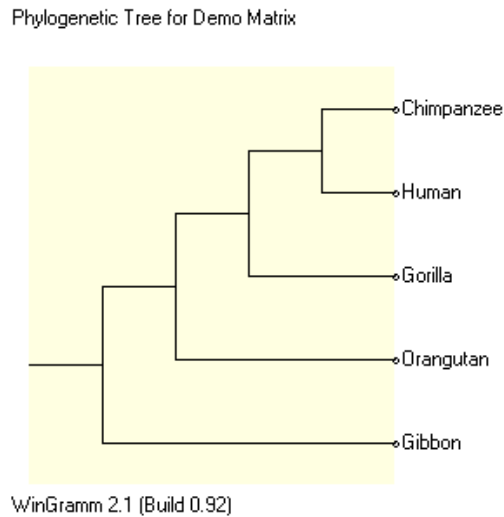


Figura 2-10. Árbol filogenético para cinco especies de primates

Dado que se está considerando una “distancia evolutiva”, esto significa qué tan distantes evolutivamente están una especie de la otra. En el árbol de la Figura 2-10 puede verse que el humano está más cerca evolutivamente del chimpancé que del gibón.

Con WinGramm 2, dicha distancia evolutiva se obtiene mediante la distancia algorítmica (pag. 14). WinGramm 2 permite obtener una matriz de distancias algorítmicas, a partir de un conjunto de secuencias y de allí construir el árbol filogenético.

2.4.5 Contenido informacional

El concepto de información y su cuantificación es esencial para entender los principios básicos de algunas técnicas de inteligencia artificial en biología molecular. Algunas técnicas de aprendizaje de máquina en particular, son excelentes para la tarea de descartar y compactar información redundante de secuencias. El análisis de redundancia y contenido de información de las secuencias biológicas ha sido fuertemente influenciado por la lingüística desde los años 1950's. La biología molecular surgió en la época en la que la metodología científica en general fue afectada por la filosofía lingüística. Muchas ideas son resultado del tratamiento filosófico y matemático de los lenguajes naturales que fueron “recicladas” para el análisis de secuencias biológicas “naturales”. La naturaleza digital de la información genética y el hecho de que las secuencias biológicas se traducen de una representación a otra en varios pasos consecutivos también han contribuido fuertemente al enlace y analogías entre las dos áreas (Baldi, 2001).

La cantidad de información en las secuencias biológicas está relacionada con su compresibilidad. Intuitivamente, las secuencias simples con muchas repeticiones pueden ser representadas usando una descripción más corta que las secuencias complejas y aleatorias que nunca se repiten.

El estudio de las propiedades estadísticas de segmentos repetitivos en secuencias biológicas y especialmente su relación a la evolución de genomas, es muy informativo. Tal análisis proporciona mucha evidencia para eventos más complejos que la fijación e incorporación de simples mutaciones generadas estocásticamente.

Una diferencia importante entre el texto común y el ADN es que las repeticiones ocurren de manera diferente. En los textos extensos, las repeticiones frecuentemente son muy chicas y cercanas unas a otras, mientras que en el ADN, las repeticiones largas pueden encontrarse alejadas unas de otras. Esto hace que los esquemas de compresión secuenciales convencionales sean poco efectivos sobre datos de ADN y proteínas. Sin embargo, se puede obtener una compresión significativa con algoritmos diseñados para otros tipos de datos como el `compress` de los sistemas UNIX, que está basado en el algoritmo de Lempel-Ziv o el algoritmo Grammar, que se implementa en este trabajo.

2.4.6 Análisis de secuencias de ADN

En esta sección se ejemplifica con secuencias de ADN el uso de la *complejidad gramatical* y la *distancia algorítmica* para el análisis de secuencias.

2.4.6.1 Ejemplo 1. Gen HUMTPA

La *complejidad gramatical*, K (ver sección 2.3.2) es una estimación del grado de compactación y redundancia de una secuencia, pero, ¿cómo interpretar su valor numérico?, para explicar esto, con WinGramm 2 se generaron 500 secuencias aleatorias de ADN de longitud (l) 50 (Sequense | Generate) y se seleccionaron aleatoriamente 500 subsecuencias fraccionadas del gen HUMTPA² (Edit | Fractionate Sequence), también de $l=50$. En la Tabla 2-4 se muestran algunos resultados de calcular la complejidad gramatical para las secuencias aleatorias y en la Tabla 2-5 para las secuencias del gen HUMTPA.

	Archivo	Secuencia	K
1	ACGT 14_1	GCCAGGTGCAAATCTACTATTAGTAGCGCTGCCCCGCAGGGTAGCGCGCT	35
2	ACGT 54_2	GTTATATTCATTAATGTACGTCATAACCCCTACACATATACTTTATCAC	36
3	ACGT 56_2	TTCTGTGGATCATCTCAGGGGAGTGCTAGACGTCCTTCATCACTAGACAT	36
4	ACGT 70_3	AGCAGGGTCGTGTAAATAATCCCGTAATTAAGTACACCACATACACCAGG	36
5	ACGT 02_5	CACAAAAAACTTAAACTCGAGGAAAGTCATAGCTGCGAGGCTACTCAA	37
...			
496	ACGT 55_3	GAAACGAATAGGATGGCCTTATACTGTTCATGCGTGACCTGCAGCTACCCG	44
497	ACGT 65_3	TCCCACCGGGAGTTAAGCTCCTCGTAAGACAGTCTATTGACCCGTGGGGT	44
498	ACGT 74_1	TAACAGTGGACTGTAGTCCAAAGAGGTTGCAATTCATATTGATCGGTCTT	44
499	ACGT 75_3	GCTGCCTCTGGTTTTACACGTCGAGGATTTAGTAGGGGAGATGTGTCCG	44
500	ACGT 82_3	ACTATCATAGTAACCTCCACAAATGCATTCCGACAGCCGTGGCTTGGAGCA	44

Tabla 2-4. 10 secuencias aleatorias ($l=50$) de menor y mayor complejidad gramatical (K)

² HUMTPA: Human Tissue Plasminogen Activator, número de acceso K03021 en GenBank

	Archivo	Secuencia	<i>K</i>
1	w339	CTTGTCTCTACACACACACACACACACACACACACACTCTCTCTCT	22
2	w482	TGATAGATAGATGATAGATTGATAGATGATTGATAGATAAATAGATGATA	24
3	w343	AGTCTCAAAAAAAAAAAAAAAAAAAAAAAAAAGAAAGAAGAAAGAAGAAA	25
4	w487	AATAGATGATAGATGATAGATGATAGGTGATAGATAGATTGATAGATGAT	25
5	w485	GATACATAGATGATAGACACATAGATGATAGACAGATTGACAGATGATAG	25
...			
496	w80	GGTTGCATTTCATTGGAGAAGCCTCTGCTGACAGAGTAAAGGGGAAAGTGA	43
497	w360	GTTTTTTTCTCTCCTTCCAGAATTTAAGGGACGCTGTGAAGCAATCATGGA	43
498	w269	GCGAACAAGCTGATTAGTTCCTTTTCCTAGGAGCTGGTTTTTATGGAGGGA	43
499	w117	CAGGCGCCTGCCACCACACCTGGTTAATTTTTGTATTTTTAGTAGAGACA	43
500	w668	ATTCAGCCCTAATACCCTGGGAAAGCACACTATTTGAGGTCCTTTGTGTG	44

Tabla 2-5. 10 secuencias (fracciones, $l=50$) del gen HUMTPA de menor y mayor complejidad gramatical (K)

Para las secuencias aleatorias, los valores de K caen en el rango 35...44 (Tabla 2-4), mientras que para las secuencias del gen HUMTPA los valores de K caen en el rango 22...44 (Tabla 2-5). Esto muestra que la complejidad gramatical de las secuencias aleatorias no baja mucho respecto a la longitud de la secuencia, ($l=50$), mientras que la complejidad gramatical para las secuencias del gen HUMTPA baja considerablemente (hasta $K=22$) para varios casos. La Figura 2-11 muestra gráficamente con qué frecuencia se encuentran secuencias con valores de K en el rango 22...44. Se puede notar por tales valores, que varias secuencias del gen HUMTPA tienen más segmentos repetitivos, que las secuencias aleatorias; aunque, también se nota que muchas secuencias del gen HUMTPA, parecen aleatorias.

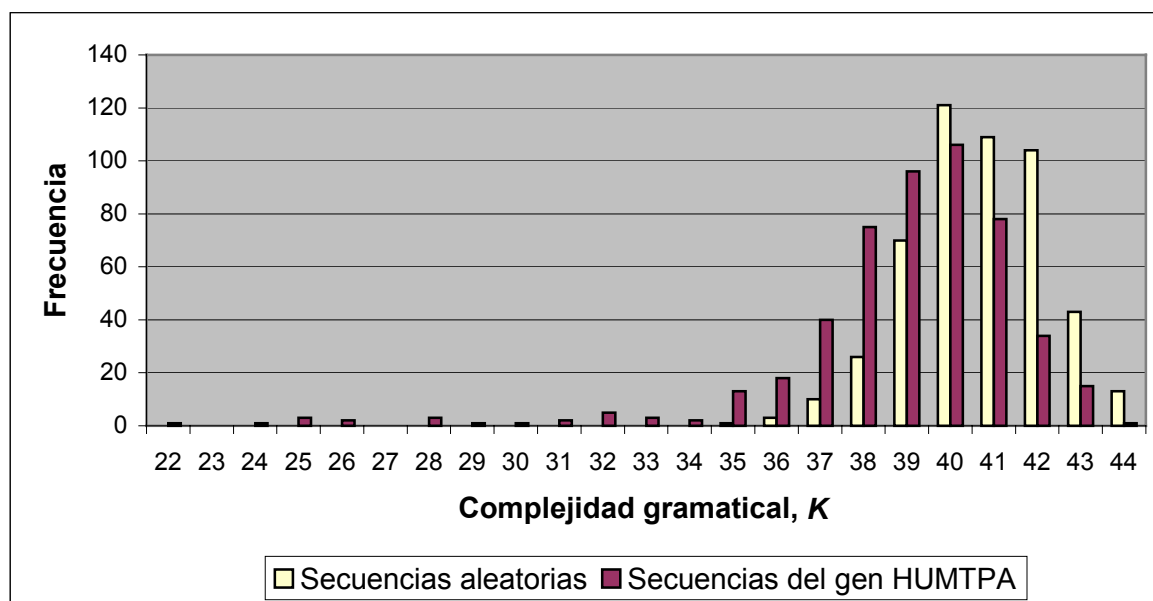


Figura 2-11. Gráfica de frecuencia de K , de 500 secuencias aleatorias y 500 del gen HUMTPA

Teóricamente, las secuencias aleatorias no deberían compactarse, pero **sí** se compactan, ya que WinGramm encuentra patrones en tales secuencias. El uso de secuencias aleatorias, es

muy importante en el análisis de secuencias, debido a que son un buen punto de comparación para encontrar o descartar redundancia.

Mediante el uso de la *distancia algorítmica* (ver sección 2.3.5) se pueden encontrar relaciones entre 2 ó mas secuencias, ya que para esta medida, las secuencias que no son muy repetitivas no cuentan, mientras que las secuencias que tienen repeticiones en común si cuentan. Por ejemplo, ¿las 5 secuencias que tienen menor K en la Tabla 2-5 tienen patrones aislados? ¿o tienen patrones en común? Para saber sobre esto, en WinGramm 2 se colocaron las 10 secuencias aleatorias de la Tabla 2-4 y las 10 del gen HUMTPA de la Tabla 2-5 en un solo grupo y se les calculó la distancia algorítmica (Calc | Algorithmic Distance), obteniéndose la matriz de distancias de la Tabla 2-6.

	ACGT 02_5	ACGT 14_1	ACGT 54_2	ACGT 55_3	ACGT 56_2	ACGT 65_3	ACGT 70_3	ACGT 74_1	ACGT 75_3	ACGT 82_3	w117	w239	w269	w343	w360	w482	w485	w487	w668	w80
ACGT 02_5																				
ACGT 14_1	62																			
ACGT 54_2	69	67																		
ACGT 55_3	63	63	60																	
ACGT 56_2	69	69	70	64																
ACGT 65_3	69	65	60	64	64															
ACGT 70_3	65	67	62	64	70	64														
ACGT 74_1	65	65	62	66	62	62	64													
ACGT 75_3	61	67	64	64	62	64	64	58												
ACGT 82_3	57	65	66	66	60	58	66	62	70											
w117	62	62	63	61	61	61	63	59	55	59										
w239	63	69	70	56	64	60	66	62	66	62	63									
w269	60	64	67	63	63	59	69	63	57	61	54	59								
w343	48	58	57	55	53	53	55	51	51	55	52	43	46							
w360	62	68	67	65	61	61	65	61	61	67	64	61	56	46						
w482	57	57	58	58	62	62	56	60	56	60	59	54	59	41	63					
w485	58	64	59	55	53	57	59	57	57	57	58	53	58	48	58	37				
w487	56	58	53	55	55	57	49	53	51	57	50	47	52	42	54	17	30			
w668	59	61	58	60	60	56	60	60	58	60	59	62	57	51	61	58	55	57		
w80	60	70	67	61	63	59	69	53	59	53	60	59	56	52	60	59	56	56	59	

Tabla 2-6. Matriz de distancias algorítmicas de 10 secuencias aleatorias (ACGT *) y 10 secuencias del gen HUMTPA (w*)

Con esta matriz de distancias, se construyó el árbol filogenético con WinGramm 2 (Matrix | Build Phylogenetic Tree), obteniéndose el que se muestra en la Figura 2-12, en donde se puede apreciar (en un rectángulo remarcado) que las secuencias w343, w239, w487, w482 y w485 (las 5 de menor K) quedan agrupadas. Esto indica que existen segmentos repetitivos en común entre esas 5 secuencias del gen HUMTPA.

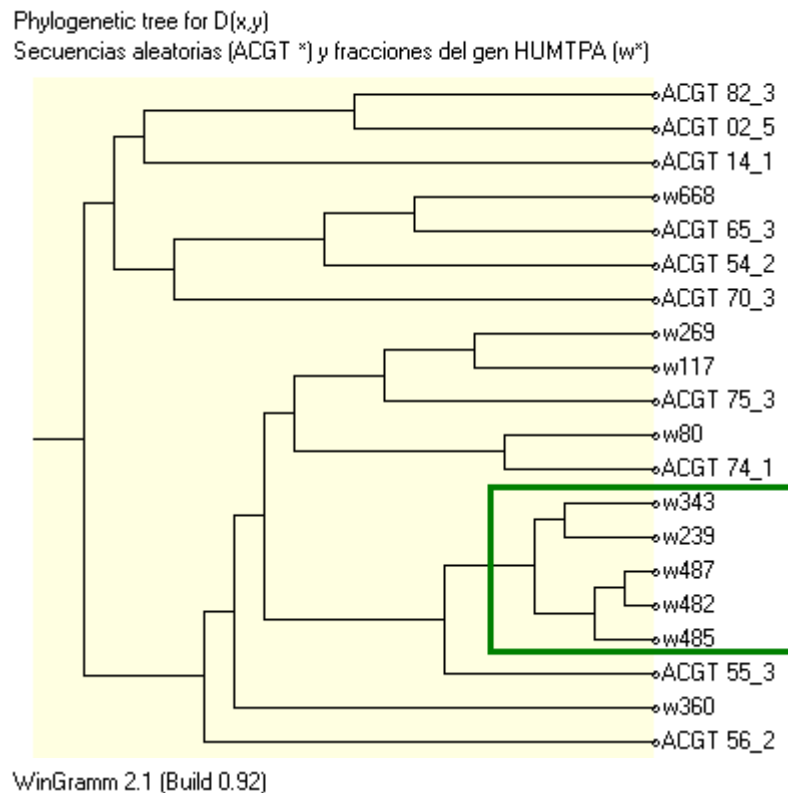


Figura 2-12. Árbol filogenético de 10 secuencias aleatorias (ACGT *) y 10 secuencias del gen HUMTPA (w*)

2.4.6.2 Ejemplo 2. Grado de relación de mamíferos

Con la finalidad de construir con WinGramm 2 un árbol evolutivo a partir de genomas mitocondriales completos y compararlo con los árboles reportados por Cao et al (1998) y Bennet, Li y Ma (2003), se obtuvieron de GenBank las 20 secuencias completas de ADN mitocondrial que usaron Cao et al (1998) y que son de las siguientes 20 especies de mamíferos:

Id	Secuencia	Long.	(en inglés)	Nombre científico	Referencia
1	Vaca	16338	cow	Bos taurus	V00654
2	Ballena de aleta	16398	fin whale	Balaenoptera physalus	X61145
3	Ballena azul	16402	blue whale	Balaenoptera musculus	X72204
4	Foca de puerto	16826	harbor seal	Phoca vitulina	X63726
5	Foca gris	16797	gray seal	Halichoerus grypus	X72004
6	Gato	17009	cat	Felis catus	U20753
7	Caballo	16660	horse	Equus caballus	X79547
8	Rinoceronte indio	16829	Indian rhinoceros	Rhinoceros unicornis	X97336
9	Ratón doméstico	16295	mouse	Mus musculus	V00711
10	Rata	16300	rat	Rattus norvegicus	X14848
11	Humano	16559	human	Homo sapiens	D38112
12	Chimpancé	16554	chimpanzee	Pan troglodytes	D38113
13	Chimpancé pigmeo	16563	Bonobo	Pan paniscus	D38116

Id	Secuencia	Long.	(en inglés)	Nombre científico	Referencia
14	Gorila	16364	Gorilla	Gorilla gorilla	D38114
15	Orangután de Borneo	16389	Bornean orangután	Pongo pygmaeus p.	D38115
16	Orangután de Sumatra	16499	Sumatran orangután	Pongo pygmaeus abelii	X97707
17	Gibón	16472	common gibbon	Hylobates lar	X99256
18	Zarigüeya	17084	Virginia opossum	Didelphis virginiana	Z29573
19	Wallaroo	16896	Wallaroo	Macropus robustus	Y10524
20	Ornitorrinco	17019	Platypus	Ornithorhynchus anatinus	X83427

Con WinGramm 2 se obtuvo la matriz de distancias de la Tabla 2-7 y el árbol filogenético de la Figura 2-13.

En el árbol obtenido, los grupos de Primates, Ferungulados, Roedores y Externos (marsupiales y ornitorrinco) están correctamente agrupados, coinciden con los agrupamientos reportados por Cao et al (1998), y Bennet, Li y Ma (2003), quienes obtienen el tipo de árbol 1 de la Figura 2-14.

La diferencia con los mencionados autores está en que WinGramm 2 relaciona a los roedores con los ferungulados, que es el tipo de árbol 3 de la Figura 2-14.

D(x,y)	Rinoceronte blanco	Ballena azul	Gato	Chimpancé	Gibón	Vaca	Ballena de aleta	Gorila	Foca gris	Foca de puerto	Caballo	Humano	Ratón	Zarigüeya	Orangután	Ornitorrinco	Chimpancé pigmeo	Rata	Orangután de Sumatra	Wallaroo
Rinoceronte blanco	10	7534	7531	7806	7792	7404	7591	7714	7532	7678	7285	7820	7572	7958	7765	7940	7816	7744	7693	7908
Ballena azul	7534	14	7673	7676	7880	7346	5321	7724	7626	7602	7557	7804	7582	7874	7761	7904	7762	7764	7787	7886
Gato	7531	7673	16	7953	7915	7547	7690	7787	7393	7469	7538	7865	7667	7997	7862	7849	7875	7765	7822	7951
Chimpancé	7806	7676	7953	16	7216	7714	7809	6054	7802	7822	7683	5590	7740	8006	7145	7940	3742	7850	7075	7946
Gibón	7792	7880	7915	7216	12	7810	7897	7192	7938	7930	7787	7304	7790	8026	7331	8174	7230	7870	7405	8010
Vaca	7404	7346	7547	7714	7810	10	7471	7650	7558	7534	7459	7684	7536	7752	7779	7834	7592	7558	7759	7844
Ballena de aleta	7591	5321	7690	7809	7897	7471	22	7775	7761	7669	7612	7875	7703	7821	7840	7863	7845	7765	7828	7941
Gorila	7714	7724	7787	6054	7192	7650	7775	10	7832	7872	7767	6258	7716	8032	7095	8000	6158	7772	7015	7896
Foca gris	7532	7626	7393	7802	7938	7558	7761	7832	12	3544	7549	7866	7680	8002	7919	7920	7816	7634	7871	7908
Foca de puerto	7678	7602	7469	7822	7930	7534	7669	7872	3544	26	7565	7832	7700	7968	7885	7978	7778	7672	7861	7886
Caballo	7285	7557	7538	7683	7787	7459	7612	7767	7549	7565	12	7741	7625	7931	7802	8023	7743	7679	7676	7925
Humano	7820	7804	7865	5590	7304	7684	7875	6258	7866	7832	7741	16	7776	8008	7113	8038	5492	7838	7029	7912
Ratón	7572	7582	7667	7740	7790	7536	7703	7716	7680	7700	7625	7776	8	7766	7749	7738	7714	7060	7785	7706
Zarigüeya	7958	7874	7997	8006	8026	7752	7821	8032	8002	7968	7931	8008	7766	14	8085	7970	7874	7900	8029	7612
Orangután	7765	7761	7862	7145	7331	7779	7840	7095	7919	7885	7802	7113	7749	8085	16	8015	7143	7863	4968	8023
Ornitorrinco	7940	7904	7849	7940	8174	7834	7863	8000	7920	7978	8023	8038	7738	7970	8015	24	7922	7918	8081	7884
Chimpancé pigmeo	7816	7762	7875	3742	7230	7592	7845	6158	7816	7778	7743	5492	7714	7874	7143	7922	12	7706	7073	7838
Rata	7744	7764	7765	7850	7870	7558	7765	7772	7634	7672	7679	7838	7060	7900	7863	7918	7706	10	7777	7768
Orangután Sumatra	7693	7787	7822	7075	7405	7759	7828	7015	7871	7861	7676	7029	7785	8029	4968	8081	7073	7777	10	8035
Wallaroo	7908	7886	7951	7946	8010	7844	7941	7896	7908	7886	7925	7912	7706	7612	8023	7884	7838	7768	8035	10

Tabla 2-7. Matriz de distancias de genomas mitocondriales completos de 20 mamíferos

Phylogenetic Tree

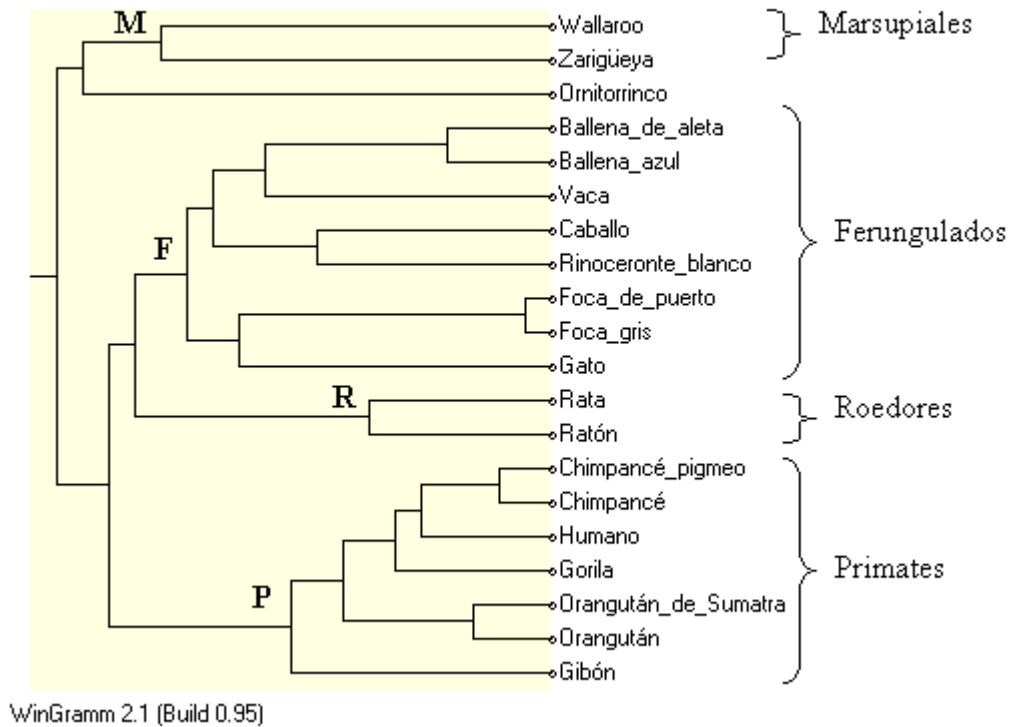


Figura 2-13. Árbol filogenético de genomas mitocondriales completos de 20 mamíferos

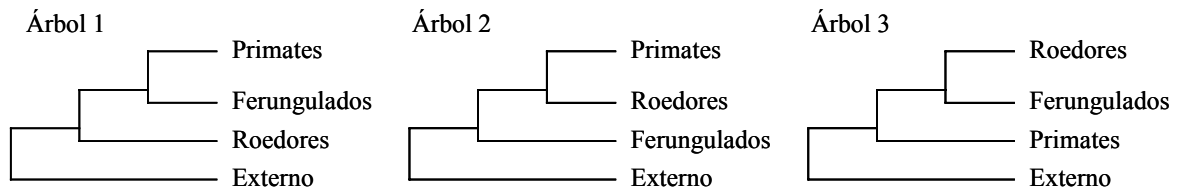


Figura 2-14. Tres posibles grupos entre primates, ferungulados y roedores

3 Desarrollo de software científico y de IA

En este capítulo se describen conceptos de ingeniería de software, la metodología de desarrollo de software seguida para la realización de WinGramm 2, se hace énfasis en el uso adecuado de técnicas de programación, tanto para software en general como científico y de inteligencia artificial (IA), la problemática de la portabilidad del software y se describen de manera general las características de WinGramm 2.

3.1 Ingeniería del Software

Es **software** es el conjunto de instrucciones o programas que se ejecutan dentro de una computadora de cualquier tamaño y arquitectura, documentos que comprenden formularios virtuales e impresos y datos que combinan números y texto y también incluyen representaciones de información de audio, video e imágenes (Pressman, 2001).

El software es el alma de la computadora, la “máquina” que conduce a la toma de decisiones en cualquier área. Sirve de base para la investigación científica moderna y de resolución de problemas de ingeniería. Es el factor clave que diferencia los productos y servicios modernos. Está inmerso en sistemas de todos tipo: de transportes, médicos, de telecomunicaciones, militares, procesos industriales, entretenimientos, productos de oficina y del hogar, etc. El software es casi ineludible en el mundo moderno.

La **Ingeniería del Software** es la aplicación de un enfoque sistemático, disciplinado y cuantificable hacia el desarrollo, operación y mantenimiento del software; es decir, la aplicación de ingeniería al software.

La ingeniería del software está compuesta de paradigmas, que son una serie de pasos que abarcan métodos, herramientas y procedimientos (Pressman, 2001). La elección de un paradigma para la ingeniería del software se lleva a cabo de acuerdo con la naturaleza del proyecto y de la aplicación, los métodos y herramientas a usar y los controles y estrategias requeridos. Se muestran a continuación cuatro paradigmas: el ciclo de vida clásico, construcción de prototipos, el modelo en espiral y el modelo de ensamblaje de componentes.

3.1.1 El ciclo de vida clásico

La Figura 3-1 muestra el paradigma del ciclo de vida clásico. Algunas veces llamado *modelo en cascada*. Este paradigma exige un enfoque sistemático y secuencial del desarrollo del software que comienza en el nivel del sistema y progresa a través del análisis, diseño, codificación, prueba y mantenimiento.

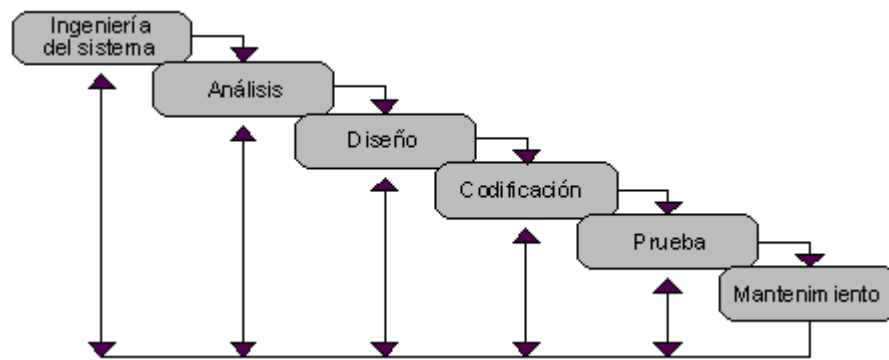


Figura 3-1. Ciclo de vida clásico.

El ciclo de vida clásico es el paradigma más antiguo y más ampliamente usado en la ingeniería del software. Entre los problemas que se presentan en este paradigma se encuentran:

1. Los proyectos reales raramente siguen el flujo secuencial que propone el modelo. Siempre hay iteraciones y se crean problemas en la aplicación del paradigma.
2. Normalmente es difícil para el cliente establecer explícitamente al principio todos los requisitos. El ciclo de vida clásico lo requiere y tiene dificultades en acomodar posibles incertidumbres que pueden existir al comienzo de muchos proyectos.
3. El cliente debe tener paciencia. Hasta llegar a las etapas finales del desarrollo del proyecto, no estará disponible una versión operativa del programa. Un error importante no detectado hasta que el programa esté funcionando puede ser desastroso.

A pesar de sus inconvenientes, es mejor que desarrollar el software sin guías.

3.1.2 Construcción de prototipos

Normalmente un cliente define un conjunto de objetivos generales para el software, pero no identifica los requisitos detallados de entrada, proceso o salida. En otros casos, el programador puede no estar seguro de la eficiencia de un algoritmo, de la adaptabilidad de un sistema operativo o de la forma en que debe realizarse la interacción hombre-máquina. En estas y muchas otras situaciones, puede ser mejor método de ingeniería de software la construcción de un prototipo (Figura 3-2).

La construcción de prototipos es un proceso que facilita al programador la creación de un modelo del software a construir. El modelo tomará una de las tres formas siguientes: (1) un prototipo en papel o un modelo basado en computadora que describa la interacción hombre-máquina, de forma que facilite al usuario la comprensión de cómo se producirá tal interacción; (2) un prototipo que implemente algunos subconjuntos de la función requerida

del programa deseado, o (3) un programa existente que ejecute parte o toda la función deseada, pero que tenga otras características que deban ser mejoradas en el nuevo trabajo de desarrollo.

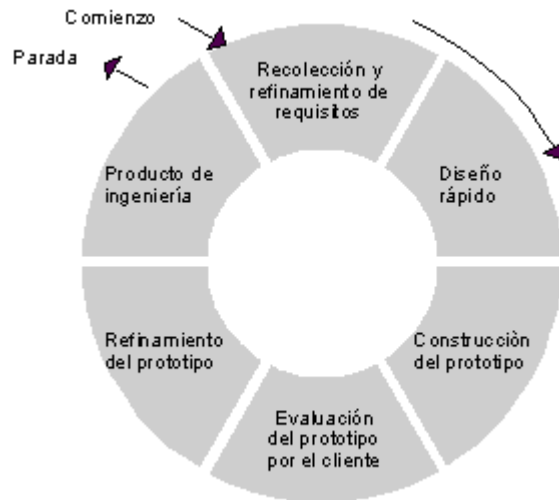


Figura 3-2. Creación de prototipos.

Idealmente, el prototipo sirve como mecanismo para identificar los requisitos del software. Si se va a construir un prototipo que funcione, el realizador intenta hacer uso de fragmentos de programas existentes o aplica herramientas (por ejemplo, generadores de informes, gestores de ventanas, etc.) que faciliten la rápida generación de programas que funcionen.

El prototipo puede servir como "primer sistema". Al igual que en el ciclo de vida clásico, la construcción de prototipos puede ser problemática por las siguientes razones:

1. El cliente ve funcionando lo que parece ser una primera versión del software, ignorando que el prototipo se ha hecho con "plastilina y alambres", ignorando que, por las prisas en hacer que funcione, no se han considerado los aspectos de calidad o de mantenimiento del software a largo plazo. Cuando se le informa de que el producto debe ser reconstruido, el cliente se vuelve loco y solicita que se apliquen "cuantas mejoras" sean necesarias para hacer del prototipo un producto final que funcione. El gestor del desarrollo del software cede demasiado a menudo.
2. El técnico de desarrollo, frecuentemente, impone ciertos compromisos de implementación con el fin de obtener un prototipo que funcione rápidamente. Puede que utilice un sistema operativo o un lenguaje de programación inapropiados, simplemente porque ya está disponible y es conocido; puede que implemente ineficientemente un algoritmo, sencillamente para demostrar su capacidad. Después de algún tiempo, el técnico puede haberse familiarizado con esas elecciones y haber olvidado las razones por las que eran inapropiadas. La elección menos ideal forma ahora parte integral del sistema.

Aunque pueden aparecer problemas, la construcción de prototipos es un paradigma efectivo para la ingeniería del software.

3.1.3 El modelo en espiral

El modelo en espiral ha sido desarrollado para cubrir las mejores características tanto del ciclo de vida clásico, como de la creación de prototipos, añadiendo al mismo tiempo un nuevo elemento: el análisis de riesgo, que falta en esos paradigmas. El modelo, representado mediante la espiral de la Figura 3-3, define cuatro actividades principales, representadas por los cuatro cuadrantes de la figura.

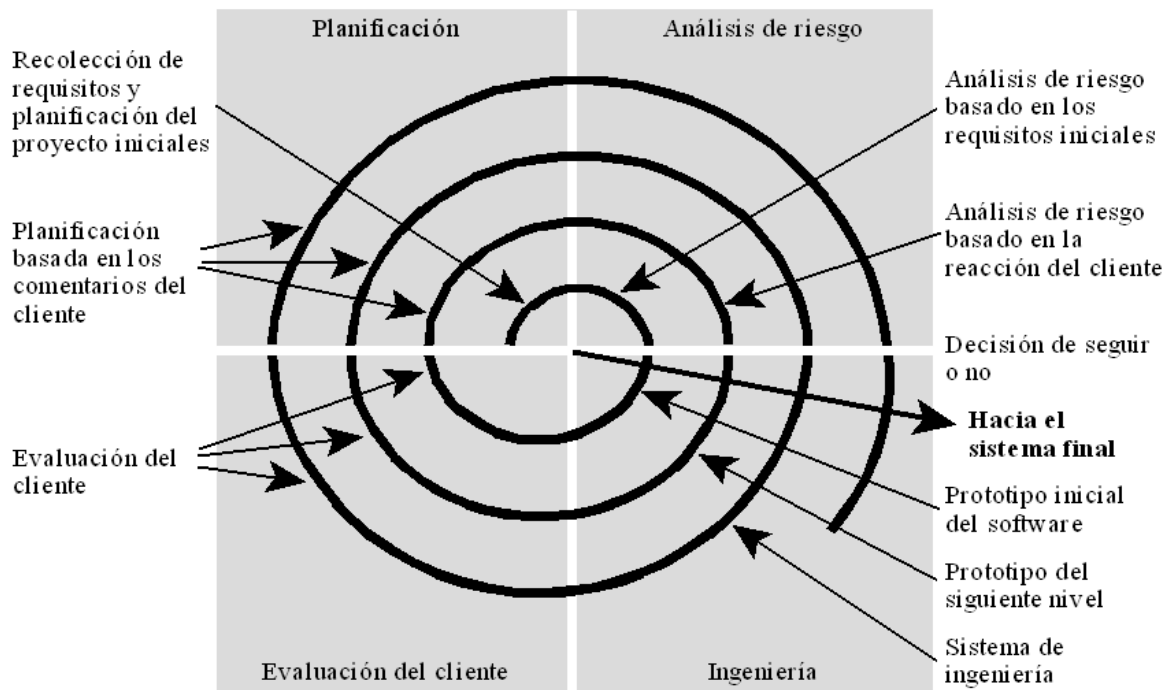


Figura 3-3. El modelo en espiral.

Con cada iteración alrededor de la espiral (comenzando en el centro y siguiendo hacia el exterior), se construyen sucesivas versiones del software, cada vez más completas.

Este paradigma es actualmente el enfoque más realista para el desarrollo de software y de sistemas a gran escala. Utiliza un enfoque "evolutivo" para la ingeniería del software, permitiendo al desarrollador y al cliente entender y reaccionar a los riesgos en cada nivel evolutivo. Utiliza la creación de prototipos como un mecanismo de reducción del riesgo, pero, lo que es más importante, permite a quien lo desarrolla aplicar el enfoque de creación de prototipos en cualquier etapa de la evolución del producto.

3.1.4 El modelo de ensamblaje de componentes

Las tecnologías de objetos proporcionan el marco de trabajo técnico para un modelo de proceso basado en componentes para la ingeniería del software. El paradigma de orientación a objetos enfatiza la creación de clases que encapsulan tanto los datos como los algoritmos que se utilizan para manejar los datos. Si se diseñan y se implementan adecuadamente, las clases orientadas a objetos son reutilizables por las diferentes aplicaciones y arquitecturas de sistemas basados en computadora.

El modelo de ensamblaje de componentes Figura 3-4 incorpora muchas de las características del modelo en espiral. Es evolutivo por naturaleza y exige un enfoque interactivo para la creación del software. Sin embargo, este modelo configura aplicaciones desde componentes (o clases) ya preparados.

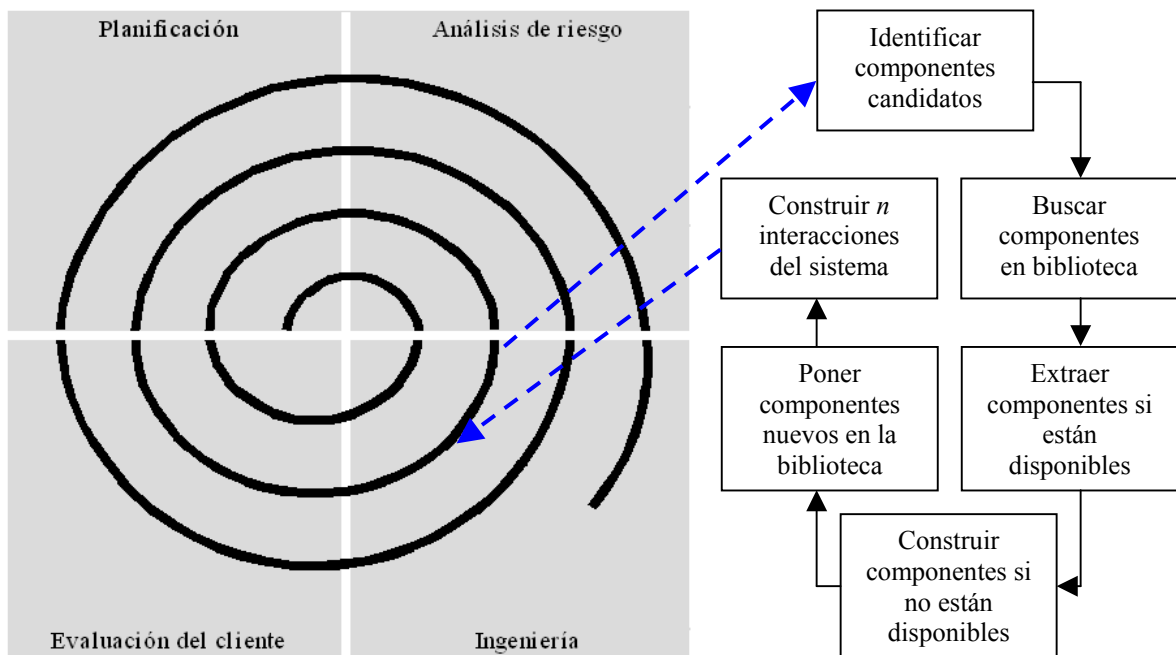


Figura 3-4. El modelo de ensamblaje de componentes.

El modelo de ensamblaje de componentes lleva a la reutilización del software, y la reutilización proporciona beneficios a los ingenieros del software. Con este modelo se puede obtener una reducción del 70 por ciento de tiempo de ciclo de desarrollo, un 84 por ciento del costo de un proyecto y un índice de productividad del 26.2, comparado con la norma industrial del 16.9 (Pressman, 2001). Aunque estos resultados están en función de la robustez de la biblioteca de componentes, no hay duda que el ensamblaje de componentes proporciona ventajas significativas para los ingenieros del software.

En la comunidad de investigación y desarrollo en modelado y simulación, hay un reconocimiento general de la necesidad de una ingeniería de software efectiva; aunque no

hay un consenso amplio en términos de procesos y métodos. Algunos miembros de la comunidad científica son partidarios de un enfoque de código abierto para software científico con la finalidad de promover la revisión del código fuente (Swain, 2001).

3.2 Software científico y de inteligencia artificial

Es innegable que el software ha influido en el desarrollo científico y tecnológico desde su aparición ya que sirve de base para la investigación científica moderna. Pressman (2001) establece algunas categorías genéricas de aplicación del software, como: software de sistemas, software de tiempo real, software de gestión, software de ingeniería y científico, software empotrado, software de computadoras personales, software basado en web y software de inteligencia artificial. De estas, son de interés para este trabajo las siguientes:

- El **software científico** (y el de ingeniería) se caracteriza por los algoritmos de manejo de números. Sus aplicaciones van desde astronomía hasta la vulcanología, desde el análisis de la presión de los automotores a la dinámica orbital de las lanzaderas espaciales y desde la biología molecular a la fabricación automática. Sin embargo, las nuevas aplicaciones del área de ingeniería/ciencia se han alejado de los algoritmos convencionales numéricos.
- El **software de inteligencia artificial** hace uso de algoritmos no numéricos para resolver problemas complejos para los que no son adecuados el cálculo o el análisis directo. Como ejemplos están los sistemas expertos, el reconocimiento de patrones (texto, imágenes, voz), las redes neuronales artificiales, prueba de teoremas, juegos entre otros.

De hecho, no se podría separar tajantemente el software científico del software de inteligencia artificial, ya que ésta es en sí parte de una ciencia (de la computación) y muchas de las aplicaciones de inteligencia artificial son para resolver problemas científicos.

3.2.1 Exactitud

Durante el diseño de una aplicación para uso científico, es bien importante seleccionar los tipos de datos básicos del lenguaje de programación usado (byte, entero, doble, real, etc). La precisión de los tipos de datos juega un papel fundamental para la exactitud de los cálculos numéricos.

A diferencia de otros tipos de aplicaciones, la selección inadecuada de tipos de datos básicos puede llevar a resultados numéricos erróneos que bien podrían pasar desapercibidos, cuando se realizan cálculos sobre un modelo matemático complejo.

En un estudio de cuatro años, Hatton y Roberts (1994) trataron de determinar qué tan consistentes son los resultados de la computación científica, y de allí, estimar su exactitud. El experimento lo llevaron a cabo en un área de las ciencias de la tierra, el procesamiento

de datos sísmicos, con 15 grandes paquetes comerciales desarrollados independientemente, que implementan algoritmos matemáticos de las mismas o similares especificaciones publicadas en el mismo lenguaje de programación (Fortran), y que habían sido desarrollados durante 20 años antes. Sus resultados son bastante reveladores. Mientras que los científicos pensaban que su código era exacto por la precisión de la aritmética usada, en dicho estudio las diferencias numéricas crecían a una tasa alrededor del 1% en promedio de diferencia absoluta por 4000 líneas de código implementado, y lo que era aún peor, la naturaleza de la diferencia no era aleatoria. Además, la industria del procesamiento de datos sísmicos tiene estándares de calidad arriba de la media para su desarrollo de software con funciones de garantía de calidad y conjuntos de datos de prueba sólidos. Compararon sus resultados con otros trabajos, mostrando tasas de fallo estadísticamente similares en software de diferentes disciplinas. Esto indica de que la realización de software científico puede ser mucho menos exacto de lo que se pudieran pensar.

3.2.2 Validación y verificación

Hay muchos aspectos de validación y verificación en software científico. Una vez que se tiene bien identificado el problema de la vida real a modelar, se realiza un modelo conceptual, a partir de este se hace un modelo matemático y de este último un modelo computacional, para finalmente escribir el código (Figura 3-5). Para verificar el software obtenido hay que contestar la pregunta: ¿está bien el modelo? y para validarlo: ¿es éste el modelo correcto?

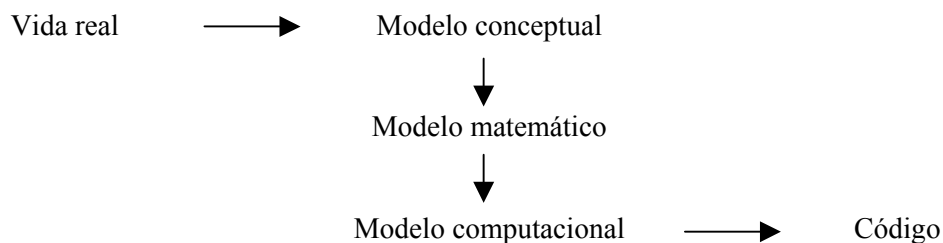


Figura 3-5. Verificación y validación del software científico

Trabajar en proyectos de software científico no solo significa desarrollar el algoritmo que éste usa. Esto incluye escribir el código, la documentación, hacer muchas pruebas y depuraciones. Las pruebas y depuraciones consumen mucho más tiempo que desarrollar el puro algoritmo. Para que otros científicos sean capaces de usar el software, normalmente también se necesitan herramientas de configuración, interfaces gráficas, una buena documentación y ayuda en línea con ejemplos. Todo esto significa trabajo adicional y mucho tiempo invertido para llegar a los mismos resultados (Orcero, 2001).

3.3 Portabilidad

Tradicionalmente se han usado lenguajes de programación como Fortran o C/C++ para desarrollar software científico. Para modelado y simulación se usan generalmente simuladores.

Dada la gran cantidad de arquitecturas de computadoras y sistemas operativos (plataformas) para operarlas, es conveniente desarrollar software que se pueda portar de una plataforma a otra. El lenguaje de programación por excelencia para esto ha sido C/C++ estándar, ya que para casi cualquier sistema operativo existe un compilador de C/C++. El inconveniente de esto es que la compatibilidad se puede asegurar para programas en modo texto. Si se quiere desarrollar un software que aproveche la interfaz gráfica en dado sistema operativo, se tienen que agregar librerías gráficas, que no siempre están disponibles para todas las arquitecturas. Aunque, si existen una gran cantidad de librerías gráficas para C/C++, como, GTK, TCL/TK, OpenGL, etc. Otra alternativa de lenguaje de programación es usar Java, que es algo parecido a C++.

Sin importar el lenguaje de programación seleccionado para desarrollar software científico, si se quiere hacerlo portable a la mayor cantidad de plataformas, siempre existirán problemas al tratar de hacerlo. Dubois, Epperly y Kumfert (2003) hablan sobre la falta de habilidad para crear software científico portable, que llaman el “problema de construir”; esto consiste en crear un sistema que entregue un sistema de software sobre una variedad de plataformas que tienen varias herramientas de programación y configuraciones. Resaltan que las dimensiones del problema de construir son difíciles de medir. De el esfuerzo gastado en código científico moderno, quizás de un 10 hasta un 30 por ciento de éste (para proyectos chicos y grandes, respectivamente) fue en tratar de resolver el problema de construir.

3.4 WinGramm 2

WinGramm 2 está desarrollado usando el modelo de ensamblaje de componentes (página 34). La programación se inició con Delphi 5 como una aplicación VCL³ para obtener WinGramm 2.0 y se continuó con Delphi 7 como una aplicación CLX⁴ para obtener WinGramm 2.1 y LinGramm 2.1, por lo que el producto de software está basado en componentes Delphi y clases construidas para este programa.

³ VCL: Visual Component Library – Para desarrollar aplicaciones Windows

⁴ CLX: Component Library for Cross-Platform – Para desarrollar aplicaciones Windows y Linux

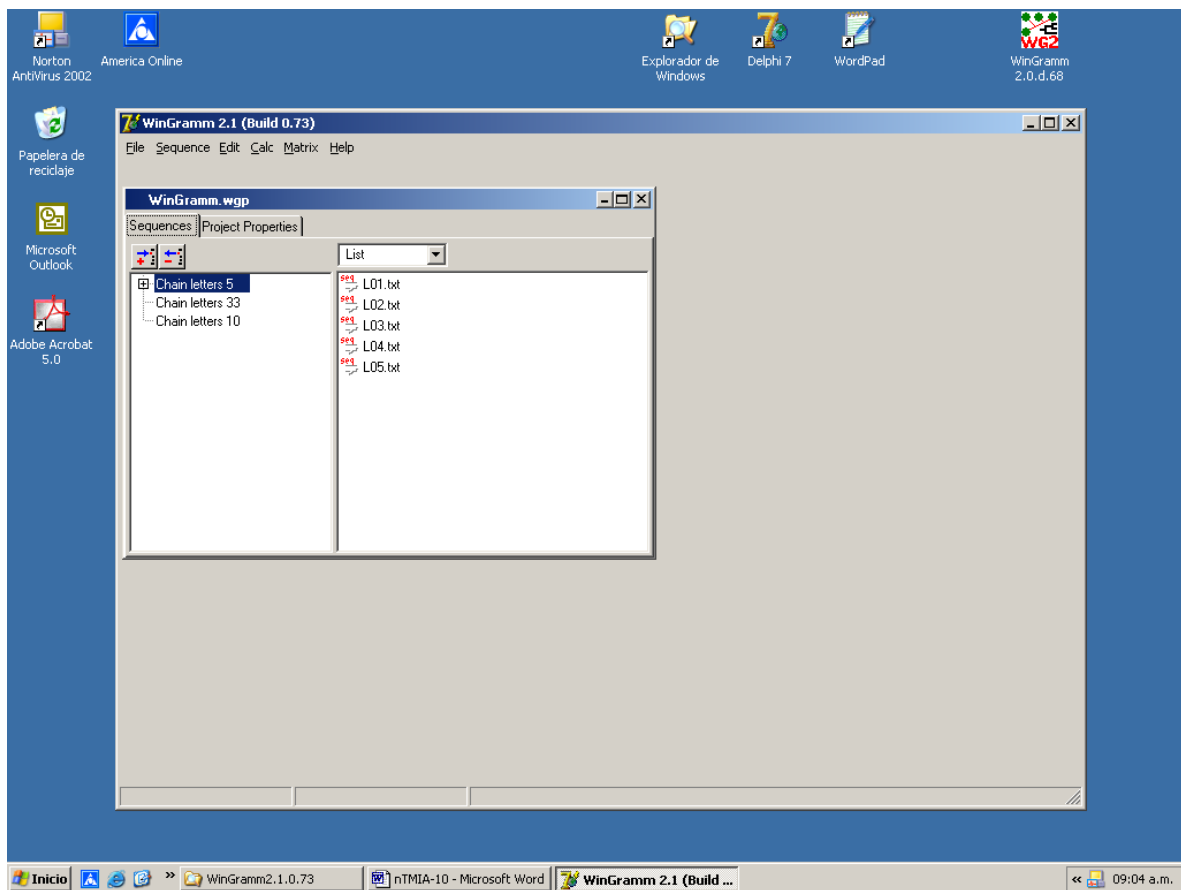


Figura 3-6. WinGramm 2.1 en un ambiente Windows XP

3.4.1 Código Orientado a Objetos

Programar WinGramm 2 orientado a objetos ha permitido optimizar gran parte de la funcionalidad respecto a WinGramm 1.

Para WinGramm 2 se crearon varias clases, por ejemplo dos de ellas son TSequence y TGrammar. En la clase TSequence se encapsula toda la funcionalidad requerida para leer una secuencia de un archivo, determinar el alfabeto de la misma, etc. (Figura 3-7).

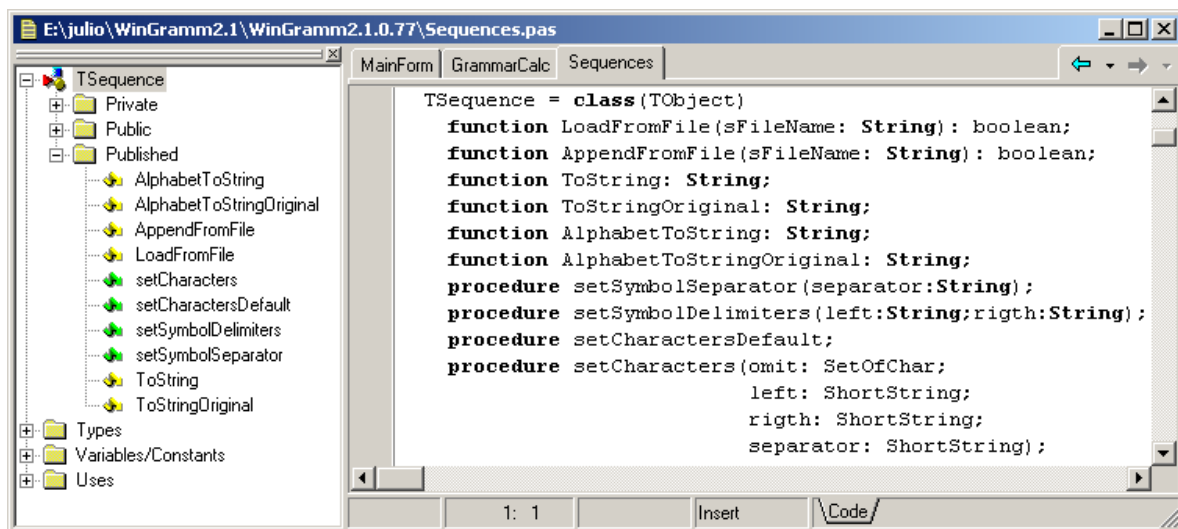


Figura 3-7. Clase TSequence

En la clase TGrammar se encapsula toda la funcionalidad requerida para calcular la complejidad gramatical de una secuencia (Figura 3-8).

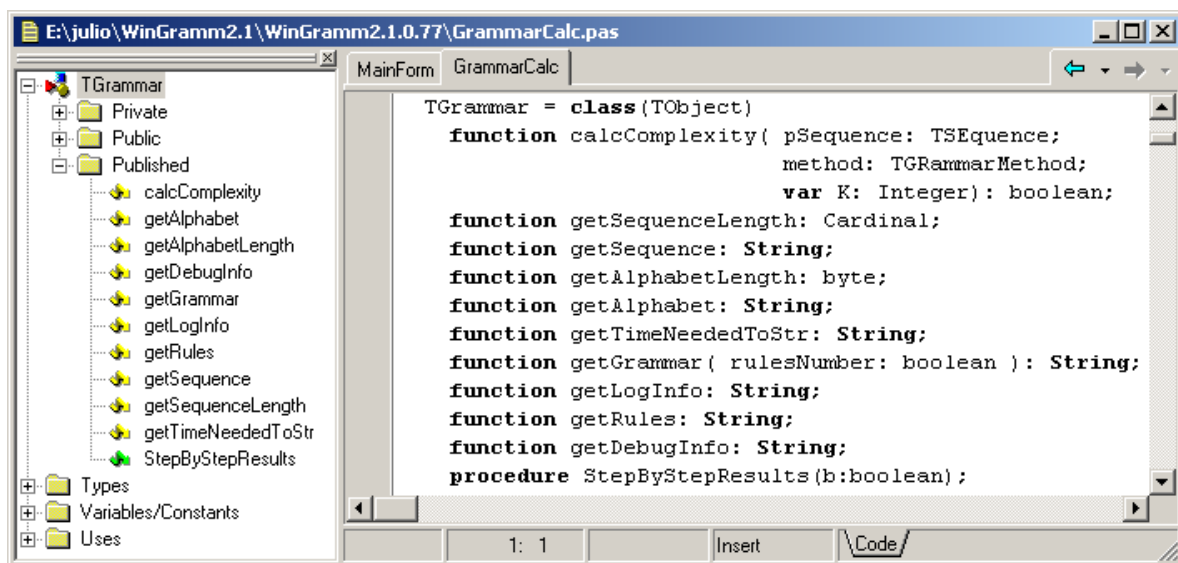


Figura 3-8. Clase Tgrammar

Cuando en WinGramm 2 se selecciona la opción **Calc | Grammatical Complexity** se ejecuta el método **GramaticalComplexity1Click**; en la Figura 3-9 se muestra el código de dicho método (un procedimiento de Delphi), omitiendo algunas líneas o conjuntos de líneas marcadas con puntos suspensivos (...). En el Apéndice A está el código completo del procedimiento. Brevemente, en este procedimiento se hace lo siguiente:

- Se declaran los objetos Sequence y Grammar, de las clases TSequence y TGrammar respectivamente (líneas 4 y 5).
- Se obtiene la lista de secuencias seleccionadas por el usuario en la ventana proyecto (línea 9).
- Se inicia un ciclo para procesar cada secuencia (línea 14).
- Se lee el archivo de una secuencia (línea 17).
- Se calcula la complejidad gramatical de dicha secuencia (línea 22).
- El valor de la complejidad gramatical, el tiempo requerido para calcularlo y la gramática obtenida (entre otros datos) se guardan en la variable sResult que se usa posteriormente para mostrar los resultados (líneas 23-29).

```

1  procedure TFormMain.GrammaticalComplexity1Click(Sender: TObject);
2  var
3      ...
4      Sequence: TSequence;
5      Grammar: TGrammar;
6      sResult: String;
7  begin
8      ...
9      tsFileNameToCalc := FormProject.getSelectedFiles;
10     ...
11     Sequence := TSequence.Create;
12     ...
13     sResult := '';
14     for i:=0 to tsFileNameToCalc.Count - 1 do
15     begin
16         ...
17         Sequence.LoadFromFile(tsFileNameToCalc.Strings[i]);
18         ...
19         Grammar := TGrammar.Create;
20         Grammar.StepByStepResults(formCalc.cbStepByStep.Checked);
21
22         if Grammar.calcComplexity(Sequence, gmNvoGram, K) then
23             sResult := sResult +
24                 ...
25                 'Grammatical Complexity = ' + IntToStr( K ) + CrLf + CrLf +
26                 'Time needed: ' + Grammar.getTimeNeededToStr + CrLf + CrLf +
27                 'Grammar Sequence: ' + Grammar.getSequence + CrLf + CrLf +
28                 'Grammar Rules Found:' + CrLf + Grammar.getRules + CrLf + CrLf +
29                 ...
30
31         Application.ProcessMessages;
32         ...
33     end;
34
35     ...
36 end; { GrammaticalComplexity1Click }

```

Figura 3-9. Código del método GrammaticalComplexity1Click

3.4.2 Mejoras

Respecto a la última versión de WinGramm 1 (la 1.7), en WinGramm 2 se hicieron las siguientes mejoras:

- **Es más rápido.** La Tabla 3-1 muestra los resultados de ejecutar WinGramm 1 y WinGramm 2 sobre un conjunto de secuencias aleatorias. Por ejemplo, en una computadora personal con procesador Intel Celeron, a 700 MHz, 184 MB RAM y Windows XP, WinGramm 1 tarda **3.1 minutos** en calcular la complejidad gramatical de una secuencia de 10000 símbolos, mientras que WinGramm 2 solo tarda **¡6.8 segundos!**.
- **Se ejecuta eficientemente.** Para algunas secuencias en particular, WinGramm 1 se cicla indefinidamente, teniendo que detener la instancia del programa desde el administrador de tareas de Windows. Para las misma secuencia WinGramm 2 se ejecuta sin ningún problema.
- **Se ejecuta en modo batch.** WinGramm 2 se ejecuta en modo batch (por lotes). El usuario seleccionar una secuencia, un conjunto o un grupo de secuencias, y entonces selecciona una opción para realizar alguna tarea sobre todas las secuencias seleccionadas.
- **Genera la matriz de distancias.** WinGramm 2 hace una matriz de distancias para algunas medidas. Por ejemplo, si se seleccionan 10 secuencias, y se elige la opción **Calc | Algorithmic Distance**, WinGramm 2 calcula la distancia algorítmica entre todos los pares que se pueden formar y genera la matriz de distancia algorítmica de 10x10 automáticamente.
- **Construye el árbol filogenético.** Con la matriz de distancias generada con WinGramm 2 o con algún programa externo, WinGramm 2 construye el árbol filogenético, mostrándolo en forma gráfica.
- **Portabilidad.** WinGramm 2.1 está programado como una aplicación CLX en Borland Delphi 7. Esto permite generar código binario ejecutable para Windows 95/98/NT/2000/Me/XP. Este tipo de aplicación Delphi es portable a Linux, por lo que se puede generar también código binario para Linux usando Borland Kylix 3, siendo la versión para Linux: LinGramm 2.1.

	Intel Celeron, 700 MHz, 184 MB RAM, Windows XP							Intel Pentium 4, 1.2 GHz, 512 MB RAM, Windows 2000		
WinGramm 1										
Longitud de secuencia	200	500	1000	2000	5000	10000	20000			
Complejidad gramatical	134	273	476	848	1820	3279	6004			
Tiempo requerido (Seconds)	0.64	1.32	2.18	6.79	43.53	187.42	1161.47			
Tiempo requerido (Minutes)	3.12367 19.3578									
WinGramm 2										
Longitud de secuencia	200	500	1000	2000	5000	10000	20000	50000	100000	200000
Complejidad gramatical	134	267	474	850	1810	3282	5971	13474	24909	46661
Tiempo requerido (Seconds)	0	0.01	0.04	0.18	1.201	6.83	44.284	385.705	1605.05	9900.75
Tiempo requerido (Minutes)								6.42842	26.7508	165.012

Tabla 3-1. Desempeño de WinGramm 1 vs. WinGramm 2

Dado que explicar detalladamente la funcionalidad de WinGramm 2 haría este trabajo muy extenso, en el Apéndice A se muestran únicamente algunas de las funciones del programa. Adicionalmente, del sitio Internet de WinGramm 2 se pueden obtener el manual, el programa y actualizaciones.

3.4.3 Actualizaciones en Internet

En este trabajo se liberó WinGramm 2.1.1.99. En Internet se pueden encontrar las actualizaciones del software en:

<http://www.geocities.com/jsandria> y

http://www.uv.mx/mia/Profesores/majm/DrJimenez_Montano.htm

Si por algún motivo, no tiene acceso a estos vínculos, puede buscar en Internet con Google, por ejemplo, las palabras: WinGramm Gramatical Complexity Analysis Tool.

En el sitio de WinGramm 2 puede encontrar el programa (para Windows y Linux), manuales y ejemplos.

3.5 LinGramm 2

LinGramm 2 es la versión para Linux de WinGramm 2. Tienen las mismas funciones y características.

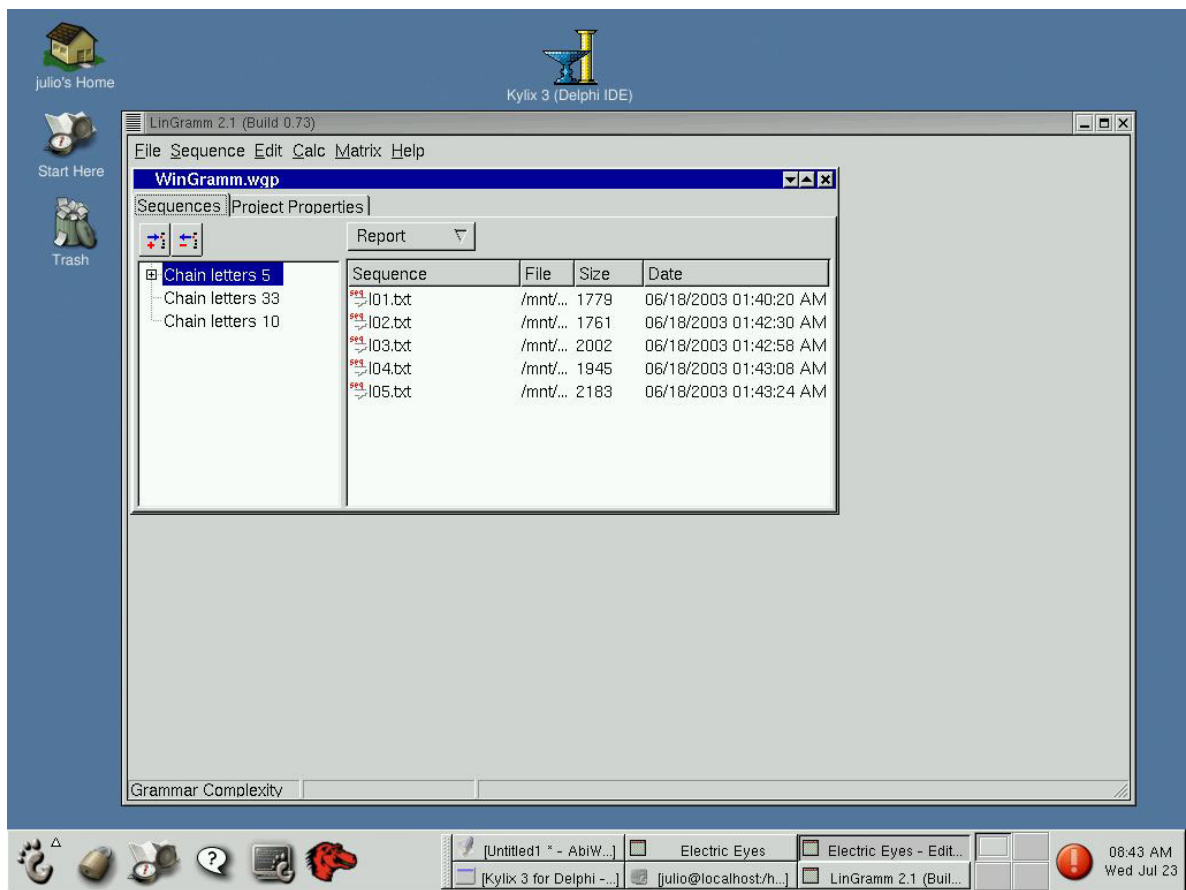


Figura 3-10. LinGramm 2.1 en un ambiente Linux Red Hat 7.3

4 Aplicaciones

En este capítulo se muestran ejemplos de análisis de secuencias, iniciando con un conjunto de cartas en cadena (secuencias de caracteres que forman palabras), secuencias de proteínas y de ADN.

4.1 Análisis de secuencias de cartas en cadena

4.1.1 Problema

En esta sección se analizan 33 versiones de una carta (Figura 4-1.a), que forman parte de una cadena, en la cual una persona recibió la carta, la fotocopió, o transcribió y la reenvió. De modo que estas cartas fueron pasadas de una persona a otra, mutando y evolucionando. Como un gen, su longitud promedio es de cerca de 2000 caracteres. Como un virus, las cartas amenazan con matar a la persona y la induce a pasarla a sus amigos y conocidos. Como una característica heredable, promete beneficios para la persona y gente a quienes la pasa. (Bennett, Li y Ma, 2003).

WITH LOVE ALL THINGS ARE POSSIBLE

This paper has been sent to you for good luck. The original copy is in New England. It has been around the world nine times. The luck has now been sent to you. You will receive good luck within four days of receiving this letter, providing, you in turn send it on. This is no joke. You will receive it in the mail. Send copies to people you think need good luck. Don't send money as fate has no price. Do not keep this letter. It must leave your hands within 96 hours. An RAF officer received \$70,000. Joe Elliot received \$40,000 and lost it because he broke the chain. While in the Philippines, Gene Walsh lost his wife six days after receiving the letter. He failed to circulate the letter. However, before her death he received \$7,755,000. Please send 20 copies of the letter and see what happens in four days. The chain comes from Venezuela and was written by Saul Anthony Decroup, a missionary from South America. Since the copy must make a tour around the world, you must make 20 copies and send them to friends and associates. After a few days you will get a surprise. This is true even if you aren't superstitious. Do note the following: Constantion Dias received the chain in 1953. He asked his secretary to make 20 copies and send them out. A few days later he won the lottery of two million dollars. Carle Dadditt, an office employee, received the letter and forgot it had to leave his hands within 96 hours. He lost his job. Later, after finding the letter again, he mailed out the 20 copies. A few days later he got a better job. Dalan Fairchild received the letter and not believing, threw the letter away. Nine days later he died. Remember, send no money, and please don't ignore this.

IT WORKS

with love all things are possible%

this paper has been sent to you for good luck. the original copy is in new england. it has been around the world nine times. the luck has now been sent to you. you will receive good luck within four days of receiving this letter, providing, you in turn send it on. this is no joke. you will receive it in the mail. send copies to people you think need good luck. don't send money as fate has no price. do not keep this letter. it must leave your hands within 96 hours. an raf officer received \$70,000. joe elliot received \$40,000 and lost it because he broke the chain. while in the philippines, gene walsh lost his wife six days after receiving the letter. he failed to circulate the letter. however, before her death he received \$7,755,000. please send 20 copies of the letter and see what happens in four days. the chain comes from venezuela and was written by saul anthony decroup, a missionary from south america. since the copy must make a tour around the world, you must make 20 copies and send them to friends and associates. after a few days you will get a surprise. this is true even if you aren't superstitious. do note the following: constantion dias received the chain in 1953. he asked his secretary to make 20 copies and send them out. a few days later he won the lottery of two million dollars. carle dadditt, an office employee, received the letter and forgot it had to leave his hands within 96 hours. he lost his job. later, after finding the letter again, he mailed out the 20 copies. a few days later he got a better job. dalan fairchild received the letter and not believing, threw the letter away. nine days later he died. remember, send no money, and please don't ignore this. it works

a) Carta original digitalizada

b) Carta transcrita modificada

Figura 4-1. Ejemplo de carta cadena

Las cartas son un intrigante fenómeno social, pero proporcionan una buena fuente de pruebas para los algoritmos usados en biología molecular para inferir árboles filogenéticos de los genomas de organismos existentes.

Bennett y colaboradores usaron GenCompress, un programa similar a WinGramm, que es lo suficientemente general para tener una amplia aplicabilidad en el análisis de secuencias, y construyeron la historia evolutiva de las 33 cartas. Para esto etiquetaron las cartas como L1 a L33. Éstas difieren significativamente. Hay 15 títulos, 23 nombres para “un empleado de oficina”, y 25 nombres para el autor original de la carta. Las cartas tienen errores ortográficos, frases faltantes o adicionales, así como oraciones y párrafos comunes. Para analizarlas, las transcribieron en archivos de computadora, completamente en minúsculas, ignorando información extra como fechas y notas marginales así como la división del texto en líneas y párrafos. Cada carta se convirtió en una cadena de caracteres continuos (Figura 4-1.b).

4.1.2 Material y métodos

Las 33 cartas se obtuvieron de internet⁵ como archivos de texto. Con WinGramm 2 se obtuvo la matriz de distancia algorítmica, parte de la cual se muestra en la Tabla 4-1.

D(x,y)	L01.txt	L02.txt	L03.txt	L04.txt	L05.txt	L06.txt	L07.txt	L08.txt	L09.txt	L10.txt	...	L33.txt
L01.txt	66	714	991	948	843	682	639	582	491	645	...	919
L02.txt	714	64	1009	972	951	634	647	646	699	647	...	953
L03.txt	991	1009	46	787	1250	1017	966	979	1014	994	...	936
L04.txt	948	972	787	46	1175	976	955	890	951	927	...	783
L05.txt	843	951	1250	1175	48	923	946	879	864	866	...	1132
L06.txt	682	634	1017	976	923	64	431	582	669	587	...	921
L07.txt	639	647	966	955	946	431	62	507	588	556	...	900
L08.txt	582	646	979	890	879	582	507	46	591	533	...	873
L09.txt	491	699	1014	951	864	669	588	591	58	606	...	918
...
L33.txt	919	953	936	783	1132	921	900	873	918	910	...	34

Tabla 4-1. Matriz de distancia algorítmica de las cartas en cadena

Con el mismo WinGramm 2 se construyó el árbol filogenético de la Figura 4-2 usando el método UPGMA.

⁵ <http://www.math.uwaterloo.ca/~mli/chain.html>

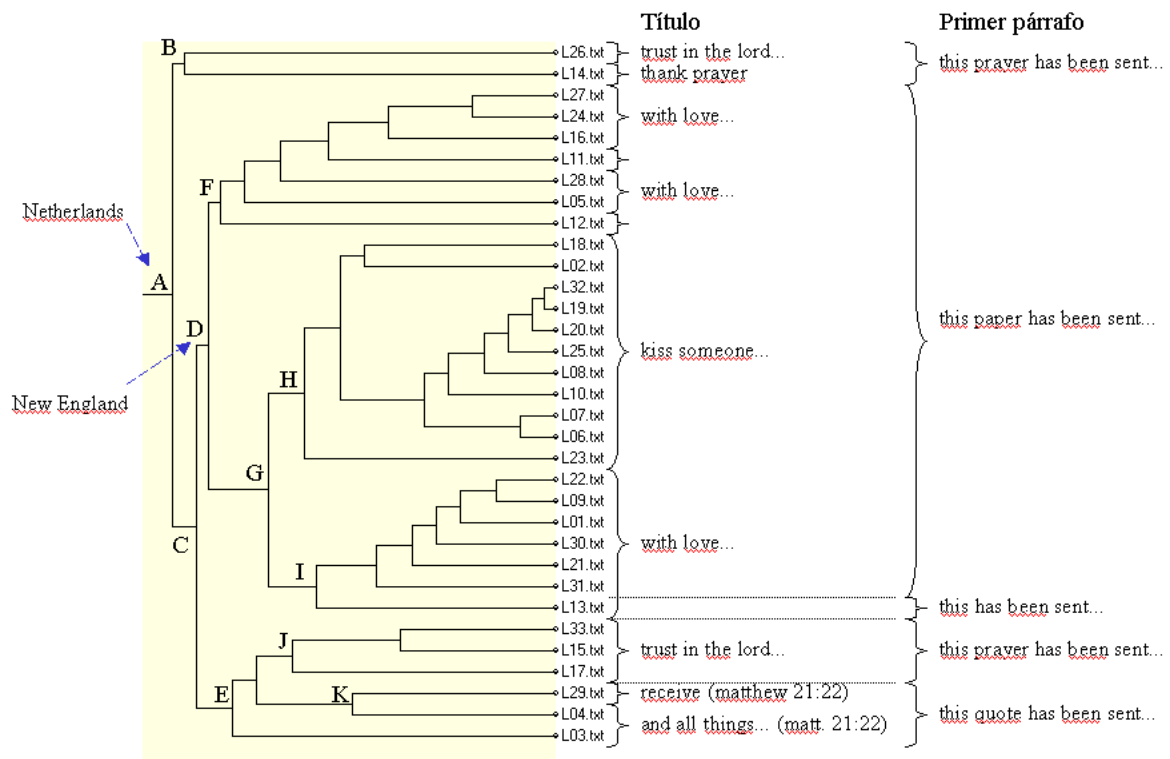


Figura 4-2. Árbol filogenético para las cartas en cadena

4.1.3 Resultados

El problema de las cartas en cadena no es un problema trivial, ya que requiere un algoritmo de compactación lo suficientemente bueno para encontrar repeticiones. Con WinGramm 2 se encuentra que las cartas más parecidas (las que tienen menor distancia algorítmica entre sí) son la L19 y L32. Leyendo ambas cartas se puede ver que tienen 10 diferencias (Figura 4-3), las cuales son errores de dedo, variaciones, abreviaciones, adiciones u omisiones de palabras. Los puntos suspensivos indican segmentos de caracteres iguales.

L19	...	gena	welsh...	...	sned...	...	aria	laddfto...	...	dagan...	works.	
L32	...	gene	Welch...	...	anthony...	...	send...	do...	arla	daddit...	again...	dala ...works!
Dif		1	2	3	4	5	6	7	8	9	10	

Figura 4-3. Diferencias entre las cartas más parecidas

El árbol evolutivo de las cartas en cadena mostrado en la Figura 4-2 tiene características interesantes. La carta inicial (punto A) de dicho conjunto indica que proviene de “netherlands”, el título dice “trust in the lord...” y el primer párrafo contiene “this prayer has been sent...”. En el punto D evolucionó el origen a “new england”, el título a “with love...” y parte del primer párrafo a “this paper has been sent...”. En el punto H evoluciona el título a “kiss someone...” y una carta pierde la frase “he failed to circulate the letter”. En el punto I una carta pierde la palabra “paper” (“this has been sent...”).

4.2 Análisis de secuencias de proteínas

4.2.1 Problema

Entender el comportamiento de las proteínas es sumamente complejo, por el número de propiedades e interacciones entre los aminoácidos que las componen. El alfabeto de las proteínas consta de 20 letras, que se pueden agrupar de 51×10^{12} maneras. Frecuentemente es necesario disminuir la cantidad de grupos mediante *alfabetos reducidos* que deben estar determinados por las propiedades fisicoquímicas de los aminoácidos, como la hidrofobicidad y el volumen, para no cambiar la estructura y por consiguiente la funcionalidad de las proteínas. Jiménez-Montaña, Del Angel, y Ramos (2003) aplicaron métodos de clustering del paquete STATISTICA a un conjunto de matrices obteniendo árboles como el de la Figura 4-4.

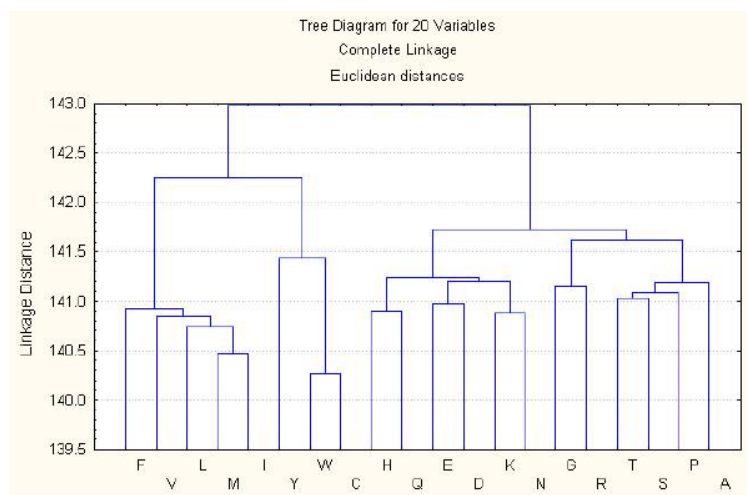


Figura 4-4. Árbol generado a partir de la matriz de Miyazawa-Jernigan

De la estructura de cada uno de los árboles definieron alfabetos reducidos de 4, 5, 9 y 12 letras, como se muestra en la tabla Figura 4-4.

4 letras

A	B	C	D
FVLM	IYWC	HQEDKN	GRTSPA

5 Letras

A	B	C	D	E
FVLM	IYWC	HQEDKN	GR	TSPA

9 Letras

A	B	C	D	E	F	G	H	I
F	V	L	M	I	Y	W	C	H
Q	E	D	K	N	G	R	T	S
P	A							

12 Letras

A	B	C	D	E	F	G	H	I	J	K	L
F	V	L	M	I	Y	C	W	H	Q	E	D
K	N	G	R	T	S	P	A				

Figura 4-5. Alfabetos reducidos para el árbol Miyazawa-Jernigan

Para comprobar si no cambia la estructura y por consiguiente la funcionalidad de las proteínas se usó WinGramm 2 para generar las matrices de distancias y árboles filogenéticos de las secuencias de mioglobina, con la idea de que los árboles formados deben mantener la misma estructura y agrupamientos. Los resultados se pueden ver en la Figura 4-6.

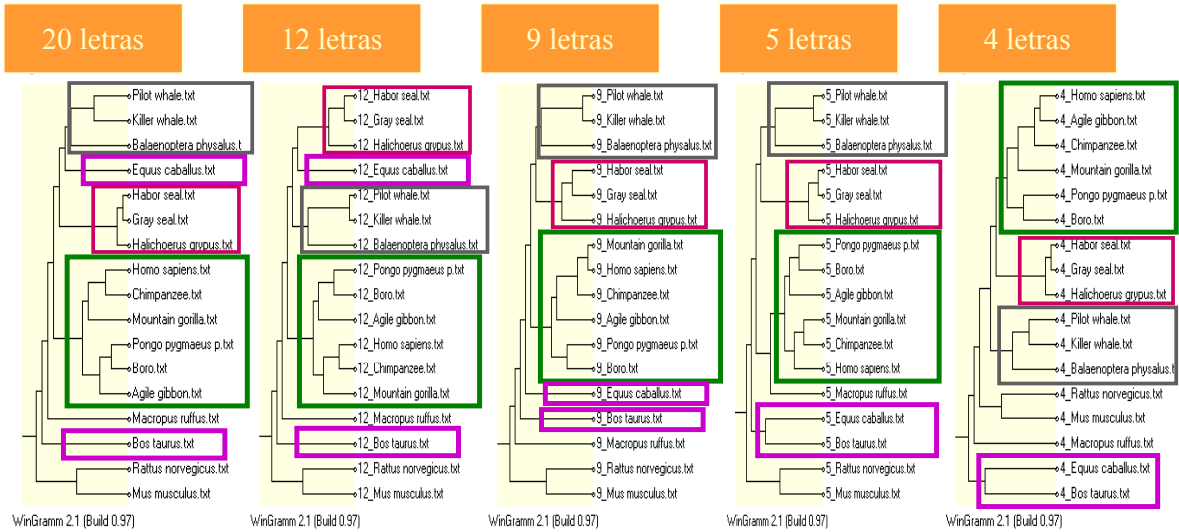


Figura 4-6. Árboles filogenéticos de las secuencias de mioglobina con los alfabetos reducidos de Miyazawa-Jernigan

Estos resultados muestran que los grupos principales no se pierden, por ejemplo los primates, las ballenas (wales) y focas continúan unidos desde el principio. Algo curioso que sucede es que en el alfabeto de 9 letras quedan cerca la vaca y el caballo, y para los alfabetos de 4 y 5 letras se agrupan.

4.3 Análisis de secuencias de ADN

4.3.1 Aplicación 1. Descubriendo secuencias simples de ADN

Muchas secuencias genómicas de ADN contienen patrones repetitivos. La detección de tales patrones es una parte importante del análisis de secuencias de ADN. En este análisis se usó la secuencia completa de ADN del gen HUMTPA, que contiene 36,594 bases. La Figura 4-7 muestra parte del sumario obtenido.

LOCUS	HUMTPA	36594 bp	DNA	linear	PRI 03-MAY-1996		
DEFINITION	Human tissue plasminogen activator (PLAT) gene, complete cds.						
ACCESSION	K03021						
VERSION	K03021.1 GI:339817						
...							
	1	ttcacacaac	tggtgctgtt	accaccatgg	gcgtctagtc	tggatcagtg	gtcctcagtc
	61	ttttttgcac	cagggaccag	ttttgtaaag	atagcttttc	cacggacaga	gggaggggag
	121	atagtttcgg	gatgattcaa	gaggattaca	tttattgtgc	actttattta	tattactatt
...							
	36421	ggctaatatg	gcatttagag	aagtaccaag	gtacagtgga	gccggtcaca	aaagggcaga

```

36481 cttgtagtag aattcagttg caagagggat tggggaatct taaggaaaa atagaatctt
36541 aaggaaaaaa taactgggtg agacgtggac tgtggacagg cgcggaag gcac

```

Figura 4-7. Parte del sumario del gen HUMTPA obtenido en GenBank

4.3.1.1 Material y métodos

De GenBank se obtuvo el gen HUMTPA en formato FASTA (Figura 4-8) y se guardó como un archivo de texto sin la primera línea descriptiva y sin caracteres de retorno de carro (carácter con código ascii 13) y avance de línea (carácter con código ascii 10).

```

>gi|339817|gb|K03021.1|HUMTPA Human tissue plasminogen activator (PLAT) gene, complete cds
TTCACACAACCTGGTGTCTTACCACCATGGGCGTCTAGTCTGGATCAGTGGTCCTCAGTCTTTTTGCAC
CAGGGACCAGTTTGTAAAGATAGCTTTCCACGGACAGAGGGAGGGGAGATAGTTTCGGGATGATTCAA
GAGGATTACATTTATTGTGCACTTTATTTATATTACTATTACATTGTATTATATAATGAAATAATGGTAT
...
ACAGAATGCTTGGTCTGATGGGCTAATATGGCATTAGAGAAGTACCAAGGTACAGTGGAGCCGGTCACA
AAAGGGCAGACTTGTAGTAGAATTGAGTTGCAAGAGGGATTGGGGAATCTTAAGGAAAAAATAGAATCTT
AAGGAAAAAATAACTGGGTGAGACGTGGACTGTGGACAGGCGCGGAAAAGGCAC

```

Figura 4-8. Parte del gen HUMTPA en formato FASTA

Con WinGramm 2 (Figura 4-9) se fraccionó la secuencia en subsecuencias de 128 bases (window=128) que se traslapan en 64 bases (overlap=64), formando 571 subsecuencias, 570 de longitud 128 y la última de longitud 114.

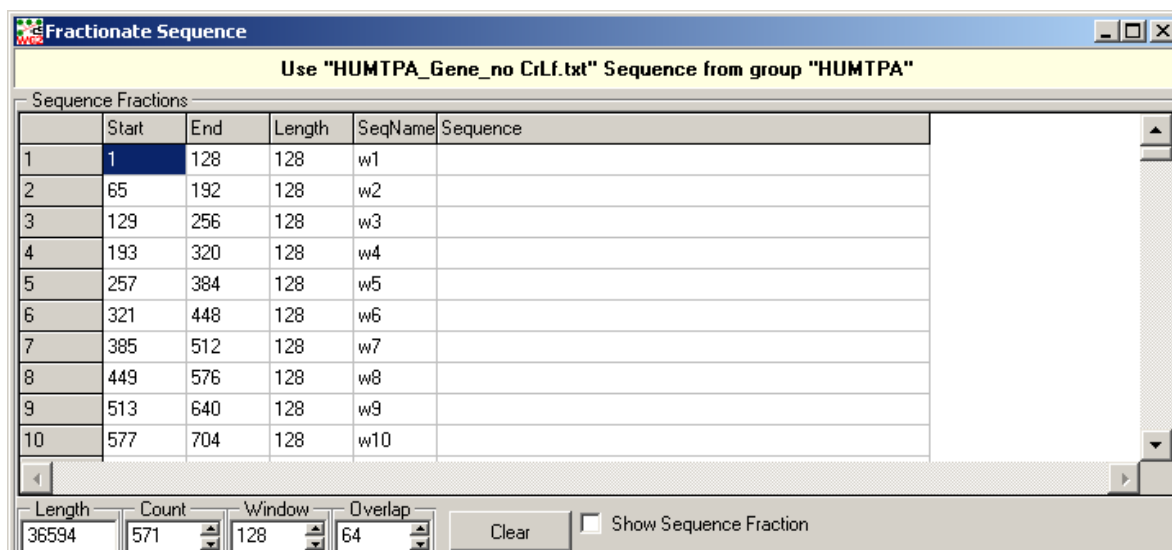


Figura 4-9. Fraccionando secuencia del gen HUMTPA

A las 571 secuencias se les calculó la redundancia algorítmica con 10 subrogados mediante la opción **Calc | Surrogate Statistics** de WinGramm 2. De los 571 resultados se muestran los 20 con redundancia algorítmica más alta en la Tabla 4-2.

4.3.1.2 Resultados

Analizando los mejores veinte resultados, se puede ver que las primeras nueve secuencias son consecutivas (w374 a w382), se encuentran entre las posiciones 23873 a 24512 del gen

HUMTPA y tienen el mismo triplete común (**GAT**). Estas secuencias coinciden con la secuencia 9 (23888-24458) de la tabla 1.1 de Milosavljevic, quien reporta como más repetida a TGATAGA, mientras que WinGramm 2 encuentra de la misma longitud a GATAGAT (w374, w374, w376, 380, 381 y 382) y GATGATA (377, 378 y 379).

	Secuencia	Inicia	Termina	<i>R</i>	Repeticiones (frecuencia)
1	W379	24193	24320	0.4275	AT (31), GAT (25), AGAT (13), AGATGAT (10), AC (6), ATAGATGAT (5), AG (3), ACATAGATGAT (3), AAATAGATGAT (2), ACATAGATGATAGAC (2)
2	W378	24129	24256	0.3835	GA (30), GAT (25), AGAT (14), AGATGAT (9), GAC (4), AGATGATA (4), TAGATGATA (3), TAGATGATAGAC (3), ACATAGATGATAGAC (2)
3	w380	24257	24384	0.3710	AT (32), GAT (27), GATA (16), GATGATA (7), ATA (4), GATT (3), ATAGATGATA (3), GGT (2), GATTGATAGATGATA (2)
4	w381	24321	24448	0.3536	GA (31), GAT (26), GATA (16), TGATA (6), GATGATA (6), TA (3), GATTGATA (3), GATTGATAGATGATA (3), GGTGATA (2)
5	w377	24065	24192	0.3342	GA (29), GAT (27), AGAT (15), AGATGAT (8), AGATGATAGAT (4), GG (3), AGA (3), GGT (3), ATAGATGATAGAT (2)
6	w374	23873	24000	0.3227	AT (30), GAT (24), GATA (14), GATAGAT (7), ATA (5), CA (3), GATAG (3), TGATAGAT (3), GATAGGTGATAGAT (2)
7	w375	23937	24064	0.3082	GA (28), GAT (24), GATA (14), GATAGAT (6), TA (5), CA (4), GG (3), GATAGGT (2), GATAAATA (2)
8	w376	24001	24128	0.2871	GA (29), GAT (25), GATA (14), TA (5), GATAGAT (5), GG (3), ATA (3), GAC (3), GATAAATA (2)
9	w382	24385	24512	0.2313	GA (34), GAT (15), GATA (9), GAGA (7), GT (6), AA (4), GATGATA (4)
10	w265	16897	17024	0.2164	CA (23), CT (19), CACA (9), CTCT (7), GG (6), AA (3), GT (3), CACACACA (3), CTCTCTCTCTCT (2)
11	w264	16833	16960	0.1986	AC (20), CT (14), AG (10), ACAC (8), GAG (5), CTCT (5), ACACACAC (4), GAGG (3), AT (3), TG (3)
12	w112	7105	7232	0.1885	TA (20), TG (18), TGTG (8), TT (6), CA (6), TATA (6), CT (4), GA (3), TAG (3), TGTGTGTG (3), TATATATA (3)
13	w113	7169	7296	0.1696	TG (21), TA (15), TGTG (8), CA (7), TATA (6), TGTGTGTG (3), TATATATA (3), GA (3), GG (3), TC (3), TTAA (2), TCAC (2)
14	w413	26369	26496	0.1662	AA (17), TT (13), TC (9), TG (7), CTT (5), GA (5), AATG (5), AAAAA (5), CTTTT (3), CTC (3), AATGGG (2), AAAAAAAAAAAAA (2)
15	w218	13889	14016	0.1246	CA (18); CT (12); CAG (8), GCT (6), AT (5), CAGG (4), CACA (4), TG (4), GCTG (4), CCT (3), CCG (2), CCTACACA (2)
16	w267	17025	17152	0.1222	AA (20), TG (10), AG (10), TC (6), AAAAA (6), CC (4), AAG (4), TGG (3), CAG (3), ATG (3), AAAAAAAAA (3), AAGAA (3), AAGAAAG (3)
17	w311	19841	19968	0.1165	CA (16), TG (12), GG (9), CC (8), CACC (5), GGG (4), TC (4), CAGGG (3), TGCC (3), TTC (3), CAG (3), CACCCACCACCTGCC (2)
18	w338	21569	21696	0.1140	AT (18), AAT (11), AG (8), AAAT (7), CT (6), GAG (4), GC (4), TT (4), ATTT (3), ATTTT (3), AATAAATAAAT (2), ACTC (2), GGAGT (2)
19	w16	961	1088	0.1109	AA (17), CT (13), GA (7), CA (7), AAAAA (7), GC (5), CTCT (4), CCA (3), GT (3), TT (3), AAAAAAAAAAAAA (2)
20	w148	9409	9536	0.1045	TG (15), CA (10), AA (8), CTG (7), CT (5), CAG (5), TGG (4), AG (4), TT (3), TCTCAGCTG (2), CAGATTAA (2)

Tabla 4-2. Subsecuencias (w=128, o=64) del gen HUMTPA con mayor Redundancia Algorítmica *R*, con sus correspondientes repeticiones y frecuencias.

Milosavljevic (1999) realizó este estudio usando su método de significancia algorítmica, que está basado en la compresión de secuencias para descubrir patrones. La Tabla 4-3 muestra las unidades repetidas obtenidas por Milosavljevic y las obtenidas con WinGramm 2 para las 20 secuencias de longitud 128 con menor *redundancia algorítmica*.

	Posición en HUMTPA Milosavljevic	Longitud	Unidades repetidas Milosavljevic	unidades repetidas WinGramm 2	Secuencias con mayor R, WinGramm 2 (Posición en HUMTPA)	Longitud
1	1017-1048	32	A			
				TA, TG	w112 (7105-7232)	128
2	7169-7296	128	GT, AT	TG, TA	w113 (7169-7296)	128
3	10497-10528	32	T			
				AC, CT	w264 (16833-16960)	128
4	16897-17024	128	AC, TC	CA, CT	w265 (16897-17024)	128
				AA, TG, AG	w267 (17025- 17152)	128
5	17089-17216	128	A, AAG	AA, AAG	w268 (17089- 17216)	128
6	19153-19192	40	A			
7	21241-21272	32	A, GAAAA			
8	21561-51592	32	TAA, TAAA	AAT, AAAT	w338 (21569- 21696)	128
9	23888-24458	571	TGATAGA	GAT, GATAGAT, GATGATA	w374 a w382 (23873-24512)	640
10	26457-26496	40	A	AA	w413 (26369-26496)	128
11	29073-29112	40	A			
				TG, GG	w464 (29633- 29760)	128
				CC, CA	w490 (31297- 31424)	128
				AA, GG	w571 (36481- 36594)	128

Tabla 4-3. Comparación de resultados de WinGramm 2 con Milosavljevic (1999)

4.3.2 Aplicación 2. Descubriendo similitud

Un patrón repetitivo común puede causar que dos secuencias parezcan similares aunque no estén relacionadas. Por ejemplo, dos secuencias de ADN que tengan tramos con múltiples bases A (AAAAA) podrían parecer similares.

En esta aplicación, se calcula la distancia algorítmica para encontrar similitud entre regiones del segmento 22,001-26,000 del gen HUMTPA.

4.3.2.1 Material y métodos

Con WinGramm 2 se fraccionó la secuencia del gen HUMTPA en subsecuencias de 200 bases (window=200) que se traslapan en 100 bases (overlap=100), entre las posiciones 22,001 y 26,000, formando 39 secuencias de longitud 200. Se hizo lo siguiente:

Experimento 1. A las 39 secuencias se les calculó la distancia algorítmica mediante la opción Calc | Algorithmic Distance de WinGramm 2, obteniendo una matriz de distancias y una lista de distancias (Tabla 4-4).

Experimento 2. A las 39 secuencias se les calculó la Complejidad Gramatical, concatenando con cada secuencia un corpus Alu, obteniendo una lista de resultados de Complejidad Gramatical (Tabla 4-5).

4.3.2.2 Resultados

Experimento 1. Excluyendo las distancias entre las mismas secuencias y las secuencias que se traslapan, se muestran los 10 resultados con menor distancia algorítmica $D(x,y)$ en la Tabla 4-4. A menor distancia algorítmica, mayor similitud entre las secuencias.

Par	D(x,y)	Ventana x	Ventana y	x	y
1	89	1901-2100	2101-2300	w20	w22
2	91	1901-2100	2201-2400	w20	w23
3	91	1901-2100	2301-2500	w20	w24
4	93	2001-2200	2201-2400	w21	w23
5	96	2101-2300	2301-2500	w22	w24
6	105	2001-2200	2301-2500	w21	w24
7	111	1801-2000	2101-2300	w19	w22
8	114	1801-2000	2001-2200	w19	w21
9	119	1801-2000	2201-2400	w19	w23
10	121	1801-2000	2301-2500	w19	w24

Tabla 4-4. Distancia algorítmica de 10 fragmentos del segmento 22,001-36,000 del gen HUMTPA

Milosavljevic (1999) realizó este estudio calculando información mutua, que permite encontrar similitud debido a las estructuras internas de las secuencias. En esta aplicación únicamente se encuentra como resultado común con Milosavljevic el par 2: 1901-2100 y 2201-2400.

Experimento 2. La Tabla 4-5 muestra los 10 mejores resultados de complejidad gramatical. Se puede observar que los mejores 9 resultados corresponden a secuencias del corpus Alu. Las secuencias w05, w04, w03, w06 y w02 que son consecutivas (w02 a w06) corresponden a la secuencia Alu 22253-22545, y las secuencias w37, w38, w36 y w39 corresponden a la secuencia Alu 25620-25911..

	Secuencia	K	Posición en Segmento	Posición en HUMTPA	Secuencia Alu correspondiente
1	w37	1991	3601-3800	25601-25800	Alu 25620-25911
2	w05	2000	401-600	22401-22600	Alu 22253-22545
3	w04	2009	301-500	22301-22500	Alu 22253-22545
4	w38	2009	3701-3900	25701-25900	Alu 25620-25911
5	w03	2020	201-400	22201-22400	Alu 22253-22545
6	w06	2024	501-700	22501-22700	Alu 22253-22545
7	w36	2025	3501-3700	25501-25700	Alu 25620-25911
8	w39	2027	3801-4000	25801-26000	Alu 25620-25911
9	w02	2031	101-300	22101-22300	Alu 22253-22545
10	w22	2032	2101-2300	24101-24300	

Tabla 4-5. Complejidad gramatical de 10 secuencias del segmento 22001-26000, cada secuencia está concatenada con el corpus Alu.

5 Conclusiones

En este trabajo se desarrolló una herramienta de software que explota el algoritmo *Grammar*, para el cálculo de la complejidad gramatical. Esta herramienta es el programa WinGramm 2.1.1 para Windows y LinGramm 2.1.1 para Linux. Ambos son el mismo programa, solo que compilados para diferentes plataformas.

WinGramm 2.1.1 es una herramienta de software para el análisis de secuencias de cualquier tipo, principalmente biosecuencias, que permite:

- Descubrir secuencias simples
- Descubrir similitudes en una secuencia
- Establecer la historia evolutiva de un conjunto de secuencias
- Identificar si una secuencia pertenece a un conjunto de secuencias
- Estudiar los efectos de los alfabetos reducidos en las secuencias
- Entre otros.

Cabe mencionar que el análisis de secuencias biológicas es un área muy explotada, pero que cada día tiene nuevas secuencias adicionales, lo que permite a gente del área de las ciencias de la computación, bioinformática o biología molecular, la posibilidad de encontrar más campos de especialización y de trabajo.

WinGramm 2 ha sido desarrollado con la idea de facilitar el trabajo de investigación en el análisis de secuencias. En gran parte se ha logrado esto; comparado con la versión anterior de WinGramm (la 1.7) es mucho más rápido, más eficiente, más automatizado, tiene funcionalidades adicionales que liberan al usuario de estar “pegado” a la computadora, como la recolección automática de resultados, que permite generar matrices de distancias. Construye árboles filogenéticos en modo gráfico y es compatible con Windows y Linux.

El desarrollo del software se ha hecho siguiendo en lo posible el modelo de ensamblaje de componentes, situación que en parte es obligada al usar una herramienta de programación visual como Delphi, ya que está basado en seleccionar componentes, arrastrarlos y usarlos. Para la mayoría de los módulos hubo una planeación, con la idea de hacer crecer el programa de manera controlada, en muchos casos no fue necesario un análisis de riesgos, porque ya estaba claro qué funcionalidades eran necesarias incorporar al programa en cada ciclo de desarrollo. Durante la ingeniería del producto se identificaban los componentes que podrían ser usados, en varios casos fue necesario crear componentes para uso exclusivo del programa que han sido reutilizados en varias unidades de la aplicación. La fase de evaluación del cliente fue muy importante, porque algunos usuarios, dieron sus opiniones y críticas que permitieron mejorar bastante el programa.

Los análisis y experimentos reportados en este trabajo se hicieron en su mayoría con WinGramm 2.1. El hacer estas pruebas, permitió encontrar y corregir muchos errores y omisiones del programa.

El autor de este trabajo ha preparado también un sitio web desde el cual se podrán obtener las actualizaciones y últimas versiones de WinGramm 2. Para el autor es importante destacar que este trabajo se concibió como un proyecto a largo plazo.

Trabajo Futuro

Al final del ciclo de desarrollo de la versión de WinGramm 2.1.1 se hizo patente la necesidad de hacerle cambios y mejoras, como los siguientes:

1. Cuando se agregan más de 500 secuencias a un grupo en la ventana Project, el programa se tarda asignando los recursos de memoria para hacer referencia a tales secuencias. Sería mejor usar otro componente para eso.
2. El análisis de secuencias de ADN y proteínas se puede automatizar aún más. Haciendo que el programa pueda excluir la primera línea de los archivos en formato FASTA, para no hacerlo manualmente.
3. Obtener secuencias de GenBank se podría automatizar, de modo que si se cuenta con el número de referencia, se use tal número como una referencia a una secuencia dentro de un proyecto WinGramm 2, para que WinGramm 2 se conecte al servidor de GenBank, obtenga automáticamente la secuencia en formato FASTA y la procese de acuerdo lo que haya indicado el usuario.
4. El procesamiento de varias secuencias sigue siendo tardado. Esta tarea se podría trabajar como un sistema distribuido.

Referencias bibliográficas

- Baldi, Pierre and Soren Brunak, 2001. *Bioinformatics. The Machine Learning Approach*. The MIT Press, London.
- Bennett, H. Charles, Ming Li, Bin Ma, 2003. *Chain Letters & Evolutionary Histories*. Scientific American. Pp. 64-69. June.
- Ebeling, Werner and Miguel A. Jiménez-Montaña, 1980. *On grammars, complexity, and information measures of biological macromolecules*, Mathematical Biosciences 52, 53-71.
- Cao, Ying, Axel Janke, Peter J. Waddell, Michael Sesterman, Osamu Takenaka, Shigenori Murata, Norihiro Okada, Svante Pääbo, Masami Hasegawa, 1998. *Conflict Among Individual Mitochondrial Proteins in Resolving the Phylogeny of Eutherian Orders*, Journal of Molecular Evolution, 47:308-322.
- Dubois, Paul F., Thomas Epperly and Gary Kumpf, 2003. *Why Johnny Can't Build*. Computing in Science & Engineering, IEEE CS and the AIP, September/Octubre, pp. 83-88.
- Fu, King-Sun and L. Booth Taylor, 1975a. *Grammatical Inference: Introduction and Survey-Part I*. IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-5, No. 1, pp 95-111.
- Fu, King-Sun and L. Booth Taylor, 1975b. *Grammatical Inference: Introduction and Survey-Part II*. IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-5, No. 5, pp. 409-423.
- Hatton, Les and Andy Roberts, 1994. *How Accurate Is Scientific Software?* IEEE Transactions on Software Engineering, Vol. 20, No. 10, pp. 785-797.
- Hendrick, Philip W. 2000. *Genetics of Populations*. Second Edition. Jones and Bartlett Publishers, Inc.
- Hopcroft, John E. y Jeffrey D. Ullman, 1996. *Introducción a la teoría de autómatas, lenguajes y computación*, CECSA, Segunda reimpresión en español.
- Jagota, Arun, 2000. *Data Analysis and Classification for Bioinformatics*. Department of Computer Science, University of California, Santa Cruz.

- Jiménez-Montaño, Miguel A, 1984. *On the syntactic structure of protein sequences and the concept of grammar complexity*, Bulletin of Mathematical Biology, Vol. 46, No. 4, pp. 641-659.
- Jiménez-Montaño, Miguel A., Rainer Feistel and Oscar Diez-Martínez, 2003. On the information hidden in signals and macromolecules I. Symbolic time-series analysis. (In Press).
- Jiménez-Montaño, Miguel A., Del Angel Ortiz, Rusalky y Ramos Fernández Antero, 2003. *Alfabetos Reducidos para la Compactación de Secuencias de Proteínas Empleando Métodos de Minería de Datos*. (En prensa).
- Lewis, Harry R. and Christos H. Papadimitriou, 1981. *Elements of the Theory of Computation*, Prentice-Hall.
- Milosavljevic, Aleksandar, 1999. *Discovering Patterns in DNA Sequences by the Algorithmic Significance Method*, in Pattern Discovery in Biomolecular Data: Tools, Techniques, and Applications. Wang, J.T.L., Shapiro, B.A., and Shasha, D. (editors), Oxford University Press, Oxford.
- Miyamoto, M.M and J. Cracraft, 1991. *Phylogenetic Analysis of DNA Sequences*. Oxford University Press, New York.
- Orcero, David Santo, 2001. *The Crisis of Free Scientific Software*. UPGRADE, Vol. II, No. 6, <http://www.upgrade-cepis.org>.
- Ouellette, B.F. Francis, 1998. *The GenBank Sequence Database*, in Bioinformatics A Practical Guide to the Analysis of Genes and Proteins, Baxevanis, Andreas D. and Ouellette, B.F. Francis (editors), John Wiley & Sons.
- Parekh, Rajesh and Vasant Honavar, 2000. *Grammar Inference, Automata Induction, and Language Acquisition*, in The Handbook of Natural Language Processing. Dale, Moisl, and Somers (editors).
- Pressman, Roger S., 2001. *Ingeniería del Software. Un enfoque práctico*, McGraw-Hill, Quinta edición, Adaptado por Darrel Ince.
- Swain, W.T., 2001. *Application of Software Engineering Practices in Computational Science*. <http://www.csm.ornl.gov/~bbd/docs/SEinCS.pdf>, Paper prepared for the UT/ORNL Science Alliance.
- Shannon, Claude E. 1948. *A Mathematical Theory of Communication*. The Bell System Technical Journal, Vol. 27, pp. 379-423, 623-656, July, October.
- Shepherd, Ellen, 1999. *Testing Existing Scientific Software*. Information Systems Engineering Center, Sandia National Laboratories.

Zamora Cortina, Luis y Julio César Sandria Reynoso, 2000, *Sobre la caracterización de poblaciones de alumnos mediante la complejidad gramatical*, Memorias del Cuarto Foro de Evaluación Educativa. Centro Nacional de Evaluación para la Educación Superior, A.C. (CENEVAL). México. Pp. 179-183.

Apéndice A. Descripción de WinGramm 2.1

Se describe una parte del funcionamiento de WinGramm 2 (versión 2.1), mostrando la interfaz gráfica y el código que se ejecuta.

A.1 Características

WinGramm 2.1 tiene programado las siguientes características:

File

- **New Project.** Crear nuevo proyecto WinGramm 2.
- **Open Project.** Abrir proyecto WinGramm 2 de disco.
- **Save Project.** Guardar proyecto WinGramm 2 a disco.
- **Save Project As.** Guardar proyecto WinGramm 2 a disco con nuevo nombre.

Sequence (para ventana Project)

- **Generate.** Generar una secuencia aleatoria
- **Open.** Abrir secuencia de disco y agregar a un grupo del proyecto en uso.
- **Add to Project.** Agregar secuencia sin tener que abrirla.

Sequence (para ventana Sequence)

- **Close.** Cerrar ventana Sequence.
- **Save.** Guardar secuencia a disco.
- **Save as.** Guardar secuencia a disco con nuevo nombre.

Edit (para ventana Project)

- **Fractionate Sequence.** Fraccionar secuencia(s) seleccionada(s).
- **Change Alphabet.** Cambiar alfabeto a secuencia seleccionada.
- **Make Surrogate Files.** Crear archivos subrogados a partir de secuencias seleccionadas.

Edit (para ventana Sequence)

- **Cut.** Cortar el texto seleccionado.
- **Copy.** Copiar el texto seleccionado.
- **Paste.** Pegar texto del portapapeles.
- **Delete.** Borrar el texto seleccionado.
- **Select all.** Seleccionar todo el texto.
- **Word Wrap.** Separar palabras al final de línea.
- **Font.** Cambiar tipo de letra.

Calc

- **Entropy.** Calcular entropía de las secuencias seleccionadas.
- **Grammatical Complexity.** Calcular complejidad gramatical de las secuencias seleccionadas.
- **Algorithmic Distance.** Calcular distancia algorítmica entre las secuencias seleccionadas, distancia relativa y factor de similitud.

- **Surrogate Statistics.** Calcular redundancia algorítmica de las secuencias seleccionadas.
- **Symbol Statistics.** Calcular frecuencia de símbolos de las secuencia seleccionada.

Matrix (para ventana Project)

- **New Matrix Window.** Crear nueva matriz en ventana Matriz.
- **Open from Text File.** Abrir matriz de archivo de texto.

Matrix (para ventana Matrix)

- **Paste from Clipboard.** Pegar matriz del portapapeles.
- **Paste Demo.** Pegar matriz ejemplo.
- **Save to Text File.** Guardar matriz a un archivo de texto.
- **Build Phylogenetic.** Construir árbol filogenético.

Help

- **About.** Cuadro de diálogo Acerca del programa WinGramm 2.1.

A.2 Ventanas principal y Project

WinGramm 2 y (LinGramm 2) es una aplicación CLX desarrollada en Delphi 7 (para Windows y Linux respectivamente) que consiste del programa principal (program WinGramm) y varias unidades (uses ...) como se muestra en el siguiente listado fuente:

```
program WinGramm;

{%ToDo 'WinGramm.todo'}

uses
  QForms,
  MainForm in 'MainForm.pas' {FormMain},
  GenerateSequenceForm in 'GenerateSequenceForm.pas' {FormGenerateSequence},
  ProjectForm in 'ProjectForm.pas' {FormProject},
  SequenceForm in 'SequenceForm.pas' {FormSequence},
  ResultsForm in 'ResultsForm.pas' {FormResults},
  MatrixForm in 'MatrixForm.pas' {FormMatrix},
  WorkingForm in 'WorkingForm.pas' {FormWorking},
  CalcForm in 'CalcForm.pas' {FormCalc},
  ChangeAlphabetForm in 'ChangeAlphabetForm.pas' {FormChangeAlphabet},
  Globals in 'Globals.pas',
  FractionateSequenceForm in 'FractionateSequenceForm.pas' {FormFractionateSequence},
  GenerateSurrogatesForm in 'GenerateSurrogatesForm.pas' {FormGenerateSurrogates},
  SelectSequenceForm in 'SelectSequenceForm.pas' {FormSelectSequence};

{$R *.res}

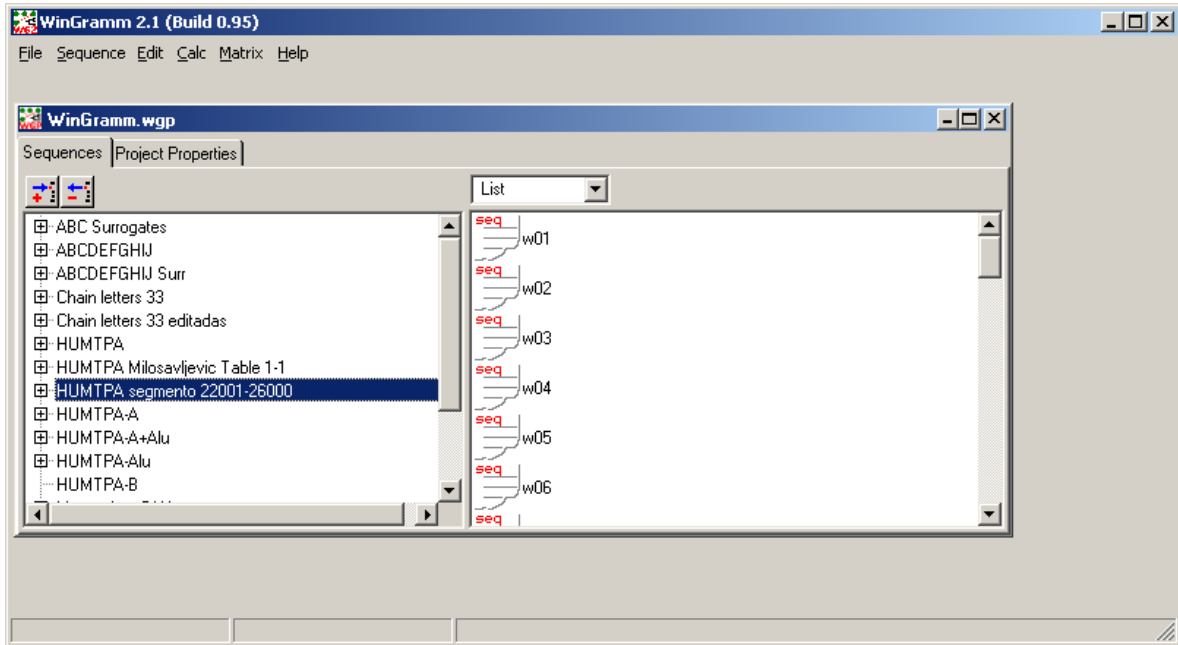
begin
  Application.Initialize;
  Application.CreateForm(TFormMain, FormMain);
  Application.CreateForm(TFormGenerateSequence, FormGenerateSequence);
  Application.CreateForm(TFormProject, FormProject);
  Application.CreateForm(TFormWorking, FormWorking);
  Application.CreateForm(TFormCalc, FormCalc);
  Application.CreateForm(TFormChangeAlphabet, FormChangeAlphabet);
  Application.CreateForm(TFormFractionateSequence, FormFractionateSequence);
  Application.CreateForm(TFormGenerateSurrogates, FormGenerateSurrogates);
```

```

Application.CreateForm(TFormSelectSequence, FormSelectSequence);
Application.Run;
end.

```

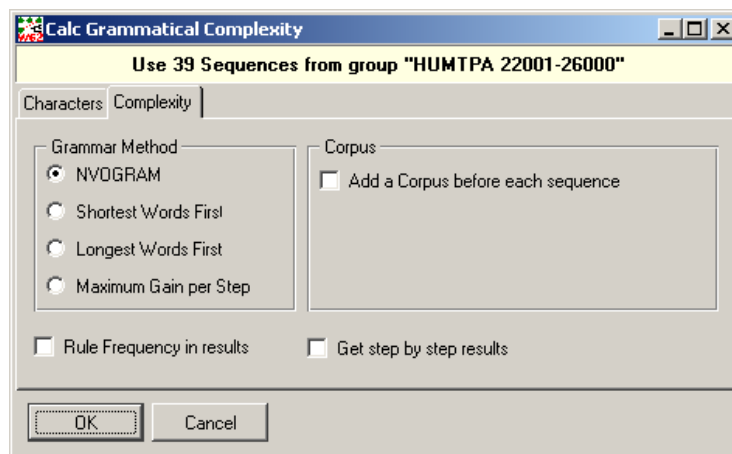
WinGramm 2 es una aplicación con una interfaz gráfica basada en menús, como lo muestra la siguiente figura:



La ventana principal de la aplicación WinGramm 2 tiene una ventana hija llamada **Project**, en la que se agrupan las secuencias a analizar.

A.3 Cálculo de la complejidad gramatical

Para calcular la complejidad gramatical de una secuencia, conjunto o grupo de secuencias, se selecciona únicamente la secuencia deseada, un conjunto de secuencias (en el panel derecho de la ventana Project) o un grupo (en el panel izquierdo de la ventana Project) y después se selecciona la opción **Calc | Grammatical Complexity** del menú principal. Esto ejecuta el procedimiento `TformMain.GrammaticalComplexityClick`, el cual abre el cuadro de diálogo **Calc**:



Cuando en este cuadro de diálogo se oprime el botón **OK**, del listado fuente mostrado a continuación (procedimiento TFormMain.GrammaticalComplexityClick) se ejecuta el código siguiente a la línea `if not (formCalc.ShowModal = mrOK) then Exit;`. Si se oprime el botón **Cancel** se termina la ejecución del procedimiento (then Exit).

```
procedure TFormMain.GrammaticalComplexity1Click(Sender: TObject);
var
  tsFileNameToCalc: TStrings;
  tsGrammaticalComplexity: TStrings;
  tsTimeNeeded: TStrings;
  tsLength: TStrings;
  tsAlphabetLength: TStrings;
  tsSequence: TStrings;

  i: integer;
  K: integer;
  Grammar: TGrammar;
  Sequence: TSequence;
  sResult, sResultCorpus, sCorpus: String;
begin
  if FormProject.getSelectedFiles.Count = 0 then
  begin
    ShowMessage('You must select at least one sequence');
    Exit;
  end;

  formCalc.MakeVisible(formCalc.tsGrammar);
  formCalc.Caption := 'Calc Grammatical Complexity';
  if not (formCalc.ShowModal = mrOK) then Exit;

  tsFileNameToCalc := TStringList.Create; { construct the list object }
  tsGrammaticalComplexity := TStringList.Create; { construct the list object }
  tsTimeNeeded := TStringList.Create; { construct the list object }
  tsLength := TStringList.Create; { construct the list object }
  tsAlphabetLength := TStringList.Create; { construct the list object }
  tsSequence := TStringList.Create; { construct the list object }

  tsFileNameToCalc.Clear;
  tsGrammaticalComplexity.Clear;
  tsTimeNeeded.Clear;
  tsLength.Clear;
  tsAlphabetLength.Clear;
  tsSequence.Clear;
```

```

tsFileNameToCalc := FormProject.getSelectedFiles;

FormWorking.Show;
FormWorking.bContinue := True;
FormWorking.Memol.Text := '';
FormWorking.ProgressBar1.Min := 0;
FormWorking.ProgressBar1.Max := tsFileNameToCalc.Count;
FormWorking.Refresh;

{ Instantiate the TSequence object }
Sequence := TSequence.Create;
Sequence.setCharacters(FormCalc.getInvalidCharacters','',', ' ');

sCorpus := ''; sResultCorpus := CrLf;
{ If user selected Corpus then fill sCorpus }
if FormCalc.cbAddCorpus.Checked then
begin
    Sequence.LoadFromFile(FormSelectSequence.SelectedFiles[0]);
    sResultCorpus := sResultCorpus + 'Corpus Sequence (CS):' + CrLf +
        FormSelectSequence.SelectedFiles[0];
    for i:=1 to FormSelectSequence.SelectedFiles.Count-1 do
    begin
        Sequence.AppendFromFile(FormSelectSequence.SelectedFiles[i]);
        sResultCorpus := sResultCorpus + CrLf +
            '+ ' + FormSelectSequence.SelectedFiles[i];
    end;
    sCorpus := Sequence.ToStringOriginal;
    sResultCorpus := sResultCorpus + CrLf + CrLf;
end;

sResult := '';
for i:=0 to tsFileNameToCalc.Count - 1 do
begin
    FormWorking.Memol.Lines.Add( 'Processing ' +
        ExtractFileName( tsFileNameToCalc.Strings[i] ) );
    FormWorking.Memol.Refresh;
    FormWorking.ProgressBar1.Position := i;

    if FormCalc.cbAddCorpus.Checked then
    begin
        Sequence.LoadFromString(sCorpus);
        Sequence.AppendFromFile(tsFileNameToCalc.Strings[i])
    end
    else
        Sequence.LoadFromFile(tsFileNameToCalc.Strings[i]);

    { Calculate Grammatical Complexity in all files in List sFileNameToCalc }
    Grammar := TGrammar.Create;
    Grammar.StepByStepResults(formCalc.cbStepByStep.Checked);
    if Grammar.calcComplexity(Sequence, gmNvoGram, K) then
    begin
        sResult := sResult +
            '*****' + #13#10 +
            formMain.WinGrammVersion + #13#10 +
            'Compute Grammar Complexity ' + DateTimeToStr(now) + #13#10;

        if FormCalc.cbAddCorpus.Checked then
        sResult := sResult +
            sResultCorpus +
            'Sequence File = CS + ' + tsFileNameToCalc.Strings[i] + #13#10 + #13#10
        else
        sResult := sResult +
            'Sequence File = ' + tsFileNameToCalc.Strings[i] + #13#10 + #13#10;

        sResult := sResult +
            'Sequence Length = ' + IntToStr(Sequence.Length) + #13#10 +
            'Alphabet Size = ' + IntToStr(Grammar.getAlphabetLength) +
            ' letters'+ #13#10 +

```

```

        'Alphabet: ' + Grammar.getAlphabet + #13#10 + #13#10 +
        'Grammatical Complexity = ' + IntToStr( K ) + #13#10 + #13#10 +
        'Time needed: ' + Grammar.getTimeNeededToStr + #13#10 + #13#10 +
        'Grammar Sequence: ' + Grammar.getSequence + #13#10 + #13#10;
    if formCalc.cbRuleFrequency.Checked then
        sResult := sResult +
            'Grammar Rules Found:' + #13#10 + Grammar.getRulesFrequency(true) +
            #13#10 + #13#10
    else
        sResult := sResult +
            'Grammar Rules Found:' + #13#10 + Grammar.getRules + #13#10 + #13#10;
    sResult := sResult +
        ifString(formCalc.cbStepByStep.Checked,
            'Step by Step results: ' + CrLf +
            Grammar.StepByStepResultsText,
            '');
end;

tsGrammaticalComplexity.Add( IntToStr( K ) );
tsTimeNeeded.Add( Grammar.getTimeNeededToStr );
tsLength.Add( IntToStr( Sequence.Length ) );
tsAlphabetLength.Add( IntToStr( Sequence.AlphabetLength ) );
tsSequence.Add( ExtractFileName( tsFileNameToCalc.Strings[i] ) );

Application.ProcessMessages;
if not FormWorking.bContinue then
begin
    Break;
end;
end;

if FormWorking.bContinue then
begin
    FormWorking.Memo1.Lines.Add( 'Done!' );

    with TFormResults.Create(Self) do
    begin
        tsImage.TabVisible := false;
        Caption := 'Results - Grammar Complexity';
        PageControll.ActivePage := tsText;
        Memo1.Text := sResult;
        PageControll.ActivePage := tsGrid;
        StringGrid1.RowCount := tsFileNameToCalc.Count + 1;
        StringGrid1.ColCount := 7;
        StringGrid1.Cells[1,0] := 'Sequence';
        StringGrid1.Cells[2,0] := 'Length';
        StringGrid1.Cells[3,0] := 'Alphabet Length';
        StringGrid1.Cells[4,0] := 'K';
        StringGrid1.Cells[5,0] := 'Time Needed';
        StringGrid1.Cells[6,0] := 'File Name';
        for i:=0 to tsFileNameToCalc.Count-1 do
        begin
            StringGrid1.Cells[0,i+1] := IntToStr(i+1);
            StringGrid1.Cells[1,i+1] := tsSequence[i];
            StringGrid1.Cells[2,i+1] := tsLength[i];
            StringGrid1.Cells[3,i+1] := tsAlphabetLength[i];
            StringGrid1.Cells[4,i+1] := tsGrammaticalComplexity[i];
            StringGrid1.Cells[5,i+1] := tsTimeNeeded[i];
            StringGrid1.Cells[6,i+1] := tsFileNameToCalc[i];
        end;
    end;
end
else
begin
    beep;
    FormWorking.Memo1.Lines.Add( 'Canceled!' );
    ShowMessage( 'Process has been canceled.' );
end;
end;

```

```

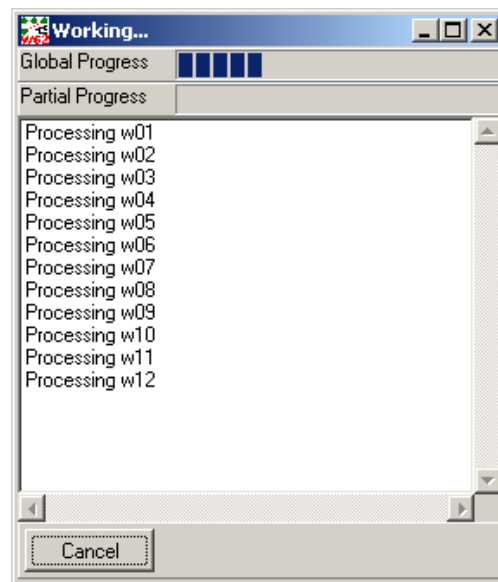
FormWorking.Close;

tsFileNameToCalc.Free;
tsGrammaticalComplexity.Free;
tsTimeNeeded.Free;
tsLength.Free;
tsAlphabetLength.Free;
tsSequence.Free;

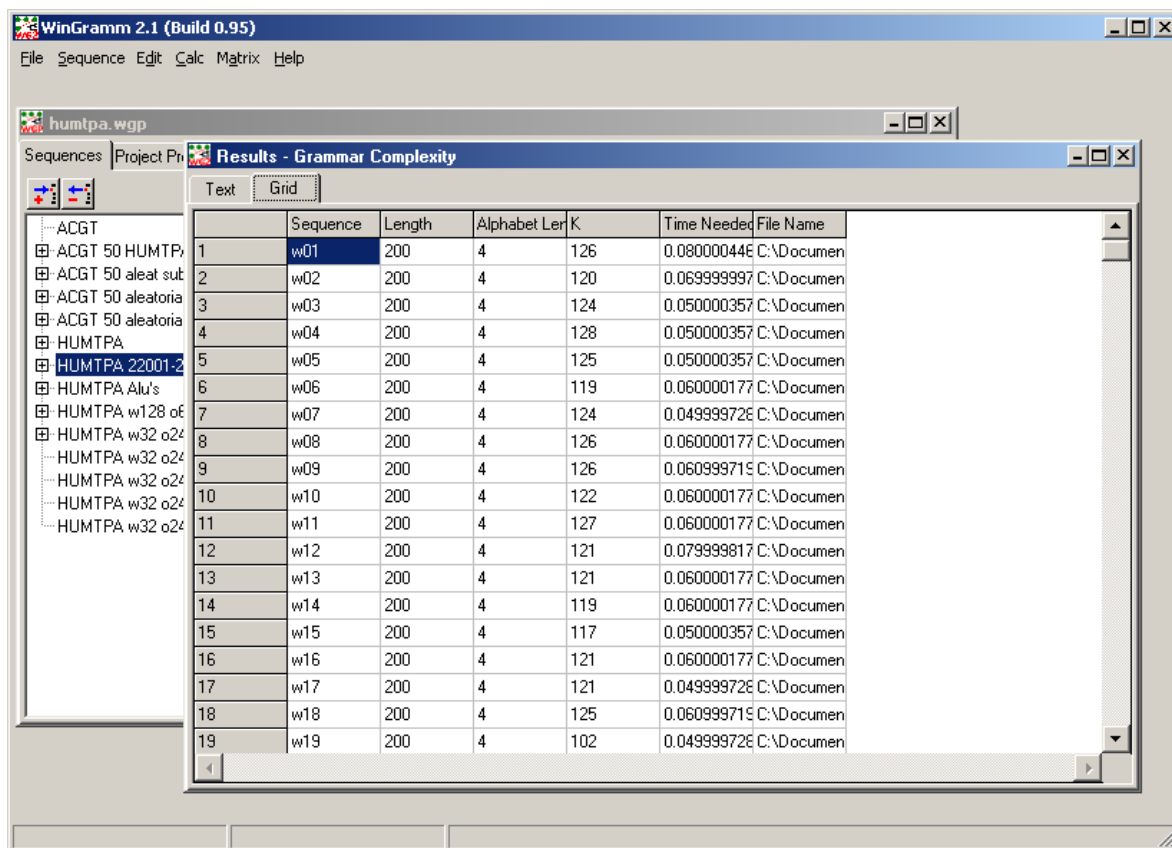
end; { GrammaticalComplexity1Click }

```

Mientras el procedimiento `TformMain.GrammaticalComplexityClick` calcula la complejidad gramatical de todas las secuencias va mostrando el avance en la ventana **Working**:



Cuando termina de calcular la complejidad gramatical de todas las secuencias, muestra los resultados en la ventana **Results**:



El f6lder **Grid** de la ventana Results despliega una cuadr6cula con seis columnas: Secuencia, Longitud, Longitud del alfabeto, K , Tiempo requerido y Nombre del archivo de la secuencia.

Estos resultados se pueden seleccionar y copiar al portapapeles (Edit | Copy) para pegarlos y procesarlos en cualquier otro programa.

En el f6lder **Text** se muestra la gram6tica obtenida, como se indica en la siguiente secci6n.

A.4 Ejemplo de gram6tica generada por WinGramm 2

Para la secuencia:

```
TCTATAGAAAGCAAGTGAGGGAGTTTTCTCAGTAGGTGCAAGGGTAGTGAGGC
ATTTTCATTTTGAGTCATACTCTGATGAGGCTGGCCTGTCTTTTCTCATAGTGATC
TGCAGAGATGAAAAAACGCAGATGATATACCAGCAACATCAGTCATGGCTGCG
CCCTGTGCTCAGAAGCAACCGGGTGGAATATTGCTGGTG
```

WinGramm 2 genera los siguientes resultados:

```
*****
WinGramm 2.1 (Build 0.95)
```

Compute Grammar Complexity 27/08/2003 05:12:41 p.m.
Sequence File = C:\Documents and Settings\jsandria\Mis documentos\Experimentos\GenBank\segmento 22001-26000\w01

Sequence Length = 200
Alphabet Size = 4 letters
Alphabet: TCAG

Grammatical Complexity = 126

Time needed: 0.0899996375665069 seconds

Grammar Sequence:

[6]T[7][4][8][12]A[4][11][13][4][14][15][16]T[4]G[10]A[4][13]T[4][11][9][7][15][7][14][11][6][19][6][5]A[11][9][5][9]C[5][6][14][15][6][7][4][5]A[6][10][4][4][17][8][8][18][9][4][17][7][19]C[12][18]A[16][6][17][9][10][9]CC[5][10][16]A[12][18]C[13]G[5]GA[7][7][10][5]GTG

Grammar Rules Found:

[4]-->AG
[5]-->TG
[6]-->TC
[7]-->AT
[8]-->AA
[9]-->GC
[10]-->[5]C
[11]-->[5][4]
[12]-->[4]C
[13]-->GG
[14]-->TT
[15]-->T[6]
[16]-->[6][4]
[17]-->A[5]
[18]-->[8]C
[19]-->[7]AC

Si en el cuadro de diálogo **Calc** se selecciona la opción **Rule Frequency in Results**, entonces en los resultados se muestra la frecuencia de las reglas encontradas:

Grammar Rules Found:

Freq.	Rule	Expands to
20	[4]-->AG	AG
20	[5]-->TG	TG
13	[6]-->TC	TC
9	[7]-->AT	AT
6	[8]-->AA	AA
6	[9]-->GC	GC
5	[10]-->[5]C	TGC
4	[11]-->[5][4]	TGAG
3	[12]-->[4]C	AGC
3	[13]-->GG	GG
3	[14]-->TT	TT
3	[15]-->T[6]	TTC
3	[16]-->[6][4]	TCAG
3	[17]-->A[5]	ATG

```

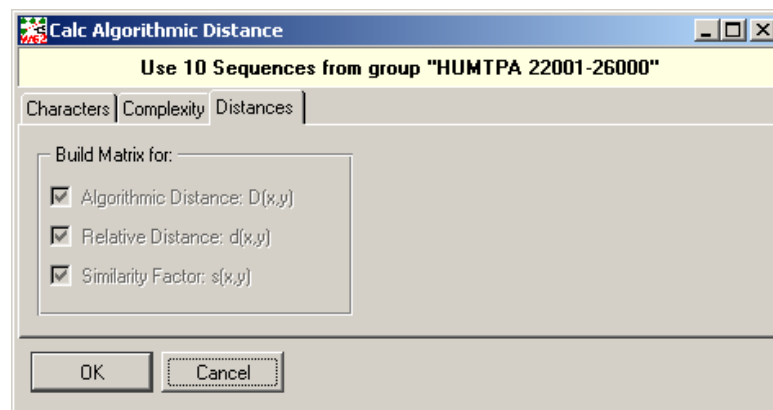
3      [18]-->[8]C      AAC
2      [19]-->[7]AC     ATAC

```

Las columnas están separadas por un carácter de tabulación, lo que permite copiarlas adecuadamente en columnas a una hoja de cálculo.

A.5 Cálculo de la distancia algorítmica, distancia relativa y factor de similitud – matrices de distancias

Se selecciona el conjunto de secuencias (en el panel derecho de la ventana Project) o un grupo (en el panel izquierdo de la ventana Project) y después se selecciona la opción **Calc | Algorithmic Distance** del menú principal. Esto ejecuta el procedimiento TFormMain.AlgorithmicDistanceClick, el cual abre el cuadro de diálogo Calc:



Cuando en este cuadro de diálogo se oprime el botón **OK**, del listado fuente mostrado a continuación (procedimiento TFormMain.AlgorithmicDistanceClick) se ejecuta el código siguiente a la línea `if not (formCalc.ShowModal = mrOK) then Exit;`. Si se oprime el botón **Cancel** se termina la ejecución del procedimiento (`then Exit`).

```

procedure TFormMain.AlgorithmicDistanceClick(Sender: TObject);
var
  ad: TAlgorithmicDistance;
  i,x,y: integer;
begin
  if FormProject.getSelectedFiles.Count = 0 then
  begin
    ShowMessage('You must select at least one sequence');
    Exit;
  end;

  formCalc.MakeVisible(formCalc.tsDistances);
  formCalc.Caption := 'Calc Algorithmic Distance';
  if not (formCalc.ShowModal = mrOK) then Exit;

  ad := TAlgorithmicDistance.Create;
  if formCalc.cbAddCorpus.Checked then
  begin

```

```

        if not ad.CalcWithCorpus(FormProject.getSelectedFiles,
                                FormSelectSequence.SelectedFiles) then exit
    end
else
    begin
        if not ad.Calc(FormProject.getSelectedFiles) then exit;
    end;

with TFormResults.Create(self) do
begin
    tsImage.TabVisible := false;
    PageControll.ActivePage := tsText;
    Caption := 'Results - Algorithmic Distance D(x,y), Relative Distance d(x,y)';
    Memol.Text := ad.log;

    PageControll.ActivePage := tsGrid;
    StringGrid1.RowCount := ad.Count + 2;
    StringGrid1.ColCount := ad.Count + 1;
    StringGrid1.FixedCols := 0;
    StringGrid1.FixedRows := 0;

    StringGrid1.Cells[0,0] := 'D(x,y)';
    StringGrid1.Cells[0,ad.Count + 2 + 0] := 'd(x,y)';
    StringGrid1.Cells[0,2*(ad.Count + 2) + 0] := 's(x,y)';
    for i:=1 to ad.Count do
    begin
        { Titles for Algorithmic Distance Matrix }
        StringGrid1.Cells[i,0] := ExtractFileName(ad.files[i-1]);
        StringGrid1.Cells[0,i] := StringGrid1.Cells[i,0];
        { Titles for Relative Distance Matrix }
        StringGrid1.Cells[i,ad.Count + 2 + 0] := StringGrid1.Cells[i,0];
        StringGrid1.Cells[0,ad.Count + 2 + i] := StringGrid1.Cells[i,0];
        { Titles for Relative Distance Matrix }
        StringGrid1.Cells[i,2*(ad.Count + 2) + 0] := StringGrid1.Cells[i,0];
        StringGrid1.Cells[0,2*(ad.Count + 2) + i] := StringGrid1.Cells[i,0];
    end;

    { Add the Algorithmic Distance Matrix to the Grid }
    for x:=0 to ad.Count - 1 do
        for y:=0 to ad.Count - 1 do
            StringGrid1.Cells[y+1,x+1] := IntToStr(ad.AlgorithmicDistance(x,y));

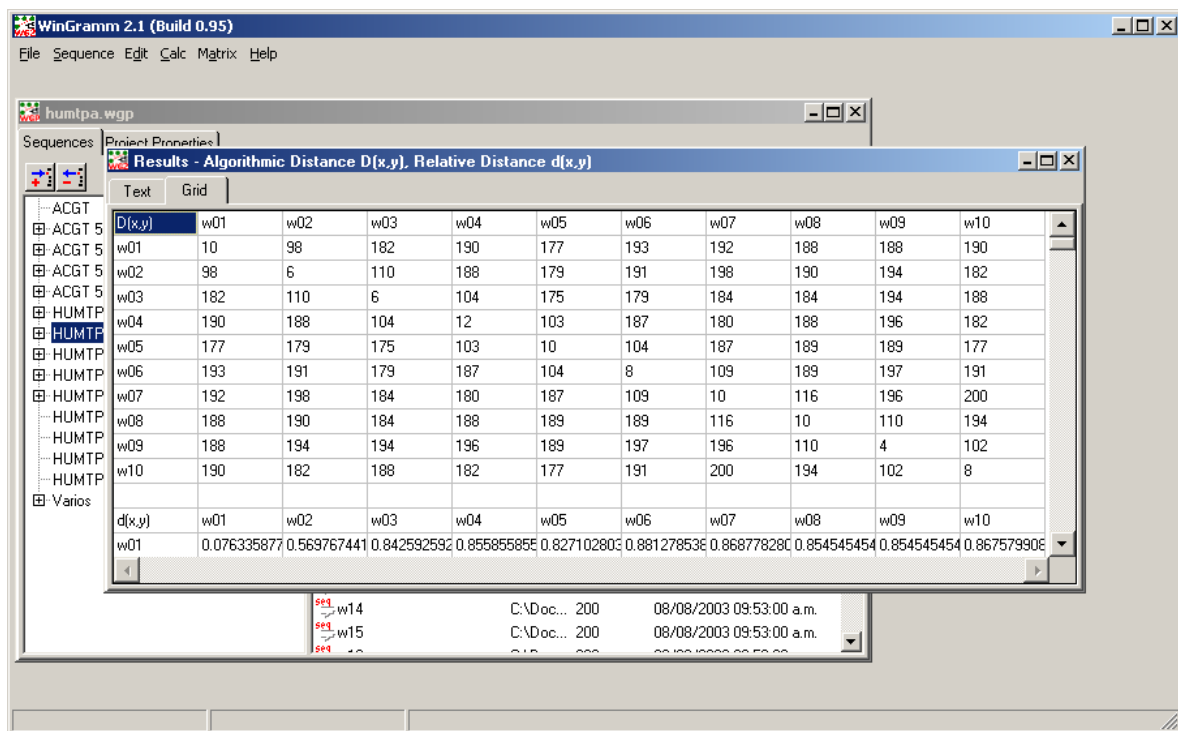
        { Add the Relative Distance Matrix to the Grid }
        StringGrid1.RowCount := StringGrid1.RowCount + ad.Count + 1;
        for x:=0 to ad.Count - 1 do
            for y:=0 to ad.Count - 1 do
                StringGrid1.Cells[y+1,ad.Count + 2 + x + 1] := FloatToStr(ad.RelativeDistance(x,y));

            { Add the Similarity Factor Matrix to the Grid }
            StringGrid1.RowCount := StringGrid1.RowCount + ad.Count + 2;
            for x:=0 to ad.Count - 1 do
                for y:=0 to ad.Count - 1 do
                    StringGrid1.Cells[x+1,2*(ad.Count + 1) + 2 + y + 1] :=
                        FloatToStr(ad.SimilarityFactor(x,y));

                PageControll.ActivePage := tsGrid;
            end; { with TFormResults.Create(self) do}
        end; { AlgorithmicDistance1Click }
    end;
end;

```

Mientras el procedimiento TFormMain.AlgorithmicDistance1Click calcula la distancia algorítmica, la distancia relativa y el factor de similitud de todas las secuencias, va mostrando el avance en la ventana **Working**. Cuando termina de calcular, muestra los resultados en la ventana **Results**:



El folder **Grid** de la ventana Results despliega una cuadrícula con las matrices de distancias calculadas.

Cada matriz se puede seleccionar y copiar al portapapeles (Edit | Copy) para pegarlos y procesarlos en cualquier otro programa. También, una vez seleccionada una matriz, se puede seleccionar la opción (Matriz | Build Phylogenetic Tree) para construir el árbol filogenético.

A.5 Construcción de árboles filogenéticos con UPGMA

Después de generar una matriz de distancias, en la ventana Results (ver sección anterior) se debe seleccionar la matriz completa, posteriormente seleccionar del menú principal la opción (Matrix | Build Phylogenetic Tree). Esto ejecuta el procedimiento `TformResults.BuildPhylogeneticTree1Click`.

```
procedure TformResults.BuildPhylogeneticTree1Click(Sender: TObject);
begin
  if not squareMatrixSelected then
  begin
    Beep;
    ShowMessage('You must select a square matrix greater or equal than
2 x 2');
    Exit;
  end;

  With TformMatrix.Create(Self) do
```

```

begin
  fillMatrix( StringGrid1, sgMatrix );
  buildPhylogeneticTree;
end;
end;

```

Si el usuario seleccionó una matriz cuadrada, entonces se crea un objeto TFormMatrix y se ejecutan sus procedimientos fillMatrix y buildPhylogeneticTree.

```

procedure TFormMatrix.buildPhylogeneticTree;
begin
  if rbUPGMA.Checked then
    begin
      { Just call calcStatistics to verify that all cells in
        sgMatrix are numbers }
      if calcStatistics( sgMatrix ) then
        begin
          copyMatrixToBuild;
          phylogeneticTree.imageTitle := EditTreeTitle.Text;
          phylogeneticTree.imageSubTitle := EditTreeSubTitle.Text;
          phylogeneticTree.imageFooter := formMain.WinGrammVersion;
          phylogeneticTree.imageWidth :=
            StrToInt( cbImageWidth.Items[cbImageWidth.ItemIndex] );
          phylogeneticTree.imageHeight :=
            StrToInt( cbImageHeight.Items[cbImageHeight.ItemIndex] );
          phylogeneticTree.showNodeLabel := cbShowNodeLabel.Checked;
          phylogeneticTree.BuildWithUPGMA( sgBuild );
        end;
      end
    else if rbNJ.Checked then
      ShowMessage( 'Neighbor Joining Method not implemented' );
    end;
end;

```

El procedimiento TFormMatrix.buildPhylogeneticTree inicializa algunas variables del objeto phylogeneticTree (de la clase TPhyloTree) que es una variables dentro de la clase TFormMatrix y después ejecuta el procedimiento phylogeneticTree.BuildWithUPGMA.

```

{ *****
* procedure TPhyloTree.BuildWithUPGMA( sg: TStringGrid )
*
* Build the Phylogenetic Tree using UPGMA
* (Unweighted Pair Group Method)
*
* procedure TPhyloTree.BuildWithUPGMA( sg: TStringGrid );
var
  j: integer;
  s: string;
  r: TRect;

  function addNewNode( x:integer; y:integer ): integer;
  var n: integer;
  begin
    n:=Length(pTree);

```

```

    SetLength(pTree,n+1);
    Inc(nodeCount);
    pTree[n].nodeType := ntNode;
    pTree[n].nodeLabel := IntToStr(n);
    pTree[n].nodeValue := cellToFloat(sg,x,y);
    pTree[n].nodeChild1 := cellToInt(sg,x,0);
    pTree[n].nodeChild2 := cellToInt(sg,y,0);
    addNewNode := n;
end; {addNewNode( x:integer; y:integer ): integer;}

procedure addNewGroup( node: integer );
var p: integer;
begin
    p := sg.ColCount;    { Position of last row/column in StringGrid sg }
    sg.ColCount := p + 1;
    sg.RowCount := p + 1;
    sg.Cells[0,p] := IntToStr(node);
    sg.Cells[p,0] := IntToStr(node);
end; {addNewGroup( node: integer );}

function getValue( x, y: integer ): real;
begin
    if x<y then getValue := cellToFloat(sg,x,y)
    else getValue := cellToFloat(sg,y,x);
end; {getValue( x, y: integer ): real;}

procedure deleteRowColumn(n:integer);
var x,y: integer;
begin
    { First delete row n }
    for y:=n to sg.RowCount-2 do
        for x:=0 to sg.ColCount-1 do
            sg.Cells[x,y] := sg.Cells[x,y+1];
        sg.RowCount := sg.RowCount - 1;

    { Second delete column n }
    for x:=n to sg.ColCount-2 do
        for y:=0 to sg.RowCount-1 do
            sg.Cells[x,y] := sg.Cells[x+1,y];
        sg.ColCount := sg.ColCount - 1;
    end; {deleteRowColumn(n:integer);}

procedure iterate;
var
    xMin, yMin, newNode, x, y: integer;
begin
    searchMin( sg, xMin, yMin );

    newNode := addNewNode( xMin, yMin ); { to pTree }
    addNewGroup( newNode );             { to sg (StringGrid) }

    { Calc average distance for last row }
    y := sg.RowCount - 1;
    for x:=1 to y - 1 do

```

```

        if (x<>y) and (x<>xMin) and (x<>yMin) then
            sg.Cells[x,y] := FloatToStr( (getValue(x,xMin) +
                                           getValue(x,yMin)) / 2 );

        if xMin > yMin then
            begin
                DeleteRowColumn(xMin);
                DeleteRowColumn(yMin);
            end
        else
            begin
                DeleteRowColumn(yMin);
                DeleteRowColumn(xMin);
            end;
        end; {iterate}
    end; {iterate}

begin { BuildWithUPGMA( sg: TStringGrid ) }
    { Init pTree }
    termCount := sg.RowCount - 1; { terminals }
    nodeCount := 0;               { nodes }
    SetLength( pTree, sg.RowCount );
    pTree[0].nodeType := ntRoot;
    pTree[0].nodeLabel := 'Root';

    for j:=1 to termCount do
        begin
            pTree[j].nodeType := ntTerminal;
            pTree[j].nodeLabel := sg.Cells[0,j];
            { Replace upper title in grid with a terminal ID }
            sg.Cells[j,0] := IntToStr(j);
        end;

    s:='';
    while sg.RowCount > 2 do
        begin
            Iterate;
            s:='Id'+#9+'type'+#9+'label'+#9+'value'
                +#9+'child1'+#9+'child2'+#13#10;
            for j:=0 to Length(pTree)-1 do
                begin
                    s:=s+IntToStr(j)+#9;
                    case pTree[j].nodeType of
                        ntRoot: s:=s+'r'+#9;
                        ntTerminal: s:=s+'t'+#9;
                        ntNode: s:=s+'n'+#9;
                    end;
                    s:=s+pTree[j].nodeLabel+#9+
                        FloatToStr(pTree[j].nodeValue)+#9+
                        IntToStr(pTree[j].nodeChild1)+#9+
                        IntToStr(pTree[j].nodeChild2)+#13#10;
                end;
            s:=s+#13#10;
            //ShowMessage(s);
        end;
    end;
end;

```

```

pTree[0].nodeChild1 := Length( pTree ) - 1;

with TFormResults.Create(nil) do
begin
  Width := imageWidth + 20;
{
  Height := imageHeight + 50;}
  tsGrid.TabVisible := false;
  tsText.TabVisible := false;
  Caption := 'Results - Phylogenetic Tree';
  Image1.Width := imageWidth;
  Image1.Height := imageHeight;
  r := Rect(0,0,imageWidth,imageHeight);
  Image1.Canvas.SetClipRect(r);
  DrawTree( Image1, showNodeLabel );
end;
end; { BuildWithUPGMA( sg: TStringGrid ) }

```

El árbol filogenético se construye en la estructuras de datos siguiente:

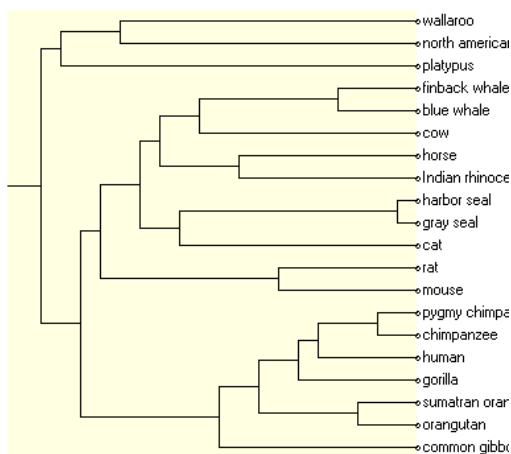
```

type
  TNodeType = (ntRoot, ntTerminal, ntNode);
  TRecPhyloTree = record
    nodeType: TNodeType;
    nodeLabel: String;
    nodeValue: Real;
    nodeParent: integer;
    nodeChild1: integer;
    nodeChild2: integer;
    x, y: integer;
  end;
var
  pTree: array of TRecPhyloTree;

```

Una vez construido el árbol, se dibuja con el procedimiento DrawTree. Dibujándose un árbol como el siguiente:

Phylogenetic tree for D(x,y)



WinGramm 2.1 (Build 0.92)