UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

CÓMPUTO EVOLUTIVO PARA UN SIMULADOR DE DIÁLOGO SIMPLE

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A:

MANUEL TENORIO FENTON

DIRECTORA DE TESIS:
DOCTORA KATYA RODRÍGUEZ VÁZQUEZ

2012

HOJA DE DATOS DEL JURADO

1. Datos del alumno
   Tenorio
   Fenton
   Manuel
   56 11 17 19
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Ciencias de la Computación
   406005005

2. Datos de la tutora
   Doctora
   Katya
   Rodríguez
   Vázquez

3. Datos del sinodal 1
   Doctora
   Sofía Natalia
   Galicia
   Haro

4. Datos del sinodal 2
   Doctor
   José de Jesús
   Galaviz
   Casas

5. Datos del sinodal 3
   M. en C.
   Gustavo
   de la Cruz
   Martínez

6. Datos del sinodal 4
   Doctor
   Iván
   Vladimir
   Meza
   Ruiz

7. Datos del trabajo escrito
   Cómputo Evolutivo para un Simulador de Díalogo simple
   89 p
   2012

Dedico este trabajo a mis papás, por su apoyo y paciencia.

Índice general

Parte I

Introducción

En el año de 1950 Alan M. Turing propuso el artículo "Computing machinery and Intelligence" (Maquinaria de cómputo e Inteligencia), para la revista "Mind" (Mente), que se ha considerado como una prueba para demostrar la existencia de inteligencia en una máquina. La prueba se basaba en la hipótesis positiva (es decir, susceptible a ser comprobada por la experiencia mediante un método científico), de que si una máquina se comporta en todos los aspectos como inteligente, entonces debe de ser inteligente. Consistía a su vez en colocar a un juez, un participante humano y una computadora en habitaciones distintas. El juez tendría que decidir cuál de los participantes era una computadora (determinando consecuentemente cuál era la persona). Solamente podía hacer preguntas, a las cuales la computadora y la persona podían responder mintiendo. La tesis de Turing es que si la persona y la computadora son los suficientemente hábiles, al juez le resultaría imposible determinar cuál de los dos era el humano. Hasta la fecha no se ha logrado que una máquina pase esta prueba en una experiencia con método científico.

Es de esta forma que el procesamiento computacional de Lenguajes Naturales[1], ha estado intrínsecamente relacionado con el desarrollo de la Inteligencia Artifical desde sus comienzos. Ahora es un área diversa y algunas de sus principales ramas son: Traducción Automática, Análisis de Textos y Producción Automática de Lenguaje Escrito, que a su vez trata prosa y diálogos por separado.

Un procesador de lenguaje natural se considera exitoso, al asumir que el comportamiento del sistema es tal, que internamente realiza las mismas cosas que uno hace al entender el lenguaje natural. Formando las mismas estructuras lógico-funcionales que nosotros los seres humanos formamos. Haciendo las mismas deducciones y notando las mismas implicaciones. Esto parece ser el mismo supuesto que se hace cuando se dice que otro ser humano, ha entendido el mensaje que hemos emitido, y sin embargo no existe evidencia directa que demuestre que la mente de otra persona funcione como la propia. Por el contrario, estudios de neurociencia recientes, sugieren que un mismo estímulo sensorial a nivel celular producen efectos observables completamente diferentes, incluso en gemelos idénticos [1].

Por otra parte, se tienen abundantes trabajos en Lingüística, que le dan un tratamiento especial a las ambigüedades propias del lenguaje natural hablado o escrito. Una ambigüedad a nivel léxico aparece cuando la misma palabra, puede tener más de un significado, que debe deducirse a partir del contexto oracional o conocimientos previos. La ambigüedad propia del lenguaje natural es una dificultad esencial para el Procesamiento de Lenguajes Naturales, que de ahora en adelante en este trabajo se abrevia PLN.

En el campo de las Ciencias de la Computación, el uso de diccionarios, gramáticas, bases de conocimientos y correlaciones estadísticas, han conducido a resul-

[1] Los Lenguajes Naturales u ordinarios, surgen de la facilidad del intelecto humano para poseer un lenguaje, que utiliza para comunicarse, tres ejemplos son el español, alemán e inglés. Estos lenguajes pueden hablarse y escribirse, y se distinguen de los Lenguajes Formales, como lo son los lenguajes de programación, y los Lenguajes Construidos, como el esperanto, que se sintetizó a partir de varios lenguajes naturales con el fin de facilitar la comunicación entre personas.

tados satisfactorios en el procesamiento de corpus reducidos [2], en otras palabras, textos bien delimitados en cuanto a extensión y contenido se refiere. Por resultados satisfactorios se entiende, que una computadora es capaz de extraer información y generar oraciones correctas a partir de ésta, siempre que exista un registro de la información que se procesa. Esto choca con las posturas de muchos lingüistas, quienes tal vez acertadamente señalan que se pierde la naturalidad propia del lenguaje.

Hasta ahora el panorama no resultaría favorable para esta tesis, si el objetivo de la misma fuera dar una explicación terminante de cómo es que el humano entiende el lenguaje natural, mostrando los procesos exactos que suceden en el cerebro, y el mecanismo mediante el cual se adquiere la habilidad de expresar ideas de forma escrita. Es resaltable que para este fin tampoco existe un consenso entre científicos de las Neurociencias y lingüistas. Estos últimos sostienen (la discusión detallada está en [2]), que el lenguaje es un producto de la mente, y por lo tanto está sujeto a la voluntad de la misma, mientras que los neurocientíficos muestran que en el acto de aprender nuevos conceptos, se modifica la estructura del cerebro humano.

Lejos de pretender resolver una controversia que se ha mantenido durante décadas de estudio, este trabajo busca aplicar técnicas de programación desarrolladas por distintos investigadores para producir comentarios de manera automática en un diálogo. Particularmente se hace uso de la programación genética aplicada a la inferencia gramatical del lenguaje objetivo, en este caso, el español. Una síntesis de la razón por la cual se optó por la programación genética sobre otros métodos computacionales para el PLN, así como una breve expliación de lo que es la inferencia gramatical, se da a continuación conforme a [2]:

En los primeros intentos serios por construir una máquina que pasara la prueba de Turing, se utilizó el método conocido como reconocedor de patrones. Éste simplemente reconocía cierta combinación de palabras, para elaborar una especie de "máscara" o molde, en el que el resto de las palabras que aparecieran pudieran ser vistas como el contenido del enunciado. Tuvo cierto éxito en sistemas que facilitan consultas a bases de datos, pero se consideró pobre cuando se trató de simular una interacción más parecida a la que se observa entre seres humanos. Esto porque no se puede abarcar la aparentemente inagotable aparición de nuevas combinaciones de palabras.

La inducción gramatical es la construcción gradual de una gramática correcta basada en un conjunto finito de una muestra de expresiones. El camino de hacer más eficientes las gramáticas inferidas introduciendo probabilidades, varía dependiendo del modelo a desarrollar. Ejemplos de gramáticas basadas en frecuencias son las Gramáticas Probabilísticas libres de contexto (Probabilistic Context Free Grammars, ó PCFG's) introducidas por Charniak en 1993 [9]. Asignan una probabilidad a cada cadena (oración) bien formada del lenguaje natural, de modo que la suma de todas las probabilidades asignadas sea igual a uno. Otro ejemplo es el de los modelos ocultos de Markov, propuesto por Rabiner y Juang en 1993 [15], y paralelamente por Kupiec en 1989 [10], que

[2] En Lingüística se refiere al texto analizado con el término "corpus".

asigna una probabilidad a cada cadena, tal que la suma de las probabilidades asignadas a cadenas con la misma longitud sea igual a uno.
Por más potente que ahora pueda parecer la inducción o inferencia gramatical puramente probabilística, trabajos recientes muestran que los mejores esfuerzos terminan por encontrarse con un problema mayor, que es el de un dominio que aprende (crece a medida que la colección de oraciones del lenguaje estudiado aumenta), mientras parece ser necesario descubrir un conjunto de categorías y un conjunto de reglas definidas sobre el mismo. Más aún, si no se tiene especial cuidado al momento de asignar probabilidades, se ocasiona el problema de "frecuencia cero", consecuencia de asignar una pequeña probabilidad a todos los posibles valores, lo que ocasiona que formas gramaticalmente incorrectas, sean tan probables como otras correctas, y cada vez más inaccesibles.
En este trabajo de tesis se aplica un algoritmo de programación genética, para adaptar gramáticas hipotéticas a un modelo más efectivo de un subconjunto del lenguaje de estudio. La efectividad o fuerza de una población de gramáticas se mide en su capacidad de *analizar sintácticamente* el conjunto de entrenamiento. El programa genético es estadísticamente sensible, pues la utilidad de patrones frecuentes se refuerza mediante la selección aleatoria de nodos del árbol, al momento de ejecutar los operadores de recombinación: cruza y mutación.
El aprendizaje del lenguaje de estudio, sigue siendo un problema muy complicado, evidente por el hecho de que las categorías léxicas emergen como nodos más eficientes para capturar regularidades en el conjunto de prueba. Además, el tamaño de las gramáticas se usa para medir la aptitud de los individuos, considerando mejor adaptadas a las más simples.

Parte II

Fundamentos Teóricos

Capítulo 1

Panorama General del PLN

1.1. Orígenes del PLN

Los recursos de procesamiento disponibles para las calculadoras programables de primera generación, no eran diferentes a aquellos de las primeras computadoras. Si tratamos de imaginar lo que sería comprar una calculadora programable barata, con el fin de hacer una traducción automática del ruso al español, podemos tener una idea de la magnitud del reto que enfrentaron los pioneros del PLN, en los años que van desde 1950 hasta 1965.
Hasta el día de hoy, las computadoras representan objetos lingüísticos de forma no lingüística. Así, las palabras se representan como una secuencia de bytes, siendo éstos las representaciones en código ASCII [1] de las letras que las conforman. Tres décadas de Ciencias de la Computación nos han dejado lenguajes de programación, que facilitan referirse a objetos lingüísticos como palabras y oraciones, tan fácil como referirse a números. Sin embargo, los primeros trabajos en Lingüística Computacional, deben verse en el contexto de los recursos que había disponibles en la época en que fueron realizados. De modo que eran trabajos que cuantificaban, por ejemplo, el número de apariciones de una palabra en un texto extenso.
Una de las primeras aplicaciones de la Computación a la Lingüística en ser planificada y patrocinada fue la Traducción Automática (TA). Las comunidades militares y de inteligencia en los Estados Unidos y del resto del mundo, tenían grandes esperanzas puestas en la TA. Desgraciadamente, pese al nivel del patrocinio, la primera generación de trabajos en TA fue muy decepcionante. Las teorías Lingüísticas que los fundamentaban eran sumamente rudimentarias; no tomaban en cuenta a la ambigüedad, ni el hecho de que el significado está involucrado. Y aún cuando se quisieran ocupar mejores teorías, los recursos necesarios no estaban disponibles, de modo que se limitaban a realizar marginalmente más que una sustitución palabra por palabra.
En gran medida, los avances en el PLN han surgido de un cambio de apreciación

[1] Del inglés: American Standard Code for Information Interchange.

con respecto de lo que es una computadora. Pues aunque son excelentes herramientas para la aritmética, conviene verlas como máquinas manipuladoras de símbolos muy generales. Los símbolos pueden representar números, o conceptos más complejos como palabras, oraciones, árboles o redes. El código que la máquina ejecuta realiza operaciones muy simples, como traspasar información de una parte de la memoria a otra, o sumar dos números. Los lenguajes de programación de alto nivel, tales como Prolog, Java o Haskell, por nombrar tres, permiten al programador especificar instrucciones en términos de conceptos más ricos y mejor orientados al problema a resolver. La existencia de compiladores que traducen de este nivel más abstracto a instrucciones primitivas, libra al programador de la carga de replantear cada idea en términos de lenguaje de máquina, y lo deja concentrarse en el problema que realmente le interesa.
Un hito en el desarrollo del PLN como se conoce el día de hoy, fue la aparición en 1971 del programa SHRDLU de Winograd, escrito en LISP, que era el lenguaje predilecto de los investigadores en Inteligencia Artifical durante la década de 1970's. Esta contribución significó una "prueba de existencia", al exhibir que la comprensión del lenguaje natural era posible para la computadora, si se restringía el dominio. SHRDLU mostraba de una manera primitiva, un número importante de habilidades, como son la interpretación de preguntas, declaraciones y órdenes, capacidad para realizar deducciones, explicar sus acciones y aprender nuevas palabras. Estas habilidades no habían sido visto juntas en un mismo programa de computadora. SHRDLU no hubiera sido posible sin lenguajes de alto nivel, menos aún considerando que fue escrito por un solo investigador [3].
Remontándonos a la época que siguió a la Segunda Guerra Mundial, pueden distinguirse dos vertientes principales al campo: el modelo de autómatas y probabilidades, y el modelo de teoría de la información. Los autómatas surgieron del trabajo de Turing sobre el cómputo algorítmico, considerado por muchos como el fundamento de las Ciencias de la Computación modernas. Esto después influyó en el trabajo de la neurona de McCulloc-Pitts, un modelo simplificado de la neurona como un elemento de cómputo que puede ser descrito con lógica proposicional. Siguiendo en orden cronológico, vinieron los trabajos de Kleene, sobre autómatas finitos y expresiones regulares, de Shannon, que aplicó modelos probabilísticos de procesos de autómatas de Markov discretos al procesamiento del lenguaje y considerando a éste, Chomsky en 1956 señaló a las máquinas de estados finitos como una forma de caracterizar gramáticas, y definió un lenguaje de estados finitos como aquel generado por un autómata de estados finitos[5].
La segunda intuición producida en esta época, fue la del desarrollo de algoritmos probabilísticos para el procesamiento del lenguaje hablado y escrito, que se relaciona a la otra contribución de Shannon: la metáfora del canal con ruido y decodificación para la transmisión de lenguaje. Shannon tomó prestado el concepto de entropía de la Termodinámica, como una forma de expresar la capacidad de transmisión de un canal, o el contenido de información de un lenguaje. Con el paso del tiempo, estas dos intuiciones originales llevaron a distintos modelos, que en la siguiente sección se describen por separado, y además, dos importantes métodos del PLN a que dieron lugar la aplicación de estos mo-

delos de forma pura o aumentada, procurando señalar las fortalezas y fallos de cada uno.

1.2. Diferentes modelos

Uno de los principales puntos de la investigación hecha durante los últimos 50 años en el PLN, es que el conocimiento contenido en el lenguaje, y fenómenos como la ambigüedad, pueden ser modelados a través de un conjunto reducido de teorías formales. Todas estas teorías o modelos derivan de herramientas estándar de las Ciencias de la Computación, Matemáticas y Lingüística. Entre los más importantes están: máquinas de estados, sistemas de reglas, lógica, modelos probabilísticos y modelos de espacios vectoriales.
En su presentación más simple, las máquinas de estados son modelos formales que consisten de estados, transiciones entre estados y una representación de entrada. Algunas de las variaciones de este modelo básico son los autómatas de estado finito y transductores, deterministas y no deterministas. Estrechamente relacionadas a este modelo, están sus contrapartes declarativas: sistemas de reglas formales. Entre los más importantes a considerar están las gramáticas regulares, relaciones regulares, gramáticas libres de contexto y gramáticas con características aumentadas.
El tercer modelo que juega un papel importante en la captura del conocimiento en el lenguaje es la lógica. La lógica de primer orden, también llamada cálculo de predicados, y formalismos relacionados como el cálculo lambda, han sido extensamente utilizados para modelar la semántica y pragmática del lenguaje, sin embargo, más recientemente técnicas derivadas de las llamadas lógicas no clásicas en semántica léxica han cobrado importancia.
Los modelos probabilísticos son cruciales para capturar cualquier tipo de conocimiento contenido en el lenguaje. Cada uno de los modelos anteriores, puede aumentarse con el uso de probabilidades. Por ejemplo, la máquina de estados puede aumentarse para producir un autómata con pesos o modelo de Markov. Más adelante se explicará el funcionamiento del modelo oculto de Markov [2], que son ampliamente usados en el campo: en el etiquetado de partes del habla, reconocimiento del voz, comprensión de diálogo, transformación de texto a voz y traducción automática. La ventaja clave de los modelos probabilísticos, es su capacidad de resolver problemas relacionados con ambigüedades. Casi cualquier problema en el PLN puede traducirse a: "Dadas N elecciones para una entrada, elegir la más probable".
Finalmente, los modelos de espacios vectoriales, basados en álgebra lineal, son la base de la obtención de información, y muchos tratamientos para el significado de las palabras. Así, por ejemplo, existen modelos que mapean conceptos del lenguaje natural a vectores, pretendiendo después representar conceptos compuestos como la composición de los vectores asociados a las partes [4].

[2] Del inglés: Hidden Markov Models (HMM's).

1.3. Dos técnicas del PLN a considerar

Regresando al experimento propuesto por Alan M. Turing mencionado en la introducción, puede agregarse que él mismo predijo que a finales del siglo XX, una máquina con diez Gigabytes de memoria podría engañar al juez, haciéndolo pensar que se trataba de un ser humano quien hacía de interlocutor. En la actualidad, contando con máquinas aún más potentes que las descritas por Turing, nuevas preguntas surgen con respecto a la suficiencia de esta prueba para la inteligencia de las computadoras. Es decir, sin tomar en cuenta el hecho de que algunas personas son notablemente más fáciles de engañar que otras, ¿es el manejo del lenguaje de forma humana suficiente por sí mismo para demostrar la existencia de inteligencia?

1.3.1. La búsqueda de patrones y ELIZA

No muy alejado temporalmente a la época de Turing, en 1966 Weizenbaum escribió un programa que nombró ELIZA [3]. Este programa entra en la categoría de sistemas de Procesamiento de Lenguaje Natural tempranos, y era capaz de realizar una conversación limitada con el usuario. Considere la siguiente sesión con ELIZA, en este caso imitando las respuestas de un psicoterapista Rogeriano:

> Usuario: Usted se parece a mi padre en algunos aspectos.
> ELIZA: ¿CUÁL ES EL PARECIDO QUE NOTA CON SU PADRE?
> Usuario: Usted no es tan agresivo, pero creo que no quiere que yo lo note.
> ELIZA: ¿QUÉ LE HACE PENSAR QUE NO SOY AGRESIVO?
> Usuario: Que no discute conmigo.
> ELIZA: ¿POR QUÉ CREE USTED QUE NO DISCUTO CON USTED?
> Usuario: Porque me tiene miedo.
> ELIZA: ¿LE AGRADA PENSAR QUE LE TENGO MIEDO?

Como bien señaló Weizenbaum, en este caso el éxito de ELIZA en este dominio se debe a que no necesita conocer algo en particular del mundo, para imitar a un psicoterapista Rogeriano. La relevancia de este caso, es que mucha gente que interactuó con ELIZA llegó a pensar que realmente los entendía a ellos y sus problemas. Muchos de ellos, incluso seguían creyendo en su capacidad de comprensión después de que se les explicaba el mecanismo mediante el cual ELIZA funcionaba. Desde entonces las teorías de la Cognición, y de la Inteligencia Artificial han avanzado lo suficiente como para tener muchas cosas que decir al respecto, una introducción a estos temas puede encontrarse en [1]. En este trabajo, me limito a describir el funcionamiento de un reconocedor de patrones, como lo fue ELIZA.

Es preciso introducir definiciones de conceptos importantes que se seguirán utilizando a lo largo de todo este trabajo:

Expresiones regulares (RE's [3]) Básicamente, una expresión regular consta de un caracter, un conjunto de caracteres, la cadena vacía que se expresa como ε, y los resultados de aplicar ciertas operaciones sobre éstos. Son una poderosa herramienta para la búsqueda de patrones. Las operaciones básicas sobre expresiones regulares incluyen la concatenación de símbolos, disyunción de símbolos, contadores, anclas y operadores de precedencia. Operadores avanzados como la estrella de Kleene y la cerradura se refieren a la concatenación arbitraria de la expresión regular a la que se le aplica. En el primer caso, la repetición va desde cero veces dando como resultado la cadena vacía o ε, que no está presente en la cerradura de la cadena.

Autómatas de estados finitos (FSA's [4]) Se define a un autómata de estados finitos, como un par de estados distinguidos: el inicial y final, el resto de los estados y las transiciones definidas entre ellos. Cada expresión regular bien formada, tiene un autómata de estados finitos asociado que lo acepta y viceversa. Un autómata define un lenguaje formal, como el conjunto de cadenas que éste acepta. Este lenguaje puede usar cualquier conjunto de símbolos, incluyendo cartas, palabras o incluso imágenes gráficas. Para mayor información sobre autómatas de estados finitos se puede consultar la fuente utilizada para esta sección [4] (capítulo 2) , o bien [5].

FSA's deterministas (DFSA's [5]) El comportamiento de un autómata de estados finitos determinista depende por completo del estado en el que se encuentra.

FSA's no deterministas (NDFSA's [6]) Un autómata de estados finitos no determinista, en ocasiones debe tomar una decisión entre múltiples caminos, bajo la misma cadena de entrada y el mismo estado actual. Cualquier NDFSA puede transformarse en un DFSA.

Estrategia de búsqueda Esto denota el orden en que un NDFSA explora los estados siguientes a explorar. La estrategia por búsqueda a profundidad (depth first search) utiliza una estructura de datos tipo pila LIFO [7]. La estrategia por búsqueda en amplitud (breath first search) utiliza una estructura de datos tipo FIFO [8].

Un importante uso de las RE's es la sustitución textual. Durante una sustitución textual, se provee al algoritmo con patrones y cadenas, de modo que cada vez que se encuentre en el texto procesado una cadena que coincida con

[3] Del inglés: Regular Expression.
[4] Del inlglés: Finite State Automaton.
[5] Del inglés Deterministic Finite State Automaton.
[6] Del inglés: Non Deterministic Finite State Automaton
[7] Del inglés: Last In First Out.
[8] Del inglés: First In First Out.

un patrón proporcionado, se sustituye por la cadena que le corresponda. Así, algunos lenguajes como PERL, facilitan la sustitución de cadenas descritas por una expresión regular, por otras especificadas en el programa. Además, muchas veces existen características extendidas de las RE's, como son los registros de memoria, mediante los cuales, se tiene acceso a las cadenas leídas que coinciden con la descripción establecida por la expresión regular.
El programa ELIZA funcionaba realizando sustituciones textuales en cascada por medio de RE's, cada una procesando una parte del texto alimentado y cambiándolo. La primera sustitución en texto, cambiaba las instancias de *"mi"* por *"su"*, y *"soy" / "estoy"* por *"usted es" / "usted está"*. La siguiente ronda de sustituciones buscaba patrones relevantes y los transformaba a una salida apropiada, de modo que *"usted está triste"* era sustituido por *"lamento escuchar que usted está triste"*, o bien *"¿por qué cree usted que usted está triste?"*. Desde luego, un buen procesamiento del texto recibido implicaría desarrollar una gran cantidad de casos especiales por separado, a pesar de que como se dijo antes, el imitador de un psicoterapista Rogeriano no requería de un conocimiento previo del mundo.

Al final de la parte dedicada al desarrollo de un algoritmo, se obtiene una estructura por medio de cómputo evolutivo, que después se utiliza para demostrar la fortaleza de esta técnica de programación, propia de la rama de la Inteligencia Artificial conocida como aprendizaje de computadoras. Esta estructura representa el árbol de derivación asociado a una gramática capaz de analizar sintácticamente la oración procesada.

1.3.2. Modelos ocultos de Markov

Podemos comenzar mencionando a los N-gramas, que son conjuntos de palabras o letras, que dado el lenguaje son más probables de encontrar que otros. Pensemos por ejemplo que en español la sucesión de palabras:

> ... repentinamente tres muchachos parados en la banqueta ...

es más probablemente leída en un texto que:

> ... tres repentinamente banqueta la muchachos en parados ...

Hay que notar que no se le está dando un tratamiento especial al significado de las palabras, más bien, el hecho de que la primera oración sea semánticamente correcta y la segunda sea incorrecta es consecuencia de las categorías sintácticas de cada palabra, de los distintos agrupamientos de las mismas, y de que suponemos que el texto leído está correctamente escrito.
Para formalizar lo anterior, decimos que un $N - grama$ ayuda a predecir la palabra que se va a leer, después de haber leído las $N - 1$ anteriores. Estos modelos estadísticos de secuencias de palabras también son llamados Modelos de Lenguajes o bien LM's [9]. Los LM's son ampliamente usados en la traducción

[9] Del inglés: Language Models

automática, como un posprocesamiento de la traducción palabra por palabra, e incluso en la corrección automática de textos.
El cálculo de probabilidades puede expresarse para una cadena de n palabras como la aplicación de la regla de la cadena:

$P(x_1x_2...x_n) = P(x_1)P(x_2|x_1)P(x_3|x_1^2)...P(x_n|x_1^{n-1})$

En está fórmula cada x_i se refiere al i-ésimo evento observado, de modo que se mide la probabilidad de que x_i suceda, dados los eventos que le precedieron. Sustituyendo cada palabra W_i por su respectiva X_i en la fórmula, la regla de la cadena nos muestra el vínculo entre calcular la probabilidad conjunta de una secuencia, y la probabilidad condicional de una palabra dadas las que la preceden. Sin embargo, viendo más detenidamente la ecuación escrita arriba, concluímos que entre más larga sea la secuencia de palabras, más improbable será leer una cierta palabra en la oración. Además, no es una forma exacta de calcular la probabilidad de que una palabra aparezca dada una secuencia larga de palabras que le preceden en un texto, y no se puede aproximar simplemente contando cuántas veces aparece la palabra después de una cadena lo suficientemente larga, porque el lenguaje natural es creativo y un contexto en particular, pudo no haber sido tomado en cuenta antes. Explicado de otra forma, no pueden tenerse todos los ejemplos posibles de cadenas para una longitud n.
Con las consideraciones anteriores, se propone el uso de bigramas para el cálculo de probabilidades. En un LM de bigramas, se aproxima la probabilidad de ocurrencia de una palabra, con la probabilidad condicional de la palabra que le precede. Es decir en lugar de calcular:

$P(x_n|x_1...x_{n-1})$ aproximamos el valor buscado con: $P(x_n|x_{n-1})$.

Aquí, la suposición de que la probabilidad de que aparezca una palabra depende solamente de la palabra que le precede, se conoce como suposición de Markov, y los modelos de Markov son la clase de modelos probabilísticos que suponen que se puede predecir la probabilidad de una unidad futura, sin tener que mirar muy atrás en la secuencia de eventos (en nuestro caso de palabras).
El uso fuerte de la probabilidad y estadística en el PLN, no está en la predicción de las palabras que pueden aparecer en un texto como tal, sino en la correcta clasificación de cada palabra en su categoría gramatical correspondiente. A esto se le conoce como el problema de etiquetado.
El problema de etiquetado es bastante viejo en proporción a la edad del PLN mismo. Stolz en 1965 fue el primero en usar probabilidades para resolver ambigüedades [10] al momento de asignar etiquetas [4]. Los modelos ocultos de Markov (HMM) aplicados a la eliminación de ambigüedades en etiquetas, son un caso especial de Inferencia Bayesiana o Clasificación Bayesiana. En ésta, se proporciona un conjunto de observaciones y el trabajo consiste en determinar a cuál clase de un conjunto establecido pertenecen. Como en el PLN se procesa la secuencia de palabras que forman una oración, se trata como una clasificación de observaciones secuenciales.
Para una oración formada por las palabras $W = w_1 \dots w_n$, se busca la asig-

[10] Una ambigüedad se presenta cuando la palabra a etiquetar, puede tener más de una categoría sintáctica.

nación de etiquetas $t_1 \dots t_n$ con $t_i \in T$ el conjunto de todas las etiquetas, que le corresponde. La aproximación bayesiana al problema comienza por considerar todas las posibles secuencias de etiquetas. De este universo enorme con $||T||^n$ secuencias, se elige aquella que sea la más probable para la secuencia de palabras dada. Es decir, se busca la $\hat{t} \in T^n$que satisfaga $\hat{t} = argmax\,(P(t_i^n|W))$, donde t_i^nes la etiqueta i en la posición n.
Hasta ahora la expresión matemática resulta bastante clara, sin embargo, no se conoce una forma directa de computar la ecuación anterior. La intuición de la Clasificación Bayesiana consiste en usar la regla de Bayes:

$$P(x|y) = \quad \frac{P(y|x)P(x)}{P(y)} \quad [ec\,1]$$

Así se obtiene un conjunto de probabilidades más fáciles de calcular.
Ahora se puede sustituir [ec 1] en la primera ecuación, obteniendo:
$\hat{t} = argmax\left(\frac{P(W|t_i^n)P(t_i^n)}{P(W)}\right)$, y después eliminar el denominador P(W), puesto que se calcula la probabilidad para cada secuencia de etiquetas $t_i^n \in T^n$, sin que P(W) cambie en uno de los cálculos. Con esta simplificación llegamos a:

$$\hat{t} = argmax\,(P(W|t_i^n)P(t_i^n)) \quad [ec\,2]$$

Por desgracia [ec 2] no es fácil de calcular tampoco. Los etiquetadores que utilizan HMM's están fundados por esta razón en dos suposiciones, la primera consiste en que la probabilidad de que una palabra aparezca depende solamente de su propia etiqueta, esto significa que es independiente de las palabras alrededor de la misma, y de sus respectivas etiquetas: $P(W|t_i^n) \approx \prod_{i=1}^{n} P(w_i|t_i)$. La segunda suposición es que la probabilidad de que una etiqueta aparezca, depende solamente de la etiqueta que le precede, esto es, se maneja un LM de bigramas: $P(t_i^n) \approx \prod_{i=1}^{n} P(t_i|t_{i-1})$.
Agregando las simplificaciones mencionadas en el párrafo anterior, la [ec 2] queda como se ve a continuación :

$$\hat{t} = argmax\,(P(t_i^n|W)) \approx argmax \prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1}) \;\; [ec\,3]$$

Es necesario mencionar que cuando las probabilidades son todas cercanas a cero, en vez de multiplicarlas se dividen, y el producto se sustituye por una división (expresada frecuentemente como $\div$). La [ec 3] contiene dos tipos distintos de probabilidades. Una es la probabilidad de transición de etiquetas, mientras que la otra es la probabilidad de que aparezca una palabra en particular, dada cierta etiqueta. A la primera la expresa $P(t_i|t_{i-1})$, y puede determinarse un estimado para casos especiales, por ejemplo, qué tanto se ve un adjetivo con etiqueta ADJ después de un sustantivo con etiqueta SUS, contando en un corpus etiquetado y suficientemente extenso cuántas veces aparece ADJ después de SUS:

$P(t_i|t_{i-1}) = \frac{C(t_{i-1},t_i)}{C(t_{i-1})}$, aquí C se refiere al conteo.
Para realizar estimados de la segunda probabilidad, se procede de la misma forma. De modo que si la etiqueta VBZ representa a un verbo conjugado en tercera persona del singular, y pretendemos determinar un estimado de la probabilidad de que sea la palabra *"es"* la que ha sido etiquetada, se procesa el corpus con la fórmula: $P(es|VBZ) = \frac{C(VBZ,es)}{C(VBZ)}$.

1.4. Comentarios sobre el capítulo

Al final de esta primera sección se revisaron de manera general dos métodos distintos de realizar PLN. El reconocimiento de patrones mediante autómatas de estados finitos puede extenderse mediante programación lógica inductiva. Esta técnica de programación, muy posterior a la primera versión del algoritmo ELIZA, permitiría el reconocimiento de nuevos patrones que no fueron agregados al conjunto de reglas base del algoritmo. La pregunta es, ¿cómo lo haría?. Para ello, requeriría determinar mediante un preprocesamiento la estructura sintáctica de la oración que se desea reconocer. Distintos métodos se describen a detalle en [3].
Los métodos probabilísticos resultan realmente útiles para que la computadora resuelva ambigüedades relacionadas con el etiquetado, es decir, situaciones en las que una palabra puede tener más de una categoría sintáctica en una oración. El limitante aparece cuando no se cuenta con un corpus bien etiquetado realmente extenso, porque queda claro que por la naturaleza creativa del lenguaje un texto limitado no puede ser representativo del mismo.
A pesar de los nuevos avances y diversos modelos de lenguajes (LM's) descritos en la literatura de PLN [4], es notable la necesidad de un paradigma menos rígido, que se adapte a la estructura y naturaleza de las oraciones que procesa el algoritmo.
Claro está, y se explicará la razón en las siguientes secciones, que el cómputo evolutivo no se libra de la necesidad de dirección por parte de un agente humano, aunque se puede evitar el trabajo tedioso de etiquetar textos para proporcionar corpus buenos.

Capítulo 2

Cómputo Evolutivo

2.1. Breve historia

Desde 1859, el concepto de la evolución Darwinista con su principio de "supervivencia del más apto", captó la imaginación popular. Este principio sirve como idea intuitiva para introducir el Cómputo Evolutivo, si se piensa que un ser vivo evolucionado demuestra un comportamiento *optimizado* complejo a cada nivel: la célula, el órgano, el individuo y finalmente la población [6].
En apariencia las especies biológicas han resuelto los problemas del caos, oportunidad, interacciones no lineales y temporalidad. El concepto evolutivo puede aplicarse a problemas que al enfrentarse con heurísticas y métodos clásicos de optimización, no producen resultados satisfactorios.
La teoría de la selección natural establece que los animales y plantas que existen hoy, son el resultado de millones de años de adaptación a las exigencias del medio ambiente. En cualquier punto del tiempo, un número indeterminado de organismos pueden coexistir y competir por los mismos recursos disponibles en un ecosistema. Aquellos más capaces de obtener dichos recursos y tener éxito procreando, serán los que tengan descendientes numerosos en el futuro. Caso contrario para los que tienen dificultades para adaptarse. Los más aptos poseen características *seleccionadas* que pasan a la siguiente generación, y con el paso del tiempo la población entera del ecosistema se dice que ha evolucionado porque se observa que posee más características que la hacen ser más apta.
En las técnicas del Cómputo Evolutivo se abstraen estos principios evolucionistas en algoritmos que pueden usarse para encontrar soluciones óptimas a una gran variedad de problemas. En los algoritmos de búsqueda, se tiene un número de posibles soluciones a un problema y se pretende hallar la mejor posible en un tiempo fijo.
En un espacio de búsqueda finito y pequeño, todas las posibles soluciones pueden procesarse en una cierta cantidad de tiempo y la mejor entre ellas queda determinada. Sin embargo, una búsqueda exhaustiva resulta inviable a medida que el espacio de búsqueda crece.

En los algoritmos de búsqueda tradicionales se realiza un muestreo aleatorio (trayectoria aleatoria) o heurístico (gradiente descendiente) del espacio de búsqueda, tomando una posible solución a la vez con la esperanza de que la óptima encontrarse. El aspecto clave que distingue un algoritmo de búsqueda evolutivo, es que está basado en una población de soluciones. A través de la adaptación sucesiva de un número grande de individuos, el algoritmo evolutivo realiza una búsqueda dirigida eficiente. Esto resulta más efectivo que la búsqueda aleatoria y es menos susceptible a estancarse en óptimos locales que cuando se utiliza un gradiente descendiente.
El Cómputo Evolutivo surgió de llevar ideas de la teoría biológica evolucionista a las Ciencias de la Computación, y continúa atento de nuevos hallazgos en la investigación en Biología para más inspiración. Como sea, aunque es cierto que se lleva a cabo una simplificación de la complejidad observada en el mapa genético de los seres vivos, las propiedades adaptativas del código genético ilustran cómo ambas comunidades pueden contribuír a un entendimiento común de las abstracciones evolutivas adecuadas.
Existen cuatro paradigmas históricos que han servido de base para mucho de lo que se ha logrado en el campo del Cómputo Evolutivo: la programación evolutiva de Fogel y otros investigadores en 1966, las estrategias evolutivas de Recheuberg en 1973, los Algoritmos Genéticos de Holland en 1975, y la Programación Genética de Koza entre 1992-1994. El algoritmo que se desarrolla en este trabajo pertenece al paradigma de Koza, es decir, es un programa genético.

2.2. Características

En el cómputo evolutivo, el investigador elige un *esquema de representación* para definir el conjunto de soluciones que forman el espacio de búsqueda del algoritmo. Se crea un número de soluciones individuales para formar la *población inicial*. Los siguientes pasos se repiten iterativamente hasta que se encuentra una solución tal que satisface los criterios de terminación predefinidos:

1. Cada individuo se evalúa usando una *función de aptitud* que es específica del problema que se desea resolver.

2. Basándose en los valores de aptitud, son elegidos cierta cantidad de individuos para ser padres.

3. Nuevos individuos o descendencia, se producen de aquellos padres usando *operadores de reproducción*. Los valores de aptitud de la descendencia se calculan con la misma función usada en los padres.

4. Finalmente, sobrevivientes de la población vieja se seleccionan para formar la nueva población junto con los descendientes. De modo que se obtiene la siguiente generación.

El mecanismo que determina cuántos y cuáles individuos seleccionar para que sean padres, cuántos descendientes crear, y cuáles individuos pasarán a la siguiente generación representa el *método de selección*. Se han propuesto muchos métodos de selección que varían en complejidad han sido propuestos. Generalmente se asegura que cada generación tenga el mismo tamaño.
Pueden introducirse tres características fundamentales del Cómputo Evolutivo en el contexto de la teoría biológica evolucionista:

1. Genes particulares y genética de poblaciones.
2. Código genético adaptativo.
3. Dicotomía entre genotipo y fenotipo [1].

El segundo punto hace referencia al hecho de que en un algoritmo propio del cómputo evolutivo, lo que evoluciona es la información genética contenida en el genotipo de los individuos. El tercer punto lleva implícito una pregunta que permanece sin respuesta en la Biología: ¿Por qué todas las formas de vida conocidas, usan dos polímeros cualitativamente distintos, los ácidos nucleicos y las proteínas, con la necesidad asociada de traducción?. Teorías recientes se concentran en el descubrimiento de que el ARN puede actuar como medio de almacenamiento genético y también como molécula catalítica. El Cómputo Evolutivo ha avanzado más en la formalización del concepto de lenguaje de representación. Una discusión más clara al respecto puede encontrarse en [6], donde se establece la necesidad de conocer a priori un método para decodificar la información contenida en el genotipo.

2.3. Ventajas

El Cómputo Evolutivo ofrece ventajas prácticas a varios problemas de optimización. Estas ventajas incluyen el enfoque simple de la aproximación, la respuesta robusta al cambio de circunstancias, su flexibilidad y más. A continuación se profundiza en algunos puntos a favor de las técnicas propias del Cómputo Evolutivo:

Enfoque simple El Cómputo Evolutivo es conceptualmente simple. El algoritmo consiste en una inicialización, variación iterativa y selección al margen de un índice de desempeño. En particular, no es necesario que un gradiente de información se presente al algoritmo. Tras las iteraciones de variación aleatoria y selección, la población puede converger a soluciones óptimas. La efectividad de un algoritmo evolutivo depende de los operadores de variación y selección aplicados a una representación, y de su inicialización adecuada.

[1] Genotipo es la totalidad de información genética que posee un individuo en forma de ADN, el fenotipo es la expresión de éste en determinado ambiente.

Múltiples aplicaciones Un algoritmo evolutivo puede usarse para cualquier problema que pueda formularse como uno de optimización de función. Se requiere una estructura de datos para representar las posibles soluciones. El diseñador humano elige la representación de acuerdo a la intuición que tenga del problema, y debe permitir que los operadores de variación mantengan un vínculo de comportamiento entre padres e hijos. Cambios pequeños en la estructura de los padres, deben producir cambios pequeños en los descendientes. Un area clásica para la aplicación de algoritmos evolutivos es la solución de los problemas de combinatoria discreta.

Paralelización La evolución es un proceso altamente paralelo. En estos tiempos, en que las computadoras que soportan procesamiento distribuído se han vuelto populares y más accesibles, es de esperarse que el potencial de aplicar algoritmos evolutivos a problemas más complejos también crezca. Generalmente las soluciones individuales se evalúan independientemente de aquellas con las que compite, esto quiere decir que la evaluación puede manejarse en paralelo, mientras que la selección solo requiere un operador serial.

Hibridación con otros métodos Los algoritmos evolutivos pueden combinarse con técnicas de optimización más tradicionales, de formas tan simples como sería usar un gradiente de minimización después de una búsqueda primaria con un algoritmo evoultivo que delimite bien el dominio de búsqueda. Además, los algoritmos genéticos pueden utilizarse para mejorar el desempeño de redes neuronales, sistemas difusos [2], sistemas de producción, sistemas inalámbricos y otras estructuras de programación.

Robustez a cambios dinámicos Los métodos tradicionales de optimización no son robustos a cambios dinámicos en el entorno, y requieren comenzar desde el principio para encontrar una solución. Por el contrario, los algoritmos evolutivos pueden adaptar las soluciones propuestas al cambio de circunstancias. La población generada de soluciones evolucionadas provee una base para seguir mejorando, y en muchos casos no es necesario reiniciar la población aleatoriamente. Se ha hecho uso extensivo de esta propiedad ventajosa del cómputo evolutivo para evolucionar redes neuronales recurrentes, que controlan el funcionamiento de agentes autónomos, como robots que se mueven con ciertas limitantes.

Capacidad para solucionar problemas sin soluciones conocidas Los algoritmos evolutivos pueden enfrentarse a problemas en los cuales no existe conocimiento humano previo. Aún cuando el conocimiento humano debería usarse cuando se requiere y está disponible, frecuentemente resulta poco adecuado para rutinas automáticas que resuelven problemas, debido a la gran cantidad de reglas que se requieren capturar. La inteligencia artificial puede aplicarse a muchos problemas difíciles que requieren alta velocidad computacional, pero no pueden competir con la inteligencia

[2] Del inglés: fuzzy systems.

humana. Ciertos problemas tienen asociado un conjunto de reglas fijo, que puede no coincidir con la experiencia humana, no ser consistente o simplemente provocar errores. Fogel declaró al respecto: "La I.A. resuelve problemas, pero no resuelve el problema de cómo resolver problemas" [11].

2.4. Algoritmos Genéticos

Los Algoritmos Genéticos, que abreviado se escribe AG, fueron inventados por John Holland y desarrollados en su libro titulado "Adaptación en sistemas naturales y artificiales" en el año de 1975 [12]. Holland propuso los AG como un método heurístico basado en la supervivencia del más apto. Los AG demostraron entonces ser una herramienta útil para resolver problemas de búsqueda y optimización. Aquí se dedica un espacio para explicar el funcionamiento de estos algoritmos, que pueden considerarse como la técnica del Cómputo Evolutivo más popular, y porque la Programación Genética retoma en gran medida muchos aspectos que primero estuvieron presentes en los AG [6].

2.4.1. El espacio de búsqueda

Muy frecuentemente uno busca la mejor solución dentro de un conjunto de soluciones. El conjunto de todas las soluciones alcanzables, donde la mejor se encuentra contenida, recibe el nombre de espacio de búsqueda. Todos y cada uno de los puntos en el espacio de búsqueda representa una posible solución. Un individuo describe una trayectoria en el espacio de búsqueda al modificar la información contenida en su genotipo, provocando cambios en su fenotipo, que se traduce a la solución particular que ese individuo representa [6].
De lo anterior, cada posible solución puede marcarse con un valor de aptitud, dependiendo de la definición del problema. Por medio de los AG's, uno busca la mejor solución entre un número de soluciones posibles, representada como un punto en el espacio de búsqueda. Las dificultades surgen de inmediato: los óptimos locales y el punto de partida de la búsqueda.

2.4.2. Un algoritmo genético simple

Un algoritmo es una serie de pasos para resolver un problema. Un AG es un método para resolver problemas que usa la genética como modelo. En él se maneja una *población* de posibles soluciones, donde cada *individuo* se representa mediante un *cromosoma* que es su abstracción. La codificación de todas las posibles soluciones en cromosomas es la primera parte, y en ningún sentido la más directa de un AG. Un conjunto de operadores de reproducción debe definirse, de modo que aplicados directamete a los cromosomas se encarguen de realizar mutaciones y recombinaciones en las soluciones del problema.
La correcta representación y los operadores de reproducción son absolutamente determinantes, ya que el comportamiento del AG depende en extremo de ellos. Frecuentemente resulta muy difícil encontrar una representación adecuada, que

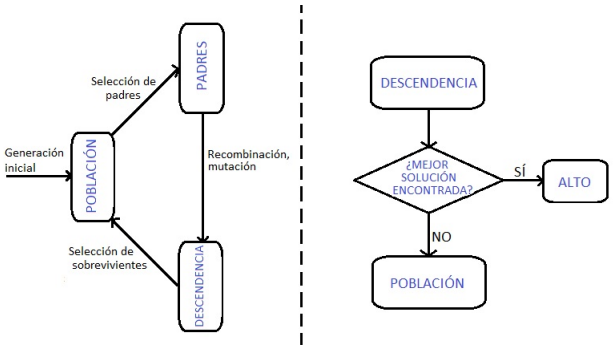


Figura 2.1: Esquema de un AG simple

respete la estructura del espacio de búsqueda, y operadores de reproducción que sean coherentes y relevantes de acuerdo a las propiedades del problema que se desea resolver.

La selección debe poder comparar cada individuo en la población, esto mediante el uso de una función de aptitud. Cada cromosoma tiene un valor que corresponde a la evaluación de qué tan buena es la solución que representa. La solución óptima es entonces la que corresponde al cromosoma que maximice la función de aptitud.

Una vez bien definidos los mecanismos de reproducción y la función de aptitud, la población presente en el AG se evoluciona de acuerdo a la misma estructura básica. Se comienza generando una población inicial de cromosomas, que debe ofrecer una amplia variedad de material genético. Los recursos genéticos, también conocidos como "pool genético", tienen que ser tan grandes como sea posible para que cualquier solución del espacio de búsqueda pueda engendrarse. Generalmente la población inicial se genera aleatoriamente. Entonces, el AG se cicla en un proceso iterado para hacer que la población evolucione. Cada iteración consiste en los siguientes pasos:

1. **SELECCIÓN**: El primer paso consiste en elegir individuos para la reproducción. Esta selección se realiza con una probabilidad que depende de la aptitud relativa de los individuos, de modo que los mejores terminan siendo más seleccionados que los malos y menos adaptados.

2. **REPRODUCCIÓN**: En el segundo paso, la descendencia se obtiene de los individuos seleccionados. Para generar nuevos cromosomas, el algoritmo puede hacer uso de la recombinación o cruza, y de la mutación.

3. **EVALUACIÓN**: La aptitud de los nuevos cromosomas se evalúa.

4. **REEMPLAZO**: Durante el último paso, individuos de la vieja población se eliminan y reemplazan por los nuevos.

Los operadores de cruza y mutación operan con probabilidades diferentes sobre los individuos seleccionados. Antes de implementar un AG, es importante considerar ciertas directrices para el diseño de algoritmos de búsqueda en general:

1. Determinismo: una búsqueda puramente determinista puede tener una varianza demasiado alta en sus resultados, pues puede quedar atorado en los peores casos y no poder salir debido a su determinismo. Más aún, la predicción de los peores casos no siempre es posible.

2. No determinismo: un método de búsqueda estocástico usualmente no sufre del potencial mencionado arriba de caer en el peor caso. Es natural pensar entonces que una búsqueda debería ser estocástica, sin embargo pueden contener una porción sustancial de determinismo. Es suficiente procurar tener suficiente no determinismo para evitar la caída en los peores casos.

3. Determinismo local: un método puramente estocástico puede ser muy lento. Es razonable hacer tantas predicciones deterministas y eficientes como sea posible, de las direcciones más prometedoras de los procedimientos locales. Esto se llama alpinismo de colinas o búsqueda avara (o glotona) [3] de acuerdo con la estrategia.

La habilidad del AG de explorar y aprovechar simultáneamente, y el éxito en su aplicación a problemas del mundo real, refuerza la conclusión de que es una técnica de optimización poderosa y robusta. En las siguientes subsecciones se desarrollan a detalle elementos de la terminología y operadores de un AG.

2.4.3. Individuos

Un individuo agrupa dos formas de soluciones: el cromosoma que es la información "genética" en bruto (genotipo) que el AG manipula, y el fenotipo que es la expresión del cromosoma en términos del modelo.
Si pensamos en un problema de optimización en $\mathbb{R}^3$ con tres variables: X, Y y Z, cada una de éstas sería un factor. Un cromosoma se subdivide en genes. En los AG's, un gen es la representación de un factor y el medio de controlarlo. Cada factor en el conjunto de soluciones se corresponde con un gen del cromosoma.
El cromosoma debería de alguna forma contener información de la solución que representa. La función de morfogénesis, que normalmente se conoce como función objetivo, mapea a cada genotipo con su respectivo fenotipo. Esto simplemente significa que cada cromosoma codifica exactamente una solución, pero no necesariamente que cada solución sea representada por solo un cromosoma. Es decir, la función objetivo no es necesariamente biyectiva, e incluso en ocasiones resulta imposible pretender que lo sea, especialmente en representación binaria para el cromosoma.
No obstante, se debe pedir que la función objetivo sea suprayectiva, es decir, que todas las soluciones posibles al problema deben de corresponderse con al menos un cromosoma, para asegurarse que el espacio de búsqueda puede explorarse en

[3] Textual del inglés: local hill climbing or greedy search

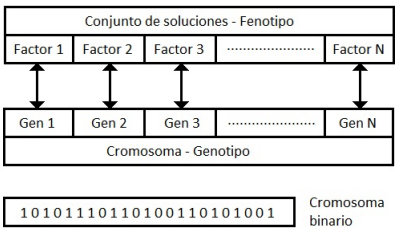


Figura 2.2: Relación entre cromosomas y las soluciones que codifican

su totalidad. Cuando la función objetivo no es inyectiva, y más de un cromosoma mapea a la misma solución, se dice que la representación está degenerada. Una degeneración ligera no es preocupante, pero una degeneración grande es un problema serio, que puede afectar el comportamiento del AG pues si varios cromosomas representan al mismo fenotipo, obviamente el significado de cada gen no se va a corresponder con una característica específica de la solución.

Los genes son instrucciones básicas para el diseño de un AG. Un cromosoma es una secuencia de genes, que pueden describir una solución posible al problema. La estructura de cada gen está codificada en un registro de parámetros del fenotipo, estos parámetros son instrucciones para el mapeo entre el genotipo y el fenotipo. Este mapeo es necesario para obtener versiones del conjuto de soluciones con las cuales el AG puede trabajar.

2.4.4. Aptitud

La aptitud de un individuo en un AG es el valor de una función objetivo para su fenotipo. Para calcular la aptitud el cromosoma debe primero decodificarse, y la función objetivo debe de evaluarse. La aptitud no solamente indica que tan buena es la solución, sino también expresa qué tan cerca está el cromosoma del óptimo.

En el caso de optimización multicriterio, la función objetivo es definitivamente más difícil de determinar. En los problemas de optimización multicriterio, muchas veces existe un dilema sobre cómo determinar que una solución es mejor que otra, ¿qué hacer cuando una solución es mejor en un critero que otra, pero peor en otro?. Aquí el dilema surge más del concepto de mejor solución, que de la forma en que se implementa un AG para encontrarla. Si a veces la combinación de multiples criterios para obtener una función objetivo puede dar buenos resultados, esto supone que la combinación de criterios debe y puede llevarse a cabo de una forma consistente.

El algoritmo desarrollado en este trabajo toma solamente un criterio en cuenta para su función de aptitud.

2.4.5. Poblaciones

Una población es una colección de individuos. Consiste en un número definido de individuos siendo evaluados, los parámetros del fenotipo que definen a los individuos y alguna información acerca del espacio de búsqueda. Los dos aspectos importantes de las poblaciones usadas en AG's son los siguientes:

1. La generación de la población inicial.
2. El tamaño de la población.

Para cada problema, el tamaño de la población dependerá de la complejidad del mismo. Además la misma es normalmente inicializada de forma aleatoria, esto quiere decir, que si los cromosomas son binarios, cada bit es inicializado con un uno o un cero con la misma probabilidad. También hay casos en que la población puede inicializarse con buenas soluciones conocidas previamente.
Idealmente, la primera población debería tener un pool genético lo suficientemente grande para explorar todo el espacio de búsqueda. Para lograr esto, en casi todos los casos la población inicial es elegida de forma aleatoria, sin embargo, en ocasiones se puede hacer uso de alguna heurística especial para sembrar a la primera población. Así, la aptitud promedio de la población ya es alta, y esto puede ayudar al AG a encontrar una buena solución más rápido.
Con todo y heurísticas, debe procurarse que el pool genético de la población sea lo suficientemente grande, de lo contrario solo se explorará una pequeña porción del espacio de búsqueda y nunca se tendrá la certeza de encontrar óptimos globales.
El tamaño de una población es un problema por sí mismo. Entre más grande es, más fácil es explorar el espacio de búsqueda. Pero ha quedado establecido que la complejidad de un AG que converge al óptimo global es $O(n \log n)$ [4] funciones de evaluación, donde n es el tamaño de la población (esta información fue tomada de [6]). Se dice que una población ha convergido cuando todos los individuos son muy similares entre sí, y el mejoramiento solo puede lograrse mediante la mutación. Goldberg también ha mostrado que la eficiencia de un AG para alcanzar el óptimo global en vez de uno local, está muy determinado por el tamaño de la población. Una población grande es útil, pero requiere mucho más costo computacional, memoria y tiempo. Generalmente se usan 100 individuos, pero es un número que puede cambiarse de acuerdo a los resultados observados y los recursos computacionales disponibles.

2.4.6. Codificación

La codificación es el proceso de representar genes individuales. El proceso puede llevarse a cabo utilizando bits, números, árboles, arreglos, listas o cualquier otro objeto. La codificación depende de la aproximación que se desea

[4] De Orden n log n significa que existe una constante positiva c tal que a partir de un valor n_0, el número de operaciones realizadas por el AG no sobrepasa a $c(n \log n)$. Para mayor información sobre notación asintótica, puede consultarse cualquier libro que trate el tema de Análisis de Algoritmos.

al resolver el problema. Se puede por ejemplo emplear directamente números reales o enteros. A continuación se describen los tres tipos de codificación más utilizados en los AG's:

- Codificación binaria: El método más común de codificación utiliza cadenas binarias para representar los cromosomas. Cada bit en la cadena puede representar alguna característica de la solución. Todas las cadenas de bits están asociadas a una solución, pero no necesariamente la mejor. Otra posibilidad es que toda la cadena represente un número. El modo de codificación con cadenas de bits varía de un problema a otro.
  La codificación binaria posibilita representar muchos cromosomas con un número pequeño de alelos [5]. Por otro lado, para muchos problemas esta codificación puede no ser natural y se torna necesario hacer correcciones después de que los operadores genéticos han actuado. La longitud de una cadena binaria depende de la precisión deseada, esto quiere decir que los enteros se representan con exactitud, los números reales requieren truncarse y la cantidad de éstos que se pretende codificar incrementan la longitud de la cadena de bits.

- Codificación entera o de permutación: Cada cromosoma es una cadena de números, que representa a su vez una secuencia específica. En ocasiones es necesaria alguna corrección después de operar genéticamente sobre los cromosomas, para evitar por ejemplo la repetición de valores en un cromosoma. La codificación de permutación es útil solamente para resolver problemas de ordenamiento, por ejemplo el problema del agente viajero. Es evidente que se requieren correcciones para algunos operadores de cruza y mutación para dejar un cromosoma consistente, que no repita valores.

- Codificación por valores: Cada cromosoma es una cadena de valores, y estos valores pueden ser de cualquier tipo siempre que estén conectados al problema, por ejemplo, pueden usarse números reales. Esta codificación produce resultados buenos para algunos problemas especiales. Sin embargo, es necesario desarrollar operadores genéticos específicos para los valores usados en el AG, y que sean adecuados para el problema. La codificación directa de valores puede usarse en problemas, en los que aparecen valores complicados como números reales.

2.4.7. Selección

La selección es el proceso de elegir a dos padres de la población para cruzarlos. Después de definir la codificación, el siguiente paso es decidir cómo se realizará la selección, en otras palabras, cómo se van a elegir los individuos de la población que van a generar la descendencia para la siguiente generación, y cuántos descendientes van a procrear.

[5] Un alelo es cada una de las formas alternativas que puede tener un gen que se diferencían en su secuencia, y que se pueden manifestar en modificaciones concretas en la función de ese gen.

El propósito de la selección es enfatizar a los mejores individuos de una generación, con la esperanza de que sus descendientes sean más aptos. En un AG, se eligen cromosomas para que sean padres, siendo la pregunta a contestar ¿cuáles cromosomas elegir? Según la teoría evolucionista de Darwin, son los más aptos los que sobreviven y producen descendencia.
Se requiere entonces un método que recoja cromosomas aleatoriamente de la población, de acuerdo a su función de evaluación. Entre más alta sea su función de aptitud, más probabilidad tiene un individuo de ser seleccionado. La *presión selectiva* se define como el grado en que los mejores individuos son favorecidos. Con una presión selectiva grande, los mejores individuos son altamente favorecidos. Esta presión selectiva conduce al AG a un mejoramiento de la aptitud de la población en las generaciones sucesivas.
La tasa de convergencia de un AG está muy determinada por la magnitud de la presión selectiva, son de hecho, directamente proporcionales. Los AG deberían ser capaces de detectar soluciones óptimas o casi óptimas bajo un amplio espectro de esquemas de selección con las presiones que éstos implican. Como sea, si la presión selectiva es muy baja, la tasa de convergencia será baja y el AG necesitará un tiempo innecesariamente largo para encontrar la solución óptima. Si la presión selectiva es muy alta, el AG puede converger prematuramente a un óptimo local, no global. Por ello, los esquemas de selección además de proporcionar presión selectiva, deben propiciar que exista y se preserve la diversidad en la población, siendo que esto evita la convergencia prematura.
Hay dos tipos de esquemas de selección, la selección proporcional y la selección basada en ordinales. La selección proporcional escoge individuos tomando en cuenta sus valores de aptitud en relación con los valores de aptitud de los demás individuos en la población. La selección basada en ordinales escoge individuos basándose no en el valor de aptitud puro, sino tomando en cuenta un "rango" dentro de la población. Esto requiere que la presión selectiva sea independiente de la distribución de aptitud de la población, y se basa solamente en el ordenamiento relativo (ranking) de la población.
Debe de haber un balance entre la selección y la variación producida con la cruza y mutación. Una selección muy fuerte significa que individuos sub-óptimos pueden tomar el control de la población, reduciendo la diversidad necesaria para el cambio y progreso; una selección muy débil significará evoluciones lentas.
El *elitismo* es cuando el mejor individuo o unos pocos entre los mejores, se copian directamente a la nueva población. Estos individuos pueden perderse si por azar no se seleccionan para procrear, o si por la aplicación de la cruza o mutación se destruyen. Con el elitismo se puede mejorar el desempeño de un AG.
A continuación se describen a detalle cuatro métodos de selección importantes y representativos de todos los que existen:

2.4.7.1. Selección por Ruleta

La selección por ruleta es una de las técnicas de selección tradicionales de los GA's. El principio de este método es una búsqueda lineal a través de la

ruleta, cuyas rebanadas están asignadas en proporción a la aptitud asignada al indivi-duo. Se fija un valor de selección, que es una porción aleatoria de la suma de todas las aptitudes de la población. La población se recorre hasta llegar al valor de selección fijado.
Este es solamente un método de selección moderado, pues los individuos aptos no necesariamente son elegidos, pero ciertamente tienen más probabilidades de serlo. Un individuo apto va a contribuir más al valor de selección fijado, pero si no lo excede, el siguiente individuo en la ruleta tiene una oportunidad de ser elegido, y puede que éste sea poco apto. Es esencial que la población no sea ordenada con respecto a su aptitud, pues esto crearía un sesgo en la selección.
Este método se implementa siguiendo los siguientes pasos:

1. Sumar el total de los valores de aptitud, o valores esperados de selección de la población. Llámese T al valor obtenido.
2. Repítanse N veces los siguientes pasos (donde N es el tamaño de la población):
   - Escoger un valor entero r entre 0 y T.
   - Recorrer los individuos de la población, sumando sus valores de aptitud hasta que la suma sea mayor o igual que r. El individuo cuyo valor de aptitud ponga la suma sobre este límite se selecciona.

La selección por ruleta es sencilla de implementar pero genera ruido. Generar ruido es una expresión usada cuando los resultados de un método se alejan de los esperados. La tasa de evolución depende de la varianza de las aptitudes en la población.

2.4.7.2. Selección por Rango

La selección por ruleta tendrá problemas si los valores de aptitud difieren mucho. Si la aptitud del mejor cromosoma es 90 %, su rebanada ocupa el 90 % de la circunferencia de la ruleta, y los demás cromosomas tendrán pocas posibilidades de ser seleccionados. La selección por rango realiza un ordenamiento relativo de la población con respecto a los valores de aptitud (ranking), y asigna a cada cromosoma una aptitud de este ordenamiento. El peor individuo tiene aptitud 1 y el mejor N. Esto lleva a una convergencia lenta, pero evita una convergencia prematura. También mantiene la presión selectiva cuando la varianza de la aptitud es baja. Preserva la diversidad y por ello conduce a una búsqueda exitosa. Aquí, la presión selectiva está dada por el número de copias máximo del mejor individuo en la siguiente generación. El ordenamiento obedece a una función establecida, lineal o exponencial, por ejemplo, que define la dominación de los individuos más aptos sobre el resto de la población.

2.4.7.3. Selección por Torneo

Una estrategia de selección ideal debe permitir ajustar la presión selectiva y la diversidad de la población, para afinar la búsqueda del AG. A diferencia del método de selección por ruleta, la selección por torneo proporciona presión selectiva llevando a cabo un torneo competitivo entre N_u individuos. El mejor individuo del torneo es el que tiene el valor de evaluación más alto, es decir, el ganador de N_u. El torneo competitivo se repite hasta que la población intermedia que va a generar la descendencia está completa. La población intermedia tiene consecuentemente un valor de evaluación promedio más alto.
Este método conduce generalmente al óptimo global, y adicionalmente puede realizarse de manera paralela en varios procesadores, lo que permite manejar poblaciones muy numerosas. Por otra parte, intruduce un factor de determinismo al provocar que por lo menos una copia del mejor individuo pase a la siguiente generación, lo que puede ocasionar una convergencia prematura del algoritmo.

2.4.7.4. Muestreo Estocástico Universal

El muestreo estocástico universal proporciona un sesgo nulo y una mínima propagación. Los individuos se mapean a segmentos contiguos de una línea, tal que la longitud de cada segmento se corresponde con la aptitud del individuo, exactamente como en el caso de la selección por ruleta. Entonces, un conjunto de apuntadores se distribuye a intervalos iguales en la línea, tantos como el número de individuos que se desea seleccionar. El muestreo estocástico universal, asegura una selección de descendencia más cerca a la justa con respecto a la aptitud de cada individuo, que la que se observa en la selección por ruleta.
Sin embargo, al igual que en el método de selección por ruleta, si la aptitud del mejor individuo en la población está demasiado por encima de la del resto de los individuos, se puede generar una población en la cual haya demasiadas copias de este individuo, poca información genética para llevar a cabo la recombinación, y por lo tanto, se converja prematuramente a un óptimo local.

2.4.8. Cruza o recombinación

La cruza es el proceso de tomar dos soluciones padres y producir a partir de ellas una hija. Después de la selección, la población está enriquecida con individuos más aptos, éstos son clones de los originales pero no hay individuos nuevos. La cruza se aplica a esta población intermedia con la esperanza de crear una mejor descendencia.
La cruza es un operador de recombinación que opera en tres pasos:

1. El operador de reproducción selecciona aleatoriamente un par de dos cadenas individuales para la cruza.

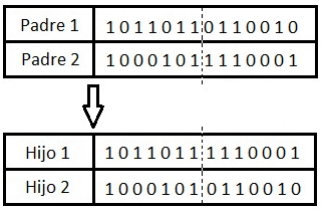
2. Un punto de corte se fija a lo largo de la longitud de la cadena. A este punto se le conoce como miasma en Genética.

3. Finalmente, los valores en las posiciones después del miasma se intercambian entre las dos cadenas seleccionadas.

El modo más simple de llevar a cabo la cruza es elegir aleatoriamente un punto de corte, y después copiar todo antes del punto de corte del primer padre y todo después del punto de corte del segundo padre, en el cromosoma del hijo. A continuación se describen algunos métodos de cruza importantes y significativos:

2.4.8.1. Cruza en un punto

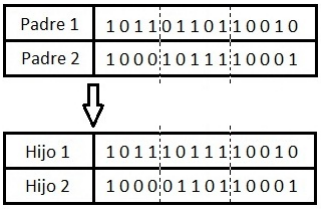
El AG tradicional utiliza cruza en un punto, donde los dos cromosomas apareados se cortan en un mismo punto y las regiones después del punto son intercambiadas. Aquí, el punto de corte se elige de forma aleatoria a lo largo de la longitud del cromosoma, que puede ser una cadena binaria, y los bits después del mismo se intercambian. Si el sitio del punto de corte se elijió bien, los descendientes serán mejores que los padres, de lo contrario puede perjudicar severamente la calidad de las cadenas.



2.4.8.2. Cruza en dos puntos

Además de la cruza en un punto, muchos algoritmos de cruza distintos han sido desarrollados, muchos involucrando más de un punto de corte. Debe notarse que añadir más puntos de corte reduce el desempeño del AG. El problema al añadirlos es que los *bloques constructores* (consultar en la sección 2.4.10. las definiciones de esquema y bloque constructor) son destruídos más constantemente. La ventaja de tenerlos es que el espacio de búsqueda puede explorarse más extensamente.
En la cruza por dos puntos, se eligen dos puntos de forma aleatoria y el contenido entre ambos se intercambia entre los padres. A diferencia de la cruza por un solo punto, el principio y final de un padre sí puede pasar directamente al hijo en este caso. Si ambas regiones, el principio y final de cromosoma, contienen buena información genética, en la cruza por un solo punto, ninguno de los dos descendientes obtendrá ambas buenas características. Usar cruza en dos puntos evita este problema, por esto es generalmente considerada mejor que la cruza por un punto. Esta observación puede generalizarse a la posición de cada gen en un cromosoma. Los genes que están cerca entre sí tienen más posibilidades de pasar juntos a la descendencia. Esto genera una correlación entre genes contiguos que no se desea tener. Por consecuencia, la eficiencia de la cruza en P puntos dependerá de la posición de los genes dentro del cromosoma. En una

representación genética buena, genes que codifican características dependientes de la solución deben estar juntos.

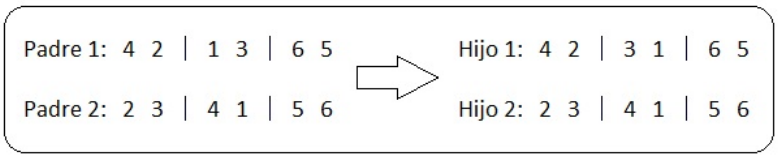


2.4.8.3. Cruza uniforme

La cruza uniforme es diferente a cruzar con P puntos. Cada segmento en el cromosoma de la descendencia se crea al copiar la región correspondiente de uno u otro padre, elegido de acuerdo a una máscara de cruza generada aleatoriamente de la misma longitud que un cromosoma. Lá máscara es una especie de esténcil. Cuando hay un 1 en la máscara, se copia el segmento del primer padre, cuando hay un 0 se copia del segundo.
Una nueva máscara de cruza se genera de manera aleatoria para cada par de padres. La descendencia por consiguiente tendrá una mezcla de genes de ambos progenitores. El número de puntos de corte efectivos no está fijo, pero debe promediar L/2, donde L es la longitud del cromosoma. Existe la posibilidad de producir dos descendientes, si se invierten los papeles de los padres con respecto a la máscara, es decir, si el primer hijo se copiaba del padre uno cuando en la máscara había un 1, el segundo hijo se copiará del padre dos. Si el primer hijo se copiaba del padre dos cuando en la máscara había un 0, el segundo hijo se copiará del padre uno.

2.4.8.4. Cruza ordenada

La cruza ordenada por dos puntos se utiliza cuando el problema está basado en permutaciones. Dados dos cromosomas padres, dos puntos de corte se fijan aleatoriamente generando una partición en tres regiones: izquierda, centro y derecha. La cruza ordenada por dos puntos se comporta de la siguiente forma: el primer hijo hereda su sección izquierda del padre uno, y su parte media queda determinada por los genes (que son valores) en la sección media del padre 1 en el orden en que aparecen en el padre dos. En la imagen de abajo se ejemplifica este método con dos permutaciones que representan a los padres.



2.4.9. Mutación

Después de la cruza, los cromosomas se mutan. La mutación evita que el algoritmo se atore un un óptimo local. La mutación juega el rol de recuperadora de información genética perdida y disruptora aleatoria de información genética. Es una póliza de seguro contra la pérdida irreversible de información genética. Se ha considerado tradicionalmente como un operador de búsqueda simple. Si la cruza debe aprovechar las soluciones actuales para encontrar mejores, la mutación debe explorar el espacio de búsqueda. También se considera un operador que en un segundo plano mantiene la diversidad genética de la población.
La mutación introduce nuevas estructuras genéticas en la población, al alterar aleatoriamente algunos de los *bloques constructores* presentes. Existen varios tipos de mutación dependiendo de la representación usada. Para cromosomas binarios, la mutación puede consistir en invertir el valor de cada bit con una pequeña probabilidad. La probabilidad usualmente tomada es $1/L$, donde L es la longitud del cromosoma. Puede emplearse una mutación que suceda solamente si mejora la calidad de la solución. Este tipo de operadores puede acelerar la búsqueda, pero igualmente si no se tiene cuidado, puede reducir la diversidad de la población y hacer que el algoritmo converja a un óptimo local. Estos son los conceptos clave que deben tomarse en cuenta cuando se habla de mutación en AG's:

Inversión o Flipping La inversión o flipping de un bit significa cambiar un 0 a 1 y un 1 a 0 en el cromosoma seleccionado para mutar.

Intercambio Dos posiciones aleatorias del cromosoma se eligen y los bits o valores correspondientes a éstas, se intercambian. Esta mutación es útil para problemas de permutaciones.

Reversa Se elige una posición aleatoria del cromosoma se elige, y se revierte el orden de los bits o valores adelante de la misma para producir al nuevo individuo mutado.

Probabilidad de mutación El parámetro importante en la técnica de mutación es la probabilidad de mutación (P_m). La probabilidad de mutación decide la frecuencia con que los componentes de un cromosoma deben mutar. Si no hay mutación, la descendencia se genera inmediatamente después de la cruza sin cambio alguno. Si la mutación se lleva a cabo, una o más partes del cromosoma se cambian. La mutación no debe ocurrir demasiado, pues entonces el AG se torna una búsqueda aleatoria común.

2.4.10. ¿Por qué funcionan los Algoritmos Genéticos?

Las heurísticas de búsqueda de los AG's están basadas en el teorema de esquemas de Holland. Un esquema se define como una plantilla que describe un subconjunto de cromosomas con secciones similares. Para codificación binaria, un esquema consiste de los bits 0, 1 y un meta caracter denotado: * ("no importa"

[6] si es 0 ó 1). Las plantillas son un medio efectivo para describir similitudes entre patrones en los cromosomas. Holland dedujo una expresión que predice la cantidad de copias de un esquema en particular que habrá en la siguiente generación, tras la aplicación de los operadores genéticos. Debe notarse que esquemas particularmente buenos se propagarán a las generaciones futuras. Por ello, los esquemas que son de *orden bajo* [7], están bien definidos y tienen un valor de aptitud por encima del promedio se denominan bloques constructores. Esto termina por implicar el principio de los bloques constructores para AG's: esquemas de orden bajo, bien definidos y de aptitud promedio se combinarán por medio del operador cruza para producir esquemas de orden alto, y aptitud por encima del promedio. Como los AG's pueden procesar muchos esquemas en un mismo ciclo evolutivo, se dice que tienen paralelismo implícito.

Ahora, antes de entrar en la explicación de la hipótesis de los bloques constructores, es necesario desarrollar de fondo algunos aspectos matemáticos muy importantes [7]:

Un esquema H describe un subconjunto de puntos en el espacio de búsqueda del problema con ciertas similitudes especificadas. En particular, si se tiene una población de cadenas binarias con longitud L cada una, sobre un alfabeto de tamaño $K = 2$ (solamente están en el alfabeto el 0 y el 1), entonces un esquema está identificado por una cadena de longitud L sobre un alfabeto extendido de tamaño $K + 1 = 3$ (se añade el símbolo *).

Habrán $(K + 1)^L$ esquemas de longitud L. Por ejemplo, cuando $L = 3$ y $K = 2$ habrán 27 esquemas diferentes. Una cadena del espacio de búsqueda pertenece a un esquema en particular, si para todas las posiciones $j = 1, ..., L$, el caracter hallado en la posición j de la cadena coincide con el caracter de la posición j del esquema, o si en esta posición hay un * en el esquema. Así, por ejemplo, las cadenas 010 y 110 pertenecen al esquema *10.

Hay una representación geométrica interesante de lo que sucede. Para $L = 3$ y un alfabeto binario es posible representar las $2^3 = 8$ cadenas como las esquinas de un hipercubo de dimensión 3, (véase la figura 2.3). Todos los esquemas que cubren las 8 cadenas, son a su vez entidades geométricas asociadas con el cubo arriba ilustrado. Particularmente cada uno de los 12 esquemas de orden 2 (con 2 bits fijos y un *), contienen dos puntos del espacio de búsqueda, y se corresponde con una de las 12 aristas en el cubo. Cada arista ha sido etiquetada con el esquema al que pertenecen los dos puntos que la delimitan.

Análogamente, a cada una de las seis caras en el cubo le corresponde un esquema de orden 1 que describe los cuatro puntos presentes en la misma. En el ejemplo se señala a la cara con esquema *0*.

Generalizando, a un esquema de orden $O(H)$, le corresponde un hiperplano de dimensión $L - O(H)$, en el hipercubo de dimensión L (la longitud de un individuo en la población de cadenas binarias).

Este esquema contiene a $2^{L-O(H)}$ individuos del espacio de búsqueda. El número

[6] Del inglés: "Don't care".

[7] El orden se refiere a la cantidad de bits fijos en la plantilla, es decir, que no son *.

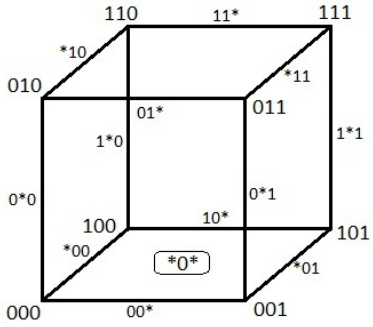


Figura 2.3: Representación geométrica del espacio de cadenas de longitud 3 en un lenguaje binario

de esquemas para un orden $O(H)$ es: $\binom{L}{O(H)} 2^{O(H)}$.

2.4.10.1. La hipótesis de los bloques constructores

La hipótesis de los bloques constructores es uno de los criterios más importantes de cómo es que un AG funciona. En el libro de Goldberg [13] se enuncia: "Un algoritmo genético logra un alto desempeño a través de la sobreposición de esquemas cortos, de bajo orden y aptitud alta, llamados bloques constructores". El concepto de lo que es un esquema con aptitud alta no está claro. La interpretación más obvia es que lo es si la aptitud promedio, de todas las cadenas que describe, está por encima del promedio de todas las cadenas presentes en el espacio de búsqueda. Esta versión de la hipótesis de los bloques constructores recibe el nombre de hipótesis estática de los bloques constructores. No es difícil encontrar contraejemplos para la misma.
Otra interpretación es que un esquema tenga aptitud alta si el promedio de las aptitudes de sus representantes en la población del AG durante algún ciclo evolutivo, es más alta que el promedio de todas las aptitudes de los individuos presentes en la población. Esta variante puede llamarse hipótesis relativa de bloques constructores.
El significado de la hipótesis de los bloques constructores se puede ilustrar considerando las "funciones trampa concatenadas" , que son un tipo de funciones de aptitud propuestas por Goldberg como problemas de prueba. Para cada función trampa, el cromosoma compuesto solamente por ceros es un óptimo global. Los esquemas que corresponden a estas cadenas son los bloques constructores. Por ejemplo, supóngase que la función trampa produce cadenas binarias aleatorias de longitud 4, y que se procede a concatenar 5 de estas cadenas para producir un individuo, que entonces tiene longitud 20. Los bloques constructores son los

esquemas:

$$000 * * * * * * * 0 * * * * * * * * *,$$
$$0 * * * 0000 * * * * * * * * * * * *,$$

etcétera. Si se producen inicialmente suficientes individuos, se espera que algunos estén en los esquemas de los bloques constructores, pero es improbable que exista un individuo que esté contenido en la mayoría de éstos. Para una población lo suficientemente grande, un AG que utilice el operador de cruza por un punto será suficiente para encontrar el óptimo global del problema.
Esta es una explicación un tanto superficial de por qué los AG's convergen a la solución óptima de un problema cuando están bien diseñados. Si el lector está interesado, puede consultarse el trabajo de Holland titulado: "Adaptación en sistemas naturales y artificiales" [12], que en las fuentes bibliográficas utilizadas está bien documentado ([6] y [7]), y se desarrolla con detalle el Teorema de esquemas y la asignación óptima de pruebas [8].

[8] Del inglés: Scheme Theoreme, Optimal Alocation of Trials.

Capítulo 3

Programación Genética

3.1. Concepto de Programación Genética

El paradigma de la Programación Genética, que de ahora en adelante se abrevia PG en esta tesis, continúa el camino marcado por los algoritmos genéticos para tratar problemas de optimización, incrementando la complejidad de las estructuras que llevan a cabo la adaptación. En particular, las estructuras que se adaptan en la programación genética son generales, programas de computadoras jerarquizados que varían dinámicamente de tamaño y forma.
En Inteligencia Artificial, Procesamiento Simbólico y Aprendizaje de Máquina muchos problemas aparentemente distintos, pueden reducirse a la búsqueda de un programa de computadora que produzca la salida deseada bajo ciertas entradas. La PG afirma que el proceso de resolver estos problemas se puede reformular como la búsqueda de un individuo altamente apto en el espacio de programas computacionales posibles. El espacio de búsqueda lo determinan las funciones y terminales apropiados, que forman a los programas para el dominio del problema a resolver. La PG proporciona una forma de encontrar a este individuo más apto. Muchas características propias de los Algoritmos Genéticos se heredan a la Programación Genética, desde el uso del principio de Darwin de la supervivencia del más apto. Más adelante en el desarrollo del algoritmo, se mostrará cómo un programa computacional que resuelve (o aproximadamente resuelve) un problema emerge de esta combinación de la selección natural de Darwin y operadores genéticos adaptados a estructuras más complejas.
La PG comienza con una población inicial de programas computacionales inicializados aleatoriamente compuestos de funciones y terminales apropiados al dominio del problema. Las funciones pueden ser operadores aritméticos estándares, operadores de programación comunes, funciones matemáticas, funciones lógicas, o bien funciones con dominio específico. Dependiendo del problema en particular, el programa de computadora puede tener valores booleanos, enteros, reales, complejos, vectoriales, simbólicos o múltiples (multivaluados). La creación de la población inicial es de hecho una búsqueda ciega en el espacio de programas de

computación asociados al problema.
Cada programa de computadora generado en la población, se mide en términos de qué tan bien se desempeña en el ambiente del problema a resolver. Esta *medida de aptitud* varía con el problema. En muchos casos queda determinada por el error producido al ejecutar el programa computacional obtenido, de modo que entre más cercano es el error a cero, mejor es el programa. Si el programador pretende reconocer patrones o clasificar ejemplos, la medida de aptitud de un programa puede ser el número de ejemplos que clasifica correctamente.
El proceso genético de reproducción sexual entre dos programas padres seleccionados en proporción a su aptitud, se utiliza para crear nuevos programas descendientes. Los programas padres son por lo regular distintos en tamaño y forma. Los programas hijos están compuestos de subexpresiones (subárboles, subprogramas, subrutinas, bloques constructores) de los padres. Esta descendencia típicamente es distinta en tamaño y forma de sus padres.
En la siguientes secciones, se pretende mostrar cómo esta técnica de programación produce al paso de las generaciones poblaciones con medida de aptitud promedio en ascenso. Adicionalmente los programas computacionales producidos pueden adaptarse a cambios en el medio de manera efectiva.
El comportamiento jerárquico de los programas que se producen mediante PG es una característica central de la misma. En muchos casos los resultados producidos son jerarquías por default, prioridades de tareas, o jerarquías en las que un comportamiento precede o suprime a otro.
La variabilidad dinámica de los programas producidos en el camino a una solución, también es una característica importante de la PG. Resulta difícil y contraproducente tratar de especificar o restringir el tamaño y forma por adelantado de la solución. Puede reducir la ventanta a través de la cual el sistema ve al mundo e impedir el hallazgo de un individuo bien adaptado.
Además, se tiene la ausencia o el bajo perfil que juega el preprocesamiento de entradas, y posprocesamiento de salidas. Las entradas, resultados intermedios y salidas, son expresados directamente en términos de la terminología natural al dominio del problema.
Finalmente, las estructuras que experimentan adaptación durante un algoritmo de PG son activas. No son codificaciones pasivas de la solución, pues son susceptibles a ser ejecutadas en su forma alcanzada.
Resumiendo, un programa genético consta de los siguientes pasos:

1. Generación de la población inicial, compuesta aleatoriamente de funciones y terminales del problema.

2. Iterar los siguientes pasos hasta que el criterio de terminación se satisface:

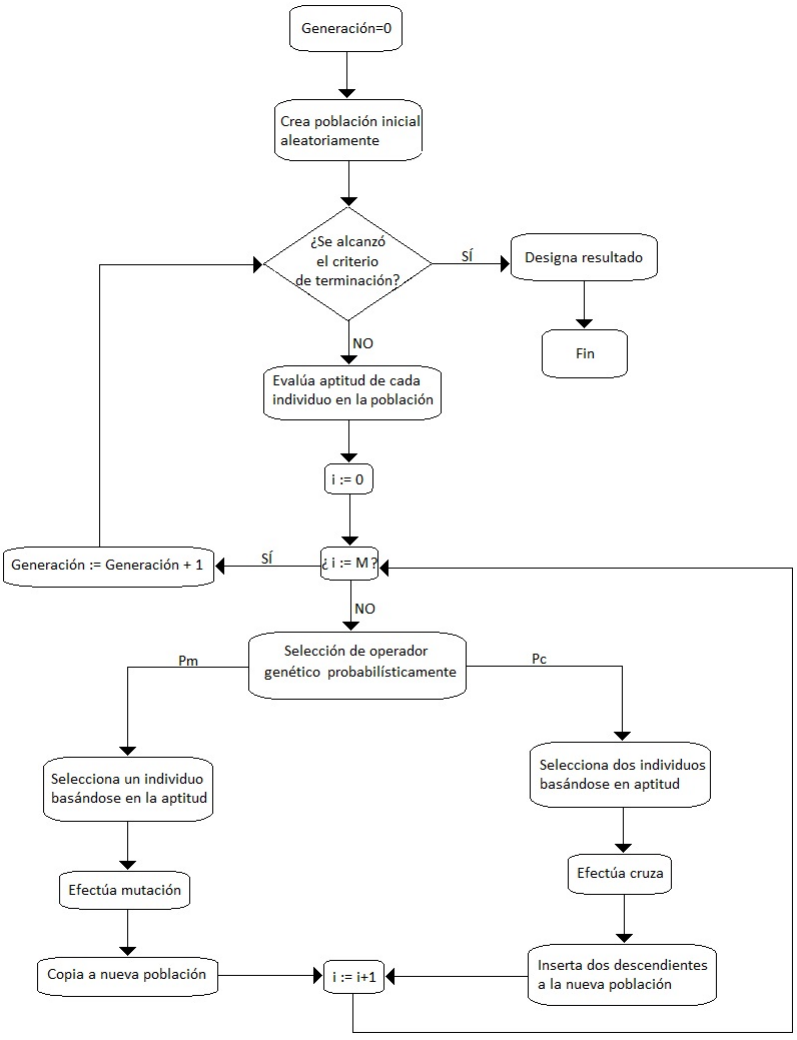
(a) Ejecutar cada programa en la población y asignarles una medida de aptitud de acuerdo a qué tan bien se desempeñan para resolver el problema.

(b) Crear una nueva población de programas de computación aplicando los dos operadores primarios siguientes, que se aplican a la población dependiendo en su aptitud.

(i) Copiar programas de computación existentes a la nueva generación.

(ii) Crear nuevos programas con recombinación genética de partes aleatoriamente elegidas de dos programas existentes.

El mejor programa computacional generado en cualquier generación, es designado como el resultado de la PG. Este resultado puede ser una solución (o una solución aproximada) al problema. Abajo se muestra un diagrama de flujo que sintetiza el paradigma de la PG.



3.2. Descripción detallada de la Programación Genética

Esta sección está dedicada a describir aspectos de la PG que son importantes y no están cubiertos con las descripciones del capítulo anterior. La PG, puede ser definida como una clase de Algoritmo Genético que usa codificación de árboles, pero por ser la técnica utilizada en la segunda parte de esta tesis, este tipo de codificación no se incluye en la sección dedicada a los AG's, y en conjunto, a la PG se le da un tratamiento más extensivo.

3.2.1. Las estructuras sometidas a adaptación

En todo sistema adaptativo o capaz de aprender, por lo menos una estructura es sometida a adaptación. Para el AG convencional y la PG, las estructuras que se adaptan son poblaciones de puntos individuales en el espacio de búsqueda. En el caso de la PG, cada individuo en la población es un programa computacional con estructura jerárquica. La forma y el tamaño de estos programas cambian dinámicamente durante el proceso.
El conjunto de estructuras posibles en la PG es el conjunto de todas las posibles composiciones de funciones, que pueden ser compuestas recursivamente del conjunto de N_{func} funciones de $F = \{f_1, f_2, ..., f_{N_{func}}\}$ y el conjunto de N_{term} terminales de $T = \{a_1, a_2, ..., a_{N_{term}}\}$. Cada función particular f_i en el conjunto F toma un número específico $z(f_i)$ de argumentos. Esto es, la función f_i tiene aridad $z(f_i)$.
Las funciones en el conjunto F pueden incluir:

- operadores aritméticos (+, -, *, etc.),
- funciones matemáticas (tales como sen, cos, exp, y log),
- operadores booleanos (tales como AND, OR, NOT),
- operadores condicionales (tales como IF _ THEN _ ELSE),
- funciones que causan iteración (tales como DO _ UNTIL),
- funciones que causan recursión, y
- cualquier función con dominio específico que deberá definirse.

Los terminales son típicamente átomos variables que representan las entradas del programa, sensores, detectores o variables de estado de algún sistema. También son átomos constantes, como el número 3.0 o la constante booleana FALSE. Ocasionalmente, los terminales son funciones que no toman argumentos explícitamente, y que su funcionalidad real está en los efectos laterales que tienen en el estado del sistema.
Considérese el conjunto de funciones $F = \{AND, OR, NOT\}$ y el conjunto terminal $T = \{D0, D1\}$, donde D0 y D1 son variables atómicas booleanas que

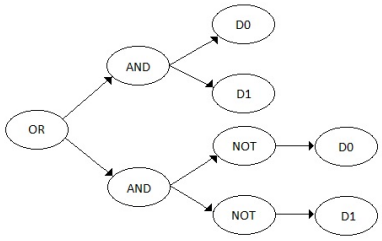
sirven de argumento para las funciones. Al combinar los conjuntos de funciones y terminales en uno solo se obtiene un conjunto que aquí se nombra C:

$$C = F + T = \{AND,\, OR,\, NOT,\, D0,\, D1\}$$

Puede verse a los terminales como funciones del conjunto C que requieren cero argumentos para evaluarse. Entonces los cinco miebros del conjunto C, pueden verse como funciones que toman 2, 2, 1, 0 y 0 argumentos respectivamente.
Como ejemplo, la función de equivalencia en álgebra boolena ($x \iff y$), que devuelve TRUE si los argumentos en ambos lados del operador son TRUE o FALSE ambos, y de otra forma devuelve FALSE. Esta función booleana en forma normal disjuntiva (FND) y notación prefija puede escribirse así:

$$(OR\,(AND\,(NOT\,D0)\,(NOT\,D1))\,(AND\,D0\,D1))$$

Además puede ilustrarse como un árbol como se muestra abajo.



Hasta este punto de la exposición ya se estableció que la PG trabaja con estructuras, y que dichas estructuras son susceptibles a ser evaluadas como se pueden evaluar expresiones aritméticas o booleanas, o ejecutarse como un programa computacional. Durante el proceso que se debe de llevar a cabo para ejecutar un programa escrito en un lenguaje computacional, está involucrado un paso importante, que es el análisis sintáctico.
El espacio de búsqueda en PG, puede verse como el espacio de árboles con raíz, de nodos etiquetados, con ramas ordenadas cuyas etiquetas que corresponden a los nodos internos son las funciones y las que corresponden a las hojas, los elementos terminales del conjunto combinado C. Ya puede notarse la existencia de jerarquía en las estructuras que se adaptan por medio de la PG. Esta es una diferencia central con aquellas estructuras que se adaptan por medio de un AG. En PG los conjuntos de funciones y terminales se seleccionan de modo que cumplan los requisitos de cerradura y suficiencia.

Cerradura de los conjuntos de funciones y terminales: La propiedad de cerradura requiere que cada función del conjunto de funciones pueda aceptar como sus argumentos, a cualquier valor y tipo de dato que con posibilidad pueda regresar alguna función dentro del conjunto de funciones, y cualquier valor y tipo de dato que pueda asumir un terminal en el conjunto de los terminales.
En apariencia, este requerimento haría imposible la producción de programas computacionales comunes por medio de PG, ya que combinan operaciones aritméticas con booleanas, cuando una operación aritmética

no puede aceptar variables o constantes booleanas como argumentos y viceversa. Además, muchas operaciones matemáticas comunes no están definidas para ciertos valores numéricos, como logaritmo de cero, o devuelven un valor no computable, como cuando se divide entre cero.
Con todo, la cerradura puede lograrse de una manera directa para casi todos los programas, manejando cuidadosamente un número limitado de situaciones según sea el caso. No se profundiza aquí en las técnicas empleadas, que involucran manejo de excepciones o escritura de funciones protegidas que checan los argumentos antes de evaluarlos.

Suficiencia de los conjuntos de funciones y terminales: La propiedad de suficiencia requiere que el conjunto de funciones y terminales sea capaz de expresar una solución al problema. El usuario de PG debe saber o creer que alguna composición de las funciones y terminales que proporciona, pueden conducir a una solución del problema. El paso de identificar variables que tienen suficiente poder explicativo para resolver un problema en particular, es común a todo problema científico. Este paso, dependiendo del problema, puede ser obvio o requerir una intuición más aguda. En algunos casos, el trabajo de identificar variables con suficiente poder explicativo resulta imposible.
Existen incluso sistemas que efectúan inducción de leyes naturales (científicas) a partir de datos empíricos. BACON, desarrollado por Langley y su equipo en 1987 [14] requiere que el usuario proporcione un repertorio de funciones, es decir, el conjunto de funciones que se usará, y que identifique las variables independientes del problema, que es el conjunto de terminales.

3.2.2. Estructuras iniciales

Las estructuras iniciales en PG consisten de los individuos en la población inicial, que están asociados a una expresión formada por elementos del conjunto combinado C de funciones y terminales, susceptible a representarse con una estructura discreta. Estas expresiones se conocen como expresiones-S por la palabra en inglés "structured" o estructurada.
La generación de cada expresión-S individual en la población inicial se realiza generando aleatoriamente un árbol con raíz, de nodos etiquetados con ramas ordenadas que la representa. Se comienza seleccionando una de las funciones del conjunto F aleatoriamente, (usando una distribución uniforme de probabilidades), para que sea la etiqueta de la raíz del árbol. Se elige un elemento del conjunto F, pues elegir una etiqueta del conjunto T para la raíz daría como resultado un árbol degenerado que consistiría de un solo nodo.
En la figura 3.1 al comienzo de la siguiente página, se muestra el comienzo de la creación de un programa con estructura aleatoria de árbol. La función '+' con aridad binaria (toma dos argumentos) fue seleccionada para ser la etiqueta del nodo raíz del árbol.
Siempre que un punto del árbol es etiquetado con una función f del conjunto

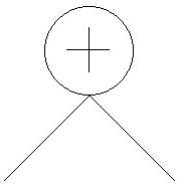


Figura 3.1: Raíz del árbol que representa al genotipo de un individuo

F, entonces $z(f)$ líneas, donde $z(f)$ es el número de argumentos que toma la función f, son creados para que partan del nodo etiquetado. Entonces, para cada una de las líneas debe elegirse un elemento del conjunto combinado C de funciones y terminales para etiquetar el nodo al que llegan. Si una función se elige para que etiquete el punto final de una de las nuevas líneas, el proceso descrito arriba continúa recursivamente. Si un terminal se elige para ser la etiqueta de cualquier nodo, ese nodo se torna una hoja del árbol y el proceso de generación se termina para este punto.
La profundidad de un árbol se define como la longitud de la trayectoria más larga desde la raíz hasta una hoja del mismo. El proceso generativo puede implementarse de varias formas, resultando en árboles iniciales aleatorios de varias formas y tamaños. Dos de los métodos básicos son los siguientes:

Generación por llenado Involucra la creación de árboles en los que la longitud de la trayectoria desde la raíz hasta cada hoja, es igual a una profundidad predefinida. Esto se logra restringiendo la elección de etiquetas al conjunto F, para los nodos cuya distancia a la raíz sea menor a la profundidad establecida, y del conjunto T para las hojas que cumplen con la profundidad máxima alcanzable.

Generación por crecimiento Involucra la creación de árboles con formas variables. La profundidad de los mismos no debe ser mayor a un valor predefinido. Esto se logra haciendo una selección aleatoria de etiquetas sobre el conjunto combinado C, de funciones y terminales, para nodos cuya profundidad no exceda el máximo, y restringiendo la elección a los elementos de T, para las hojas del árbol, en la profundidad máxima.

El número de elementos del conjunto de funciones F, y del conjunto de terminales T presentes en un individuo, determinan su profundidad, o en otras palabras, la distancia máxima entre la raíz y una de las hojas del árbol.
El método de generación considerado como el mejor en un amplio rango de problemas, es uno llamado: "Mitad y mitad escalonada". En programación genética usualmente no se conoce, o no se desea especificar el tamaño y la forma de una solución al comienzo del algoritmo. El método de mitad y mitad escalonada produce una variedad de árboles en cuanto a tamaño y forma se refiere.
Este método incorpora a ambas, la generación por llenado y por crecimiento. Requiere la creación de un mismo número de árboles usando como parámetro de

profundidad valores que van desde 2 hasta un máximo predefinido. Por ejemplo, si la profundidad máxima es 6, 20 % de los árboles tendrán profundidad 2, 20 % 3, y así hasta 6. Entonces, para cada valor de profundidad, 50 % de los árboles se crean por medio de generación por llenado y 50 % por medio de generación por crecimiento.

Nótese que para todos los árboles creados por medio de generación por llenado, para una profundidad específica, todas las trayectorias de la raíz del árbol a las hojas, tienen la misma longitud, y por lo tanto, todos los árboles tienen la misma forma. En contraste, para los árboles creados por medio de generación por crecimiento, el único control es la longitud máxima de una trayectoria desde la raíz hasta una hoja. Esto significa que para una profundidad máxima establecida, estos árboles pueden y seguramente variarán en forma. De ahí que el método de generación por mitad y mitad escalonada, creará árboles que serán variados en cuanto a forma y tamaño.

Individuos duplicados en la población aleatoria inicial, son inútiles, desperdician recursos computacionales y reducen la diversidad genética de la población. Es deseable pero no necesario evitar individuos duplicados en la población inicial. En PG, se observa que es más probable crear individuos duplicados en la población inicial cuando la profundidad máxima establecida es pequeña. A veces se recomienda checar cada expresión-S producida para que sea única en la población, antes de que se inserte. Ocasionalmente, se debe insertar un árbol de mayor tamaño cuando se han agotado todos los posibles árboles para cierta profundidad.

La variedad de una población es el porcentaje de individuos para los cuales no hay un duplicado exacto en la población. Si el chequeo de duplicados se lleva a cabo bien, la variedad de la población inicial es de 100 %. En generaciones subsecuentes, la creación de individuos duplicados es parte inherente del proceso genético.

Comparativamente, en el AG convencional con individuos de longitud fija, cada gen en los cromosomas de la población inicial es típicamente creado usando un valor binario aleatorio. Entre más grande es el cromosoma de un individuo y el espacio de búsqueda, menos necesario es checar si se crean individuos duplicados en la población inicial.

Opcionalmente se pueden sembrar individuos en la población inicial. Lo anterior, si no se realiza con cuidado y se introducen individuos con aptitud relativamente alta en la población inicial, creada aleatoriamente y por consecuencia con aptitud promedio baja, después de una generación resultará en casi una dominación total de la población por copias y descendientes de los individuos sembrados. En términos de diversidad genética, el resultado será después de una sola generación, similar a comenzar con un tamaño de población igual al número relativamente bajo de individuos sembrados. Si se pretende hacer este tipo de siembra, 100 % de la población inicial debería sembrarse con individuos con un nivel de aptitud parecido.

3.2.3. Aptitud

La aptitud es la fuerza que dirige la selección natural de Darwin en los Algoritmos Genéticos convencionales y la Programación Genética. En la naturaleza, la aptitud se traduce en la probabilidad con que un individuo llega a la edad reproductiva y tiene descendencia. Esta medida puede ponderarse para considerar el número de descendientes. En el mundo artificial de algoritmos matemáticos, medimos la aptitud en cierta forma y se usa esta medida para controlar la aplicación de operaciones que modifican estructuras en nuestra población artificial. La aptitud puede medirse de varias maneras. La más común es crear una medida explícita para cada individuo en la población. Esta aproximación se usa en la gran mayoría de aplicaciones de los AG's. Cada individuo en la población tiene asignado un valor escalar de aptitud, que produce un proceso evaluador explícito bien definido.
También puede calcularse la aptitud por un proceso co-evolutivo, como cuando la aptitud de la estrategia para un juego se calcula aplicando determinada estrategia contra una población de estrategias contrarias en el mismo juego.
El hecho de que un individuo sobrevive y se reproduce es indicativo de su buena adaptación. Esta definición implícita de aptitud es usada frecuentemente en la investigación de vida artificial. Aquí solo se describirán cuatro medidas explícitamente calculadas de la aptitud:

- aptitud en bruto,
- aptitud estandarizada,
- aptitud ajustada y
- aptitud normalizada.

3.2.3.1. Aptitud en bruto

Esta es la aptitud establecida en la terminología natural del problema mismo. Qué tan bien las expresiones-S evaluadas se adaptan al comportamiento esperado.
La aptitud es usualmente evaluada sobre un conjunto de *casos de aptitud*. Estos casos de aptitud proveen una base para evaluar la aptitud de las expresiones-S en la población, sobre un número de situaciones representativas diferentes suficientemente grande, tal que un rango amplio de aptitudes numéricas en bruto pueda obtenerse. Los casos de aptitud son generalmente un subconjunto finito del espacio de dominio, que puede ser enorme o infinito. En el caso de funciones booleanas con pocos argumentos, es práctico usar todas las combinaciones posibles de valores de los argumentos. Los casos de aptitud deben ser representativos del espacio de dominio en su totalidad, porque son la base para generalizar los resultados obtenidos.
Uno puede minimizar el efecto de seleccionar un conjunto particular de casos de aptitud, al cambiar el mismo para computar la aptitud de cada generación. Con todo y esto, el beneficio potencial de esta aproximación se ve reducido por

el inconveniente, de que el desempeño de un individuo pueda no ser comparable a través de las generaciones. Un buen diseño procura que los casos de aptitud sean elegidos al principio de cada corrida del algoritmo, y que no varíen de generación en generación.
La definición más común de aptitud en bruto, es que ésta significa error. Es decir, que la aptitud en bruto de una expresión-S individual es la suma de distancias, al valor correcto esperado. Si la expresión-S evaluada toma valores enteros, por ejemplo, la suma de distancias es la suma de los valores absolutos de las diferencias entre los números involucrados. Cuando la aptitud en bruto es error, se denota $r(i,t)$ para cualquier expresión-S individual i, al tiempo generacional t, para una población de tamaño M. La ecuación que debe resolverse es:

$$r(i,t) = \sum_{j=1}^{N_e} |S(i,j) - C(j)|$$

donde $S(i,j)$ es el valor que devuelve una expresión-S i para el caso de aptitud j (de N_e casos) y donde $C(j)$ es el valor correcto para el caso de aptitud j. Si la expresión-S toma valores booleanos o simbólicos, la suma de distancias es igual al número de valores que no coinciden con la salida esperada bajo cierta entrada.
Como la aptitud en bruto se expresa en la terminología natural del problema, el mejor valor será el menor o el mayor, dependiendo si la aptitud en bruto es el error o beneficio alcanzado por el individuo evaluado.

3.2.3.2. Aptitud estandarizada

La aptitud estandarizada $s(i,t)$ replantea la aptitud en bruto de modo que el mejor valor siempre es el más bajo. Si en un problema en particular, uno intenta minimizar el error, el individuo con valor de aptitud en bruto igual a 0 será el mejor. En este caso, el valor de la aptitud estandarizada es igual a la aptitud en bruto: $s(i,t) = r(i,t)$. Es conveniente y deseable hacer el mejor valor de la aptitud estandarizada igual a 0. Si este no es el caso, pueden ajustarse los valores sumando o restando una constante a todos los valores de aptitud.
Si en un problema en particular, la aptitud en bruto es mejor mientras mayor sea, la aptitud estandarizada se calcula a partir de la aptitud en bruto de esta forma:

$$s(i,t) = r_{max} - r(i,t)$$

Donde r_{max} es igual al máximo valor posible que alcanza la aptitud en bruto, y $r(i,j)$ es la aptitud en bruto observada al tiempo generacional t, para el individuo i. Para los casos en que no se tiene una cota superior para la aptitud en bruto, la aptitud estandarizada puede calcularse directamente de ésta tomando en cuenta los valores observados. Similarmente cuando no se tiene una cota inferior y el menor valor de aptitud en bruto es el mejor, los valores de aptitud se invierten y se procede igual.

3.2.3.3. Aptitud ajustada

La aptitud ajustada $a(i,j)$ se calcula a partir de la aptitud estandarizada $s\,(i,t)$ como se indica a continuación:

$$a(i,j) = \frac{1}{1+s(i,t)}$$

Aquí $s\,(i,t)$ es la aptitud estandarizada de un individuo i al tiempo t. El valor de la aptitud ajustada se encuentra entre 0 y 1, y es mayor para los mejores individuos en la población. Tiene el beneficio de exagerar pequeños cambios en la aptitud estandarizada a medida que ésta toma valores cercanos al 0, como frecuentemente ocurre en las últimas generaciones del algoritmo. Esta exageración es evidente cuando la aptitud estandarizada alcanza de hecho al 0, cuando una solución perfecta al problema se encuentra.
Debe notarse que para ciertos métodos de selección distintos a los proporcionales a la aptitud, como selección por torneo, la aptitud ajustada no es relevante y no se utiliza.

3.2.3.4. Aptitud normalizada

Si el método de selección empleado es proporcional a la aptitud, el concepto de aptitud normalizada $n\,(i,t)$ es necesario. Ésta se calcula a partir de la aptitud ajustada $a\,(i,t)$ para una población de tamaño M de esta forma:

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{M} a(k,t)}$$

La aptitud normalizada tiene tres características deseables: va de 0 a 1, es más grande en los mejores individuos de la población y la suma de los valores de las aptitudes normalizadas de toda la población es igual a 1.
Claramente, tampoco será relevante para métodos de selección que no son proporcionales a la aptitud como el torneo, y en esos casos no se debe usarse.

3.2.4. Selección glotona

Una población de tamaño 500 es suficiente para resolver casi todos los problemas que tratan de optimización en PG. Problemas más complejos requieren poblaciones más grandes para ser resueltos. Estos problemas generalmente conllevan un consumo de tiempo excesivo para calcular las aptitudes de los individuos. Esto aunado al incremento en el tamaño de la población resulta en algoritmos menos eficientes.
En PG y AG's es posible mejorar el desempeño de un algoritmo por medio de la selección glotona. Esto es, cuando los individuos son seleccionados de la población para que se les aplique un operador genético, los mejores individuos tienen una posibilidad todavía mayor a la que reciben con una asignación de aptitud normalizada [7].

3.2.5. Operadores primarios para modificar estructuras

Esta subsección describe los dos operadores primarios usados para modificar a las estructuras que deben adaptarse en la programación genética:

1. Reproducción de Darwin.
2. Cruza (recombinación sexual).

3.2.5.1. Reproducción

El operador de reproducción en la programación genética es el motor básico de la selección natural de Darwin, y de la supervivencia del más apto. Este operador es asexual, por lo que opera sobre solo una expresión-S padre y produce solo una expresión-S hija en cada aplicación. Consiste en dos pasos: primero, una expresión-S se selecciona de la población de acuerdo a algún método de selección basado en la aptitud. Después el individuo seleccionado es copiado, sin alterarlo, de la población actual a la siguiente generación.
Ya se mencionaron en el capítulo anterior algunos métodos de selección. En este trabajo de tesis se presta especial atención al método de torneo, que no es proporcional a la aptitud. En él, un número específico de individuos (generalmente dos) se seleccionan aleatoriamente de la población, y el de mejor aptitud es seleccionado.
Los padres pueden y generalmente son seleccionados más de una vez en la reproducción de la generación actual. Se puede ahorrar una cantidad considerable de recursos computacionales, si no se calcula la aptitud de cada individuo que aparece en la generación actual como resultado de reproducción de la generación anterior. Su aptitud no cambiará y por ello no es necesario que se vuelva a calcular, a menos que los casos de aptitud varíen de generación en generación. Esto puede ahorrar en promedio un 10 % de operaciones.

3.2.5.2. Cruza

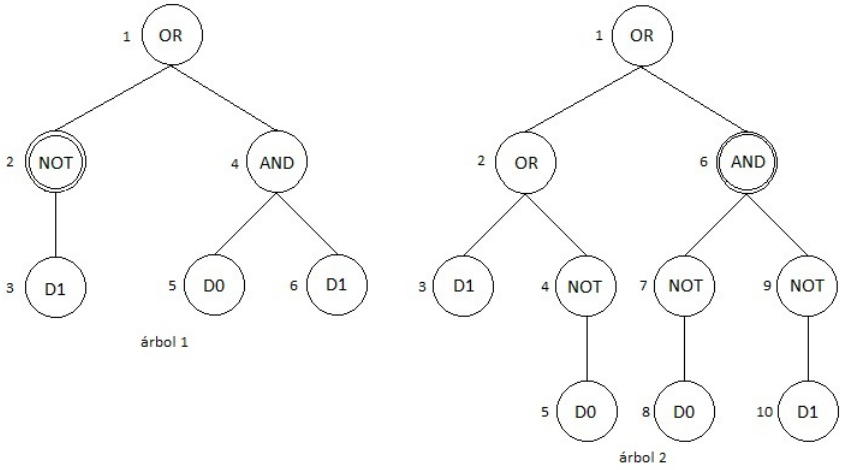
La cruza o recombinación sexual en programación genética crea variación en la población, introduciendo unos cuantos individuos que contienen partes tomadas de cada padre. El operador de cruza comienza con dos expresiones-S padres. Retomando la terminología biológica, dado que este método de recombinación genética involucra a más de un progenitor, se le conoce como un operador sexual.
Los padres son elegidos de la población intermedia generada por medio del operador de reproducción aplicado a la población actual. La operación comienza al seleccionar independientemente, usando una distribución uniforme de probabilidad, un nodo de cada padre que será el punto de cruza para éste. Es típico que los padres no tengan el mismo tamaño.
El fragmento de cruza de un padre en particular es el subárbol que contiene su raíz, y excluye al subárbol que está por debajo del punto de cruza seleccionado. Este último se conoce como fragmento de recombinación.

La primera expresión-S hija se produce borrando el fragmento de recombinación del primer padre e insertando el fragmento de recombinación del segundo padre en el punto de cruza del primero. La segunda expresión-S hija se produce de forma simétrica. Como ejemplo considérense las siguientes expresiones-S que son funciones booleanas en notación prefija:

$$E1 : (OR\,(NOT\,D1)\,(AND\,D0\,D1))$$

$$E2 : (OR\,(OR\,D1\,(NOT\,D0))\,(AND\,(NOT\,D0)\,(NOT\,D1)))$$

Aparecen las funciones AND, OR y NOT y los terminales que son los argumentos booleanos D0 y D1. Abajo pueden verse las estructuras de árbol que se corresponden con cada expresión.

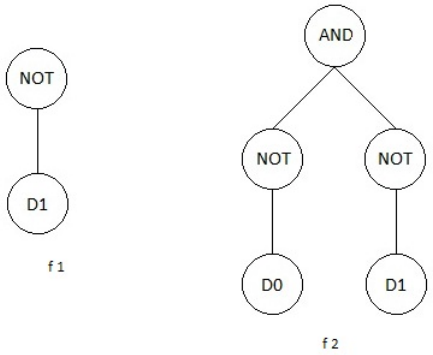


Asúmase que los nodos de ambos árboles están numerados a profundidad de izquierda a derecha, y que los nodos 2 y 6 del primer y segundo árbol respectivamente son seleccionados aleatoriamente. Esto quiere decir que los puntos de cruza son la función NOT y la función AND para el primer y segundo árbol respectivamente. Los correspondientes fragmentos de recombinación serían los siguientes:

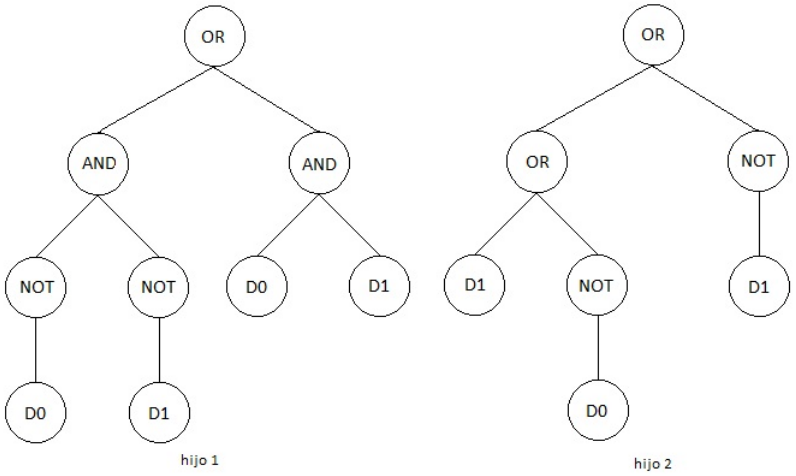
$$f1 : (NOT\,D1)$$

$$f2 : (AND\,(NOT\,D0)\,(NOT\,D1))$$

Los subárboles asociados se muestran abajo.



Los descendientes que resultan de efecuar la cruza serán entonces los siguientes:



Es importante señalar que debido a la cerradura del conjunto de funciones, se producen expresiones sintácticamente correctas. Además es preferible seleccionar como punto de cruza siempre un nodo interno del árbol, es decir, que no sea ni la raíz ni una hoja. Es sencillo ver la razón de esto, puesto que al seleccionar la raíz de un árbol como punto de cruza, se inserta a partir de la misma un subárbol que proviene de la pareja en el proceso de cruza, perdiendo de esta forma información del padre que pudiera ser buena. Cuando se selecciona una hoja en cambio, se lleva a cabo una operación idéntica a la llamada "mutación puntual", que se ve a detalle más adelante.
Cuando un individuo se cruza insestuosamente con otro que es idéntico a él, la descendencia en PG será generalmente diferente. Esto porque los puntos de cruza seleccionados son casi siempre distintos. Este hecho es de gran impor-

tancia cuando se desea contrastar un algorimto de programación genética, con un AG clásico, con individuos de longitud fija. En el AG clásico, la cruza incestuosa puede conducir a una convergencia prematura, dando como resultado un subóptimo global. En PG, la cruza entre dos individuos idénticos produce descendientes diferentes a los padres, exceptuando el caso poco probable en el que los puntos de cruza seleccionados de forma aleatoria en ambos progenitores son los mismos.
Para tener un control sobre los recursos computacionales y de tiempo que se utilizarán al ejecutar un algoritmo de programación genética, usualmente se fija un tamaño máximo de los individuos, esto en términos de su profundidad. Existen diferentes formas de llevar a cabo este control. Una muy simple es abortar la producción de un descendiente cuando se detecta que éste va a exceder la profundidad máxima permitida, copiando al padre en la nueva generación.

3.2.6. Operadores secundarios

Existen en la programación genética además de los operadores genéticos primarios, cinco operadores secundarios que son opcionales que se mencionan en este trabajo [7]:

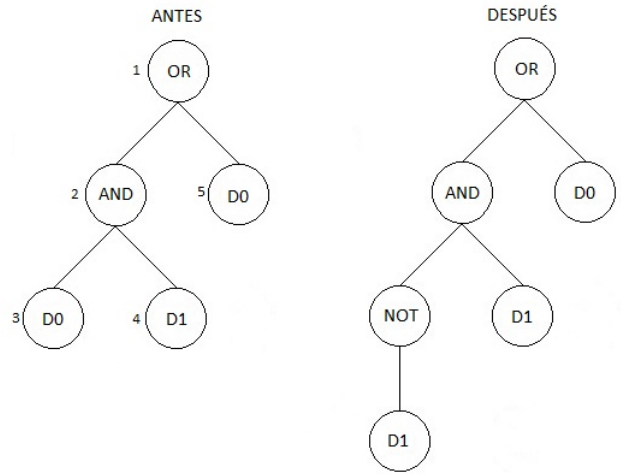
- mutación,
- permutación,
- edición,
- encapsulamiento, y
- destrucción.

La aplicación de cada uno depende del tipo de problema que se pretende resolver.

3.2.6.1. Mutación

El operador de mutación introduce cambios aleatorios en las estructuras presentes en la población. En algoritmos genéticos convencionales que operan con individuos codificados en cadenas lineales, la mutación beneficia a la población reintroduciendo diversidad genética, evitando así una convergencia prematura. Por otra parte, para problemas con espacios de búsqueda no lineales, se observan casos en los que un fragmento de cromosoma que desaparece en generaciones tempranas por medio de mutación, es requerido en generaciones posteriores para alcanzar un resultado óptimo. Es importante reconocer que el operador de mutación tiene poca importancia relativa en el AG convencional, y que está asociado a una técnica de búsqueda semialeatoria para probabilidades de mutación menores al 100 %, y a una búsqueda ciega para el caso en que todos los individuos mutan en una generación.
La mutación es un operador genético asexual, es decir, que opera en solamente un individuo, que se selecciona en base a su aptitud normalizada.

Similarmente como se procede en la cruza, debe seleccionarse al azar un punto de mutación dentro de la expresión-S asociada con el individuo seleccionado. Este punto puede ser interno (una función), o una hoja (un terminal). El operador de mutación borra lo que sea que está por debajo del punto de mutación seleccionado, e inserta un subárbol generado aleatoriamente en ese punto.
Esta operación se controla por un parámetro que define el tamaño máximo, en términos de la profondidad, para el subárbol creado aleatoriamente y que se insertará en el punto de mutación.
Un caso especial de mutación ocurre cuando se inserta un terminal en el punto de mutación aleatorio. Esta mutación puntual ocurre ocasionalmente durante el operador de cruza cuando ambos puntos de cruza son ambos terminales.
Como ejemplo vemos en la imagen de abajo, el diagrama titulado con "ANTES", cuyo nodo 3 está seleccionado como punto de mutación, en el que se encuentra el terminal D0, y el subárbol generado de forma aleatoria (NOT D1), que es insertado para producir al individuo con la expresión-S que se representa con el árbol titulado "DESPUÉS".



A diferencia de los efectos positivos que tiene la mutación en los AG's clásicos, en PG es realmente improbable que una función o terminal desaparezca por completo de una población. Esto reduce la importancia del operador de mutación, como restaurador de la diversidad genética en una población.

3.2.6.2. Permutación

Para explicar cómo funciona este operador secundario de la Programación Genética, es necesario mencionar el operador de inversión en Algoritmos Genéticos. Éste funciona reordenando los valores entre dos puntos del cromosoma invirtiendo su orden. Esto acerca algunos alelos, mientras que aleja a otros. Cuando se aplica a individuos con buena aptitud relativa, el operador de inversión ayuda a establecer relaciones genéticas estrechas entre combinaciones de alelos que se desempeñan bien cuando están juntos dentro del cromosoma. Se debe recordar aquí, que en AG's los alelos tienen significado porque ocupan un lugar específico

en el cromosoma.
La permutación es un operador asexual. Comienza seleccionando un punto interno de la expresión-S seleccionada, si la función en el punto seleccionado tiene k argumentos, se debe seleccionar una de las posibles $k!$ permutaciones de los mismos. Si la función en el punto seleccionado es conmutativa, el operador de permutación no tendrá efecto alguno.
La utilidad del operador de permutación no se ha demostrado de forma conclusiva.

3.2.6.3. Edición

El operador de edición provee una forma de editar y simplificar una expresión-S mientras la ejecución se lleva a cabo. Es un operador asexual. Funciona aplicando recursivamente un conjunto de reglas dependientes e independientes del dominio a cada expresión-S de la población.
La regla universal independiente del dominio es la siguiente: si una función no tiene efectos laterales, no depende del contexto y toma como argumentos valores constantes, el operador de edición debe evaluar la función y sustituirla con el valor obtenido. Por ejemplo, la expresión aritmética en preorden: (+2 1) debe ser sustituida por el valor 3.
Además, el operador de edición aplica un conjuto de reglas establecidas para cada dominio en específico. Para el dominio de los problemas numéricos, se podría insertar un 0 cada vez que una subexpresión es sustraída de sí misma, o la expresión cuando se le suma cero o se multiplica por 1. En un dominio booleano, uno podría aplicar reglas de edición como las siguientes:

$$(AND\, X\, X) \rightarrow X$$
$$(OR\, X\, X) \rightarrow X$$
$$(NOT(NOT\, X)) \rightarrow X$$

Estas reglas son conocidas como tautologías en la lógica de primer orden y se utilizan para simplificar expresiones, se podrían aplicar también las leyes de De Morgan.
La aplicación recursiva de la edición consume muchos recursos computacionales y tiempo. No tiene equivalente en AG's, donde los individuos ya codificados tienen longitud fija y son de complejidad estructural uniforme. Se puede aplicar con dos motivaciones; la primera es cosmética y consiste en producir resultados más legibles, la segunda persigue mejorar la eficiencia global del programa, al hacer que éste procese individuos más simples. Este operador se controla mediante un parámetro de frecuencia, que especifica si el operador de edición debe o no aplicarse a determinada generación.
Existe un argumento en contra del uso de este operador, y dice que puede degradar la eficiencia del programa reduciendo prematuramente la variedad de estructuras disponibles para la recombinación. Como sea, el efecto del operador de edición es muy poco claro.

En esta subsección se introdujo un concepto importante en Programación Genética, que es el de los *introns*. Se trata de una comparación más con un concepto presente en la Genética de los seres vivos, y se refiere a la aparición de código redundante en el genotipo de un individuo. Sin embargo, en los seres vivos, los introns pueden ser varias expresiones que no necesariamente pueden simplificarse a una expresión más corta, como sucede en los dominios típicos de la Programación Genética.

3.2.6.4. Encapsulación

La encapsulación sirve para identificar automáticamente un subárbol potencialmente útil, y darle un nombre para hacer referencia a él y usarlo después.
Un asunto clave en Inteligencia Artificial y aprendizaje de máquinas, es cómo escalar técnicas prometedoras que tienen éxito resolviendo subproblemas, para resolver así el problema mismo. Una forma de resolver un problema grande es descomponerlo en problemas jerárquicamente menores. Identificar estos subproblemas que descomponen al problema original en partes útiles es un paso vital. Es esto lo que por medio de la Inteligencia Artificial y el aprendizaje de máquinas se pretende automatizar.
La encapsulación es un operador genético asexual. El individuo a operar se selecciona generalmente con una probabilidad proporcional a la aptitud normalizada. Comienza seleccionando un nodo interno del árbol asociado a la expresión-S aleatoriamente, es decir, una función de la expresión-S. El resultado de la operación es una expresión-S hija, y una nueva definición de subárbol.
Durante la operación se remueve el subárbol que está por debajo del punto seleccionado, y crea una función nueva para permitir referencias al subárbol elegido, en otras palabras, se agrega un elemento nuevo al conjunto de funciones F. Las nuevas funciones encapsuladas no tienen argumentos, y son nombradas E_o, E_1, E_2, ... según se van creando. Dependiendo del problema, para mayor eficiencia las funciones encapsuladas pueden ser compiladas y en su lugar insertarse directamente el código de máquina que les corresponda.
Lo anterior implica que a través del operador de mutación, las funciones recientemente encapsuladas pueden ser insertadas en el individuo mutado. Además, por tratarse de un punto indivisible de un árbol, una función encapsulada no puede ser objeto del efecto destructivo que puede tener la cruza.
En trabajos tempranos de Koza, el operador de encapsulamiento fue conocido como la operación de definir bloques constructores.

3.2.6.5. Destrucción

Para algunos problemas complejos, la distribución de valores de aptitud puede estar sesgada, de modo que un gran porcentaje de los individuos pueden tener muy poca o nula aptitud en bruto. Este sesgo puede ocurrir en poblaciones en las que los individuos pueden recibir una penalización en su aptitud, porque de otro modo consumirían una cantidad infinita de tiempo en ser evaluados, como se observa en problemas de control o aquellos que involucran ciclos itera-

tivos. Más aún, cuando ocurre una distribución de aptitudes altamente sesgada, los pocos individuos con una aptitud marginalmente mejor, comienzan dominando la población y la diversidad genética cae.
El operador de destrucción ofrece una forma más rápida que la cruza para enfrentar el problema expuesto antes. Se controla mediante dos parámetros, un porcentaje y una condición que especifica cuándo debe de emplearse. Por ejemplo, el porcentaje puede ser 10 % y el operador puede invocarse en la generación 0. En ese evento, inmediatamente después del cálculo de la aptitud para la generación 0, todos los individuos excepto el 10 % son eliminados. Empleado en la generación 0 con un porcentaje de 10 %, se recomendaría generar al principio 10 veces más individuos de los que se desearía tener en las generaciones siguientes. La selección de los individuos en la operación de destrucción, se realiza de forma probabilística basándose en la aptitud. La reselección no está permitida, para maximizar la diversidad en la población.

Parte III

Desarrollo

Capítulo 4

Diseño del algoritmo

4.1. Planteamiento del problema

Se ha establecido en los capítulos anteriores la teoría básica para sustentar el desarrollo de un algoritmo que mediante la Programación Genética, cree estructuras capaces de analizar sintácticamente[1] varias oraciones del lenguaje natural, que a su vez tengan asociadas gramáticas libres de contexto.
Resulta fundamental en este punto aclarar que no se pretende generar una gramática de ese tipo para el lenguaje natural, que como se sabe, es dependiente del contexto. Más aún, la técnica de programación elegida siendo no determinista, puede arrojar distintos resultados en cada ejecución, dejando la pregunta abierta de cuál de esas estructuras generadas es la correcta y por qué razón.
Gramáticas libres de contexto para contextos específicos, han sido utilizadas con éxito en el Procesamiento de Lenguajes Naturales, para sistemas que utilizan un subconjunto bien definido de oraciones, como lo puede ser por ejemplo, un sistema que proporciona el horario de llegadas de un aereopuerto. Puede consultarse el libro de Jurafsky [4] sobre PLN y Lingüística Computacional.
Aquí podría plantearse la siguiente pregunta: ¿Cuál es la motivación del uso de Programación Genética, para elaborar un programa que resuelva un problema que bien podría enfrentarse con un sistema que utilice un conjunto finito de reglas?.
En efecto, para el mismo subconjunto de oraciones en español que se pretende analizar, podría programarse un módulo bastante simple en Perl, por nombrar un lenguaje eficiente para la tarea, que utilizando una tabla donde se almacene la oración leída, proporcione la respuesta adecuada. La cuestión es esta: ¿Qué pasaría si al programa se le presentara una oración que no está presente en la memoria?
Mientras que con un conjunto finito de reglas se posee una memoria estática, la

[1] En inglés "parse" significa analizar, es parte importante del proceso de traducir una expresión al lenguaje de máquina.

Programación Genética permite la creación de estructuras modificables, susceptibles en este caso, de ser guardadas cada vez que se enfrentan a una estructura gramatical que desconocen. La Programación Genética se gana su clasificación dentro de las técnicas de programación propias del Aprendizaje de Máquinas.
Este trabajo busca precisamente la implementación de un algoritmo, capaz de aprender primero un conjunto de oraciones de entrenamiento sin relación entre ellas. Después de hacer observaciones sobre los resultados obtenidos en esta primera fase, se procederá a que el algoritmo pueda asociar una respuesta a la entrada proporcionada por un usuario, dependiendo del resultado de analizar la misma.

4.2. Hipótesis y soluciones esperadas

Como hipótesis para estre trabajo se tiene que el Cómputo Evolutivo, será capaz de producir y almacenar un conjunto de conocimientos, que en forma de una estructura de árbol, asociada a una gramática libre de contexto, sean capaces de procesar oraciones del español. La gramática asociada a las estructuras obtenidas, tiene limitaciones en su capacidad para capturar la complejidad del lenguaje natural, pero servirá como representación interna de un subconjunto de producciones escritas que el usuario proporcione en una sesión de entrenamiento.

4.3. Primera fase de desarrollo

La inducción gramatical consiste en la construcción gradual de una gramática libre de contexto basada en un conjunto de expresiones como muestra. Esta sección está basada en el artículo de Tony C. Smith e Ian H Witten de la universidad de Waikato, en Hamilton, Nueva Zelanda [2].
Como ya se mencionó en la introducción de esta tesis, el uso del Cómputo Evolutivo como técnica de programación exhibe ventajas claras sobre aproximaciones probabilísticas puras, entre las cuales destaca la capacidad de inferir conjuntos de reglas y categorías sobre las oraciones proporcionadas como entrada, de manera simultánea. Esto debido a la sensibilidad estadística propia de los individuos presentes en un algoritmo evolutivo, que pueden poseer patrones frecuentes útiles y reforzarlos durante la selección aleatoria de nodos.
Algunos esfuerzos históricos por inducir gramáticas libres de contexto con AG's han resultado exitosos. En 1992, Koza escribió un algoritmo para detectar exones con cinco bases nitrogenadas de longitud en el ADN[2]. Tras 35 generaciones, el árbol resultante de 61 nodos era 100 % efectivo para identificar todos los exones en un segmento de 1000 bases nitrogenadas. Por otra parte, en 1991 Wyard escribió un AG para inducir la gramática del lenguaje de los paréntesis bien balanceados con anidaciones permitidas, por ejemplo la cadena: ()(())((())).

[2]Un exón es un segmento de ADN que no se rompe durante el proceso de segmentación y empalme, por ello se mantienen en el ARN maduro. Además, en los genes que codifican una proteína cada exón codifica una parte específica de la proteína completa.

Este algoritmo obtuvo una gramática correcta y concisa en solamente 3 generaciones para un conjunto de prueba que contenía 20 cadenas [7]. Wyard también intentó producir un algoritmo que indujera la gramática para el lenguaje que contiene a las cadenas con el mísmo número de A's y B's, que es más complejo. En este caso no fue posible producir una gramática exitosa, y se observaba que el modelo se estancaba en máximos locales.
En el desarrollo de un AG para la inferencia o inducción de una gramática libre de contexto que describiera un lenguaje más complejo, como el inglés en el caso del artículo de Smith y Witten, se retomó la representación de Koza de una gramática por medio de una expresión-S formada de los operadores lógicos AND, OR y NOT en el conjunto de nodos no terminales, y las palabras presentes en la oración para los nodos terminales.
El algoritmo desarrollado aquí solamente utilizará los operadores AND y OR. La exclusión del operador NOT se debe a que desde el comienzo, se pretende apuntar las estructuras sometidas a la evolución, para que modelen gramáticas libres de contexto, y en las reglas de producción de dichas gramáticas se consideran solamente productos válidos. Además, no se tiene pensado aplicar un operador secundario de edición sobre la población, con lo que el operador NOT podría introducir redundancias y extensiones innecesarias en los genotipos de la población.
En este punto para mejorar la exposición, resulta necesario introducir algunos conceptos propios de la Teoría de Compiladores. Esta materia por sí sola es una rama de las Ciencias de la Computación que abarca a su vez varias áreas en común con las demás, por esta razón y porque la parte teórica de este trabajo ya se ha concluído, los conceptos se exponen durante la especificación del algoritmo como pies de página. Para estudiar los temas a detalle se recomienda consultar el libro sobre compiladores de Keith D. Cooper y Linda Torczon[8].
Muchas gramáticas en Forma Normal de Chomsky, se traducen fácilmente a expresiones-s lógicas. Por ejemplo, la gramática que se muestra al comienzo de la siguiente página, en FNC y en la forma de Backus-Naur[3]:

[3]La notación tradicional para representar una gramática libre de contexto se llama forma de Backus-Naur, o BNF por su nombre en inglés. En este trabajo, un símbolo no terminal comienza con mayúscula, un terminal solo consta de minúsculas y el signo $'\rightarrow'$ quiere decir "deriva", e indica las reglas de producción.

Las gramáticas libres de contexto en Forma Normal de Chomsky tienen reglas de producción de la siguiente forma:

$$\begin{aligned} A &\rightarrow \quad B \quad C \\ A &\rightarrow \quad \alpha \\ S &\rightarrow \quad \varepsilon \end{aligned}$$

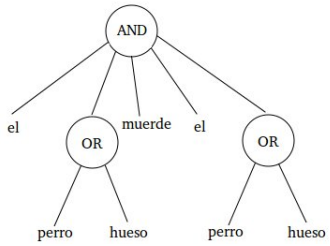
En esta gramática los símbolos 'A', 'B' y 'C' son no terminales, 'B' y 'C' deben ser distintos del símbolo inicial, S es el símbolo inicial, α es un terminal y ε es la cadena vacía. La tercera producción aparece si ϵ está en el lenguaje. Cualquier gramática libre de contexto puede ser transformada a FNC [5].

$$
\begin{aligned}
O &\rightarrow S \quad P \\
S &\rightarrow Art \quad Sust \\
P &\rightarrow Verbo \quad S \\
Art &\rightarrow el \\
Verbo &\rightarrow muerde \\
Sust &\rightarrow perro \\
Sust &\rightarrow hueso
\end{aligned}
$$

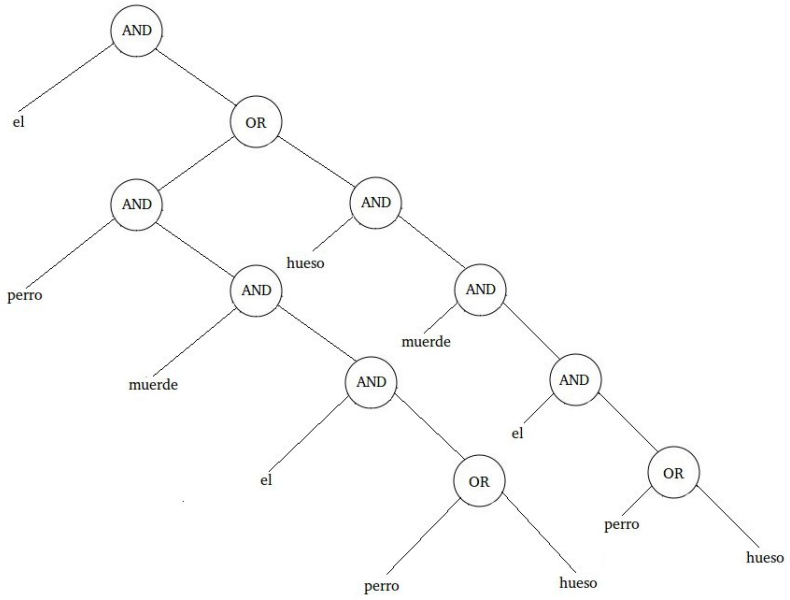
Se expande a la siguiente gramática:

$$
\begin{aligned}
O &\rightarrow Art\,Sust\,Verbo \quad Art\,Sust \\
Art &\rightarrow el \\
Verbo &\rightarrow muerde \\
Sust &\rightarrow perro \\
Sust &\rightarrow hueso
\end{aligned}
$$

Está asociada a la siguiente expresión-S lógica en caso de que no haya restricciones al número de argumentos que recibe el operador AND:



En este trabajo, se limita al operador AND para que reciba solamente dos parámetros y el primero sea siempre una palabra, contenida en el conjunto de los símbolos terminales. Al operador OR de opción, se le permite recibir un número de parámetros que varíe entre 1 y el número de símbolos terminales presentes en esa iteración de la ejecución del algoritmo, y además no puede recibir como parámetro a más operadores OR. La razón de requerir estas limitaciones tiene que ver con el tipo de gramáticas libres de contexto que se desea producir, cuestión a la que se regresa más adelante. Con lo anterior, la gramática libre de contexto del ejemplo, estaría asociada a la siguiente expresión-S lógica:



Aquí es importante señalar que esta representación no es una verdadera expresión-S lógica, simplemente porque el operador AND se usa para indicar concatenación, e impone un orden estricto a sus argumentos. El operador OR indica que cualquiera de sus parámetros puede sustituir a todo el nodo, es decir, es un operador de opción. Lo establecido en estos párrafos será de gran importancia para el resto de este trabajo.
Además, no es casual que se limitara la aridad del operador AND a dos argumentos. Comparando las dos imágenes anteriores se ve claro que en la segunda el árbol tiene más profundidad que en la primera. La jerarquización es un efecto deseado en la Programación Genética. Esto va a permitir que se extraiga más y mejor información de las oraciones que se pretende analizar sintácticamente.
Ahora bien, una vez que el algoritmo, logra producir una estructura capaz de analizar sintácticamente una oración del conjunto de prueba, esta estructura debe de **unificarse** al árbol logrado hasta este punto, que será entonces capaz de analizar sintácticamente todas las oraciones procesadas. La necesidad de introducir un operador de unificación de esta clase, se explica de manera sencilla: pensemos que si se pretendiera conseguir un individuo capaz de analizar sintácticamente todas las oraciones en el conjunto de prueba, a través de la aplicación de operadores genéticos que funcionan de manera aleatoria, el nodo raíz sería especialmente sensible, al proporcionar la primera elección de derivación.
Por otra parte, el operador de unificación asegura que tras procesar todas las oraciones se obtendrá una gramática cuyos componentes fueron las estructuras, que estadísticamente se desempeñaron mejor en una población de individuos a

los que se les pedía analizar una oración en particular.
Adjunto a este trabajo puede encontrarse un disco, en el cual hay una carpeta con el nombre de "Clases". Dentro de esta carpeta, pueden consultarse los archivos: **dialogo.txt** y **dialogoINFO.xml**. En el primero se encuentra un conjunto de oraciones que se procesó para ser mostrado a manera de ejemplo, y en el archivo con extensión xml, está la estructura que el algoritmo produjo. Cabe aclarar que el proceso por medio del cual se genera esta estructura se explica en las siguientes subsecciones, y que los archivos con extensión xml están explícitamente estructurados como árboles.

4.3.1. Aptitud de los individuos

Debido a que se pretende generar una estructura asociada a una gramática libre de contexto, que modele un subconjunto del lenguaje objetivo donde estén las oraciones analizadas, es más apta una estructura que acepte todas las oraciones del conjunto de entrenamiento como elementos del lenguaje que modela. El mecanismo necesario para contestar la pregunta de pertenencia al lenguaje modelado, es una parte esencial de un compilador. Se conoce como analizador sintáctico o con el término técnico "parser" (del inglés)[4]. Para el objetivo de este algoritmo es vital que el analizador sintáctico sea eficiente. Será el responsable de determinar el valor de aptitud de cada individuo, sin el cual no se puede llevar a cabo ninguna clase de Cómputo Evolutivo. Es posible hacer una partición en el universo de las gramáticas libres de contexto, en una jerarquía basada en la dificultad de análisis sintáctico de las gramáticas.
Una gramática libre de contexto arbitraria produce oraciones analizables en $O(n^3)$ para el peor de los casos, donde n es la longitud en palabras de la oración que se analice.
El algoritmo que determina la aptitud de un individuo en este trabajo lleva a cabo un análisis sintáctico descendente o "top-down parsing"[5] como se le conoce

[4]Un compilador debe inferir una derivación para la cadena de entrada, o determinar que dicha cadena no pertenece al lenguaje generado por la gramática que modela el lenguaje en cuestión. La raíz del árbol que representa el análisis sintáctico de la oración es conocida, se trata del símbolo inicial de la gramática. En el ejemplo sería el no terminal 'OR'. Las hojas de ese árbol son conocidas, pues deben coincidir de izquierda a derecha con las palabras que forman la oración procesada.

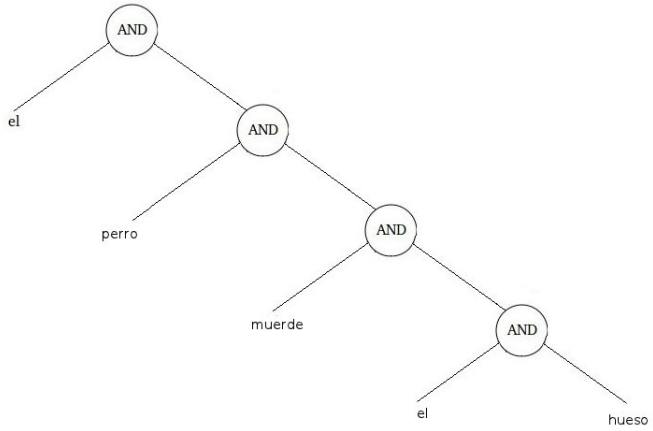
[5]Un analizador sintáctico descendente (top-down parser) comienza en la raíz del árbol, y sistemáticamente crece el árbol hasta que sus hojas coinciden con las palabras contenidas en la oración de entrada. La eficiencia de un analizador sintáctico descendente depende de forma crítica en su habilidad de seleccionar la regla de producción correcta, cada vez que expande un signo no terminal. En el peor de los casos, la estructura de una gramática libre de contexto puede ocasionar que un parser no termine. Un analizador sintáctico descendiente promedio, al seguir una secuencia de reglas de producción y no poder llegar al punto de terminación, que es cuando las hojas del árbol producido son exactamente las palabras de la oración, leídas de izquierda a derecha, recurre a lo que se conoce en inglés como "backtrack". Esto es, retrocede hasta el punto donde había más de una regla de producción posible a elegir y la toma, hasta que agote las posibilidades.

La eliminación de la recursión izquierda de las reglas es el primer paso para transformar una gramática a una forma LL(1), después, mediante el uso de la siguiente palabra en la oración que se procesa, puede determinarse la regla correcta a utilizar, eliminando el backtrack. Se

en inglés. Requiere que las gramáticas que evalúa sean de la clase LL(1). Esta clase es un subconjunto de las gramáticas libres de contexto que procesan de manera lineal y de izquierda a derecha ("left to rigth" en inglés), las palabras en la oración. Derivando siempre el símbolo no terminal que esté más a la izquierda, y generando el árbol de forma descendente de la raíz a las hojas.
Se asignarán valores de aptitud no nulos a los individuos que logren generar árboles de análisis sintáctico para una porción de la oración de entrada, siempre y cuando esta porción sea un prefijo de la misma, es decir, una secuencia contínua de palabras desde el principio hasta la penúltima palabra. Si la función que evalúa a cada individuo se llama EV, para un individuo x, se tendría lo siguiente:

$$EV(x) = \frac{n}{N} * 10$$

donde N es la longitud de la oración procesada y n la longitud en palabras del prefijo que logra analizar el individuo x, de modo que el valor máximo alcanzado por la función EV es 10. Este valor será el asignado a individuos que generan con éxito el árbol de análisis sintáctico para toda la oración. Tomando en cuenta el método de selección usado en este algoritmo que será el de torneo binario, y dado que este método no es proporcional a la aptitud del individuo con respecto al resto de la población, el valor de aptitud será igual al valor devuelto por la función EV de evaluación. Para ilustrar un ejemplo de individuo exitoso, en el caso de que la cadena de entrada fuera: "el perro muerde el hueso", la gramática de la imagen anterior produciría el siguiente árbol de análisis sintáctico:



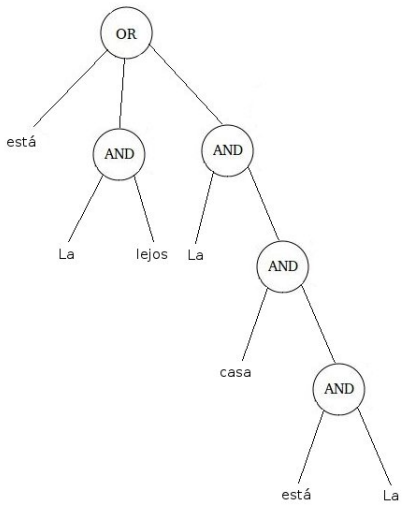
Finalmente se logra tener individuos que están asociados a gramáticas libres de contexto, que al procesar una oración contenida en el conjunto de entrenamiento, producen un árbol de análisis sintáctico correcto, o indican que el lenguaje que modela su gramática no contiene a dicha oración. Hay que notar que aquí no aparece el operador lógico OR, pues éste indica la opción entre

recomienda consultar el capítulo 3 de la fuente [8], para una exposición detallada de cada paso.

varias reglas de producción.
En el contexto de analizadores sintácticos, la ambigüedad significa que se puede llegar a la misma oración por medio de más de una secuencia de producciones diferentes. Cuando se desea obtener la estructura sintáctica asociada a un programa computacional, es absolutamente importante que la gramática del lenguaje usado no sea ambigua. En el caso del algoritmo desarrollado en esta tesis, controlar que no exista ambiguedad resulta un problema que puede disparar la complejidad. Además, la no ambigüedad de las gramáticas producidas es producto de las restricciones estructurales impuestas a los individuos.

4.3.2. Un ejemplo de evaluación

Se ha mencionado en este trabajo que el objetivo del programa genético que se desarrolló, es producir estructuras de árbol que a su vez tienen asociadas gramáticas libres de contexto capaces de analizar sintácticamente la oración de entrada.
Para mayor claridad, es absolutamente importante remarcar que en ningún momento de la ejecución del algoritmo se dispone de la gramática en forma Backus-Naur. Ahora bien, como en la subsección anterior se habló del criterio de evaluación sin dar un ejemplo, se abre la oportunidad de ejemplificar la manera en que se utiliza una estructura de árbol obtenida, con una gramática libre de contexto asociada, para evaluar la aptitud del individuo del cual es genotipo. Suponiendo que la oración de entrada fuera la siguiente: *La casa está lejos*, podría darse el caso de que un individuo tuviera por genotipo la siguiente estructura de árbol:



Esta estructura tiene asociada una gramática libre de contexto que en forma de Backus-Naur estaría descrita de la siguiente forma:

$$\begin{aligned} S &\rightarrow N_1|N_2|está \\ N_1 &\rightarrow La\, lejos \\ N_2 &\rightarrow La\, N_3 \\ N_3 &\rightarrow casa\, N_4 \\ N_4 &\rightarrow está\, La \end{aligned}$$

Claramente este individuo sería incapaz de analizar sintácticamente la oración de entrada en su totalidad. A través del método de evaluación utilizado, que es un análisis sintáctico recursivo descendente, en una verificación cuya complejidad no sería mayor a la altura del árbol más una constante, se sabría que la longitud del segmento que este individuo es capaz de analizar es igual a 3. Ahora bien, de acuerdo con la fórmula establecida en la subsección anterior: $EV(x) = \frac{n}{N} * 10$, donde para este caso $N = 4$ y $n = 3$, el valor de evaluación obtenido para este individuo sería igual a 7.5.
Las gramáticas en Forma Normal de Chomsky, como las que están asociadas a los individuos que conforman las poblaciones del algoritmo desarrollado en este trabajo, se limitan a capturar el orden en que aparecen las palabras en la oración, como puede verse en la imagen del ejemplo.

4.3.3. Dos esquemas de presión selectiva

La presión selectiva ayuda al algoritmo genético a seleccionar individuos con la misma aptitud bajo criterios de selección adicionales, por ejemplo, el número de nodos presentes en el árbol que representa a la gramática libre de contexto. Se debe de tener cuidado al incluir presión selectiva en el valor de evaluación que se piensa utilizar al momento de llevar a cabo la selección. En el presente caso, dado que se está usando un torneo determinista como método de selección, si la presión selectiva es muy agresiva, puede ocasionar que individuos con pocos nodos y no tan aptos dominen a la población en pocas generaciones. Esto provocaría que escaseara información genética útil para la recombinación genética, y que en un lapso no mayor a P generaciones, donde P es el tamaño de la población[6], el algoritmo llegara a un máximo local (lo cual sería muy contraproducente).
Por lo expuesto en el párrafo anterior, para el algoritmo que se expone en esta tesis se optó por introducir dos esquemas de presión selectiva. En el primer esquema, se penaliza el valor de evaluación de los individuos con un número de nodos que está por encima del promedio de la población con un valor fijo de 0.25, si está por debajo del promedio de la población se distinguen dos casos nuevamente, ***los individuos que no reciben penalización*** *son aquellos que*

[6]La razón de esto es simple. Pensemos que en un torneo determinista, el mejor individuo de la población siempre pasa a la generación siguiente tantas veces como deba repetirse el proceso de selección. Si se realizan 2 torneos para llenar la población intermedia que servirá para la recombinación genética, en ella aparecerá 2 veces dicho individuo. Para la siguiente generación, aparecerá hasta 4 veces, y así sucesivamente.

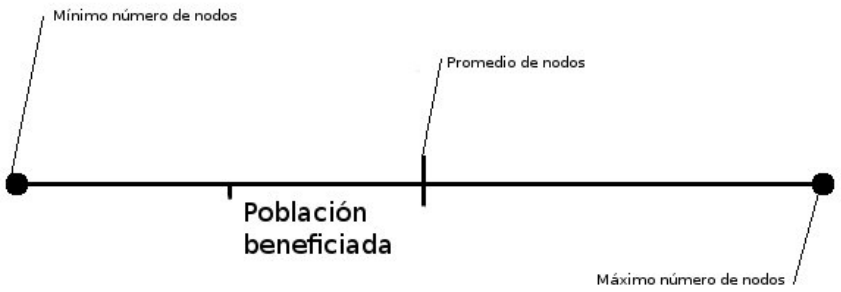


Figura 4.1: Sector de la población beneficiado por el primer esquema de presión selectiva, menos agresivo

con respecto al resto de la población, no tienen un exceso de nodos, pero poseen suficientes para aportar información genética al momento de hacer la recombinación. La figura 4.1 al inicio de esta página, muestra el sector de la población beneficiado por este primer esquema de presión selectiva.

Llamando β al valor de penalización fijado en 0.25, o bien 0 para el sector de la población que se pretende beneficiar, la función de evaluación quedaría como sigue:

$$EV(x) = \frac{n}{N} * 10 - \beta$$

Aquí x es nuevamente un individuo cualquiera de la población, n es la longitud de la oración que logra analizar sintácticamente y N es la longitud en palabras de la oración analizada. Debe enfatizarse que para un conjunto de individuos, el valor de β será 0.
Una vez que se logra obtener un individuo que logra analizar toda la oración de entrada y su número de nodos está por debajo del promedio de la población, es decir, que su valor de EV es 10. Se procede a presionar a la población de una forma más agresiva para que los individuos pierdan la mayor cantidad de nodos posible. Esto se logra con el segundo esquema de presión selectiva que se expone a continuación.
Se usará una variable que se llamará Γ, dependiente del número de nodos promedio que tienen los individuos de la población : $\overline{NODOS}$, y el número de nodos presentes en el individuo que se evalúa: *nodos*. Así, la expresión de la función

EV queda de la siguiente forma:

$$\begin{aligned} EV(x) &= \frac{n}{N} * 10 + \Gamma \\ donde \quad \gamma &= \overline{NODOS}/nodos \\ y \quad \Gamma &= \gamma/\gamma_{max} \end{aligned}$$

El término γ_{max} se refiere al máximo valor alcanzado por γ en toda la población. Este valor sirve para escalar el valor de Γ a 1.0, que de otra forma podría ser muy alto y perder su utilidad. De esta forma se introduce una forma de presión selectiva que ocasionará que entre individuos con la misma capacidad para analizar oraciones, se conserven los que posean menor número de nodos.
Ahora bien, este esquema de presión selectiva proporciona igualmente una forma de saber cuándo la población no puede presionarse más; si el valor de Γ para el individuo menos apto en la población es igual a 1.0, la ejecución del algoritmo debe detenerse y devolver la estructura obtenida.
El énfasis en controlar el número de nodos en la población, y la suposición de que entre las estructuras con mismo valor de aptitud, aquellas con menor número de nodos son más efectivas, no solamente tiene su razón en el ahorro de recursos computacionales. En Programación Genética es muy conocido el efecto de hinchamiento (del inglés "Bloating"). Este efecto se debe a que a través de las generaciones, generalmente se observa que los individuos con mejor aptitud tienen más nodos, que el promedio de nodos en la población. Esto hace que conforme avanzan las iteraciones del algoritmo, el número de nodos presentes en los individuos vaya en aumento. Para un tratamiento detallado de este interesante tema, puede consultarse [7].

4.3.4. Operador de cruza

Cada vez que se agrega una nueva oración del conjunto de entrenamiento, se debe renovar el conjunto de símbolos terminales, que de ahora en adelante llamamos diccionario. El diccionario es sencillamente un conjunto de palabras del lenguaje objetivo.
Dos gramáticas de la población existente se eligen de manera aleatoria para la reproducción sexual. Un nodo elegido al azar de un padre se intercambia por uno del otro padre, para generar dos hijos. Es importante respetar las restricciones estructurales impuestas a los árboles que representan las gramáticas, durante la generación de los individuos. Debe evitarse que después de la cruza se produzcan individuos con operadores OR que a su vez tengan descendientes OR, que un nodo terminal posea hijos o que un nodo AND tenga un primer hijo distinto a un terminal. Si en la selección aleatoria de punto de cruza en ambos padres se selecciona un nodo terminal, o nodos con el mismo operador, ya sea OR o AND, la cruza puede realizarse directamente. De lo contrario debe de hacerse un procesamiento que involucra a los nodos hijos o al nodo padre de alguno de los puntos de cruza seleccionados. Abajo se muestra una tabla con la explicación del proceso que se lleva a cabo en algunos de los casos. Los casos que no se

muestran son simplemente simétricos a los que se consideran:

Primer padre	Segundo padre	Explicación del proceso
terminal	terminal	Cruza directa
AND	AND	Cruza directa
OR	OR	Cruza directa
terminal	AND	Si el padre del nodo con un terminal es un operador AND, usar este último como punto de cruza del primer padre. Si es un operador OR, usar el primer hijo del nodo AND como punto de cruza del segundo padre
terminal	OR	Si el primer hijo del operador OR es un terminal, usar este último como punto de cruza del segundo padre. Si es AND, se reduce al caso de terminal contra AND
AND	OR	Si el primer hijo del operador OR es un operador AND, se usa este último como punto de cruza del segundo padre. Si es un terminal, se reduce al caso terminal contra AND

Además, se tiene la complicación adicional de preservar el número de hijos permitido a cada operador. En el caso del operador AND, se sustituyen los subárbol hijo por el segmento de recombinación que provienen del otro padre. En el caso del OR, como se mencionó antes, se permiten tantos hijos como número de palabras en la oración procesada. Aquí se permite agregar más hijos hasta que se alcanza este el límite, y a partir de entonces se observa un comportamiento de "carrusel", en el que el nuevo descendiente se agrega al final del listado, y se elimina al primer descendiente del listado de subárboles hijos. El proceso procura ser no destructivo, por lo que se observa que los hijos obtenidos sean al menos tan aptos como los padres que los originaron. Si no pasan esta prueba, no ocupan el lugar de los padres en la población intermedia.

4.3.5. Operador de mutación

Siempre que dos individuos se seleccionan para efectuar una reproducción sexual, otro se selecciona al azar para llevar a cabo una mutación. La selección aquí, contrariamente con respecto al resto del algoritmo, se hace directamente proporcional al tamaño del individuo (es más probable elegir gramáticas de mayor tamaño). Esto bajo la suposición de que las gramáticas más grandes no mejoran la aptitud de la población vista en conjunto. Más aún, un mayor tamaño indica la presencia de nodos que pueden beneficiarse de la mutación. Así, a individuos cuyos árboles sintácticos poseen una estructura interna correcta, pero que sus hojas contienen elementos del diccionario en orden equivocado, se les da la oportunidad de corregirse.
El proceso comienza eligiendo cinco individuos aleatoriamente de la población,

de los cuales se conserva aquel con el mayor número de nodos. Sabiendo cuál individuo mutará, un nodo hoja se selecciona al azar, y se reemplaza por un símbolo terminal aleatorio, es decir, una palabra del diccionario. El individuo mutado se agrega a la población existente si puede analizar sintácticamente un prefijo mayor o igual, que el analizado por el individuo original, que será entonces sustituído.
En Algoritmos Genéticos tradicionales, se asigna una probabilidad pequeña a la mutación para evitar que el algoritmo se estanque en máximos locales. En este caso, se pensó en la mutación como un operador que facilite la introducción de las palabras nuevas del diccionario a la población de forma pasiva, o bien, sin agragerlas directamente a los individuos presentes. De esta forma, se pretende que no le tome demasiado tiempo a la población encontrar gramáticas bien adaptadas.

4.3.6. Población inicial

Si N es la longitud de la oración analizada en palabras, se inicializa una población de $POB = N * 150$ individuos por medio del método de mitad y mitad escalonada. En caso de que se esté utilizando el método de generación por crecimiento, dado que solamente se tienen dos operadores (AND y OR), se tendrá cuidado de que la posibilidad de elegir entre un símbolo terminal y uno no terminal sea la misma, es decir 50 % y 50 % respectivamente.
En cuanto al nivel máximo de niveles permitidos a los árboles que codifican a los individuos, regresando a las imágenes anteriores podemos apreciar que para una oración de 5 palabras, se necesitó un árbol con profundidad 5, que representara una gramática capaz de analizarlo sintácticamente, así pues, el número de elementos presentes en el conjunto de símbolos terminales, o bien, de palabras, que llamamos $N_{palabras}$, resulta un horizonte confiable para marcar el límite al que pueden crecer los individuos generados. Si M es el máximo número de niveles, entonces:

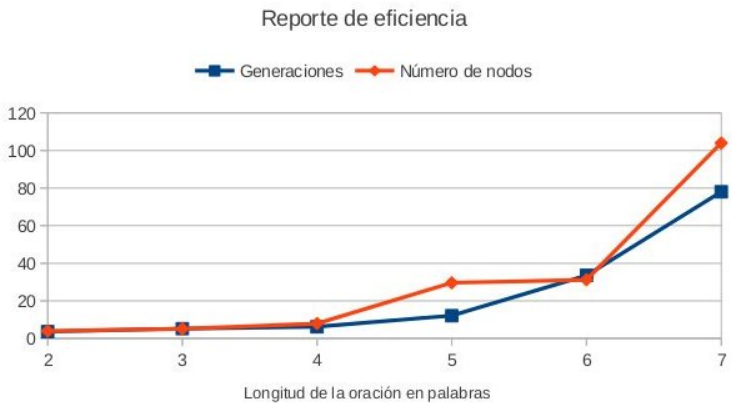
$$M = N_{palabras} + 1$$

de modo que se permitirá que los individuos vayan un nivel más abajo del óptimo en busca de una solución. Además se usará el operador secundario de destrucción con un parámetro porcentual del 25 %, por lo que al inicio de cada ejecución, se generarán POB * 3 individuos adicionales a los POB existentes, es decir, un total de POB * 4 individuos de los cuales solamente el 25 % más apto pasará a formar parte de la población. Se va a aplicar el operador cada vez que una oración se agregue desde el conjunto de entrenamiento, y bajo el supuesto de que en la iteración anterior se consiguió una estructura que lograba analizar la oración anterior. En este punto se reinicializará la población. Una oración será agregada desde el conjunto de entrenamiento al algoritmo, para que las gramáticas presentes en la población se adapten de modo que puedan analizarla sintácticamente. Como atributo a la clase que define los parámetros del problema, se agregó un entero que especifica el número máximo de iteraciones extras que se le permite realizar a cada ejecución del programa genético, para

que por medio del esquema de presión selectiva agresivo encuentre un individuo apto con el menor número de nodos posible.
Por lo anterior, en la generación 0 se generan POB * 4 individuos de los cuales POB pasarán a la siguiente generación. Si menos de POB individuos tienen una aptitud mayor que cero, para ocupar el residuo se llevará a cabo una selección aleatoria de individuos que no estén presentes en la población intermedia.
Finalmente, es importante mencionar que por ser la programación genética una técnica de programación no determinista, a pesar de plantear un número de individuos por generación a partir de la complejidad del problema a resolver, no es completamente seguro que se alcance formar al individuo o solución deseada en un número predefinido de generaciones. Por esto, se desarrolló un esquema adaptativo que permite la recuperación cuando el algoritmo cae en máximos locales. Por medio de un parámetro del algoritmo, se especifica el máximo de generaciones permitidas por proceso evolutivo para encontrar al individuo capaz de analizar sintácticamente toda la oración. De no alcanzarse este objetivo, la población se reinicializa. Así se sale de un caso en el que características genéticas de un individuo que no obtiene la aptitud deseada, dominan a toda la población.

4.3.7. Reporte de eficiencia

Habiendo concluído la implementación del programa genético descrito en las subsecciones anteriores, es posible realizar pruebas con oraciones de distintas longitudes. Esto es necesario para plantear el funcionamiento global del programa genético, de modo que alcance su objetivo sin acaparar tiempo y recursos computacionales. Después de haber hecho diez pruebas con oraciones de seis longitudes distintas, los promedios de generaciones requeridas y número de nodos en el mejor individuo obtenido fueron los que se muestran en la siguiente gráfica:



En regresión simbólica, es común encontrarse con umbrales como el que se exhibe en las oraciones de siete palabras. La cantidad de símbolos no terminales que deben encadenarse de manera correcta para llegar al mejor resultado va en aumento mientras mayor sea el espacio de búsqueda. Además la explosión en las posibles combinaciones de los elementos terminales en una solución, hace que el algoritmo sea menos eficiente. En el caso de las oraciones con ocho palabras o más, no se puede garantizar incluso que el algoritmo encuentre un individuo capaz de analizar toda la oración pasados varios minutos de que el algoritmo comenzó su ejecución.

Esto no es aceptable para lograr la funcionalidad que se esparaba obtener del programa genético en un principio. Afortunadamente, la causa de este problema, que es con seguridad las limitaciones estructurales impuestas, también proporcionan una forma de solucionarlo.

Al comienzo de la primera fase de desarrollo se mencionó la **unificación** que se lleva a cabo cuando al final de una ejecución, en la que la estructura de árbol obtenida, que representa una gramática libre de contexto capaz de analizar la oración recién procesada, se une a **una gramática libre de contexto que concentra el conocimiento adquirido por ejecuciones pasadas**, de modo que después de está unión la gramática resultante puede procesar todas las oraciones procesadas hasta ese punto.

Dado que las ejecuciones del algoritmo que procesan oraciones de hasta seis palabras de longitud, se llevan a cabo en pocos segundos, la solución propuesta ya se puede intuír. El procesamiento de oraciones arbitrariamente largas se podría llevar a cabo componiendo varias ejecuciones del algoritmo hasta que se produzca una estructura de árbol, asociada a una gramática libre de contexto capaz de analizar la oración en su totalidad.

Se trata de maximizar el tamaño de los bloques en los que se separa la oración

original. Así por ejemplo la oración: "Los soldados marchan hacia el frente de batalla", implicaría dos ejecuciones del algoritmo; en la primera se procesaría la oración: "Los soldados marchan hacia el frente" y en la segunda la oración: "frente de batalla ". Los mejores individuos de ambas ejecuciones se unirían para al final obtener una estructura capaz de analizar la oración en su totalidad.
El mecanismo que sigue el método unificador se explica en la siguiente subsección. Por lo expuesto aquí, se entiende que sea de gran importancia para este trabajo.

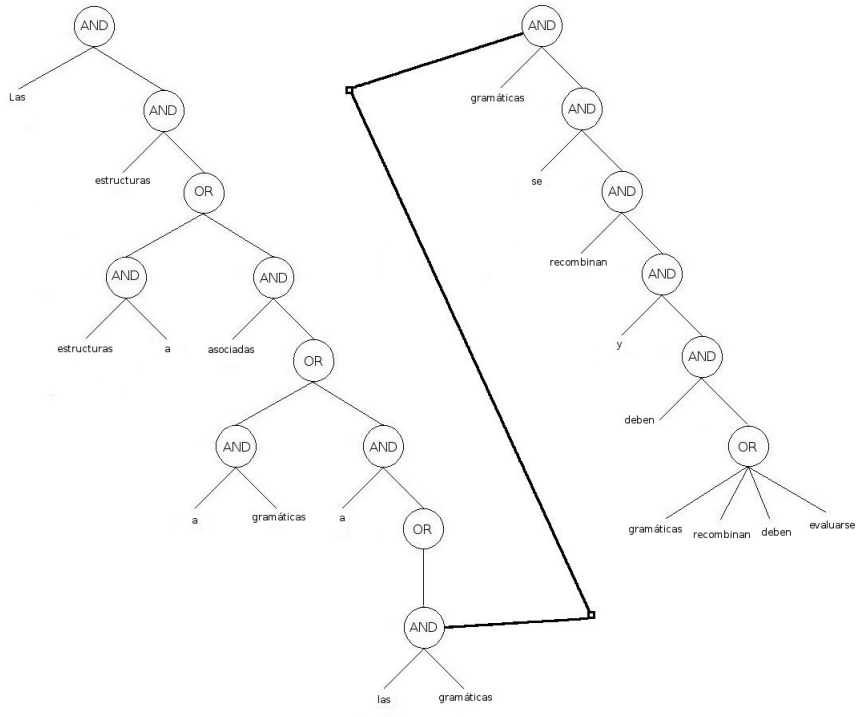
4.3.8. El método unificador

Suponiendo que se deseara producir una gramárica libre de contexto capaz de analizar sintácticamente la oración: *"Las estructuras asociadas a las gramáticas se recombinan y deben evaluarse"*, que claramente tiene más de seis palabras, procederíamos a generar una población incial distinta, y ejecutar un programa genético para cada uno de los segmentos: *"Las estructuras asociadas a las gramáticas"* y *"gramáticas se recombinan y deben evaluarse"*. Ahora, si los mejores individuos obtenidos de dichas ejecuciones fueran los descritos por las cadenas:

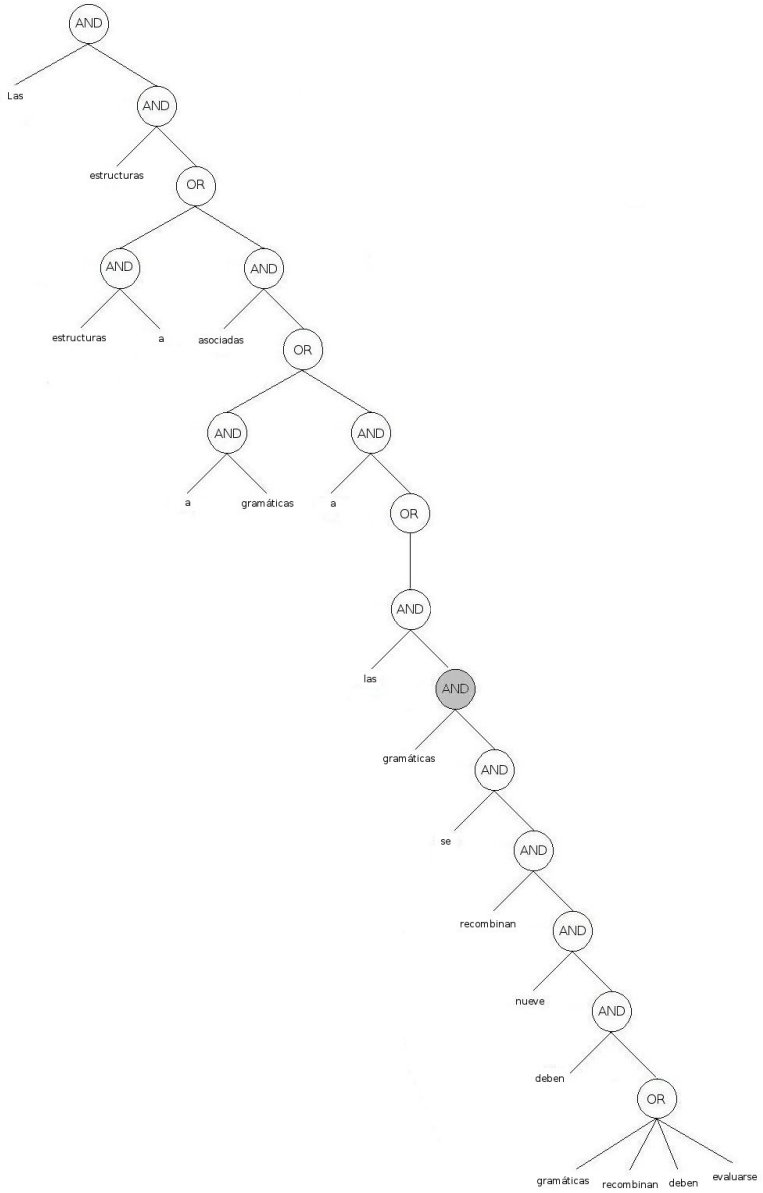
$$AND(Las,\ AND(estructuras,\ OR(AND(estructuras,\ a), AND(asociadas,\ OR(AND(a,\ gramáticas),\ AND(a,\ OR(AND(las,\ gramáticas))))))))$$

$$AND(gramáticas,\ AND(se,\ AND(recombinan,\ AND(y,\ AND(deben,\ OR(gramáticas,\ recombinan,\ deben,\ evaluarse))))))$$

La siguiente imagen muestra a las estructuras de árbol correspondientes a estas descripciones, con una línea que indica el nodo por el que el operador unificador va a juntar toda la información contenida en ambos árboles:



Después de unir ambas estructuras, se obtiene un árbol asociado a una gramática libre de contexto capaz de analizar sintácticamente la oración en su totalidad. La raíz del segundo árbol, resaltada en un tono gris en la siguiente imagen, pasa a ser el segundo hijo de un nodo del primer árbol, que a su vez es padre de la última palabra que éste logra analizar, en el primer segmento de la oración:



La cadena que describe a la estructura que se muestra arriba es la siguiente:

$$AND(Las,\ AND(estructuras,\ OR(AND(estructuras,\ a),\ AND(asociadas,\ OR(AND(a,\ gramáticas),\ AND(a,\ OR(AND(las,\ AND(gramáticas,\ AND(se,\ AND(recombinan,\ AND(y,\ AND(deben,\ OR(gramáticas,\ recombinan,\ deben,\ evaluarse))))))))))))))$$

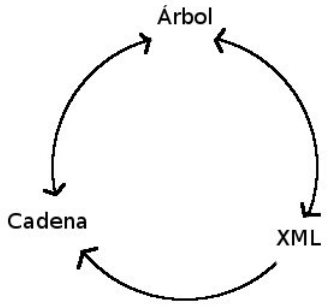
Los detalles de implementación no se especifican, pero se resalta la necesidad de calcular bien la longitud de los segmentos que se alimentaran a cada proceso evolutivo, de modo que al momento de unificar, la última palabra que lograba analizar el mejor individuo de la iteración anterior, sea la primera palabra que analiza el mejor individuo del programa genético que recién ha terminado.

4.3.9. Almacenamiento y recuperación de la información

Producir una estructura capaz de analizar sintácticamente una cantidad considerable de oraciones, en el orden de miles, puede tomar varias horas en una computadora de escritorio comercial sin un procesador que tenga tecnología multi-núcleo. Una vez llevado a cabo este trabajo, sería una gran desventaja si al terminar la ejecución se perdiera la información obtenida y se tuviera que repetir el proceso. Por esta razón, la clase llamada ArbolXML.java posee métodos capaces de transformar la información a un formato que puede reportarse en un archivo con extensión XML. Este formato, tiene una estructura de árbol explícita, y permite además la especificación de atributos para cada nodo que pueden determinarse durante el proceso de transformación. Así, los nodos con símbolos terminales, es decir, aquellos que contienen palabras del lenguaje natural analizado, pueden recibir una etiqueta única, dependiendo de la palabra que contengan. Adicionalmente, pueden numerarse las versiones de los árboles que se están reportando, y los nodos que contienen símbolos no terminales, o bien, operadores OR y AND. Como ejemplo, la siguiente imagen muestra al archivo con extensión XML que resulta de la ejecución que procesa la oración: “agua para el que agua pide”.

```
<Arbol version="003">
 −<OR id="NT1">
  −<AND id="NT2">
     <terminal id="aaaa0">agua</terminal>
    −<AND id="NT3">
       <terminal id="aaaa1">para</terminal>
      −<AND id="NT4">
         <terminal id="aaaa2">el</terminal>
        −<AND id="NT5">
           <terminal id="aaaa3">que</terminal>
          −<OR id="NT6">
            −<AND id="NT7">
               <terminal id="aaaa0">agua</terminal>
              −<OR id="NT8">
                 <terminal id="aaaa0">agua</terminal>
                 <terminal id="aaaa4">pide</terminal>
                 <terminal id="aaaa0">agua</terminal>
                 <terminal id="aaaa1">para</terminal>
               </OR>
             </AND>
           </OR>
         </AND>
       </AND>
     </AND>
  </OR>
</Arbol>
```

Una vez realizada la transformación y escrito el archivo, es posible en ejecuciones futuras recuperar la información obtenida realizando la transformación inversa, del formato XML a la representación interna de árbol, conservando la identidad única de los nodos hojas, y la numeración de los nodos internos que se reportó en el archivo.
El algoritmo está listo para ser utilizado con el objetivo original de recibir un entrenamiento para simular un diálogo. Si la estructura de árbol que se obtiene del archivo con extensión XML requiere adiciones, producto del programa genético, es necesario regenerar el etiquetado y actualizar la versión del archivo donde se concentra la información. Para este punto, una transformación del formato XML a una representación más simple que no tome en cuenta atributos extras es necesaria. La cadena que describe al árbol en notación infija será suficiente.
De manera resumida, las transformaciones de la información aprendida por el algoritmo durante distintas ejecuciones se describen en el siguiente esquema:



4.4. Segunda fase de desarrollo

Hasta este punto se ha logrado evolucionar una población con un número bien definido de individuos, hasta que se obtiene un individuo capaz de analizar sintácticamente la oración de entrada proporcionada por el usuario. No solamente esto, sino que el genotipo de este individuo, que es el árbol asociado a la gramática libre de contexto capaz de llevar a cabo la tarea antes mencionada, tiene características especiales que permiten hacer el análisis sintáctico de manera muy eficiente.
Por parte del cómputo evolutivo, se han obtenido los resultados esperados: un algoritmo que por medio de la programación genética, genera una estructura de árbol asociada a una gramática libre de contexto que reconoce la oración de entrada. Y es bueno mencionar que las estructuras reportadas en el archivo XML, donde además las palabras del lenguaje natural estudiado tienen etiquetas únicas, tienen un gran potencial de alimentar a un algoritmo que las procese y produzca otra estructura cuyos nodos no terminales posean atributos sintetizados a partir de los terminales que sean sus descendientes.
Ahora bien, el trabajo de desarrollar un simulador de diálogo no es algo sencillo, y en este trabajo solamente se da el primer paso. En la clase con nombre

Dialogo.java, existen los métodos necesarios para iniciar y concluir una sesión de entrenamiento, en la que se solicita al usuario que introduzca oraciones. Si la respuesta correcta está almacenada en memoria, el algoritmo responde, de lo contrario, a través del proceso que involucra programación genética y que se expuso en la sección anterior, el algoritmo aprende la respuesta que pide al usuario, y el comentario inicial si es necesario.
Al finalizar una sesión de entrenamiento, los archivos de texto dialogo.txt y agenda.txt se actualizan con el nuevo conocimiento (o informacion) adquirida. En el primero está una sucesión alternada de entradas y salidas, mientras que en el segundo se almacenan las etiquetas de los nodos que han de recorrerse para producir la resupuesta adecuada.
Para ver ejemplos completos se recomienda consultar los archivos dialogo.txt, agenda.txt y dialogoINFO.txt, que se encuentran dentro de la carpeta con nombre "Clases", en el disco adjunto a este trabajo de tesis. En el archivo de texto agenda.txt pueden observarse las sucesiones de nodos que deben seguirse en la estructura reportada en el archivo dialogoINFO.xml, de acuerdo con los resultados obtenidos al procesar el conjunto de entrenamiento reducido presente en el archivo dialogo.txt, con el fin de facilitar la exposición del algoritmo.
Se trata realmente de un simulador simple, sin embargo gracias a que está sustentado en un proceso que transforma cadenas del lenguaje natural a una estructura que la computadora puede procesar eficazmente, el potencial de crecimiento es grande. Para alguien que acostumbra trabajar con estructuras de datos, debe ser evidente que una estructura de árbol además de proporcionar más información que una cadena, facilita por medio del etiquetado de nodos, la manipulación futura del conocimiento almacenado.

4.4.1. En busca de patrones

Cuando se introduce la cadena: "Conoces a Cecilia de clase de discretas?", con su respectiva respuesta: "No conozco a ninguna Cecilia", se espera que en una segunda sesión de entrenamiento, si el algoritmo recibe la cadena: "Conoces a Gabriela Mistral de la clase de discretas?" responda: "No conozco a ninguna Gabriela Mistral". Para lograr esto, se trabaja sobre los recorridos en el árbol que están marcados por las secuencias de nodos que aparecen en el archivo agenda.txt. Así, si estas fueran las únicas dos oraciones que hubiera procesado el algoritmo hasta este punto, el árbol reportado en el archivo con extensión xml podría ser el siguiente (hay nodos contraídos por cuestión de espacio):

```
<Arbol version="051">
−<OR id="NT1">
 −<AND id="NT2">
    <terminal id="aaaa0">Conoces</terminal>
  −<AND id="NT3">
    <terminal id="aaaa1">a</terminal>
   −<AND id="NT4">
    <terminal id="aaaa2">Cecilia</terminal>
   −<AND id="NT5">
    <terminal id="aaaa3">de</terminal>
   −<OR id="NT6">
   −<AND id="NT7">
    <terminal id="aaaa4">la</terminal>
   −<OR id="NT8">
    <terminal id="aaaa3">de</terminal>
    <terminal id="aaaa5">clase</terminal>
    <terminal id="aaaa0">Conoces</terminal>
   −<AND id="NT9">
    <terminal id="aaaa5">clase</terminal>
   −<AND id="NT10">
    <terminal id="aaaa3">de</terminal>
    <terminal id="aaaa6">discretas?</terminal>
   </AND>
   </AND>
   </OR>
   </AND>
  +<AND id="NT11"></AND>
   </OR>
   </AND>
   </AND>
   </AND>
 −<AND id="NT13">
    <terminal id="aaaa7">No</terminal>
  −<OR id="NT14">
   +<AND id="NT15"></AND>
   +<AND id="NT21"></AND>
   −<AND id="NT23">
    <terminal id="aaaa8">conozco</terminal>
   −<AND id="NT24">
    <terminal id="aaaa1">a</terminal>
   −<AND id="NT25">
    <terminal id="aaaa9">ninguna</terminal>
    <terminal id="aaaa2">Cecilia</terminal>
   </AND>
   </AND>
   </AND>
   −<AND id="NT26">
    <terminal id="aaaa8">conozco</terminal>
    <terminal id="aaaa8">conozco</terminal>
   </AND>
   </OR>
  </AND>
 </OR>
</Arbol>
```

Y el archivo agenda.txt reportaría entonces las secuencias de nodos:

$$\textit{Para la primera oración}$$
$$NT1 : NT2 : aaaa0 : NT3 : aaaa1 : NT4 : aaaa2 : NT5 : aaaa3 : NT7 : aaaa4 : NT9 : aaaa5 : NT10 : aaaa3 : PA : aaaa6$$
$$\textit{y para la segunda}$$
$$NT1 : NT13 : aaaa7 : NT23 : aaaa8 : NT24 : aaaa1 : NT25 : aaaa9 : PA : aaaa2$$

Notamos que la marca “PA” se utiliza cuando se reportan dos palabras que son hijas del mismo padre con operador “AND”. Ahora bien, al introducir el enunciado “Conoces a Gabriela Mistral de la clase de discretas?” el algoritmo detecta que el árbol difiere en el contenido del nodo aaaa2 de lo que está pro-

porcionando el usuario. Una propuesta es, seguir leyendo la oración de entrada, hasta que encuentre una palabra que sea igual a la que contiene el nodo aaaa3, es decir, en este caso la palabra "de", si la encuentra, reserva en la memoria el segmento que no reconoció: "Gabriela Mistral", y utilizarlo cada vez que encuentra un nodo aaaa2 en la respuesta.
Tomando en cuenta este esquema sencillo de búsqueda de patrones, se modificó la dinámica de las sesiones de entrenamiento, para que el algoritmo pudiera proponer distintas oraciones de respuesta, dependiendo de los patrones que encuentra.
Cuestiones más avanzadas, como lo son el género y número de un sustantivo, o la conjugación correcta de los verbos, pertenecen a la Lingüística Computacional y escapan al alcance de este trabajo. Pero el poder de adaptación del cómputo evolutivo podría sin duda proponer valores para el conjunto de etiquetas de un árbol lo suficientemente robusto. Definir este conjunto de etiquetas y la función de evaluación para distinguir a un individuo no apto de uno apto, requeriría sin duda de un estudio detallado del problema.
Puede tomarse en cuenta, que con el resultado obtenido de este trabajo, el planteamiento de un simulador de diálogo tipo ELIZA, es decir, un buscador de patrones, es casi directo.

4.4.2. Una sesión de entrenamiento

Llevada a cabo la implementación de un simulador de diálogo que soporta el reconocimiento de patrones, vale la pena hablar de los alcances de este nuevo algoritmo, ya que inmediatamente se hace evidente, que a pesar de la gran eficiencia que se tiene para adquirir nuevo conocimiento, la necesidad de establecer reglas propias al dominio es grande, que en este caso serían reglas gramaticales del español. Primero que nada, al ejecutar la clase Dialogo, se presenta al usuario la opción de iniciar una sesión de entrenamiento que soporte o no el reconocimiento de patrones, en la imagen que sigue, se elige la opción 2 para experimentar con los patrones que el algoritmo sea capaz de encontrar:

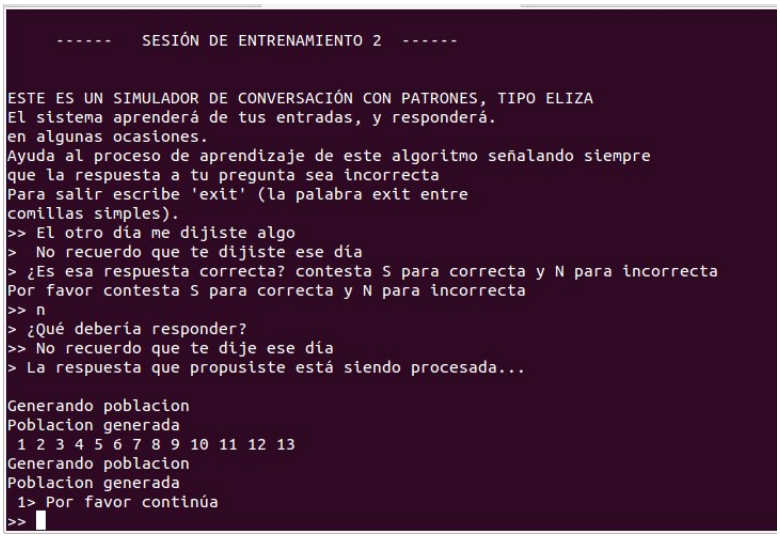
```
mtf@mtf-P43-ES3G:~/Escritorio/ALGparser$ java Dialogo
Selecciona el tipo de entrenamiento que deseas realizar:
1  -  Diálogo SIN manejo de patrones
2  -  Diálogo CON manejo de patrones
Por favor contesta 1 o 2 para elegir una de las opciones
>> 2

     ------   SESIÓN DE ENTRENAMIENTO 2  ------


ESTE ES UN SIMULADOR DE CONVERSACIÓN CON PATRONES, TIPO ELIZA
El sistema aprenderá de tus entradas, y responderá.
en algunas ocasiones.
Ayuda al proceso de aprendizaje de este algoritmo señalando siempre
que la respuesta a tu pregunta sea incorrecta
Para salir escribe 'exit' (la palabra exit entre
comillas simples).
>>
```

Por ejemplo, en una primera sesión de entrenamiento, se introduce: "¿Dónde se puede comer sushi en la UNAM?", y se le asigna la respuesta: "se puede comer sushi en la Facultad de Ciencias", en una sesión posterior como se muestra en la imagen de abajo, con el nuevo esquema de entrenamiento sucede lo siguiente con la entrada: "¿Dónde se puede comer una torta en la UNAM?":

Cuando pregunta si la respuesta es correcta, el usuario decidirá que sí lo es en este caso, y no será necesario que se lleve a cabo un proceso evolutivo para obtener el conocimiento necesario para analizar, la oración formada por medio de reconocimiento de patrones.
Ahora bien, para exhibir las limitaciones de este algoritmo, en una primera sesión de entrenamiento se introdujo la oración: "El otro día me *verbo* algo", donde se pretende introducir un verbo cualquiera en la palabra *verbo*, y si como respuesta se le asignara: "No recuerdo qué te *verbo* ese día", al momento de introducir la oración: "El otro día me dijiste algo", se observó lo siguiente:



Desde luego, la oración: "No recuerdo qué te dijiste ese día" es incorrecta. Al marcarlo así, el usuario ordena que se lleve a cabo un proceso evolutivo que produzca una estructura capaz de analizar la oración correcta: "No recuerdo qué te dije ese día".
La cantidad de combinaciones de palabras que podría existir en un diálogo humano cotidiano, aunado a la capacidad de las personas de elegir las palabras que dicen por su significado, más que por alguna familiaridad con una oración que hayan dicho en el pasado, y cuestiones mucho más avanzadas como los cambios de tema, dejan claro que la información requerida para simular con éxito una conversación, va mucho más allá que un simple etiquetado de nodos.

4.4.3. La disyuntiva entre aprendizaje computacional efectivo, y producción de gramáticas lingüísticamente correctas

Como se mencionó al comienzo de la primera fase de desarrollo, en un algoritmo evolutivo es esencial contar con un método que evalúe a los individuos para determinar su aptitud en determinado ambiente. En el artículo que se menciona ahí mismo, Tony C. Smith e Ian H Witten de la universidad de Waikato [2], realizan inducción gramatical sobre una oración del inglés por medio de programación genética, y concluyen exhibiendo una de las estructuras que obtienen, que el algoritmo es capaz de inducir gramáticas con características lingüísticamente correctas, como la distinción entre sujeto y predicado, por nombrar la más evidente.
Pese a que es un artículo de gran importancia para este trabajo, el aspecto de la implementación del algoritmo que usaron no es lo suficientemente claro como para reproducirlo. Por esta razón, se planteó un método de evaluación que dependía de restricciones estructurales fuertes en los genotipos de los individuos.

El aprendizaje de máquina observado es eficiente, pero a modo de reivindicación del área de Inteligencia Artificial llamada Lingüística Computacional, es necesario decir que las gramáticas aquí inducidas son lingüísticamente incorrectas. Es decir, tomando una oración al azar de un periódico o libro, lo más probable es que los conocimientos adquiridos y almacenados en el archivo con extensión XML, no fueran capaces de realizar un análisis sintáctico sobre ésta, pero sí generar una estructura con el diseño requerido y aprenderla.

4.5. Comentarios finales y futuros trabajos

Este trabajo pudo mostrar la versatilidad del cómputo evolutivo, enfrentando un problema difícil como lo es la producción de una gramática libre de contexto para el lenguaje natural.
Se observó que la eficiencia del análisis sintáctico era de gran importancia para la función de evaluación, y por ello se impusieron limitaciones morfológicas fuertes a los genotipos de los individuos. Al final se logró un algoritmo que produjera una estructura de árbol asociada a la gramática deseada, pero, ¿qué tan buena es realmente esta gramática?.
Esta pregunta puede responderse de diferente manera. Si se piensa en qué tanto logra capturar la complejidad y el significado del lenguaje natural, está claro que se queda corta. Es más, por un teorema de Teoría de la Computación [5] sabemos que una gramática libre de contexto lineal por la derecha, es equivalente a un autómata de estados finitos. Y debería de quedar claro, pues al ver el árbol que se reporta en los archivos con extensión xml, no estamos viendo otra cosa que un autómata de estados finitos. Por otro lado, si pensamos en términos de aprendizaje computacional, se logró una gramática que puede extenderse sin límite, con potencial de ser procesada nuevamente para extraer conocimiento más refinado. Estas pueden ser tal vez las palabras finales de este trabajo: debemos esperar que las computadoras tengan un proceso de aprendizaje similar al de los humanos, en el cual el conocimiento adquirido deba de ser reprocesado para depurarlo y elevar su valor.
En trabajos futuros, será esencial contar con un corpus lo suficientemente robusto para la extracción de información. En el caso de este trabajo, se proporcionó al algoritmo un número relativamente pequeño de líneas en el archivo diálogo.txt.
A lo largo de esta sección se despliega el diagrama de clases UML del algoritmo desarrollado durante este trabajo. Para que se puedan apreciar las letras pequeñas, el diagrama está segmentado en tres partes, pero igualmente se adjuntó un disco con el código y las clases compiladas en carpetas con el nombre de Código y Clases, respectivamente, y el archivo del diagrama completo. Si la versión es electrónica, se econtrarán las mismas carpetas y el diagrama, además del archivo PDF de este trabajo. Todo el código está comentado para javadoc, y adicionalmente en algunos métodos se comentaron partes para diferenciar los diferentes casos que se resolvieron.

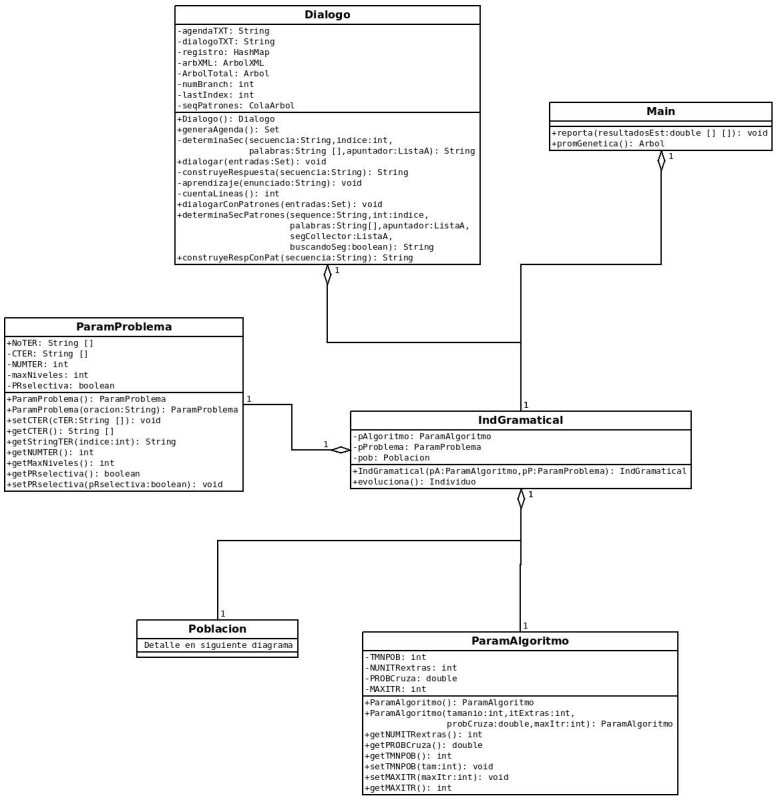


Figura 4.2: Primer segmento del diagrama de clases

4.5.1. Instrucciones para usar el código

El código puede entrar en funcionamiento ejecutando desde la línea de comandos las clases Dialogo, o Main. Es decir, introduciendo en la terminal cualquiera de las siguientes líneas:

java Dialogo

java Main

Desde luego, se debe de estar en el directorio donde estén las clases compiladas que se desea ejecutar, y tener la máquina virtual de java instalada.
En el primer caso, la clase Dialogo lleva a cabo una sesión de entrenamiento. En el segundo caso, la clase Main permite cargar un número arbitrario de oraciones a la estructura de árbol que almacena el conocimiento adquirido, sin necesidad de interacción con el usuario.

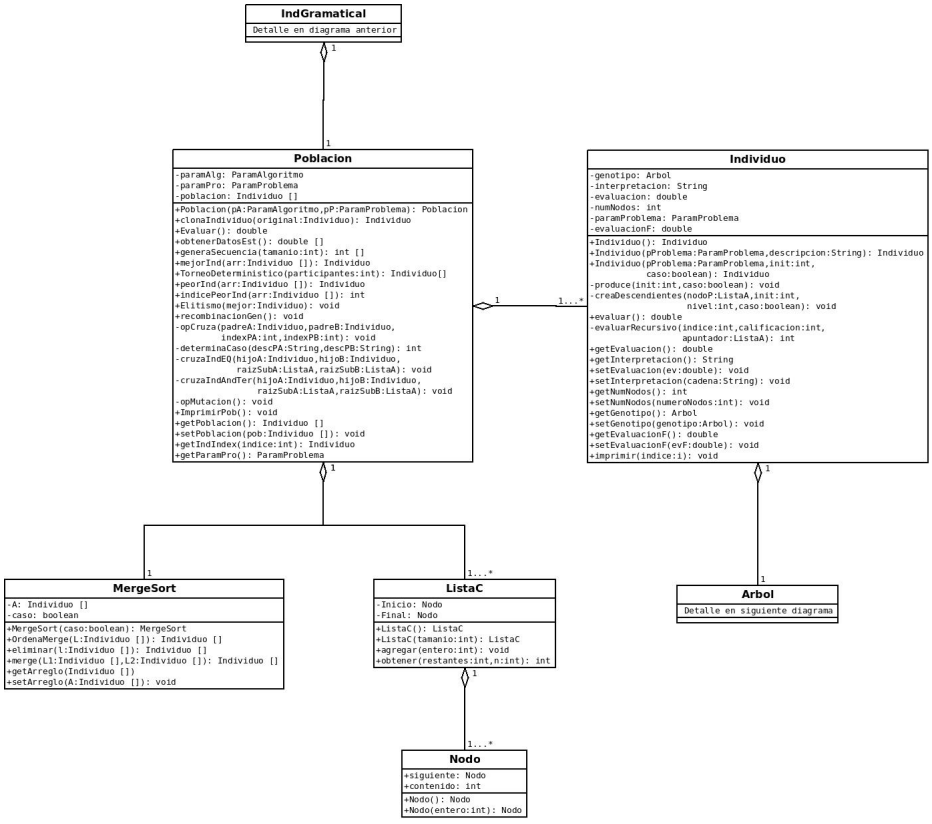


Figura 4.3: Segundo segmento del diagrama de clases

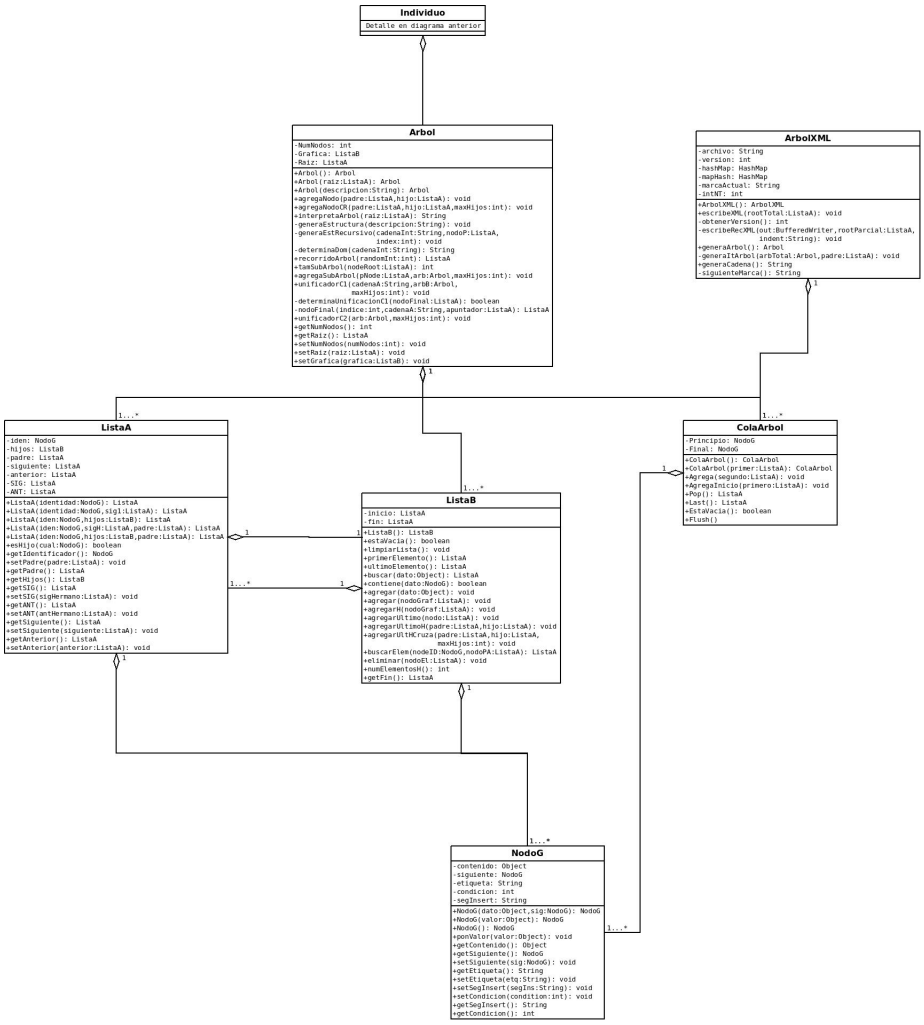


Figura 4.4: Tercer segmento del diagrama de clases.

Bibliografía

[1] NATURAL LANGUAGE PROCESSING, H. M. Noble, Editorial Blackwell Scientific Publications,1988.

[2] A GENETIC ALGORITHM FOR THE INDUCTION OF NATURAL LANGUAGE GRAMMARS, Tony C. Smith, Ian H. Witten. Departamento de Ciencias de la Computación, Universidad de Waikato, Hamilton, Nueva Zelanda.

[3] NATURAL LANGUAGE PROCESSING IN PROLOG, Gerald Gazdar, Chris Mellish, Editorial Addison-Wesley, 1994.

[4] SPEECH AND LANGUAGE PROCESSING: AN INTRODUCTION TO NATURAL LANGUAGE PROCESSING, COMPUTATIONAL LINGUISTICS, AND SPEECH RECOGNITION, Daniel Jurafsky, James H. Martin, Editorial Prentice Hall, 2009.

[5] INTRODUCCIÓN A LA TEORÍA DE LA COMPUTACIÓN, Elisa Viso Gurovich, Editorial Las Prensas de Ciencias, Primera Edición, 2008.

[6] INTRODUCTION TO GENETIC ALGORITHMS, S. N. Sivanandam, S. N. Deepa, Editorial Springer, 2010 .

[7] GENETIC PROGRAMMING: ON THE PROGRAMMING OF COMPUTERS BY MEANS OF NATURAL SELECTION, John R. Koza, Editorial MIT Press, 1992.

[8] ENGINEERING A COMPILER, Keith D. Cooper, Linda Torczon, Editorial Morgan Kaufmann, Segunda Edición, 2012.

[9] STATISTICAL LANGUAGE LEARNING, Eugene Charniak, Editorial MIT Press, 1996.

[10] ROBUST PART-OF-SPEECH TAGGING USING A HIDDEN MARKOV MODEL, Julian Kupiec, 1993.

[11] EVOLUTIONARY COMPUTATION, David B. Fogel Editorial John Wiley & Sons, Tercera Edición, 2005.

[12] ADAPTATION IN NATURAL AND ARTIFICIAL SYSTEMS, John H. Holland, Editorial MIT Press, 1992.

[13] GENETIC ALGORITHMS IN SEARCH, OPTIMIZATION, AND MACHINE LEARNING, David. E. Goldberg, Editorial Addison-Wesley, 1989.

[14] COMPUTATIONAL MODELS OF SCIENTIFIC DISCOVERY AND THEORY FORMATION, P. Langley, J. Shrager, Editorial Morgan Kaufmann, 1990.

[15] FUNDAMENTALS OF SPEECH RECOGNITION, L. Rabiner, B Juang, Editorial Prentice Hall, 1993.