80
241

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

DESARROLLO EN JAVA DE UNA MAQUINA PARA PROGRAMACION GENETICA

T E S I S

4 COMPRIMIDO DE UN DISKETTE 3 1/2

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A:

ANTONIO PEÑA GUTIERREZ

Directora: Dra. Ana María Vázquez Vargas

Ciudad Universitaria 1997

Agradecimientos:

Quiero agradecer, en primer lugar, a Hugo Hernández Mora e Iván Proaño Muciño cuyo apoyo me permitió llevar a cabo este trabajo.

A María Waleska Vivas Mendoza por su ayuda con la revisión de estilo.

Al Colegio Nacional de Educación Profesional Técnica (CONALEP) por permitirme el uso de los recursos de cómputo necesarios para la realización de esta tesis.

CONTENIDO

Presentación

En la década de los 50 Arthur Samuel, uno de los precursores en el campo de la inteligencia artificial, se planteó el problema de cómo puede una computadora aprender a solucionar problemas para los que no ha sido programada explícitamente. Los esfuerzos destinados a dar respuesta a esta interrogante han constituido una rama del conocimiento denominada "inteligencia artificial" que agrupa a un variado conjunto de técnicas, herramientas y metodologías de solución de problemas.

Dentro de la inteligencia artificial existe una rama de estudio conocida como "computación evolutiva" que agrupa técnicas diseñadas para explorar el espacio solución de un problema a través de una metáfora artificial del proceso de selección natural guiado por el principio de supervivencia de los individuos más aptos. Las ideas básicas de la computación evolutiva están inspiradas en los mecanismos de evolución descritos por Charles Darwin en *El Origen de las Especies* y, por lo tanto, proponen la búsqueda de mecanismos de solución de problemas basadas en la evolución de un caldo primigenio de posibles soluciones, utilizando un proceso continuo de evaluación, selección y transformación de éstas.

El trabajo presentado en esta tesis esta relacionado directamente con una de las técnicas de la programación evolutiva llamada "Programación Genética" (PG) descrita inicialmente por John R. Koza en 1992. La técnica de programación genética no es necesariamente superior a otras técnicas de computación evolutiva o de inteligencia artificial en general. Históricamente, los distintos métodos propuestos por la investigación en computación evolutiva han tenido éxitos y fracasos y la programación genética no es una excepción a esta regla. Se trata, más bien, de una propuesta más dentro de este tema de estudio.

El abordar un trabajo de desarrollo de una herramienta de solución de problemas basada en el paradigma de programación genética surge, pues, del interés por el

estudio de dicha técnica y busca estimular en otros este interés, poniendo a disposición de los lectores los resultados del presente trabajo.

La tesis esta constituida por un primer capitulo que presenta el funcionamiento de la técnica de programación genética basado en los trabajos de J. R. Koza (1992 y 1994) y menciona las motivaciones del autor original al proponer esta nueva técnica. Adicionalmente, se discute la implementación de dos tipos de máquinas de programación genética que cumplen fundamentalmente las mismas tareas que la que se desarrollará a lo largo de este trabajo.

El segundo capitulo busca justificar la construcción de una nueva herramienta de programación genética utilizando el lenguaje de programación "Java" y presenta el diseño desarrollado para su implementación.

El tercer capitulo demuestra el uso de la herramienta a través de la solución de un problema básico con el fin de demostrar su correcto uso y funcionamiento.

El último capitulo hace una reflexión sobre los resultados del trabajo y el diseño del motor producido, así como sobre la experiencia en torno al uso de la herramienta elegida para el desarrollo.

1. Introducción a la programación genética

El objetivo de este capítulo es introducir al lector a los elementos y procedimientos involucrados en la técnica de programación genética. En él se describirán las estructuras bajo adaptación utilizadas, para después presentar los procedimientos que operan sobre éstas para promover su adaptacion.

1.1. La idea fundamental detrás de la programación genética

El proceso básico utilizado por la programación genética, común a otras técnicas de computación evolutiva, es el uso de mecanismos aleatorios o pseudoaleatorios[1] para la creación de poblaciones de individuos y para su evolución hacia generaciones cada vez más aptas en la solución de un problema.

Más allá de esta similitud, una de las caracteristicas de diferenciación de las técnicas evolutivas es el concepto de individuo utilizado por cada una de ellas, es decir, la definición estructural y semántica de aquello que se considera un individuo en evolución. Por ejemplo, el algoritmo genético original define un individuo como una cadena de bits de longitud fija que, bajo un enfoque de interpretación específico, expresa una posible solución a un problema determinado. En este renglón la programación genética define a un individuo como una estructura de árbol cuyos nodos son bloques funcionales que operan sobre el conjunto de operandos representados por las ramas que penden de él.

Otra parte fundamental en el funcionamiento de las técnicas evolutivas es el promover la convergencia de sus individuos hacia poblaciones con mejores caracteristicas de solución del problema planteado, a pesar de la característica aleatoria subyacente en el proceso. Este efecto es logrado mediante la presencia de

[1] En este trabajo usaré los términos "aleatorio" y "pseudoaleatorio" indistintamente para referirme a procesos algorítmicos para la generación de series de números.

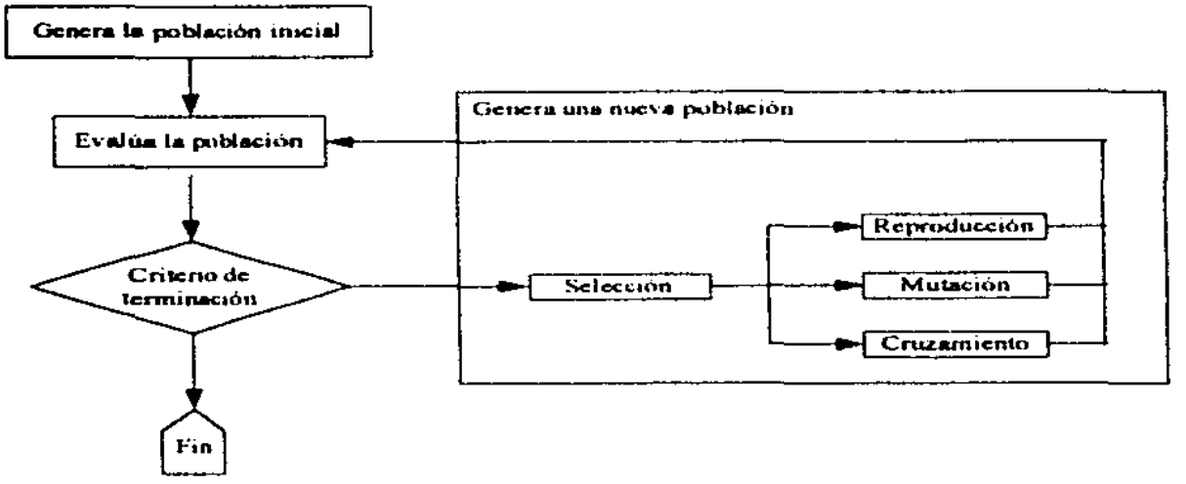
una presión evolutiva que favorece, en una forma no determinística, la supervivencia de los individuos más aptos y, por lo tanto, tiende a reducir la esperanza de vida de individuos menos aptos.

La presión evolutiva mencionada es implementada a través de un mecanismo capaz de determinar una métrica de aptitud[2] para cada individuo de una población. Este mecanismo, llamado generalmente función de evaluación, es utilizado para clasificar a los individuos de una generación y llevar a cabo sobre ellos un proceso de transformación para conformar una nueva generación de individuos. Los individuos de una generación son seleccionados en forma aleatoria para ser trasformados y pasar a formar parte de la nueva generación. El mecanismo de selección aleatoria es sesgado porque favorece la selección de individuos con mayor nivel de aptitud.

Es así como el proceso de selección aleatoria sesgada produce un efecto en la convergencia de las poblaciones que resulta superior a una metodología de búsqueda ciega, es decir, a una metodología completamente aleatoria.

En resumen, la técnica de programación genética representa la exploración aleatoria dirigida de un espacio solución conformado por todos los posibles programas formulables en términos de la combinación de un conjunto finito de bloques funcionales. Esta exploración se lleva a cabo en forma repetida a lo largo de generaciones de individuos hasta que se cumple algún criterio de terminación predeterminado. El proceso general es esquematizado en la siguiente figura:

[2]El término "metrica de aptitud" se refiere a un valor comparable que califica la capacidad de un individuo para desenvolverse en un entorno determinado, en el caso de la programación genética se utiliza específicamente para expresar la capacidad de solución de un problema.



1.2. Las estructuras bajo evolución: los individuos

En la sección anterior se señala que una de las características únicas de la programación genética en comparación con otras técnicas de computación evolutiva es la estructura y semántica de los elementos bajo evolución, es decir, los individuos. En esta sección se presentará el tipo de estructuras sujetas al proceso evolutivo en la técnica de programación genética.

1.2.1. Bloques constructivos: los funcionales

En la técnica de programación genética, los individuos son estructuras de árbol cuyos nodos son bloques constructivos y operativos conocidos como funcionales. Un funcional es concebido como un elemento que aplica una operación especifica a un número finito (posiblemente cero) de operandos para producir un resultado. Los operandos utilizados por cada bloque funcional son a su vez subárboles de funcionales que penden de él en la estructura.

Esta descripción plantea una clasificación importante de los funcionales entre los que utilizan cero operandos contra los que utilizan uno o más de ellos; a los primeros se

le conoce como funcionales terminales y a los segundos como funcionales no terminales. Evidentemente, la nomenclatura de la clasificación está directamente vinculada al hecho de que un funcional terminal (que no utiliza operandos) representa un nodo hoja en la estructura del árbol

Así, en la estructura que conforma a un individuo, el objetivo de un funcional no terminal es el de efectuar alguna operación sobre sus operandos y/o sobre el estado del problema en cuestión y producir un resultado. Por otro lado, el objetivo de los funcionales terminales es generalmente el de representar valores constantes o el valor de las variables de estado del problema, es decir, producir un resultado sin el uso de operandos. Semánticamente, la estructura completa describe una composición de las funciones individuales de cada uno de los nodos del árbol.

Los bloques funcionales son comúnmente representados mediante una notación como la siguiente:

(operación operando$_1$, operando$_2$, ... , operando$_n$).

Por ejemplo, las notaciones alternativas *(suma op_1, op_2)* o *(+ op_1, op_2)* pueden ser usadas para representar un funcional no terminal que efectúa y devuelve la suma del valor de sus dos operandos; la notación *(3.1415)* representa un funcional terminal que devuelve una aproximación al número π., o *(temp)* como un funcional que devuelve la temperatura de algún elemento en el universo del problema.

Estos ejemplos nos hacen relacionar el comportamiento de los funcionales con el de funciones matemáticas tradicionales en dominios numéricos. Pero, imaginemos por ejemplo *(if op_1, op_2, op_3)* como un funcional que dependiendo del valor de su primer operando, evalúa y devuelve en forma selectiva el valor de su segundo o tercer operando. En realidad, es posible concebir funcionales capaces de realizar tareas complejas en dominios no necesariamente numéricos.

Utilizando estos bloques constructivos, la técnica de programación genética buscará estructuras de composición de funcionales capaces de realizar una tarea predeterminada. Los tipos y comportamientos de los funcionales que serán aplicados en una corrida de programación genética son diseñados en forma predeterminada por el usuario de la técnica y ésta no contempla ningún mecanismo para determinar la pertinencia del conjunto utilizado. De hecho, es ésta la labor creativa que el usuario debe aportar en la búsqueda de una solución a un problema específico.

1.2.2. El conjunto de funcionales y el problema de suficiencia

Así, al abordar un problema, se plantea la importante labor de la elección de un conjunto de funcionales terminales y no terminales suficiente para expresar una solución al problema propuesto. Por ejemplo, es matemáticamente demostrable que el conjunto de funcionales booleanos no terminales:

$\{(or\ op_1,\ op_2),\ (and\ op_1,\ op_2)\}$

resulta insuficiente para componer ciertas funciones booleanas independientemente del conjunto de funcionales terminales especificados.

Por otro lado, el conjunto:

$\{(nand\ op_1,\ op_2)\}$

será suficiente para este mismo fin siempre y cuando el conjunto de funcionales terminales sea también suficiente en términos de las variables relevantes del problema.

Es importante insistir aquí en que la técnica de programación genética no hace ninguna evaluación de los funcionales elegidos por el usuario. Su correcta elección, diseño e implementación deben surgir directamente de él a través de sus conocimientos sobre el problema. Este requerimiento, del cual depende directamente

el éxito de la técnica, parece en principio una debilidad importante del método, sin embargo, su creador menciona que es equiparable a otras prácticas necesarias en diferentes técnicas de aprendizaje automático que en ocasiones no son identificadas y discutidas por los investigadores al describir dichos paradigmas (Koza, 1992, p. 88). De esta manera, la necesidad inicial de una correcta elección de funcionales no hace a la técnica de programación genética inferior a otras metodologías existentes para la solución de problemas.

1.2.3. El conjunto de funcionales y el problema de cerradura

En el diseño de un conjunto de funcionales para atacar un problema, además de enfrentarse al problema de suficiencia, el usuario debe enfrentarse a la solución de un requerimiento de cerradura entre el dominio y el codominio de las funciones o mapeos implementados por sus funcionales.

Debido a que las estructuras formadas expresan composiciones de los mapeos específicos de los funcionales involucrados, es necesario que el conjunto de resultados generado por cualquiera de ellos sea aceptable como valor de operando para cualquier otro.

Este problema puede ser ejemplificado de manera sencilla en un funcional:

(divide op_1, op_2*)*

que devuelve el valor resultante al dividir el valor de su primer operando entre el valor del segundo. Cuando el segundo operando tiene un valor de cero, el resultado de este funcional no está matemáticamente definido y puede no ser aceptable como valor de operando para otro funcional.

Para garantizar la cerradura en un conjunto de funcionales, pueden utilizarse distintas técnicas dependiendo del dominio del problema abordado. En el caso específico de la división, muchos programadores de programación genética eligen producir un

resultado cero cuando el valor del segundo operando es cero. Sin embargo, también es posible extender el dominio y codominio de los funcionales para codificar y manejar valores especiales como *nil* y definir, por ejemplo:

(divide op_1*, 0)* = *nil*,
(suma nil, op_2*)* = *nil*,
(resta op_1*, nil)* = *nil*,
. . . etc.

El abordar la idea de cerradura conduce pues, directamente, a mencionar el dominio del problema y, por lo tanto, el dominio y codominio del conjunto de funcionales. Cabe mencionar que, si bien a primera vista se antoja pensar en implementaciones de programación genética sobre dominios numéricos o booleanos -ambos ya mencionados-, a través de un diseño e implementación adecuados es posible plantear conjuntos de funcionales para operar en forma suficiente y cerrada sobre dominios más complejos como pueden ser listas, árboles, redes, o cualquier estructura de datos que se desee.

1.2.4. Las estructuras de composición de funcionales

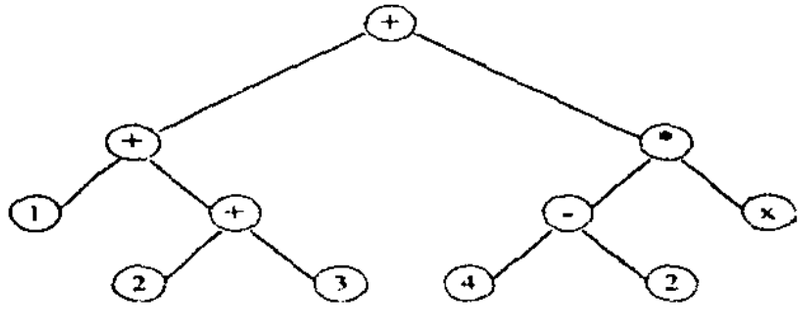
Una vez presentadas las ideas básicas en torno a las características de los elementos funcionales utilizados y las restricciones que deben cumplir para poder formar composiciones entre ellos, es interesante mostrar como dicha composición en varios niveles de estos elementos da lugar a estructuras que pueden ser representadas adecuadamente en diagramas de árbol. Esta es, de hecho, la estructura de elección para su almacenamiento y manejo en la técnica de programación genética. Por ejemplo, la composición:

(suma (suma 1, (suma 2, 3)), (multiplica (resta 4, 2), x))

basada en el conjunto de funcionales:

{ *(x), (1), (2), (3), (4), (5),*
(suma op_1, op_2*), (resta* op_1, op_2*),*
(multiplica op_1, op_2*), (divide* op_1, op_2*)* }

implementa la función lineal $f(x) = 2x + 6$, que es claramente representable en el árbol de composición mostrado en la siguiente figura:



1.3. La generación inicial de una población

Ahora bien, todo el proceso de evolución de los individuos de una población llevado a cabo por la técnica de programación genética inicia con la producción de una población inicial.

La construcción del árbol de composición de un individuo se basa en la elección aleatoria de instancias de los tipos de funcionales determinados por el usuario. Estas elecciones son efectuadas bajo un conjunto de restricciones que controlan el resultado global del proceso. Las restricciones están asociadas a un par de parámetros que especifican la longitud (profundidad) máxima de construcción que puede tener el árbol y a un indicador que determina si el árbol debe tener o no una estructura balanceada, es decir, si todas sus ramas deben ser de la misma longitud. Así, el proceso de construcción de un individuo puede desglosarse como sigue:

1. Elegir aleatoriamente un funcional no terminal para el nodo raiz del árbol; esta exclusión de los funcionales terminales se hace con el objeto de no generar árboles formados por un sólo nodo.

2. Elegir aleatoriamente suficientes funcionales para ocupar el lugar de cada uno de los operandos del funcional recién generado, de acuerdo a las siguientes reglas:

   - si el nivel de profundidad a ocupar por el operando es igual a la profundidad máxima permitida, el funcional se elige de entre el subconjunto de funcionales terminales, de manera que el árbol no crezca más allá de dicha profundidad.

   - si el nivel de profundidad es aún menor a la profundidad máxima y el árbol debe ser balanceado, el funcional se elige de entre el subconjunto de funcionales no terminales para garantizar que todas las ramas continúen su crecimiento hasta alcanzar el nivel máximo de profundidad.

   - si el nivel de profundidad es menor a la profundidad máxima y el árbol no requiere necesariamente ser balanceado, el funcional se elige -con igual probabilidad- de entre cualquiera de los funcionales del problema.

3. Se repite el paso número 2 hasta que todos los funcionales del árbol cuenten con los operandos que requieren para su funcionamiento.

Este proceso permite la construcción aleatoria de árboles de composición pero, a la vez, limita su crecimiento descontrolado más allá de la cota de profundidad máxima. Por supuesto, el número de funcionales que conforma a cada árbol (su complejidad) varía de acuerdo al número de operandos requeridos por cada uno de los funcionales elegidos.

A su vez, la generación de una población completa de individuos está controlada principalmente por el número de individuos a generar y por un parámetro conocido como el método de generación, del cual existen cinco variantes.

- el método "completo", en el cual todos los individuos son balanceados y, por lo tanto, crecen uniformemente hasta la profundidad máxima;
- el método de "crecimiento", en el cual los individuos no son balanceados, de manera que no necesariamente llegan a alcanzar la profundidad máxima,
- el método "completo en rampa", en el cual los individuos son balanceados y se construyen en grupos de profundidad creciente que van desde 2 niveles hasta la profundidad máxima, creando el mismo número de individuos para cada nivel de profundidad. Es decir, si la población debe constar de 1,000 individuos con una profundidad máxima de 5 niveles, se construyen 250 individuos de profundidad 2; 250 de profundidad 3; 250 de profundidad 4 y 250 individuos de profundidad 5;
- el método de "crecimiento en rampa", el cual utiliza la técnica de profundidades crecientes, pero los individuos no son balanceados y;
- el método de "rampa combinada", en el cual los niveles de profundidad crecen de la misma manera que en los dos métodos anteriores, sin embargo, el parámetro de balanceo se elige aleatoriamente con una probabilidad igual para cada una de las dos opciones de balanceo.

De estos cinco métodos, el de "rampa combinada" produce poblaciones con una variedad máxima de tamaños y configuraciones de sus individuos, y es el método favorecido por J. R. Koza en la presentación de sus trabajos (1992, p. 93).

1.4. Evaluación de los individuos de una población

La discusión anterior cubre completamente la metodología básica descrita por Koza (*ibid.*) para la creación de poblaciones para su posterior evaluación y modificación en ciclos repetitivos. A continuación, se presenta la manera en la que se lleva a cabo la tarea de evaluación.

Una vez que se cuenta con una nueva población, ya sea una población inicial o una nueva generación, el siguiente paso consiste en hacer una evaluación de la aptitud de sus individuos para solucionar el problema deseado.

Como se mencionó con anterioridad, la implementación de este proceso queda en manos del usuario de la técnica de programación genética y, por lo tanto, su funcionamiento es específico al problema que se analiza.

En general, la tarea consiste en someter a los individuos -aislados o en grupos- a un proceso "calificador" que los evalúa. Este mecanismo calificador debe ser diseñado por el usuario para otorgar a cada individuo una medida de aptitud que refleje de la manera más veraz posible su capacidad para cumplir la tarea predeterminada.

1.5. El uso de una métrica de aptitud comparable

Por supuesto, existen muchas maneras diferentes para expresar la medida de aptitud requerida para la diferenciación de los individuos de la población. El uso de valores numéricos fácilmente comparables, por ejemplo, es una alternativa válida, pero aún así queda a discusión el dominio, rango e interpretación de los valores que se utilizarán al expresar esta aptitud.

Por ejemplo, en un problema en el que se evalúen varios casos tipo para cada individuo, puede usarse una medida que refleje el número de decisiones acertadas registradas. Bajo este escenario, las calificaciones serían números enteros con un

rango desde cero hasta el número de casos evaluados, donde una calificación mayor denota un mayor grado de aptitud

De manera similar, pueden plantearse situaciones en las que la medida de aptitud corresponda a una cantidad en el dominio de los números reales negativos, o negativa y positiva o, donde cantidades menores denoten mayores grados de aptitud.

Siguiendo el ejemplo de Koza (1992, p. 97), la metodología elegida en este trabajo hace uso de una "medida de aptitud ajustada" (f_a) representada por un valor real positivo, de acuerdo con la regla de que un valor mayor denota un nivel de aptitud superior. La medida recibe el calificativo de "ajustada" en virtud de que en muchos esquemas de evaluación es necesario expresar una regla de transformación para convertir el valor de aptitud obtenido en un número real positivo que refleje la regla de ordenamiento señalada.

La medida de aptitud ajustada aplicada a cada individuo permite el cálculo, para cada individuo, de una segunda cantidad llamada "medida de aptitud estandarizada" (f_s):

$$f_s(i) = f_a(i) \,/\, \Sigma f_a$$

dónde $f_x(i)$ denota la medida de aptitud "x" del i-ésimo individuo de la población.

Esta otra medida ofrece una característica especial: la sumatoria de f_s sobre todos los individuos de la población tiene un valor total de uno, esto permite hacer una selección de individuos -tal y como se detallará más adelante- haciendo uso de números con valores reales comprendidos en el intervalo [0, 1].

1.6. La selección de individuos basada en su aptitud

Una vez concluido el proceso de evaluación de toda la población de manera acorde con las reglas mencionadas, la técnica de programación genética describe la manera de producir una nueva generación, es decir, una nueva población, a través de la

aplicación de métodos de mutación, recombinación o reproducción sobre los individuos recién evaluados. Estos procesos de mutación y recombinación son aplicados a individuos elegidos aleatoriamente con un sesgo que favorece la elección de aquéllos que presentaron una mayor aptitud durante el proceso de evaluación.

Antes de comenzar el proceso de selección de individuos, la población es ordenada de acuerdo a su medida de aptitud estandarizada.

Después de efectuar este ordenamiento, el mecanismo de elección sesgada puede realizarse fácilmente mediante la elección de un número aleatorio a en el rango [0, 1]. Esto permite extraer de la población al individuo $I(n)$ para el cual el área bajo la curva descrita por las f_i es igual o mayor al número elegido:

$$n_{\text{mínimo}} \quad \text{tal que} \quad \sum I(i) \geq a \quad \text{con } i = (1, 2, \ldots, n)$$

1.7. Operaciones de transformación

Valiéndose del método de selección de individuos antes descrito, la técnica permite extraer de la población copias de uno o más individuos para someterlos a un proceso de reproducción, mutación o cruzamiento e insertar sus resultantes en una nueva generación.

Esta tarea de selección de individuos, aplicación de transformaciones e inserción en la nueva población continúa hasta que la nueva población alcanza o rebasa el número de integrantes definidos por el parámetro de tamaño poblacional. Al concluir este proceso repetitivo, se cuenta con un nuevo grupo de individuos listos para ser sometidos al proceso de calificación de aptitud.

La elección de uno de los tres métodos de transformación que se utilizará se logra en forma aleatoria de acuerdo a una distribución de probabilidades para la ocurrencia de

cada tipo de transformación, la cual es determinada por el usuario al inicio de la corrida.

1.7.1. Operaciones de transformación: reproducción

La operación de reproducción requiere de la elección de un individuo de la población actual para ser duplicado fielmente en la siguiente generación de individuos.

1.7.2. Operaciones de transformación: mutación

El caso de la mutación es ligeramente más complicado.

Después de haber elegido un individuo de la población actual, se elige al azar un nodo de su árbol de composición. El nodo elegido y todo el subárbol que pende de él es removido y sustituido por una nueva estructura generada aleatoriamente en forma similar a como fueron generados inicialmente los árboles de los individuos.

Para efectuar esta operación, existe la restricción de que la profundidad del árbol resultante no exceda el valor de un parámetro de operación conocido como la profundidad máxima de operación del árbol.

Koza (1992, p. 106), en particular, respeta esta restricción generando un nuevo subárbol de profundidad aleatoria en un rango tal que no viole la restricción mencionada. La característica de balanceo de dicho árbol es determinada aleatoriamente con igual probabilidad para ambos valores del parámetro.

El individuo obtenido de este proceso de transformación es incluido en la nueva población.

1.7.3. Operaciones de transformación: cruzamiento

El proceso de cruzamiento o reproducción sexuada se lleva a cabo eligiendo a dos individuos de la población original, eligiendo aleatoriamente nodos en sus árboles de funcionales e intercambiando los subárboles elegidos

Los dos árboles resultado de la operación de cruzamiento, igual que en la operación de mutación, deben respetar la restricción de profundidad máxima de operación. Cuando un resultado viola la restricción, Koza (*ibid*, p. 104) elige insertar en la nueva población no el resultado transgresor, sino una copia fiel del padre que le dio origen. Así, la operación de cruzamiento puede convertirse en un cruzamiento y una reproducción directa o en dos reproducciones directas.

La implementación de la operación de cruzamiento diseñada para la máquina de programación genética que aquí se presenta recurre a la repetición, tantas veces como sea necesario, de la elección de los puntos de cruzamiento en los individuos originales hasta obtener una combinación que arroje dos resultados válidos. Considero que esta variante preserva con mayor fidelidad la proporción de reproducciones, mutaciones y cruzamientos elegidos por el usuario para realizar la corrida de su problema.

Al finalizar la operación de cruzamiento, ambos resultados son incluidos en la nueva generación de individuos.

1.8. El criterio de terminación

Todo el mecanismo de evaluación y transformación de poblaciones se repite para un número máximo de generaciones o, en algunos casos, hasta que se alcanza un criterio de terminación típicamente basado en la obtención de una calificación de aptitud máxima por uno de los individuos de la población en alguna de las generaciones producidas.

1.9. La extensión de ADFs sobre la técnica básica

Hasta este punto, la estructura presentada para un individuo consiste de un único árbol de composición de funcionales y, la técnica básica presentada explora el espacio de posibles árboles de composición en busca de una instancia capaz de solucionar un problema especifico.

Como complemento a la técnica básica, Koza y Rice (1992) han presentado y discutido una modificación conocida con el nombre de "Definición Automática de Funciones" (ADFs por sus siglas en inglés). Esta extensión provee a la programación genética de la capacidad de definir subfunciones para utilizarlas como parte de una solución, de la misma manera en la que un programador usa subrutinas o subprogramas.

El uso de ADFs, al igual que el uso de subprogramas en un lenguaje de programación, facilita la expresión de soluciones a problemas cuya resolución requiere de un alto grado de regularidad en ciertas funcionalidades parciales. Dicho de otra manera, el método permite resolver problemas cuyo árbol de solución contiene un gran número de secciones que realizan exactamente la misma función.

Cuando un programador de computadoras identifica la necesidad de una cierta rutina en más de un punto de su programa, generalmente, escribe un subprograma que es invocado desde dos o más puntos dentro del programa principal, evitando asi codificar el mismo comportamiento en ocasiones repetidas. Este es exactamente el beneficio que las ADFs brindan a la técnica de programación genética.

Al abordar un problema con una alta regularidad en su solución, la técnica básica de programación genética debe encontrar un árbol de composición dentro del cual existan varias estructuras funcionalmente equivalentes que expresen la regularidad de la solución. Este fenómeno hace que la probabilidad de éxito se reduzca conforme aumenta el número de ocurrencias de estructuras funcionalmente equivalentes, esto

se debe a que crece el número de veces que debe evolucionar un mismo comportamiento en diferentes puntos del árbol solución.

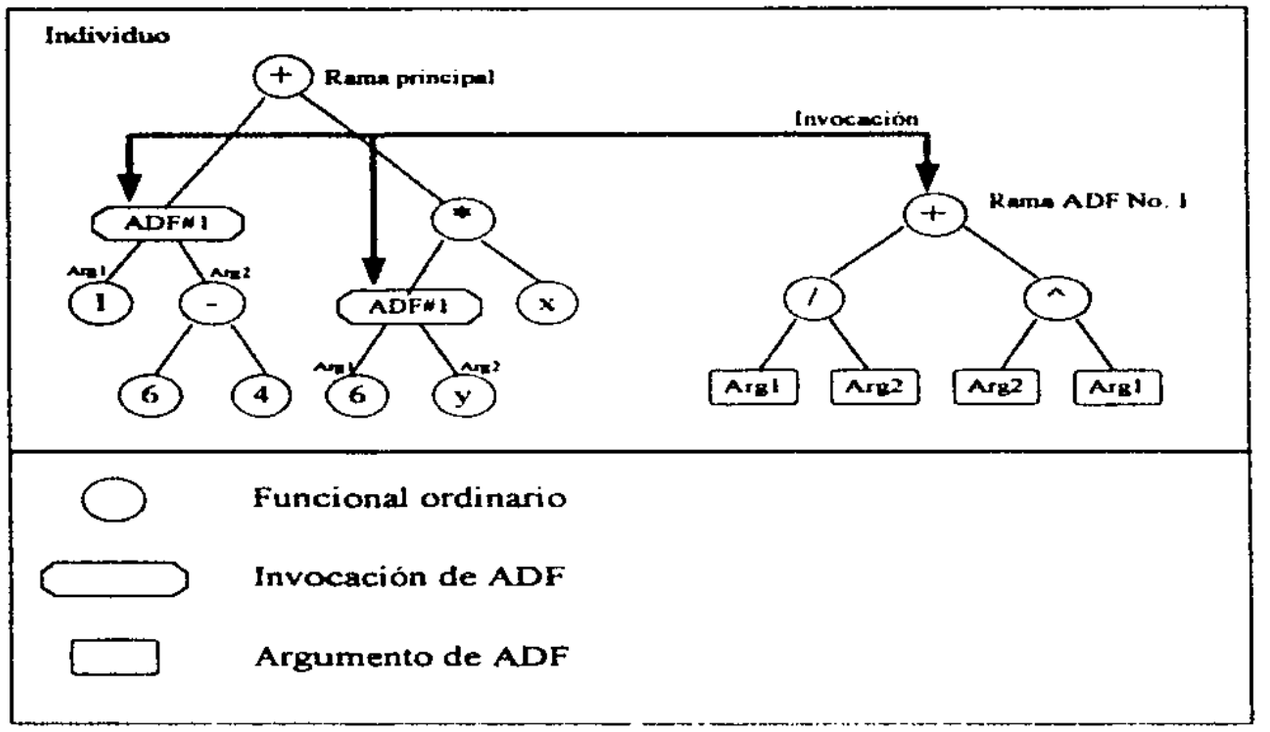
1.9.1 Implementación y uso de ADFs

En programación genética, la invocación del comportamiento de una ADF se expresa como un funcional de uno o más operandos, esto permite que un árbol contenga un gran número de copias de este funcional para representar la regularidad de la solución.

Ahora bien, este funcional no terminal que invoca un cierto comportamiento difiere de los funcionales diseñados por el usuario en que su comportamiento no está predeterminado al inicio del proceso de solución. Los funcionales que invocan el comportamiento de una ADF son representaciones simbólicas de un árbol de composición adicional contenido también dentro del individuo.

Este, o estos árboles adicionales, están sujetos también a los procesos de mutación y cruzamiento de la técnica, con lo que se persigue la evolución automática de funciones que formen parte de la solución global del problema.

De esta manera, la técnica de uso de ADFs modifica la estructura de un individuo de manera que éste contiene ahora ya no un árbol de composición de funcionales sino varios. Uno de ellos es el árbol principal usado en la técnica básica y los demás expresan ADFs que pueden ser invocadas como parte de la solución del problema.



Por ser un subprograma, una ADF implementa una cierta funcionalidad que opera sobre un conjunto de argumentos. Esto se expresa mediante el uso de funcionales terminales en el árbol de la ADF que representan los argumentos sobre los cuales el resto de la estructura debe actuar. El valor específico de cada argumento no se determina sino en el momento de invocación, de acuerdo a los operandos del funcional que invoca a la ADF.

1.9.2 Definición constructiva de una ADF

De manera similar al caso de la rama[3] principal en la técnica básica, la construcción de una ADF está determinada por una profundidad máxima de construcción, una profundidad máxima permitida y un conjunto de funcionales no terminales. En el

[3]Uso aquí, y en el resto del trabajo, el término "rama" como sinónimo de los teérminos "arbol" y "subarbol".

caso de una ADF, ésta recibe, además, un nombre y un número fijo de argumentos. Este conjunto de parámetros constituye el tipo de una ADF.

Para completar los elementos necesarios para su construcción, todas las ADFs utilizan el mismo conjunto de terminales, el cual cuenta con un sólo elemento que sirve para representar los argumentos que ésta recibe.

En una corrida especifica se dota a cada individuo del mismo número y tipo de ADFs, además de su rama principal. Es decir, no sólo cada individuo contiene el mismo número de ADFs sino que, además, las caracteristicas de las ADFs de un individuo particular se ven replicadas en cada uno de los individuos de la población.

En general, los conjuntos de no terminales utilizados en la construcción de cada rama son diferentes; esto plantea una restricción sobre la operación de cruzamiento usada en la técnica básica.

1.9.3 Cruzamiento utilizando ADFs

Es un hecho que las diferentes ramas que conforman a un individuo tienen caracteristicas únicas que es deseable preservar. Por ejemplo, si la operación de cruzamiento tradicional actúa sobre dos individuos en puntos que pertenecen a ADFs de tipo distinto, los resultados de la cruza probablemente violarán la determinación del conjunto de no terminales que deben constituirlas.

La solución propuesta a este problema es el uso de una operación de cruzamiento que restringe la elección de las ramas a cruzar de acuerdo a sus caracteristicas de construcción.

Así, la técnica de programación genética con ADFs modifica la operación de cruzamiento de manera que se elige aleatoriamente un punto de cruzamiento en el primero de los padres involucrados. Posteriormente, y siempre de manera aleatoria, se elegirá otro punto de cruzamiento en el segundo padre, sólo dentro del conjunto

de nodos que conforman a la rama del mismo tipo a la que corresponde el nodo del primer padre.

De esta manera, la nueva operación de cruzamiento conserva las características utilizadas en la construcción de las ramas involucradas. Esto es, una estructura perteneciente a la rama principal del primer padre se intercambia solamente por una estructura de rama principal en el segundo padre, o una estructura de la n-ésima ADF del primer padre se intercambia sólo por una estructura de la n-ésima ADF del segundo padre.

1.10 En torno a la técnica de programación genética

En su presentación del paradigma de programación genética, Koza (1992) hace un breve recuento del desarrollo histórico de las variantes de la técnica de algoritmo genético tradicional y muestra cómo los elementos en evolución, elegidos por diversos autores, han ido creciendo en complejidad estructural. En este contexto, Koza ofrece la idea de la programación genética como una extensión natural a este proceso de uso de estructuras cada vez más complejas y, a la vez, argumenta que su propuesta reformula el problema en forma general como la búsqueda de un programa de computadora en el espacio de posibles programas enumerables con base en un conjunto de elementos funcionales.

Koza argumenta en favor de su técnica la naturalidad con la que es posible expresar soluciones a problemas en dominios variados, así como ofrecer la capacidad inherente de los programas de computadora para:

- efectuar operaciones de manera jerárquica,
- ejecutar operaciones alternas basadas en el resultado de cálculos intermedios,
- implementar métodos de iteración y recursividad,
- manejar resultados de tipos diversos y

- definir valores y subprogramas para ser utilizados repetidamente.

Además, la técnica de programación genética no requiere de una especificación *a priori* de la estructura o complejidad (en términos de número de funcionales) para la búsqueda de una solución. El algoritmo mismo ajusta dinámicamente estas características durante su ejecución.

2. Presentación del diseño de la máquina

2.1. La idea de un motor para programación genética

De la presentación hecha en el capítulo anterior sobre los elementos y procesos que conforman la técnica de programación genética, se desprende la existencia de dos actividades distintas en la búsqueda de una solución a través de este método.

La primera, de tipo creativa, involucra el diseño y desarrollo de un conjunto de elementos funcionales y la preparación de un mecanismo de evaluación capaz de dar una calificación de adecuación a cada uno de los individuos de una población.

La segunda actividad es de tipo repetitiva y consiste en la generación de una población inicial para después, de forma repetida, evaluarla, ordenarla y modificarla.

Como fue ya mencionado, el diseño e implementación del conjunto de funcionales para expresar una solución al problema, así como el mecanismo de evaluación de individuos, son tareas que la técnica de programación genética deja completamente en manos del usuario.

La actividad repetitiva de evaluación y transformación de poblaciones de individuos, por su parte, puede ser expresada en una forma general adecuada a cualquier tipo de problema. Esto ofrece la ventaja de poder utilizar un mismo núcleo de evaluación/transformación para cualquier conjunto de funcionales y sistema de evaluación deseados. A este sistema, concebido para llevar a cabo las tareas repetitivas y predefinidas del paradigma de programación genética lo llamaremos una "maquina de programación genética" o un "motor de programación genética".

2.2. Implementaciones existentes de máquinas de programación genética

Como parte del desarrollo de la técnica de programación genética, Koza (1992) ha desarrollado una máquina implementada en el lenguaje LISP para la exploración de la capacidades de éste paradigma. La elección de la herramienta de desarrollo hecha por el autor esta basada en la capacidad del lenguaje para manipular programas expresados en LISP como estructuras de datos y la sencillez con la que se pueden ejecutar estructuras de datos que almacenan programas. Estas dos caracteristicas son centrales en la manipulación y evaluación de los árboles de composición utilizados por la técnica de programación genética. La máquina desarrollada por Koza es una de las herramientas existentes para la exploración de problemas por medio de la técnica de programación genética.

Por otro lado, existe otra implementación de esta herramienta desarrollada por Adam P. Fraser[4] que utiliza el lenguaje de programación C++. Esta máquina de programación genética ofrece acceso al paradigma en un lenguaje de uso común (en comparación con LISP) y agrega los beneficios de eficiencia de procesamiento alcanzables utilizando un lenguaje ejecutable en código binario, lo cual contrasta con el uso de LISP, que es típicamente un lenguaje interpretado.

El código fuente para las dos herramientas mencionadas esta disponible al público y puede ser utilizado y distribuido libremente en actividades no lucrativas. Esto ofrece, a cualquier interesado, una base de arranque para la exploración de problemas partiendo de una plataforma de disponibilidad completa e inmediata.

2.3. Una nueva implementación en lenguaje Java

A partir de lo ya explicado, cobran importancia en el presente trabajo dos preguntas distintas: ¿porqué considerar siquiera el reproducir la implementación de las máquina

[4]Véase http://www.io.com/~ftp/genetic-programming/code/

de programación genética ya existentes? y, de hacerlo, ¿porqué considerar hacerlo utilizando el lenguaje de programación Java?. La respuesta esta directamente relacionada con varias de las capacidades que el nuevo lenguaje Java ofrece.

En primer lugar, Java promete ser -al igual que C++- una herramienta disponible para casi cualquier equipo de cómputo del que se disponga y, además, ofrece ciertos estándares de implementación que, si bien no son completos, si resultan un gran apoyo para el desarrollador en la tarea de producir código altamente portable. Esto contribuye al diseño e implementación de un motor de programación genética que pueda ser usado en cualquier plataforma disponible, sin la necesidad de adecuar el código fuente existente.

Por otro lado, la definición sintáctica y semántica de Java, cercana a la del lenguaje C++, ofrece un entorno de programación familiar a un gran número de programadores, a la vez que corrige una serie de desventajas que C++ heredó del antiguo lenguaje C. Estas modificaciones básicamente semánticas implementadas por Java ofrecen un marco de desarrollo un poco más estricto que contribuye a mejorar la legibilidad y calidad del código.

Java ofrece un sistema de manejo de memoria basado en un recolector de basura. El uso de recolectores que reaprovechan automáticamente los recursos consumidos por elementos dinámicos que dejan de ser necesarios durante la ejecución de un programa, libera al desarrollador de la tediosa tarea de liberar explícitamente estos recursos, eliminando, además, la posibilidad de cometer errores en la administración de dichos recursos.

Sobre estas caracteristicas, Java ofrece además una serie de capacidades "modernas" requeridas en los nuevos lenguajes de programación; entre las cuales se encuentra, por un lado, la posibilidad de un sencillo desarrollo de procesos en tramas múltiples

de programación genética ya existentes? y, de hacerlo, ¿porqué considerar hacerlo utilizando el lenguaje de programación Java?. La respuesta esta directamente relacionada con varias de las capacidades que el nuevo lenguaje Java ofrece.

En primer lugar, Java promete ser -al igual que C++- una herramienta disponible para casi cualquier equipo de cómputo del que se disponga y, además, ofrece ciertos estándares de implementación que, si bien no son completos, si resultan un gran apoyo para el desarrollador en la tarea de producir código altamente portable. Esto contribuye al diseño e implementación de un motor de programación genética que pueda ser usado en cualquier plataforma disponible, sin la necesidad de adecuar el código fuente existente.

Por otro lado, la definición sintáctica y semántica de Java, cercana a la del lenguaje C++, ofrece un entorno de programación familiar a un gran número de programadores, a la vez que corrige una serie de desventajas que C++ heredó del antiguo lenguaje C. Estas modificaciones básicamente semánticas implementadas por Java ofrecen un marco de desarrollo un poco más estricto que contribuye a mejorar la legibilidad y calidad del código.

Java ofrece un sistema de manejo de memoria basado en un recolector de basura. El uso de recolectores que reaprovechan automáticamente los recursos consumidos por elementos dinámicos que dejan de ser necesarios durante la ejecución de un programa, libera al desarrollador de la tediosa tarea de liberar explícitamente estos recursos, eliminando, además, la posibilidad de cometer errores en la administración de dichos recursos.

Sobre estas caracteristicas, Java ofrece además una serie de capacidades "modernas" requeridas en los nuevos lenguajes de programación; entre las cuales se encuentra, por un lado, la posibilidad de un sencillo desarrollo de procesos en tramas múltiples

(*multithreaded*) y, por el otro, la implementación de código "viajero" para producción de sistemas de cómputo distribuido[5].

Históricamente, estas capacidades han sido ofrecidas por los desarrolladores de sistemas operativos como conjuntos de librerías de servicios las cuales, si bien similares en su concepción, varían entre sistemas operativos e incluso entre diferentes implementaciones y/o versiones de un mismo sistema operativo. A esta diversidad difícil de manejar, Java responde incluyendo estos servicios como parte de la definición base del ambiente de ejecución del lenguaje. Esto garantiza la disponibilidad de estas capacidades en una forma completamente consistente, independientemente de la plataforma o implementación utilizada.

La posibilidad de producir un sistema de multiprocesamiento o procesamiento cooperativo tiene una ventaja definitiva en la técnica de programación genética, en la cual una gran población de individuos puede ser sometida a un proceso de evaluación en forma paralela, beneficiándose tanto de equipos multiprocesadores como de conjuntos de equipos de cómputo interconectados por una red de datos.

Por otro lado, el gran inconveniente del lenguaje Java es el hecho de su ejecución en forma interpretada. Esto lo hace un sistema de cómputo de menor potencia al ser comparado con el eficiente código binario producido por un compilador de C++. Sin embargo, esta tendencia esta cambiando últimamente con el surgimiento de interpretes provistos de compiladores de tecnología JIT (*Just In Time*), capaces de producir bajo demanda y selectivamente versiones de código binario de fragmentos de pseudocódigo (llamado *byte code* en Java), lo cual incrementa significativamente su eficiencia de ejecución.

Este grupo de factores dan sentido a la idea de construir una máquina para programación genética basada en este nuevo lenguaje. Así, este trabajo se aboca al

[5]La idea de código "viajero" se refiere a código que puede ser transmitido entre equipos de cómputo en una red de datos para, de manera cooperativa, distribuir las actividades de solución de una tarea.

diseño e implementación de esta herramienta buscando un modelo funcional capaz de explotar los beneficios ofrecidos por el lenguaje Java.

2.4. Discusión del diseño

El resto de este capítulo esta dedicado a la revisión de las características del diseño de la máquina de programación genética que aquí se presenta. La presentación cubre temas de interés especial en el ámbito de la implementación, sin revisar de manera completa todo el código que la constituye.

Para aquellos interesados en conocer más profundamente los detalles de la implementación, el código en Java, así como una fuente de documentación hipertextual de la jerarquía de clases están disponibles en forma de anexos a este trabajo.

2.4.1. Representación de funcionales

Dado que el funcional representa el bloque constructivo básico para la expresión de soluciones en programación genética, su implementación merece especial atención.

Por un lado, el comportamiento específico de cada funcional utilizado en un problema debe ser especificado por el usuario de la técnica. Por otra parte, los procesos que actúan sobre estructuras de funcionales requieren de una interfase de trabajo para la manipulación de éstas.

Esta dualidad de requerimientos fue resuelta en el diseño produciendo una clase llamada `Funcional` que ofrece los métodos básicos requeridos por la máquina de programación genética para la manipulación de estructuras de estos elementos. Partiendo de esta clase, todo funcional diseñado por un usuario debe descender directa o indirectamente de esta clase.

La definición para variables y métodos de la clase funcional es la siguiente:

```
public abstract class Funcional extends Object implements Cloneable {

// Variables
   protected Funcional[]     operandos = null;

// Metodos
   public abstract Object clone();
   public int getNOp();
   public Funcional getOp( int n );
   public void setOp( int n, Funcional op );
   public int getNNodos();
   public int getProf();
   protected int getProf( int p_act, Entero p_max );
   public AtomLink buscaNodo( int n );
   protected AtomLink buscaNodo( Entero n, int prof );
   public int getNArg();
   public Funcional expandeADF( Funcional[] op );
}
```

En primera instancia, la clase Funcional es abstracta, es decir, no puede ser instanciada directamente. Esto refuerza su concepción como una base con unas ciertas caracteristicas que deben ser extendidas para implementar funcionales especificos a cada problema que se desee estudiar.

La clase cuenta con el arreglo `operandos` cuyo fin es permitir a todo funcional almacenar referencias a otros funcionales que hagan las veces de sus operandos de entrada. El tamaño del arreglo no esta especificado como parte de la definición debido a que cada clase de funcional derivada de ésta deberá especificar, como parte de su constructor, el número de operandos que requiere. Por ejemplo, una clase que implemente un funcional para la operación suma de dos operandos debe instanciar su arreglo de operandos como:

```
public class Suma extends Funcional {

// Constructor
   public Suma() {  operandos = new Funcional[ 2 ];  }

// Metodos
   public Object clone() {
      Suma clon = new Suma();
      clon.operandos[ 0 ] = this.operandos[ 0 ].clone();
      clon.operandos[ 1 ] = this.operandos[ 1 ].clone();

      return clon;
   }


   public <TIPO> eval( Individuo ind, Object ambiente ) {

      <IMPLEMENTACION>

   }


   public String toString() {
      return "(suma "
       + operandos[ 0 ] == null ? "null" : operandos[ 0 ].toString()
       + ", "
       + operandos[ 1 ] == null ? "null" : operandos[ 1 ].toString()
       + ")";
   }
}
```

Debe notarse que el constructor de la clase `Suma` no recibe ningún parámetro y que sólo establece el tamaño del arreglo para contener a sus operandos, y no ejecuta acciones para instanciarlos. Es parte de la labor de la máquina el interrogar el número de operandos requeridos por el funcional e instanciarlos aleatoriamente a partir del conjunto de funcionales del problema.

El método `clone()` mostrado debe implementarse en la nueva clase para permitir a la máquina obtener copias del funcional y de la estructura que pende de él. Generalmente, este método hace en todo funcional exactamente lo que se muestra en este ejemplo, devuelve una copia de si mismo y de la estructura de funcionales que conforman sus operandos.

Adicionalmente, la nueva clase muestra parcialmente la implementación de un método llamado `eval(Individuo, Object)` para su evaluación, pero esto no es un requerimiento fundamental del resto de la máquina, sino el mecanismo que se utilizará para invocar el comportamiento del funcional. También puede incluirse un

método `toString()` que devuelva una representación en cadena de caracteres del funcional e invoque a este mismo método para cada uno de sus operandos.

Por lo demás, los métodos de la clase `Funcional` ofrecen el comportamiento necesario para que el motor de programación genética escudriñe y manipule los árboles de funcionales que conforman a los individuos en evolución.

2.4.2. Estructuras de composición de funcionales

Como ya se mencionó, las estructuras formadas por composiciones de elementos funcionales son árboles caracterizados por sus parámetros de:

- profundidad de construcción inicial,
- característica de balanceo,
- profundidad máxima permitida,
- conjunto de elementos funcionales terminales permitidos y,
- conjunto de elementos funcionales no terminales permitidos.

Esto se ve reflejado en el diseño a través de una clase contenedora de estos parámetros cuyo nombre es `TipoRama`, y su implementación es la siguiente:

```
public class TipoRama extends Object implements Cloneable {

// Variables
   protected boolean               balance = false;
   protected int                   prof_ini = 0;
   protected int                   prof_max = 0;
   protected FuncionalFactory      terminales = null;
   protected FuncionalFactory      noterminales = null;

// Constructores
   public TipoRama();
   public TipoRama( boolean b, int pi, int pm,
                    FuncionalFactory t, FuncionalFactory nt );

// Metodos
   public Object clone();
   public boolean getBalance();
   public int getProfIni();
   public int getProfMax();
   public FuncionalFactory getTerminales();
   public FuncionalFactory getNoTerminales();
   public void setBalance( boolean b );
   public void setProfIni( int pi );
   public void setProfMax( int pm );
   public void setTerminales( FuncionalFactory t );
   public void setNoTerminales( FuncionalFactory nt );
   public String toString();
}
```

Las variables (o campos) de la clase reflejan los parámetros que caracterizan a una estructura de funcionales.

Tanto `terminales` como `noterminales` merecen mención especial por tratarse de variables de tipo `FuncionalFactory`. Baste por ahora decir que estas clases son fábricas productoras de instancias de funcionales de conjuntos predefinidos, que determinan el tipo de funcionales que pueden aparecer dentro de la estructura en cuestión.

El objeto de esta clase es constituir un contenedor de parámetros para la construcción y operación sobre estructuras de elementos funcionales representadas por la clase `Rama`:

```
public class Rama extends Object implements Cloneable {

// Variables
   protected Funcional     raiz = null;
   protected int           nodos_totales = 0;
   protected int           prof = 0;
   protected TipoRama      tipo = null;

// Constructores
   public Rama();
   public Rama( TipoRama tip );
   public Rama( TipoRama tip, int prf, boolean bal );

// Metodos
   public Object clona();
   public int getNNodos();
   public int getProf();
   public Funcional getRaiz();
   public Funcional setRaiz( Funcional nueva );
   public Funcional sustituye( AtomLink corte, Funcional nuevo );
   private Funcional nuevaRama( int p_act, int p_max, boolean bal );
   public AtomLink buscaNodo( int n );
}
```

Cada instancia de una rama contiene campos que almacenan:

- una referencia al funcional que es el nodo raiz de la rama,
- un conteo del total de nodos que la conforman,
- un indicador de la profundidad màxima actual de la rama y,
- una referencia a los parámetros usados para su construcción.

Los constructores `Rama(TipoRama tip)` y `Rama(TipoRama tip, int prf, boolean bal)`, hacen uso del método `nuevaRama()` para crear instancias de la rama con su árbol de funcionales ya construido y, Son utilizados por clases superiores para, en una sola operación, generar nuevas ramas provistas de sus árboles de funcionales.

Los demás métodos de la clase permiten el manejo de la estructura para las operaciones de transformación que la máquina deba aplicar sobre ella.

Así, esta clase representa la implementación completa utilizada para expresar a la rama principal de funcionales de un individuo.

Como se muestra a continuación, las ramas adicionales a la principal, cuyo papel es representar ADFs, requieren de algunos parámetros distintos encapsulados en la clase `TipoADF` que es una subclase formal de la clase `TipoRama`:

```
public class TipoADF extends TipoRama implements Cloneable {

// Variables
   protected String                          nombre;
   protected int                             nargs;
   protected static String[]                 t_adf = { "GP.ArgADF" };
   protected static FuncionalFactory         t = null;

// Constructores
   public TipoADF();
   public TipoADF( boolean b, int pi, int pm,
                   FuncionalFactory nt, int na, String nomb )

// Metodos
   public Object clone();
   public String getNombre();
   public int getNArgs();
   public void setNombre( String nomb );
   public void setNArgs( int na );
   public String toString();
}
```

Esta clase -descendiente directa de `TipoRama`- cuenta con los campos y métodos de su antecesora, a los cuales añade un nombre identificador, un número que indica cuantos argumentos debe recibir la ADF y un par de variables de clase usadas para contener el conjunto de terminales permitidos en su estructura.

Recordemos aquí que los funcionales terminales en la estructura de una ADF son solamente referencias simbólicas a los parámetros formales utilizados al momento de su invocación. En el diseño de la máquina, este papel es desempeñado por instancias de la clase `ArgADF`, cuyo nombre es el único citado por la variable `t_adf` que almacena los nombres de las posibles clases para expresar funcionales terminales de la estructura.

De manera similar a las instancias de la clase `TipoRama`, las instancias de esta clase encapsulan los parámetros de creación y manipulación para elementos de la clase `ADFStru`, la cual se presenta a continuación:

```
public class ADFStru extends Rama implements Cloneable {

// Variables
   protected TipoADF     tipo;

// Constructores
   public ADFStru();
   public ADFStru( TipoADF tip );
   public ADFStru( TipoADF tip, int prf, boolean bal );

// Metodos
   public Object clone();
   public String getNombre();
   private Funcional nuevaRama( int p_act, int p_max, boolean bal );
   public static Funcional expandeADF( Individuo ind, Funcional fun );
}
```

Esta sencilla clase expresa una parte importante de la funcionalidad requerida en la implementación de ADFs para la programación genética. Esto no es evidente en un primer análisis debido a la sencillez de su estructura, sin embargo, esta clase hereda completamente la funcionalidad de la clase `Rama` y adicionalmente puede echar mano de las capacidades de las clases `TipoADF` y `Funcional`.

Al igual que la clase `Rama`, sus constructores producen automáticamente el árbol de funcionales contenido en la clase.

El rol principal de una instancia de la clase `ADFStru` es el contener un árbol de composición de funcionales en la variable `raiz`, heredada de la clase `Rama`. En la estructura de este árbol todos los nodos hoja son instancias de la clase `ArgADF` que representan los argumentos de invocación de la función.

Eventualmente, la invocación de la ADF requiere de la sustitución de estos argumentos simbólicos por los argumentos formales de tiempo de invocación. Esta tarea es realizada por el método de clase `expandeADF(Individuo, Funcional)`, el cual devuelve a la instancia que lo invoca una copia de la ADF con la correcta sustitución de parámetros simbólicos por parámetros formales, de acuerdo a los operandos referenciados por `Funcional`.

2.4.3. La definición de una especie

La construcción de una población de individuos que contienen una rama principal y un cierto número de ADFs requiere de la especificación de parámetros de profundidad, balanceo y tipo de terminales a utilizar para cada una de estas ramas. Como mencioné antes, estas características son iguales para todos y cada uno de los inidivduos de la población.

Para facilitar el manejo de estos parámetros, la clase `Especie` es capaz de almacenar las características de construcción de los individuos de una población:

```
public class Especie extends Object {

// Variables
   protected String       name = null;
   protected TipoRama     def_rpb = null;
   protected Vector       defs_adfs = null;

// Constructores
   public Especie(  String nomb,
                    boolean b,  int pi,  int pm,
                     String t[], String nt[] )
       throws ClassNotFoundException;
   public Especie( String nomb, TipoRama def_rama );

// Metodos
   public TipoRama getTipoRPB();
   public int getNADFs();
   public TipoADF getTipoADF( int n );
   public void addADF( boolean b,  int pi,  int pm,
                       String nt[],  int nargs,  String nomb )
    throws ClassNotFoundException;
   public void addADF( TipoADF def );
   public String toString();
}
```

El constructor de una especie recibe un nombre para la especie (nomb) y, las características a utilizar en las ramas principales de los individuos de la población (balanceo: `b`, profundidad inicial: `pi`, profundidad máxima: `pm`, funcionales terminales: `t` y, funcionales no terminales: `nt`). Otra manera de construir una definición de especie es usar el constructor que recibe estos mismos parámetros agrupados en una instancia de `TipoRama`.

El método `addADF()` permite agregar a la especie las características de ADFs que deben estar presentes en los individuos de la población.

Los demás métodos permiten extraer la información almacenada en los campos de la clase.

2.4.4. La implementación de un individuo

Cada individuo de una población esta constituido básicamente por un árbol de funcionales principal y varios árboles que expresan ADFs para el problema. Las características de cada una de estas ramas esta especificada por una instancia de la clase `Especie`. Adicionalmente, cada individuo lleva consigo sus últimas calificaciones de adecuación ajustada y estandarizada (f_a y f_s), como se muestra a continuación:

```
public class Individuo implements Ordenable, Cloneable {

// Variables
   protected Especie       especie;
   protected Rama          rpb;
   protected ADFStru[]     adfs;
   protected float         adjusted_fitness;
   protected float         normalized_fitness;

// Constructores
   public Individuo();
   public Individuo( Especie esp, int prof, boolean bal );
```

```
// Metodos
    public Object clone();
    public float getFa();
    public float getFn();
    public void setFa( float f );
    public void setFn( float f );
    public Rama getRPB();
    public ADFStru getADFStru( String name );
    public int getNNodos();
    public AtomLink buscaNodo( Entero n_nodo );
    public int indiceRama( int n_nodo );
    public Rama referenciaRama( int n_nodo );

    public void mutacion( Poblacion pob );
    public void cruzamiento( Individuo ind, Poblacion pob );
    public int orden( Ordenable cmp );
    public String toString();
}
```

El constructor de un individuo recibe una especificación de especie y parámetros de profundidad máxima y balanceo para su rama principal. Haciendo uso de los constructores de las clases `Rama` y `ADFStru`, crea automáticamente las ramas (principal y ADFs) que lo han de conformar.

El grueso de los métodos de esta clase giran en torno al acceso y manipulación de las calificaciones y ramas del individuo, pero los cuatro métodos: `clone()`, `mutacion()`, `cruzamiento()` y `orden()` implementan algunas de las operaciones base de la programación genética.

El método `clone()` representa la capacidad de reproducción del individuo, mientras que el método `mutacion()` inserta en la población indicada una mutación del individuo sobre el que se invoca. El método `cruzamiento()` efectúa el cruce de dos individuos e inserta su descendencia en la población pasada como parámetro y, finalmente, el método `orden()` determina el criterio de ordenamiento de los individuos por medio de la comparación de sus medidas de aptitud estandarizada (f_s).

2.4.5. Poblaciones de individuos

Para manejar las colecciones de individuos, el motor utiliza la clase `Poblacion`:

```
public class Poblacion extends Object {

// Variables
   private Vector     individuos;

// Constructores
   public Poblacion( int num_i, Especie esp, String metodo );
   public Poblacion( int num_i );

// Metodos
   public void addIndividuo( Individuo ind );
   public Individuo getIndividuo( int n );
   public void setIndividuo( Individuo ind, int n );
   public void delIndividuo( int n );
   public int getNPoblacional();
   public void cruza( float p_cruza, Poblacion pob );
   public void muta( float p_mutacion, Poblacion pob );
   public void reproduce( float p_reprod, Poblacion pob );
   public void sort();
   public int fitnessSeek( float fn );
}
```

El sistema de almacenamiento de los individuos es una instancia de la clase estándar `java.util.Vector` para permitir variaciones en el número de individuos en las poblaciones de cada generación.

En esta clase se implementa la metodologia global de transformación de las poblaciones con apoyo en los métodos de la clase `Individuo`. Adicionalmente, el método `sort()` permite el ordenamiento de la población a través de un algoritmo de QuickSort, mientras que `fitnessSeek()` representa el método de selección sesgada porporcional a la aptitud usado para la selección de individuos en las operaciones de reproducción, mutación y cruzamiento.

2.4.6. El control de evaluación

Como se mencionó en el capítulo anterior, el mecanismo de calificación de los individuos es especifico a cada problema y debe ser diseñado por el usuario del motor, quien debe implementar un método de comunicación con éste. El diseño de la máquina logra esto fijando la restricción de que toda metodología de evaluación para un problema debe ser una subclase de la clase abstracta `Entrenador`:

```
public abstract class Entrenador implements Runnable {

// Variables
   protected Individuo[]     cobayos;

// Constructor
   public Entrenador();

// Metodos
   public Individuo[] getCobayos();
   public void setCobayos( Individuo[] conejos );
   public abstract float[] fitness();
   public void startPobEval( Poblacion pob );
   public void endPobEval( Poblacion pob );
   public void run();
}
```

En las instancias de esta clase, el arreglo `cobayos` contiene referencias a un grupo de individuos a ser evaluados. Tipicamente, este arreglo almacena a un sólo individuo para su evaluación, sin embargo, hay técnicas de calificación que requieren operar a más de un individuo a la vez y, en este caso, el motor puede ser instruido para entregar los individuos al evaluador del usuario en n-adas.

Para evaluar a un individuo o una n-ada de individuos, el motor almacena en `cobayos` al conjunto que será evaluado y después invoca al método `fitness()`, esperando como valor de retorno un arreglo de calificaciones de adecuación ajustada para cada uno de ellos. Las calificaciones devueltas por `fitness()` deben ser siempre números positivos que representen mejores adecuaciones usando valores numéricamente mayores. La subclase diseñada por el usuario debe forzosamente proveer el método `fitness()` en el que se lleva a cabo el proceso de evaluación del problema.

Adicionalmente, los métodos `startPobEval()` y `endPobEval()` son invocados al inicio y fin de la evaluación de la población y pueden ser reescritos por el usuario para ganar acceso a la población antes y después del proceso de evaluación.

Las instancias de la clase del usuario utilizadas para la evaluación de los individuos son invocadas por un mecanismo de control global de la evaluación. Este sistema de control es implementado en la clase `Evaluador`:

```
public class Evaluador extends Object {

// Variables
   protected Class     cls_trainer;

// Constructor
      public Evaluador( String nomb_c )
        throws ClassNotFoundException;

// Metodos
   public void evalua( Poblacion pob, int g_size, int max_trainers )
     throws IllegalAccessException, InstantiationException;
}
```

La clase almacena el nombre de la clase de evaluación del usuario en el campo `cls_trainer`, y recibe este mismo nombre como parámetro de su constructor.

El método `evalua()` es el único método de la clase y controla la evaluación de la población `pob` en grupos de evaluación de `g_size` individuos, la evaluación se lleva a cabo usando `max_trainers` tramas de ejecución paralela. Es aquí donde el motor usa las capacidades *multithreading* de Java y sus mecanismos de control para lograr la evaluación paralela de los individuos.

2.4.7. El control global del proceso

El proceso completo de producción inicial, evaluación y transformación de las poblaciones descrito en el capítulo anterior es implementado por la clase `Motor`. Como presentaré en el siguiente capítulo, una aplicación Java diseñada por el usuario debe instanciar esta clase e invocar a su método `run()` para llevar a cabo una corrida de la máquina de programación genética.

```
public class Motor {

// Variables
   private Poblacion   pob;

// Constructor
   public Motor( int rnd_seed, int nInd, Especie especie, String metodo
)
    throws ClassNotFoundException;

// Metodos
   public void run( int nGen, int g_size,
                    float pcruza, float pmutacion, float preprod,
                    String entrenador, int max_entrenador )
    throws ClassNotFoundException,
           InstantiationException,
           IllegalAccessException;
   private void display( Poblacion p, int generacion );
}
```

El constructor de esta clase crea una población inicial de `nInd` individuos de características determinadas por `especie` mientras que `metodo` expresa el tipo de crecimiento de los individuos que serán utilizados en la generación. Adicionalmente, `rnd_seed` especifica una semilla para el generador de números aleatorios del que se obtienen todos los números pseudoaleatorios del proceso

El método `run()` lleva a cabo la iteración sobre los procesos de evaluación, ordenamiento y transformación de las poblaciones a lo largo de `nGen` generaciones. El proceso de evaluación es llevado a cabo en grupos de `g_size` individuos por `max_entrenador` copias de ejecución paralela de la subclase de `Entrenador` especificada por `entrenador`. Durante el proceso de transformación las operaciones de reproducción, mutación y cruzamiento se llevan a cabo de acuerdo con las probabilidades especificadas por `preprod`, `pmutacion` y `pcruza`.

Finalmente, el método `display()` es usado para reportar la evolución de la ejecución de una corrida. La salida produce, para cada generación, en *stdout*, un listado con el número de individuos en la población, el número de nodos, la profundidad y medida de adecuación ajustada del individuo más apto encontrado, así como el valor promedio de la medida de adecuación ajustada para toda la población. Adicionalmente, envía hacia *stderr* una representación en cadena de caracteres de la estructura del individuo más apto de cada generación.

2.4.8. Clases misceláneas

Para la generación de los números aleatorios usados en el proceso de programación genética, el motor utiliza la clase `Random`, que produce dichos números usando la clase estándar `java.util.Random` del ambiente de ejecución de Java.

```
public class Random {

// El ganerador
   static private java.util.Random     rnd_gen;

// Metodos
   public static void init();
   public static void init( int seed );
   public synchronized static int randomInt( int range );
   public synchronized static float randomFloat();
}
```

La clase `Sort` y la interfase `Ordenable` permiten al motor el ordenamiento de las poblaciones de individuos de acuerdo a su medida de adecuación estandarizada, usando un algoritmo de QuickSort.

```
public class Sort {

   public static void vQSort( Vector v );
   private static void vQS( Vector v, int lo, int hi );
}

public interface Ordenable {

// Metodos
   public int ordan( Ordenable cmp );
}
```

Finalmente, la clase `Entero` es usada para el paso por referencia de un parámetro de tipo `int` a ciertos métodos del motor que así lo requieren.

```
public class Entero extends Object {

// Variables
   public int     valor = 0;

// Constructores
   public Entero();
   public Entero( int i );
}
```

3. Uso de la herramienta

En este último capítulo presentaré, a través de un ejemplo, los pasos a seguir para el uso del motor de programación genética en la solución de un problema teórico.

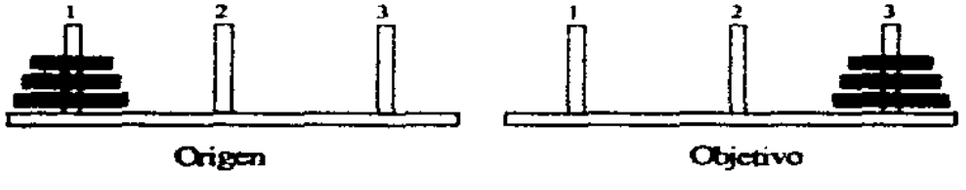
El análisis y la solución de este problema representan más bien un ejemplo de utilización de la herramienta que un exponente típico de la clase de problemas solucionables usando esta metodología.

3.1. El problema de torres de Hanoi

El problema de torres de Hanoi es un juego de ingenio que puede ser descrito de la siguiente manera: el área de juego consta de una tablilla provista de tres postes verticales (1, 2 y 3) y un conjunto de discos de diferentes diámetros (A, B, C, etc.). Los discos cuentan con hoyos en su centro para poder ser apilados en los postes de la tablilla. Inicialmente, todos los discos están apilados sobre el poste 1 en orden decreciente de diámetro de abajo hacia arriba, es decir, el disco A ocupa la cima de la torre y bajo él están los discos B, C, D, etc., en ese orden. El objeto del juego es mover, de uno en uno, todos los discos al poste 3 bajo las siguientes restricciones:

- solo puede moverse el disco superior de un poste y,
- no puede ponerse un disco sobre otro de menor diámetro.

La siguiente figura muestra las configuraciones inicial y objetivo para un problema con tres discos.



A pesar de que este es un problema citado comúnmente en la literatura de inteligencia artificial para ejemplificar técnicas recursivas de solución por reducción de problemas, su solución puede también lograrse a través de búsquedas sobre su espacio solución (Nilsson, 1971, pag.81)

Adicionalmente, existe una solución iterativa al problema que puede ser expresada de la siguiente manera:

- mueve el disco más pequeño un poste hacia la izquierda,
- mueve el segundo disco más pequeño al único poste al que puede ser movido.

Para ejecutar esta secuencia se considera que el poste 3 está a la izquierda del poste 1 y que el poste 1 está a la derecha del poste 3.

Esta solución específica transporta los discos del poste 1 al poste 3 cuando hay un número impar de discos en la tablilla, y del poste 1 al poste 2 cuando el número de discos es par. Si el disco más pequeño se mueve siempre a la derecha, esta relación se invierte, es decir, se pasa del poste 1 al 2 para un número impar de discos y del 1 al 3 para un número par de discos.

Este método de solución al problema de torres de Hanoi es sencillo y requiere de poca inteligencia por parte del agente de solución en comparación con los métodos de reducción de problemas y búsqueda del espacio solución. La principal razón por la cual no es mencionado en la literatura es por ser de escaso interés teórico en inteligencia artificial.[6]

Por otro lado, este método ofrece una vía de solución al problema fácilmente implementable en términos de los elementos del juego, sin necesidad de recurrir a representaciones superiores del problema o de su estado. Es por esto que elegí este enfoque como modelo para desarrollar una representación del juego y un conjunto de

[6]Véase http://www.math.toronto.edu/mathnet/plain/questionCorner/towerpatterns.html.

funcionales que permitan probar el funcionamiento del motor de programación genética descrito en el capítulo anterior.

3.2. La representación del juego

Para buscar una solución al problema de torres de Hanoi a través de programación genética diseñé un modelo del juego en clases de Java sobre el cual pudieran actuar un conjunto de funcionales que describiré más adelante.

3.2.1. La representación del juego: las piezas del tablero

El elemento base en el problema es la pieza de tablero. Para describirla use la clase `hnPieza` que se esboza a continuación:

```
public class hnPieza extends Object {

// El identificador de la pieza
   char    id = ' ';

// Constructor
   public hnPieza( char identificador );

// Metodos
   public boolean equals( hnPieza cmp );
   public int compareTo( hnPieza cmp );
   public String toString();
}
```

Las instancias de esta clase permiten representar piezas (discos) de distintos tamaños comparables gracias al identificador `id` y a los métodos `equals()` y `compareTo()`. El método `toString()` es una adición que permite obtener una representación en cadena de caracteres del identificador de la pieza.

3.2.2. La representación del juego: los postes

Para continuar con la representación del tablero, la clase `hnPoste` implementa un poste del tablero de la siguiente forma:

```
public class hnPoste extends Object implements Cloneable {

// Una pila de piezas
   Stack    piezas;

// Constructor
   public hnPoste();

// Metodos
   public Object clone();
   public void ponPieza( hnPieza pieza );
   public hnPieza quitaPieza();
   public int nPiezas();
   public boolean vacio();
   public hnPieza piezaSup();
   public boolean equals( hnPoste cmp );
   public String toString();
}
```

En esta clase, un poste está implementado a través de una pila (stack) llamada `piezas` a la que pueden ser agregadas o removidas piezas usando los métodos `ponPieza()` y `quitaPieza()`. Los métodos `nPiezas()`, `vacio()` y `piezaSup()` son informativos y permiten, respectivamente, contar el número de piezas en el poste, saber si el poste esta vacio y conocer la pieza superior de un poste sin removerla de éste

Adicionalmente, el método `equals()` compara dos postes y determina si contienen piezas idénticas apiladas en el mismo orden.

El método `clone()` produce una copia del poste y las piezas que contiene para obtener réplicas de operación de una configuración dada.

3.2.3. La representación del juego: el tablero

El tablero completo es una clase que agrupa tres postes y es capaz de realizar movimientos de piezas entre ellos.

```
public class hnTablero extends Object implements Cloneable {

// Arreglo de postes en el tablero
   hnPoste[]     postes = null;

// Numero de movimientos efectuados
   int           mov = 0;

// El ultimo poste afectado en una movida
   hnPoste       ultimo = null;


// Constructor
   public hnTablero( int n_postes );

// Metodos
   public Object clone();
   public void setPoste( int n, hnPoste poste );
   public void setUltimo( hnPoste ult );
   public int nPiezas();
   public int idxPoste( hnPoste poste );
   public boolean equals( hnTablero cmp );
   public hnPoste muevePieza( hnPoste orig, hnPoste dest );
   public String toString();
}
```

La clase contiene el campo `postes` que es un arreglo de instancias de la clase `hnPoste`. Adicionalmente, `mov` y `ultimo` le permiten llevar una cuenta del número de piezas movidas en el tablero y una referencia al último poste destino de una operación de movimiento. Los métodos `setPoste()` y `setUltimo()` son usados para afectar el valor de los campos del mismo nombre.

El método `nPiezas()` informa el número total de piezas en todos los postes del tablero mientras que `idxPoste()` devuelve el indice de un poste en el arreglo de postes del tablero. El método `equals()`, por su parte, compara dos tableros para determinar si tienen la misma configuración de piezas.

Finalmente, `muevePieza()` lleva una pieza de un poste origen a un poste destino, contabiliza el movimiento en el campo `mov` y actualiza el valor del campo `ultimo`.

3.2.4. La representación del juego: el ambiente de evaluación

Para contar con un ambiente de evaluación de los funcionales que describiré más adelante, la clase `hnAmbiente` agrupa las características significativas de un problema específico:

```
public class hnAmbiente extends Object implements Cloneable {

// El tablero origen
   hnTablero    origen;

// El poste objetivo
   hnPoste      objetivo;

// El minimo numero de movidas para ir de origen a objetivo
   int          mov_min = 0;


// Constructores
   public hnAmbiente();
   public hnAmbiente( hnTablero org, hnPoste obj, int movs );

// Metodos
   public Object clone();
   public boolean ya();
   public String toString();
}
```

La clase contiene un tablero origen que es el punto de partida del problema y un poste objetivo sobre el cual deben apilarse las piezas para lograr una solución. Adicionalmente, se almacena el número mínimo de movimientos requeridos para solucionar este problema.

El método `ya()` determina si el tablero origen ha alcanzado el estado solución.

3.3. Funcionales para la solución del problema

Para buscar una solución al problemas de torres de Hanoi, diseñe un conjunto de funcionales que actúan sobre el tablero origen del ambiente del problema. El dominio de operación de los funcionales es de tipo `hnPoste`, es decir, el resultado de la evaluación de un operando siempre representa una referencia a alguno de los tres postes del tablero. Esto es un ejemplo de lo mencionado en el primer capítulo: aquí

el tipo de valores que opera y devuelve cada funcional no es un tipo simple. Adicionalmente, la evaluación de un operando puede tomar el valor especial `null` el cual es una referencia inválida.

La cerradura de los funcionales está garantizada en su comportamiento ya que todos tienen la capacidad de operar sobre postes del tablero o manejar el valor nulo como aceptable para cualquiera de sus operandos.

3.3.1. Funcionales para la solución del problema: los terminales

En este ejercicio, sólo se incluyen dos funcionales terminales para la expresión de la solución: `Minimo` y `Ultimo`. El primero devuelve una referencia al poste del tablero que contiene al disco de menor diámetro en el tablero y el segundo devuelve una referencia al último poste afectado en la última operación de movimiento efectuada en el tablero.

Para expresar los distintos tipos de funcionales del problema, y de acuerdo con lo mencionado en el segundo capítulo, diseñé la clase `hnNodo` descendiente de `Funcional` del motor:

```
public abstract class hnNodo extends GP.Funcional {
   public abstract hnPoste eval(GP.Individuo sujeto, Object ambiente);
   public abstract String toString();
}
```

Esta clase cumple con el requerimiento de extender la clase `Funcional` del motor y establece que toda subclase debe implementar un método `eval()` con tipo de retorno `hnPoste`.

La estructura de las clases que implementan los funcionales `Minimo` y `Ultimo`, derivadas de la clase `hnNodo`, se presentan a continuación:

```
public class hnMinimo extends hnNodo {

// Constructor
   public hnMinimo();

// Metodos
   public Object clone();
   public final hnPoste eval( Individuo sujeto, Object ambiente );
   public final String toString();
}

public class hnUltimo extends hnNodo {

// Constructor
   public hnUltimo();

// Metodos
   public Object clone();
   public final hnPoste eval( Individuo sujeto, Object ambiente );
   public final String toString();
}
```

3.3.2. Funcionales para la solución del problema: los no terminales

Además de los dos terminales, diseñe cinco funcionales no terminales: `PIzq`, `PDer`, `Mayor`, `Menor` y `Mueve`; implementados por las clases `hnPIzq`, `hnPDer`, `hnMenor`, `hnMayor` y `hnMueve`, respectivamente. Estas clases también son subclases de `hnNodo` y comparten los mismos métodos que los terminales ya presentados. Por esta razón, omitiré el formato estructural de cada clase para sólo discutir el comportamiento del método `eval()` de cada una de ellas.

`PIzq` Este funcional devuelve una referencia al poste inmediatamente a la izquierda del obtenido por la evaluación de su único operando. Por otro lado, si dicha evaluación devuelve el valor nulo (`null`), el funcional también devuelve nulo.

`PDer` Casi idéntico en funcionamiento a `PIzq`, este funcional devuelve el poste a la derecha de su operando.

`Menor` Evalúa sus dos operandos y devuelve, de los dos, aquel que contiene la pieza de menor diámetro. El funcional devuelve nulo en el caso de que ambos operandos evalúen a nulo o que ninguno de ellos contenga piezas.

`Mayor` En forma similar a `Menor`, este funcional devuelve el operando que contiene la pieza de mayor diámetro.

`Mueve` Mueve la pieza superior de su primer operando a su segundo operando, siempre que el movimiento sea válido de acuerdo a las reglas del juego. Adicionalmente, este funcional incrementa el número de movimientos del tablero (`mov`) y fija el valor del campo `ultimo` al poste destino del movimiento. Si uno de sus operandos evalúa a nulo, el poste origen esta vacio o el movimiento es inválido, no se realiza ninguna operación y el funcional devuelve nulo.

Asi, {`Minimo`, `Ultimo`, `PIzq`, `PDer`, `Mayor`, `Menor`, `Mueve`} es el conjunto completo de funcionales para el problema, los cuales reciben cero, cero, uno, uno, dos, dos y dos operandos, respectivamente.

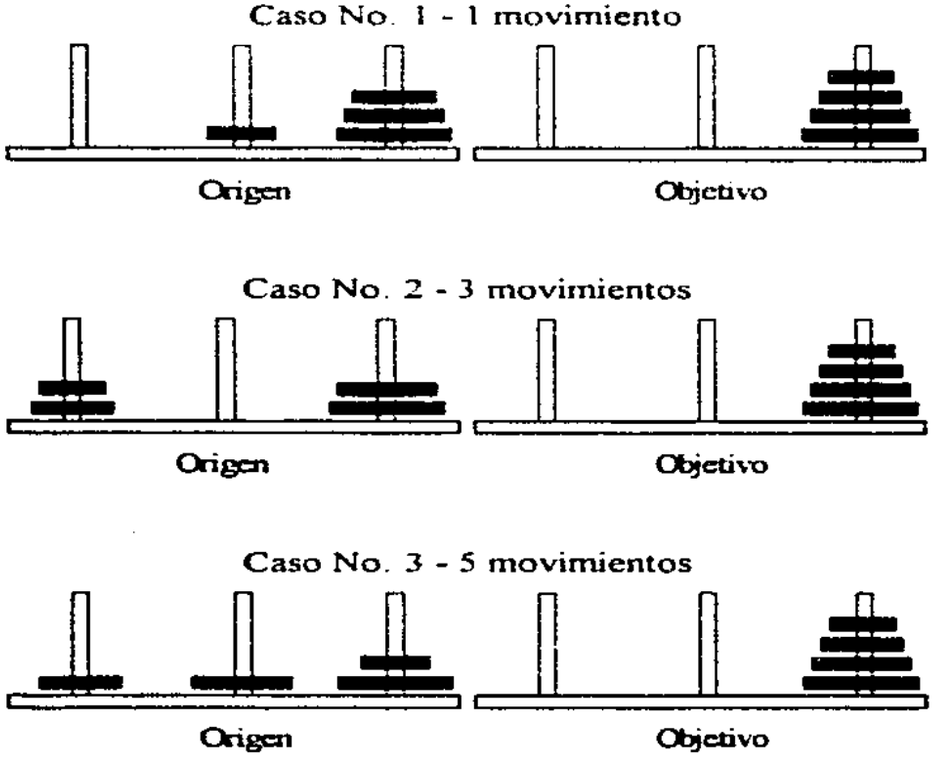
Nunca llevé a cabo un análisis de a garantía de suficiencia de este conjunto de funcionales, sin embargo, conociendo la descripción de la solución iterativa al problema de torres de Hanoi se antoja pensar que el conjunto utilizado tiene la capacidad suficiente para expresar soluciones a este tipo de problemas.

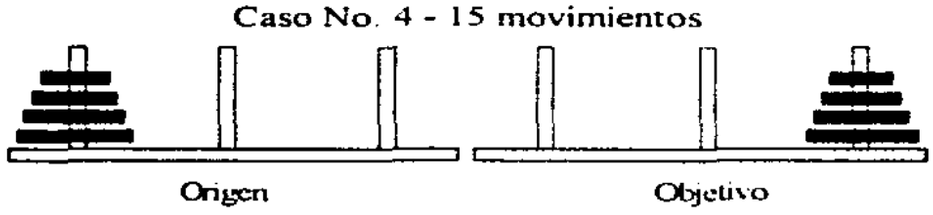
3.4. La metodología de evaluación

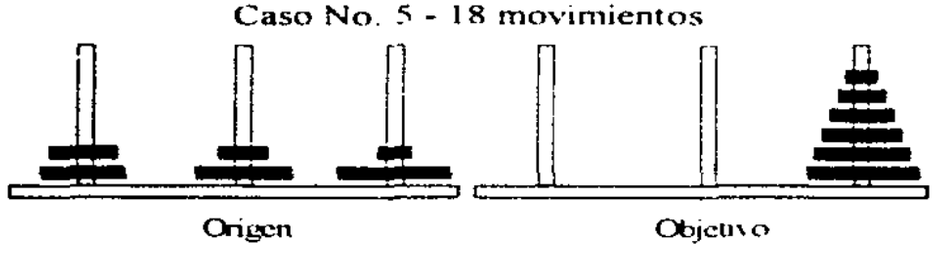
El mecanismo de evaluación de los individuos bajo evolución es implementado por la clase `hnTrainer` que, tal como se menciona en el capitulo anterior, cumple con la restricción de ser una subclase de la clase `Entrenador` del motor:

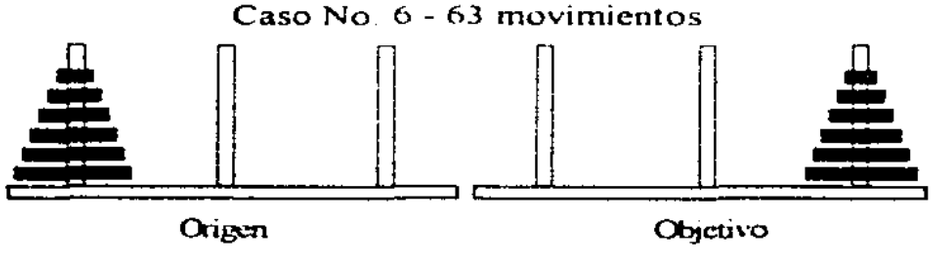
```
public class hnTrainer extends GP.Entrenador {

// Variables
   private static int                casos;
   private static hnAmbiente[]       ambientes;

// Metodos
   public float[] fitness();
   private double fd( Individuo ind );
   public static void init();
}
```

La clase ofrece una implementación para el método `fitness()`, que es la interfase con el motor para la evaluación de los individuos, mientras que la funcionalidad real de la evaluación está contenida en el método `fd()`. El método `init()` debe ser invocado al inicio de una corrida para inicializar un conjunto de casos de evaluación en el arreglo `ambientes`. En este problema, elegí seis escenarios que son parte del desarrollo de una solución para los problemas de 4 y 6 discos. Estos se presentan a continuación:









Utilizando esta clase, cada individuo es evaluado en forma individual sobre un grupo de casos del problema de torres de Hanoi. Cada uno de estos casos está representado por un ambiente que establece una configuración de tablero y un poste objetivo sobre el cual deben apilarse todos los discos.

Cada individuo específico es expuesto a todos los casos propuestos en el arreglo `ambientes`, y cada caso es evaluado invocando en forma repetida al método `eval()` del funcional raiz de la rama principal del individuo. Esta invocación se repite un número máximo de veces (2^{n-1}) determinado por el número de discos (n) en el tablero origen, o hasta que las piezas del tablero estén colocadas en el poste objetivo; lo que suceda primero.

Al finalizar el ciclo de invocaciones repetidas, el entrenador determina si el caso en turno ha sido solucionado por el individuo. De ser así, el individuo recibe una calificación acumulativa de hasta un punto que se calcula de la siguiente forma:

- El individuo recibe una calificación base de 0.75 puntos por haber solucionado el caso en turno.
- A la calificación base se agrega un factor igual a 0.25 puntos multiplicado por la relación que guardan el número mínimo de movimientos necesarios para resolver el caso contra el número de movimientos utilizados por el individuo para la solución.

$$\text{calificación} = 0.75 + 0.25\ mov_{minimos} / mov_{utilizadas}$$

En esta forma, una solución lograda en un número mínimo de movimientos merece una calificación de un punto. Esta calificación decrece conforme el número de movimientos utilizados se vuelve mayor. La estrategia de puntajes busca promover a individuos que no sólo logran soluciones al problema sino que las logran en un número mínimo de movimientos.

3.5. Ejecución del motor

Para llevar a cabo una corrida del motor, utilicé una clase encargada de fijar los parámetros de operación del motor e invocarlo sobre el problema. Esta es la clase `StartHanoi`:

```
public class StartHanoi {

// La especificacion de la especie para la poblacion
   static Especie     especie;

// Las clases de funcionales terminales del arbol de programa
   static String[]    t_rpb  = { "hnUltimo", "hnMinimo" };

// Las clases de funcionales no terminales del arbol de programa
   static String[]    nt_rpb = { "hnMueve",   "hnMenor",   "hnMayor",
                                 "hnPIzq",    "hnPDer" };
```

```
// Parametros de la poblacion
   public static int             deep = 5;
   public static int         max_deep = 10;
   public static boolean      balance = true;
   public static int             nInd = 500;
   public static String         metodo = "HRAMP";

// Parametros para evaluacion
   public static int             nGen = 100;
   public static int           g_size = 1;
   public static int        max_train = 16;
   public static float         pcruza = (float)0.8;
   public static float      pmutacion = (float)0.2;
   public static float      preproduc = (float)0.0;


// El procedimiento principal
   public static void main( String[] argv )
    throws ClassNotFoundException,
           InstantiationException,
           IllegalAccessException
   {

   // Inicia los casos del entrenador del problema
      hnTrainer.init();

   // Crea la especificacion de la especie
      especie = new Especie( "Hanoi",
                             balance, deep, max_deep,
                             t_rpb, nt_rpb );

   // Crea una instancia del motor
      seed = (int)(Math.random() * 2147483647);
      System.out.println("Run seed: " + seed);
      Motor m = new Motor( seed, nInd, especie, metodo );

   // Arranca el proceso de GP en el motor
      m.run( nGen, g_size,
             pcruza, pmutacion, preproduc,
             "hnTrainer", max_train );
   }
}
```

Esta clase expresa una aplicación de Java que almacena los parámetros de ejecución para el motor de programación genética. En su método principal lleva a cabo las siguientes tareas:

- Crea una definición de `Especie` para crear los individuos de la población con los parámetros de balance, profundidad de creación, profundidad máxima, conjunto de funcionales terminales y no terminales, especificados por las variables `balance`, `deep`, `max_deep`, `t_rpb` y `nt_rbp`, respectivamente.

- Obtiene una semilla arbitraria para el generador de números aleatorios del motor y la almacena en la variable `seed`
- Crea una instancia de la clase `Motor` proporcionando la información de semilla aleatoria, número de individuos en la población, definición de especie y método de crecimiento de la población de acuerdo a los parámetros `seed`, `nInd`, `especie` y `metodo`, respectivamente.
- Finalmente, inicia la ejecución del motor de programación genética al invocar al método `run()` del motor creado. La invocación indica el número de generaciones a evaluar, el tamaño de grupo a usar en la evaluación de los individuos, la distribución de probabilidades a utilizar para las operaciones de reproduccion, mutación y cruzamiento, el nombre de la clase entrenadora especifica para el problema y, el número de entrenadores a ser ejecutados en paralelo por el motor.

Este proceso se encarga de la inicialización necesaria para una corrida del motor e invoca su ejecución. La creación y ejecución del motor desencadenan todas las actividades necesarias para la implementación del paradigma de la programación genética.

La ejecución del motor termina al alcanzar la evaluación del número de generaciones especificadas o, al ser detenido por algún criterio de terminación en el entrenador del problema (`hnTrainer`).

3.6. Parámetros de control

Las corridas efectuadas para la solución del problema de torres de Hanoi se llevaron a cabo usando poblaciones de 500 individuos con una profundidad máxima de construcción de 5 niveles y una profundidad máxima de operación de 10 niveles. La generación de la población inicial se hizo por medio del método de rampa combinada con el fin de maximizar la variedad de configuraciones estructurales presentes en la

población. El proceso de transformación-evaluación se repitió a lo largo de 100 generaciones sin un criterio adicional de terminación

Los procesos de evaluación se llevaron a cabo en individuos aislados ejecutando simultáneamente 16 instancias de la clase `hnTrainer` con el fin de aprovechar la capacidad de cómputo de un equipo provisto de 4 procesadores.

La transformación de las poblaciones contempló una probabilidad de cruzamiento del 80% y una probabilidad de mutación del 20% mientras que la operación de reproducción no fue utilizada por considerar que reduce la "diversidad genética" en las poblaciones de individuos y provoca una tendencia hacia convergencias tempranas con soluciones subóptimas del problema.

3.7. Ambiente de cómputo utilizado

Para la ejecución del código de Java se utilizaron inicialmente -en la etapa de pruebas- dos intérpretes diferentes. Por una parte se uso el Java Development Kit de SunSoft en versión 1.1.2 sin capacidades reales de multiprocesamiento; y en segunda instancia se utilizó el interprete de Java "Kaffe" de dominio público en versión 0.97, provisto de un compilador JIT (Just In Time) incapaz de ejecutar tramas en forma distribuida sobre varios procesadores.

Ambas herramientas de ejecución fueron usadas en forma preliminar sobre plataformas de cómputo Intel 486 y Pentium ejecutando SunSoft Solaris versión 2.5.1 y Linux en la distribución de RedHat versión 4.2. Las configuraciones de hardware variaron desde equipos 486 a 33MHz con 16MB RAM hasta Pentium a 166MHz con 64MB RAM. Otros equipos utilizados con estas herramientas fueron monoprocesadores Sun-Microsystems modelos Ultra I y Ultra II, de procesadores UltraSPARC a 167MHz, provistos de 128MB RAM o más.

Las corridas finales fueron efectuadas sobre un equipo Sun-Microsystems modelo Enterprise 4000 provisto de 4 procesadores UltraSPARC a 167MHz, 512MB de memoria principal y 768MB de memoria virtual. El sistema operativo utilizado fue Sun-Solaris versión 2.5.1.

Sobre estas plataformas multiprocesadoras se uso el Java Development Kit de SunSoft en versión 1.1.3 provisto de un compilador JIT (*Just In Time*) con capacidad de utilización de tramas de ejecución en forma nativa (*native threads*) sobre el sistema operativo Solaris, el cual es capaz de distribuir las tramas de ejecución sobre los procesadores disponibles en el equipo.

Bajo los parámetros de control presentados se fijó, para todas las ejecuciones, un tope de utilización de memoria de 150MB. Este tope es muy superior a los requerimientos mínimos de espacio en estado estable de las corridas, que están tasados entre 40 y 70MB. Sin embargo, la holgura adicional ahorra en tiempo de ejecución por requerir una menor actividad por parte del recolector de basura de Java.

Los resultados de ejecución generados por el motor en cuanto a la evolución de las poblaciones y la estructura del mejor individuo de cada generación fueron registrados en archivos de salida separados, consumiendo en conjunto hasta un total de 50KB por ejecución.

3.8. Resultados

Para la solución del problema propuesto se realizaron 50 corridas independientes, de las cuales 18 (un 36%) lograron encontrar uno o más individuos capaces de solucionar en forma óptima los 6 casos propuestos.

La convergencia a la solución fue relativamente temprana. De las corridas exitosas, la primera solución óptima fue encontrada en promedio en la generación número 54.

Las generaciones posteriores siempre fueron evaluadas en busca de individuos óptimos formados por un número menor de funcionales, es decir, individuos capaces de solucionar el problema haciendo uso de una menor complejidad estructural.

Probablemente, la respuesta a este comportamiento reside en el hecho de que la complejidad estructural nunca estuvo involucrada en la métrica de aptitud de los individuos. La única presión evolutiva en esta dirección es que el ordenamiento de dos individuos con la misma medida de aptitud favorece siempre al individuo de menor complejidad estructural, y todo parece indicar que este criterio no es suficiente para producir una presión eficaz hacia la producción de individuos de menor peso.

El apéndice de resultados al final de este trabajo muestra un conjunto de gráficas extraídas de corridas selectas de las cincuenta corridas realizadas. Las gráficas muestran la evolución de aptitud y peso en los individuos de estas corridas.

4. Conclusiones

A través de estas conclusiones pretendo presentar no solamente una evaluación de la aportación del proyecto realizado sino también mencionar aspectos interesantes relativos a la experiencia ganada a lo largo de su desarrollo.

El diseño de la máquina de programación genética logrado se apega a la descripción de la técnica hecha por J. R. Koza (1992 y 1994). Considero que la interfase concebida para el enlace con el diseño de los funcionales y el método de evaluación del usuario resulta sencilla y flexible a la vez. De hecho, las únicas restricciones impuestas son los requerimientos de derivar a todo funcional de la clase `Funcional` y al método de evaluación de la clase `Entrenador` provistas por el motor mismo.

Este sencillo método de comunicación entre código del usuario y el motor tiene un bajo costo para el usuario a la vez que le ofrece completa flexibilidad para elegir el dominio y ambiente de operación para la implementación de la solución. Ejemplo de esto es el problema de torres de Hanoi abordado, en el que el dominio de operación de los funcionales del problema es un tipo complejo que representa a un poste del tablero.

A la vez, la segmentación del motor en clases que abstraen cada uno de los elementos conceptuales del paradigma (funcional, rama, individuo, población, evaluador, entrenador y motor) agrupa cada una de las actividades de la técnica en módulos autocontenidos. En un futuro, este diseño podría resultar útil para simplificar la tarea de comprensión y manipulación del código que conforma a la máquina.

Creo que el diseño logrado es sencillo y refleja correctamente la organización que se ha propuesto para cada uno de los elementos y comportamientos que conforman la técnica de programación genética.

En cuanto a la confiabilidad de la versión final de la implementación, el funcionamiento parece estable y libre de errores, sin embargo, no puede descartarse la posibilidad de que existan errores residuales aun no detectados.

La experiencia de trabajo en Java para la implementación del sistema me deja una opinión positiva de las capacidades del lenguaje para abordar proyectos de desarrollo de gran escala. El sencillo manejo y portabilidad de las funciones para multiprocesamiento de Java resultaron invaluables para la eficaz implementación del sistema de evaluación paralela de la aptitud de individuos de una población.

Por otro lado, las características de administración dinámica de memoria de Java, que utiliza un recolector de basura, resultan cómodas durante el desarrollo aunque tienen efectos negativos en la ejecución del sistema. La frecuencia de creación y obsolescencia de los cientos de miles de objetos creados durante una corrida del sistema requieren de una gran actividad por parte del recolector. En este contexto, existen técnicas de administración de memoria que, si bien son menos amables para el desarrollador, ofrecen una mayor eficiencia en la asignación y liberación de recursos.

El cuestionamiento sobre la eficiencia de ejecución del código interpretado de Java (con o sin los beneficios de un compilador JIT) en comparación con la eficiencia del código binario, no fue abordada como parte del proyecto. Su análisis profundo requeriría de una versión del sistema de diseño comparable implementada en un lenguaje completamente compilable.

Salvo por el impacto negativo de la administración de memoria y la ejecución interpretada del lenguaje, creo que el balance global de la elección de Java como herramienta de desarrollo resultó positiva.

Al margen del sistema en el que se centra este trabajo y los resultados obtenidos, la experiencia de trabajar con un lenguaje como Java en la exploración de técnicas de

multiprocesamiento aplicadas a una metodologia de inteligencia artificial es un beneficio personal obtenido a lo largo del proyecto

La experiencia de trabajo con varias herramientas para el uso de Java sobre diversas plataformas es un beneficio adicional que, creo, probará ser de gran valor profesional en los próximos años.

Bibliografía

Booch, Grady
1991 *Object Oriented Design.* Redwood City, California:The Benjamin/Cummings Publishing Company.

Flanagan, David
1996 *Java in a Nutshell.*Sebastopol, California: O'Reilly & Associates, Inc.

Groner, Daniel, K.C. Hopson, Harish Prabandham y Todd Sundsted
1996 *Java Language API Superbible*. Corte Madera, California: Waite Group Press.

Koza, John R.
1994 *Genetic Programming II. Automatic Discovery of Reusable Programs.*Cambridge y Londres: The MIT Press.
1992 *Genetic Programming*. Cambridge y Londres: The MIT Press.

Koza, John R. y J.P. Rice
1992 *A Non-Linear Genetic Process for Data Encoding and for Solving Problems Using Automatically Defined Functions*. U. S. Patent Application, citado en Koza, John R,1994, *Genetic Programming II. Automatic Discovery of Reusable Programs.*Cambridge y Londres: The MIT Press.

Vanderburg, Glenn
1996 *Tricks of the Java Programming Gurus*. Indianapolis:Sams.net Publishing.

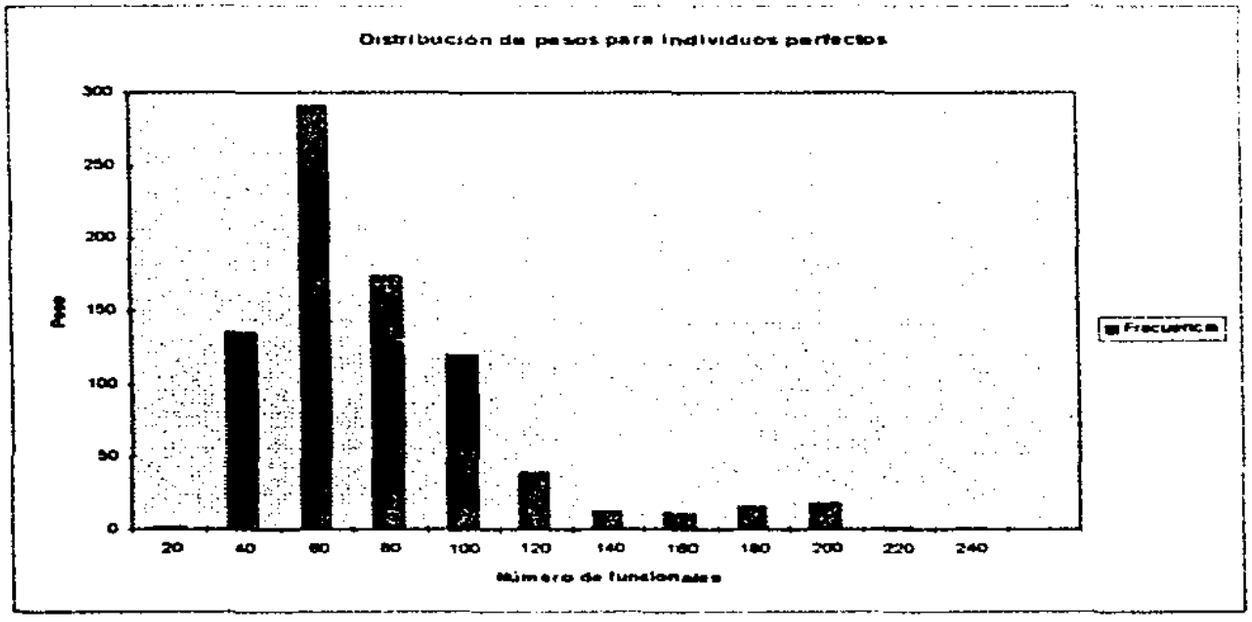
Apéndice de resultados

A continuación presento un conjunto de gráficas obtenidas a partir de diez de las cincuenta corridas efectuadas para el problema de torres de Hanoi.

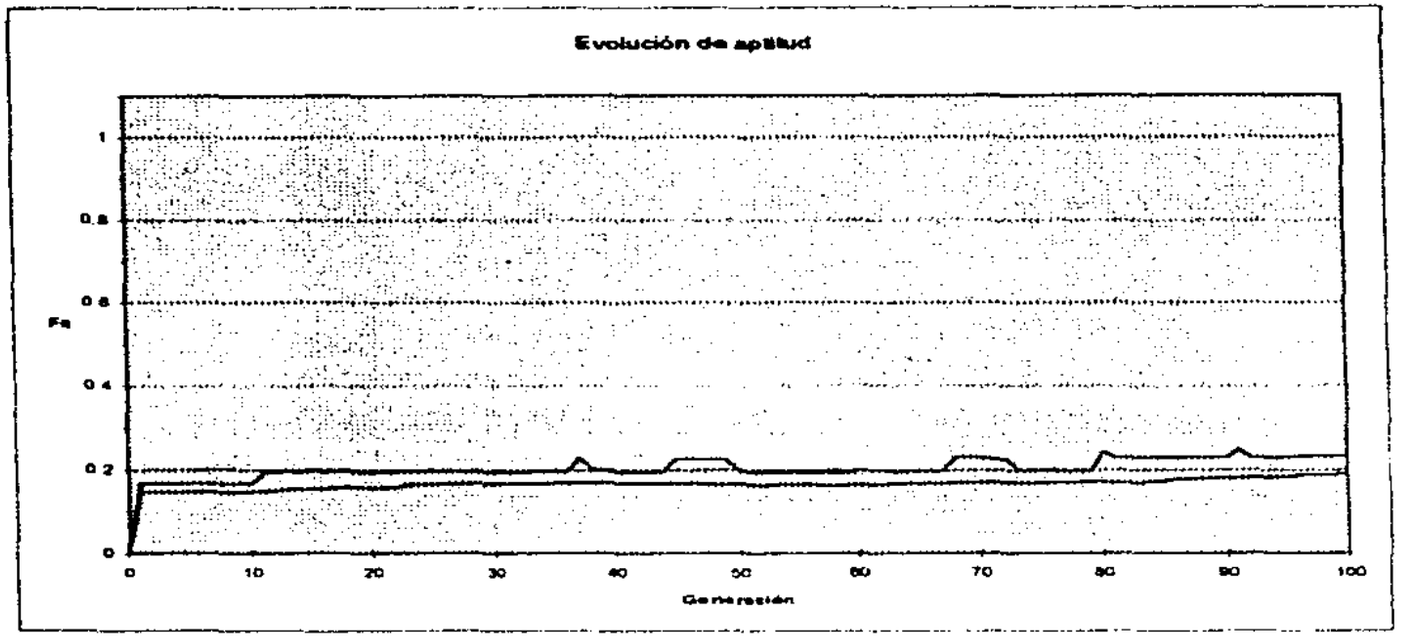
Uno de los tipos de gráficas muestra la evolución de la adecuación ajustada para el mejor individuo de cada generacion en una corrida, así como el valor medio de este parámetro para todos los individuos de la poblacion

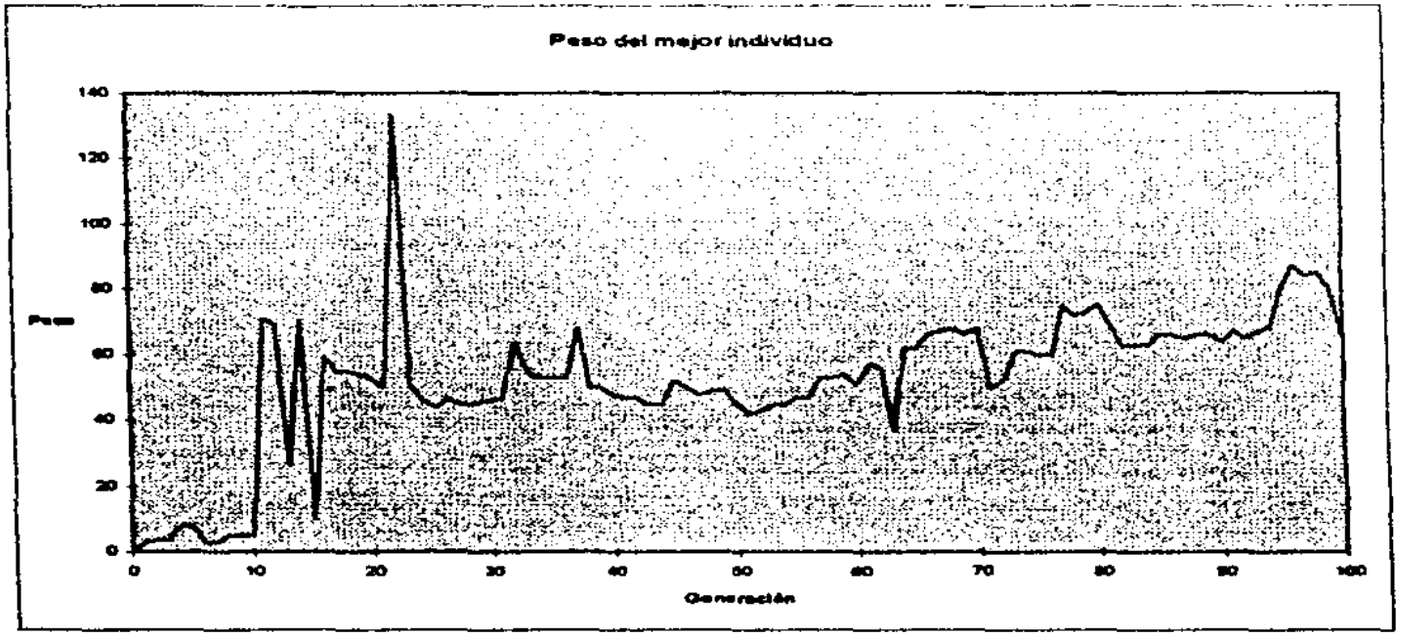
El segundo tipo de gráfica muestra, para una corrida, el peso, en numero de nodos, del árbol de composición del mejor individuo de cada generación

Adicionalmente, presento un grafico que muestra un histograma de distribución de pesos para los individuos perfectos encontrados en todas las corridas.

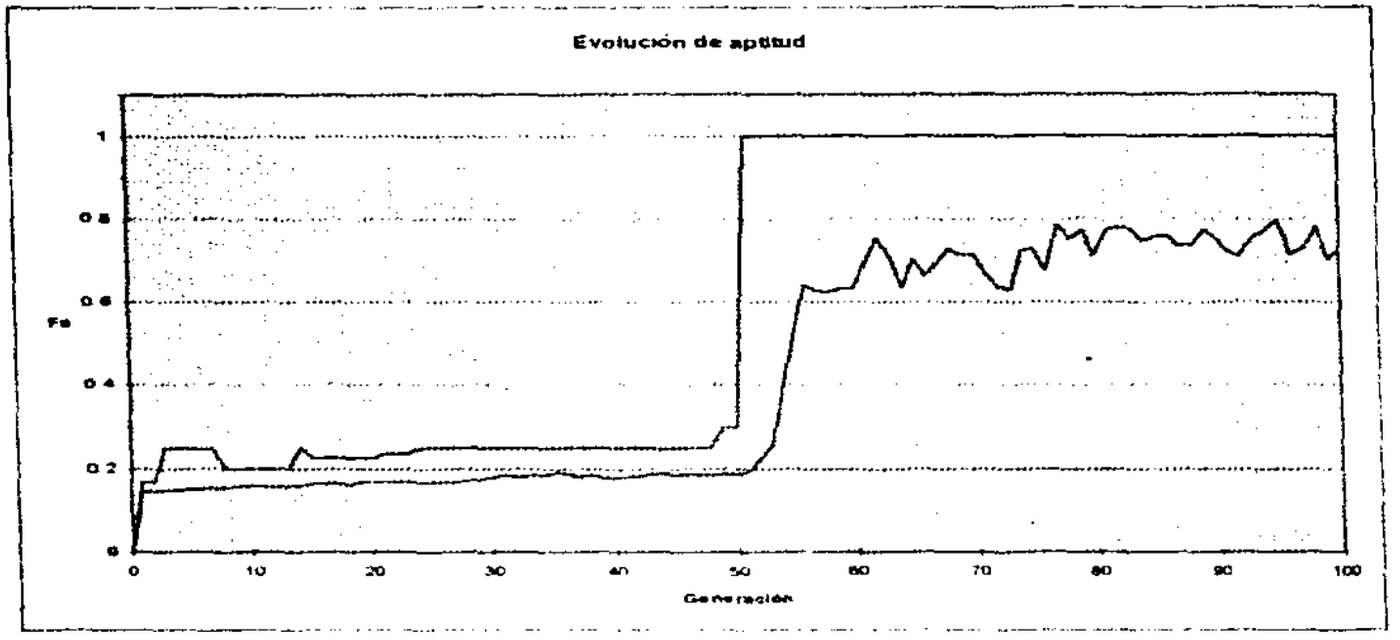


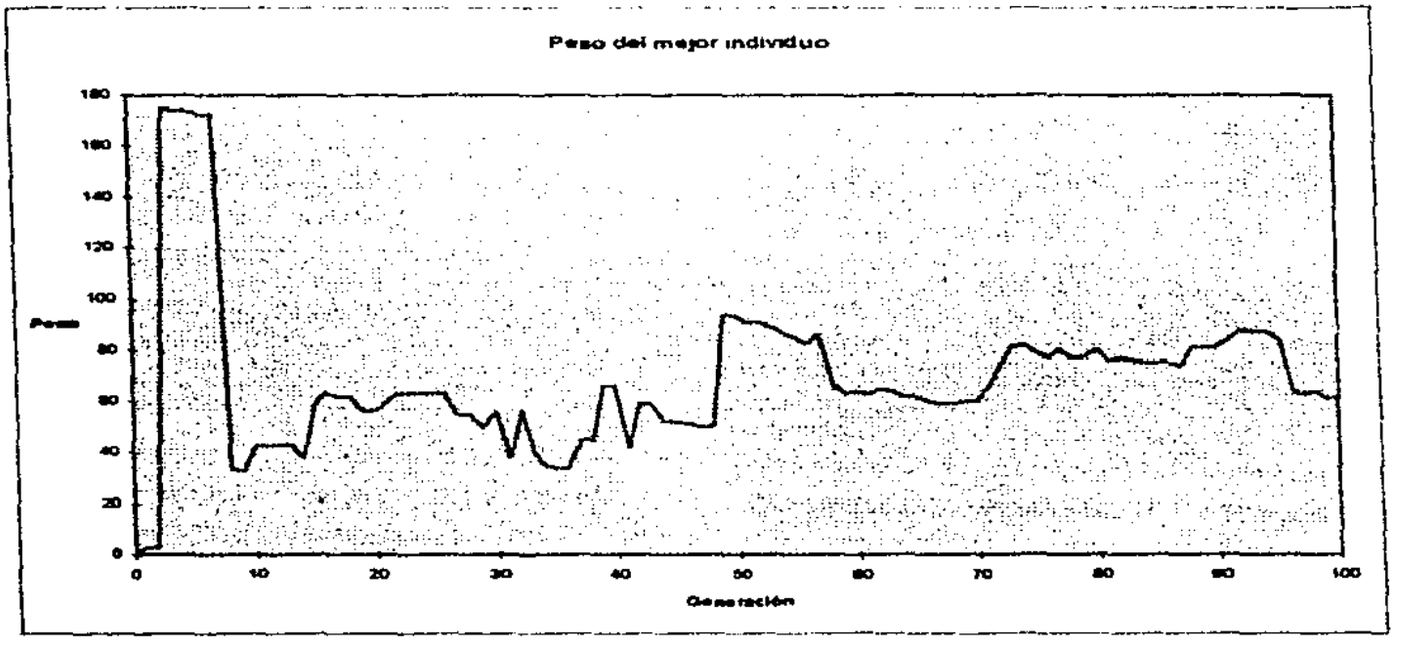
Corrida No. 1: aptitud y peso



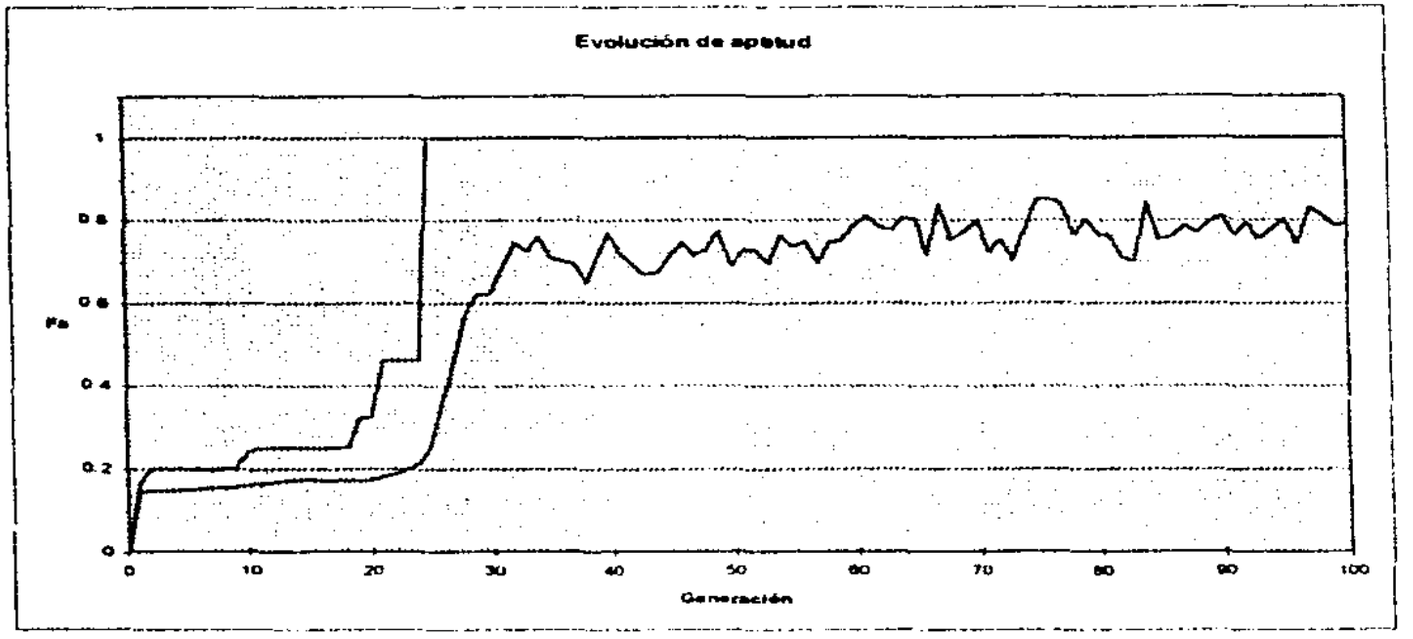


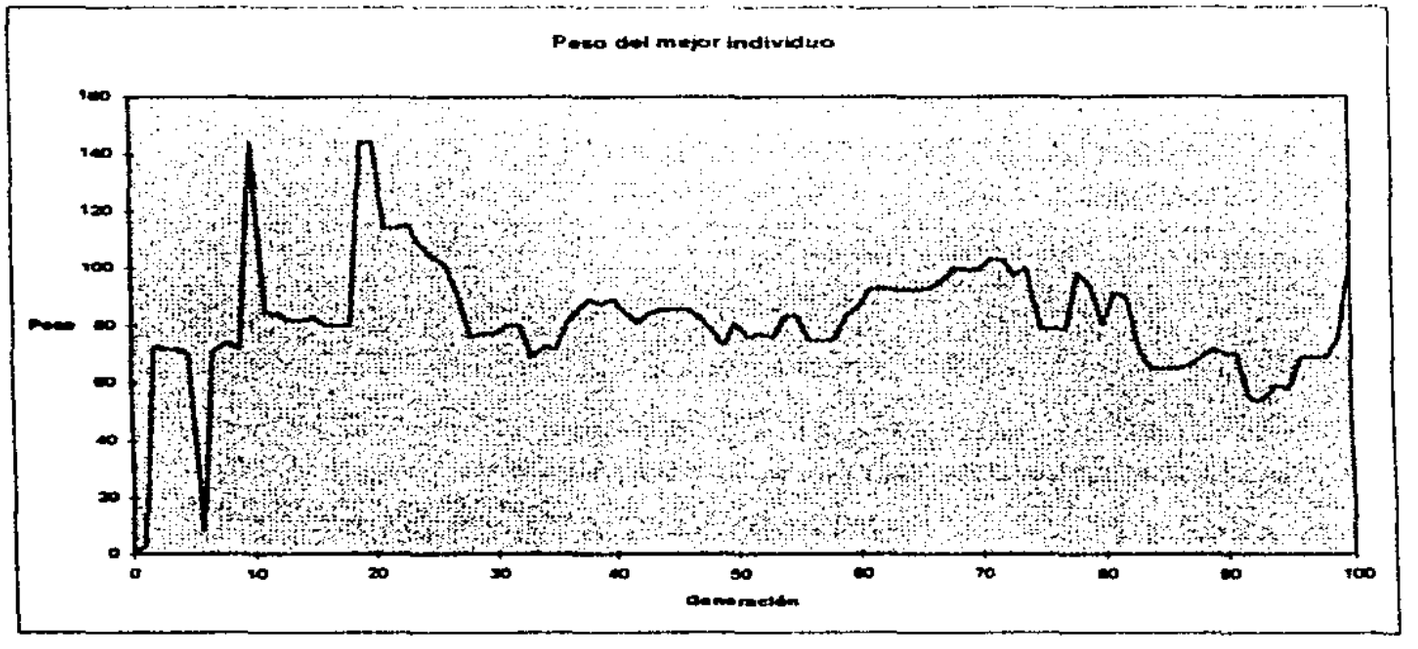
Corrida No. 7: aptitud y peso



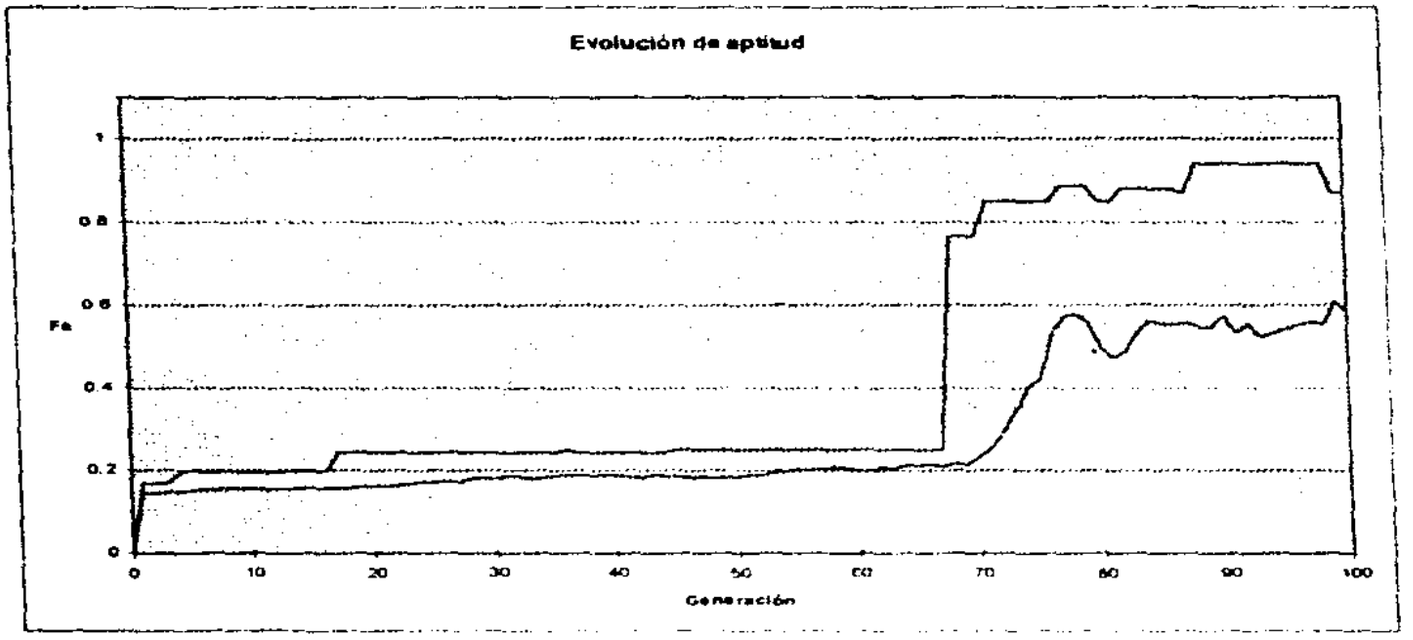


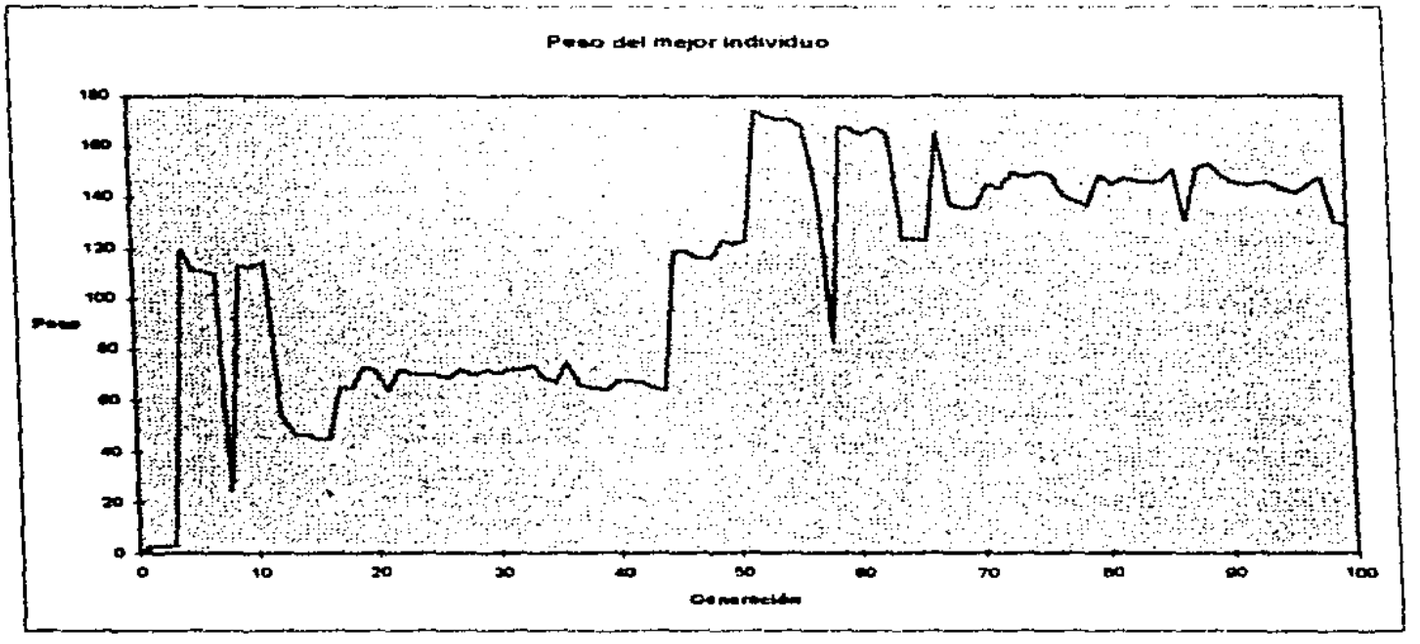
Corrida No. 12: aptitud y peso



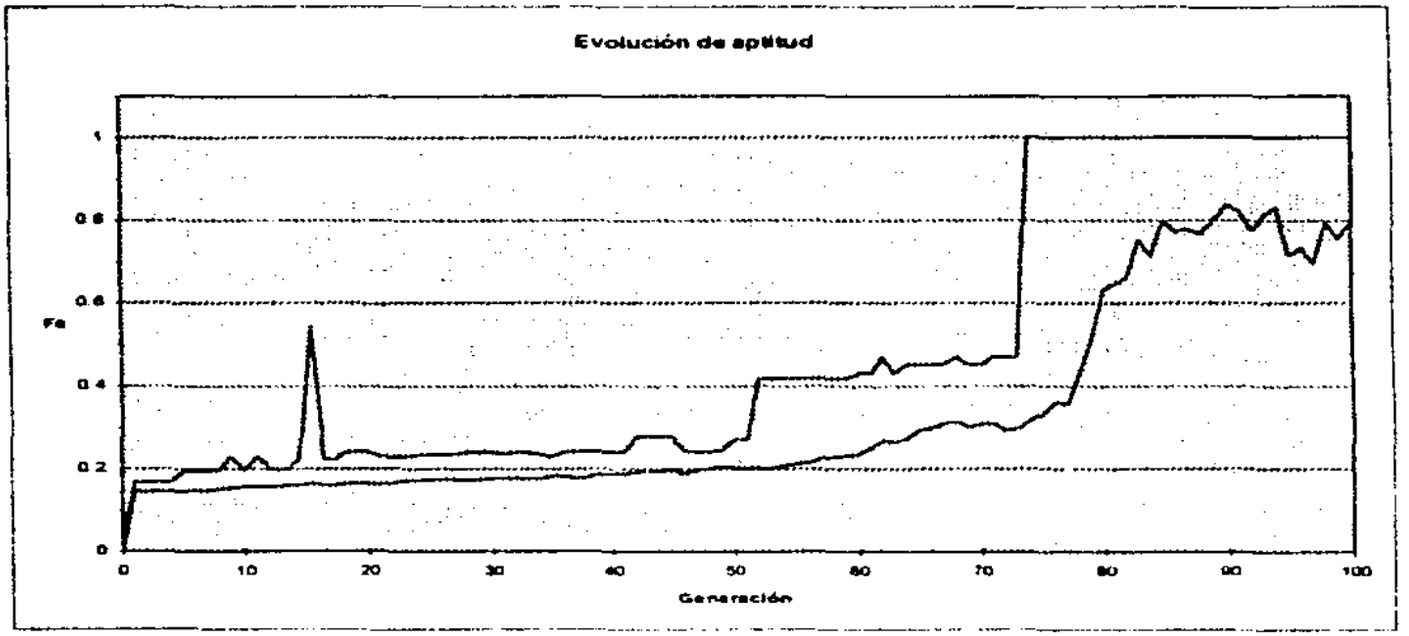


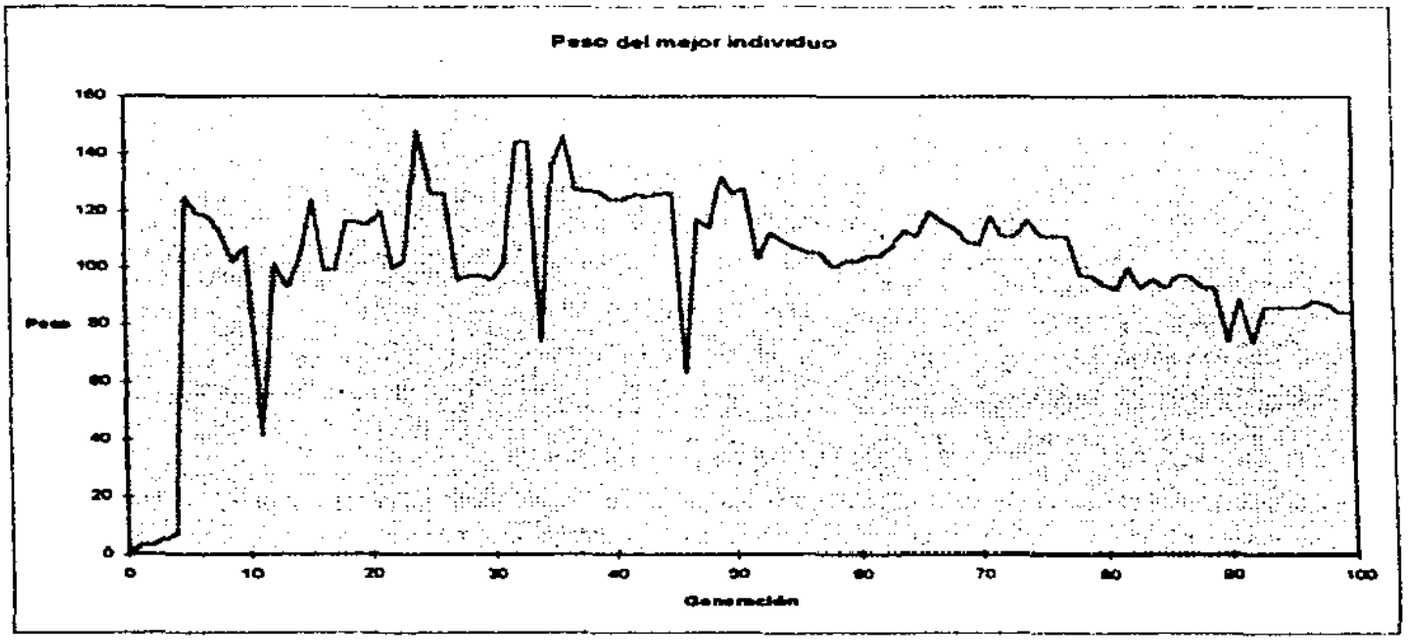
Corrida No. 23: aptitud y peso



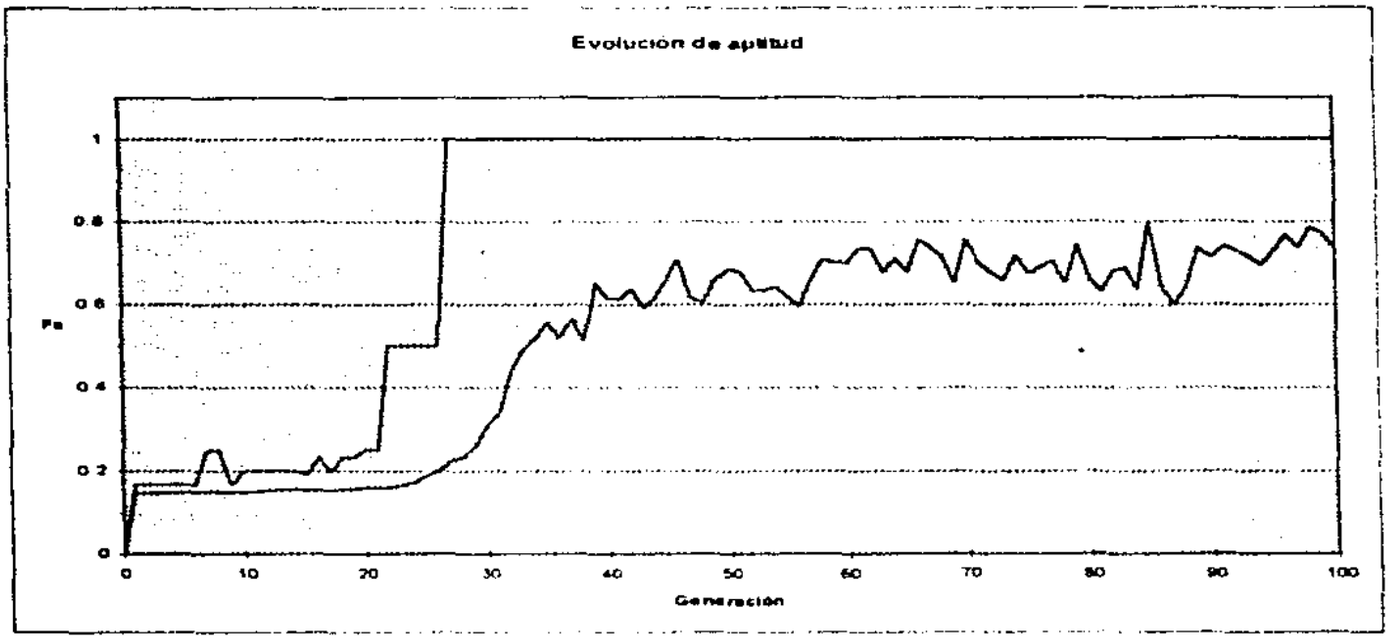


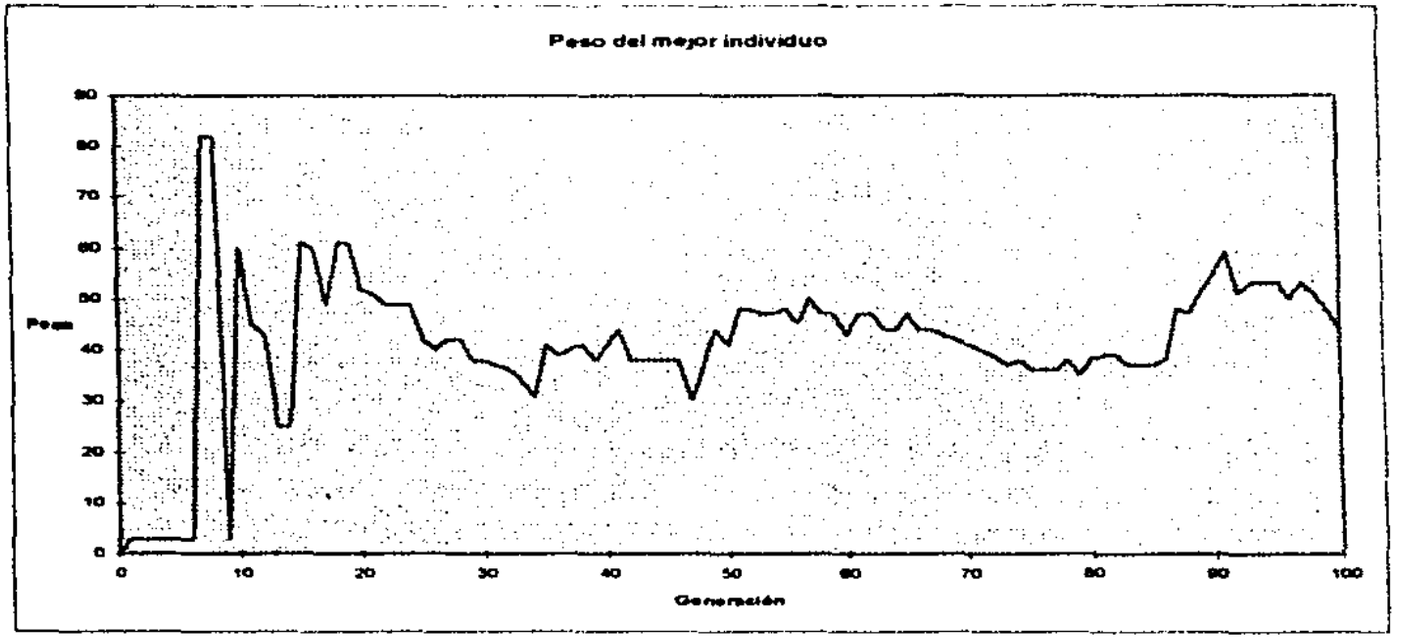
Corrida No. 36: aptitud y peso



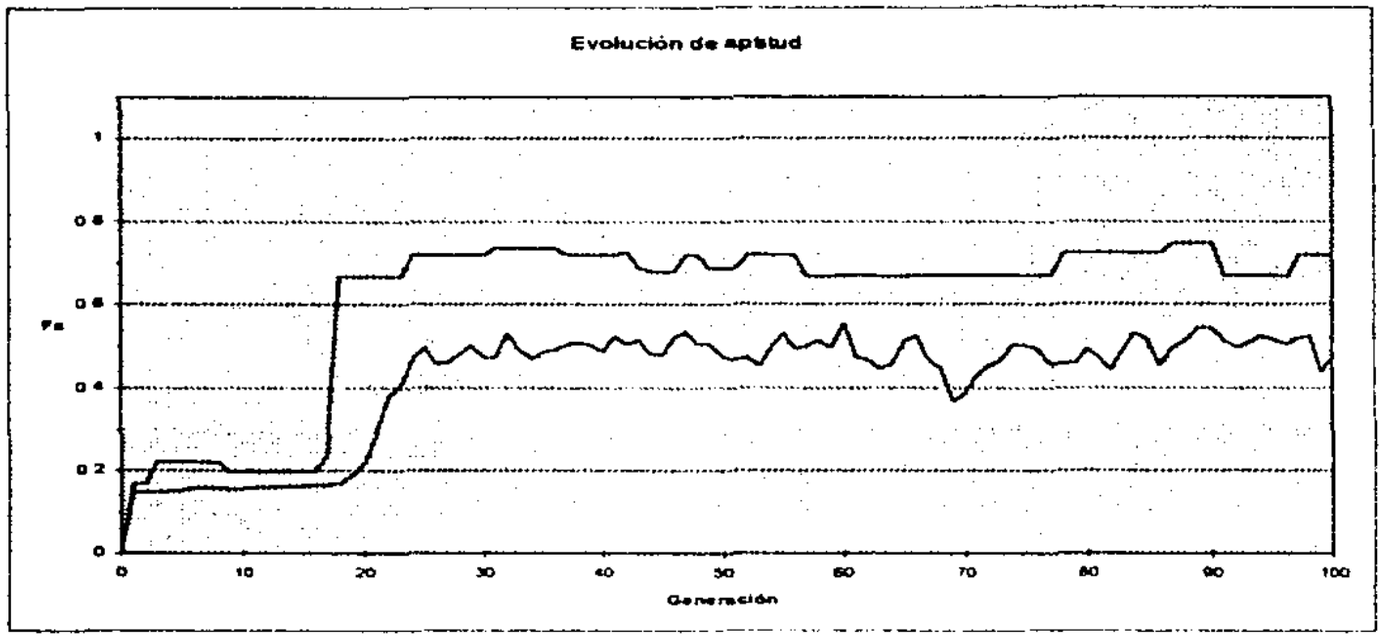


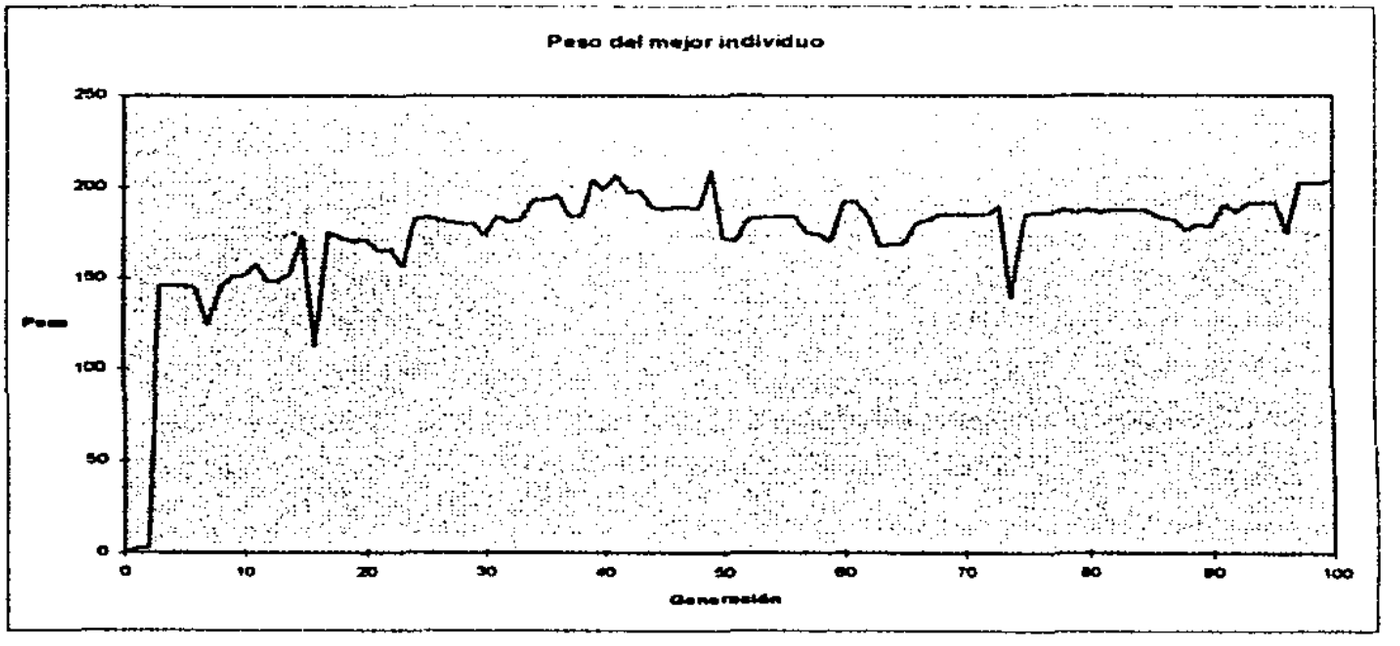
Corrida No. 39: aptitud y peso



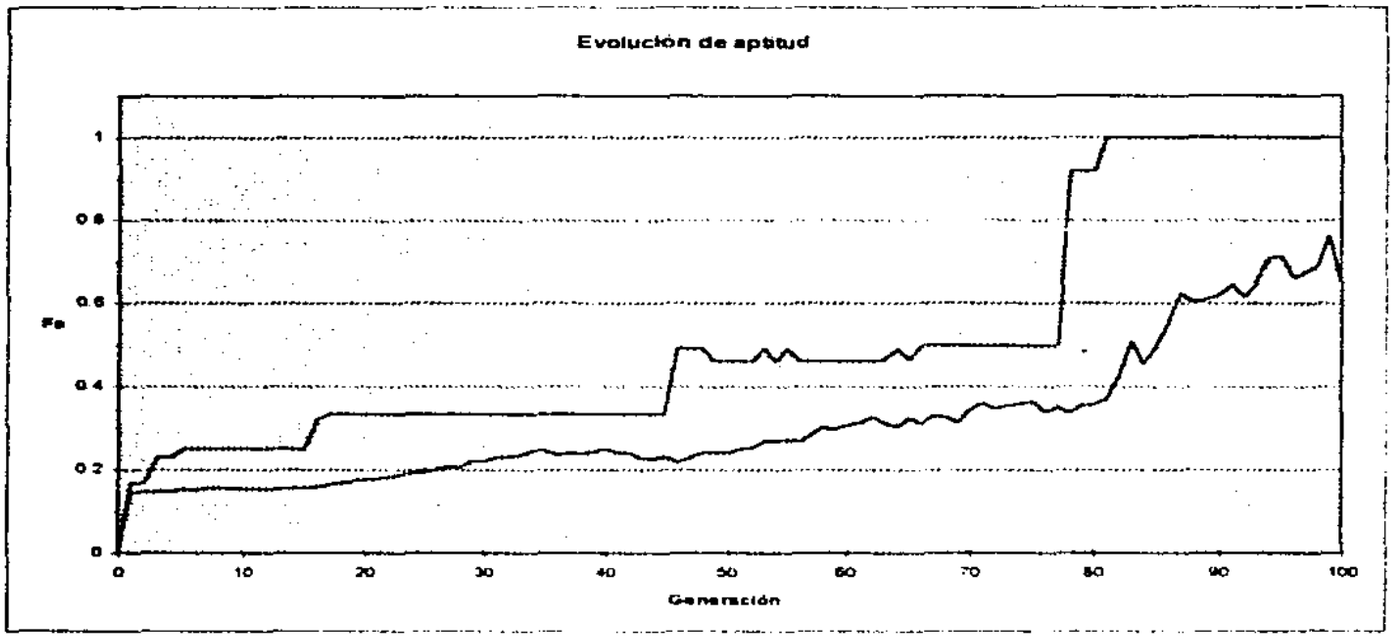


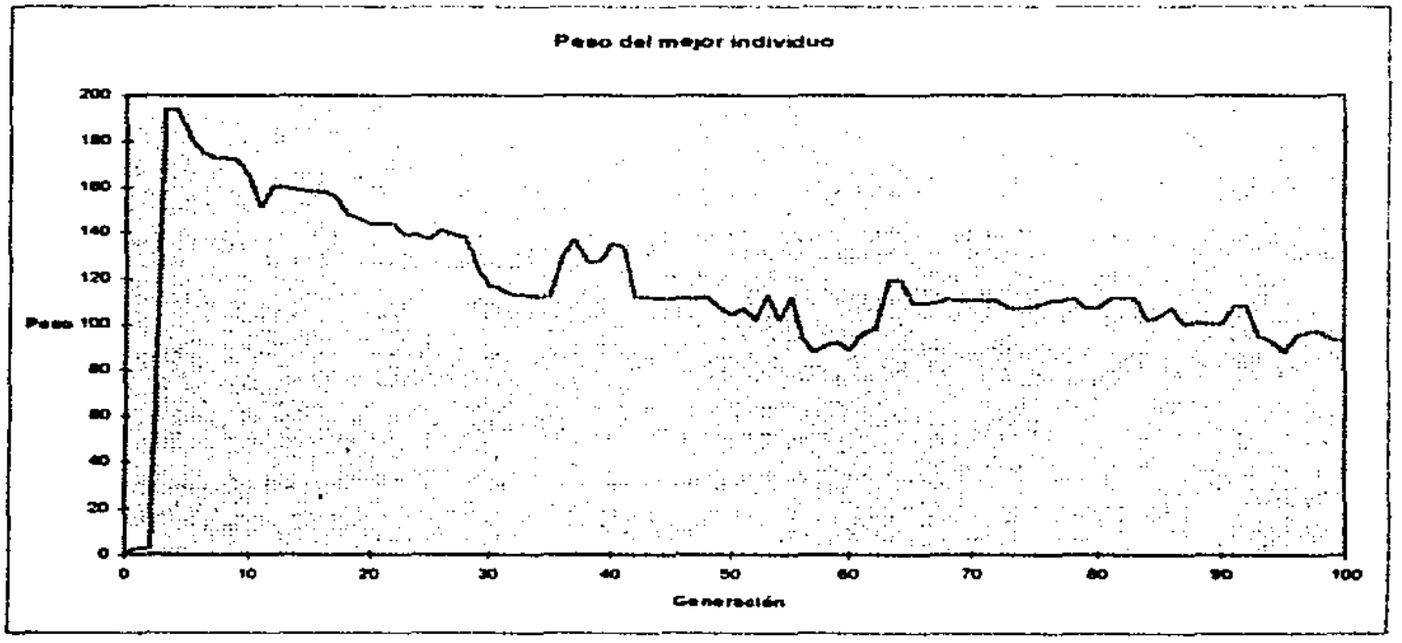
Corrida No. 42: aptitud y peso



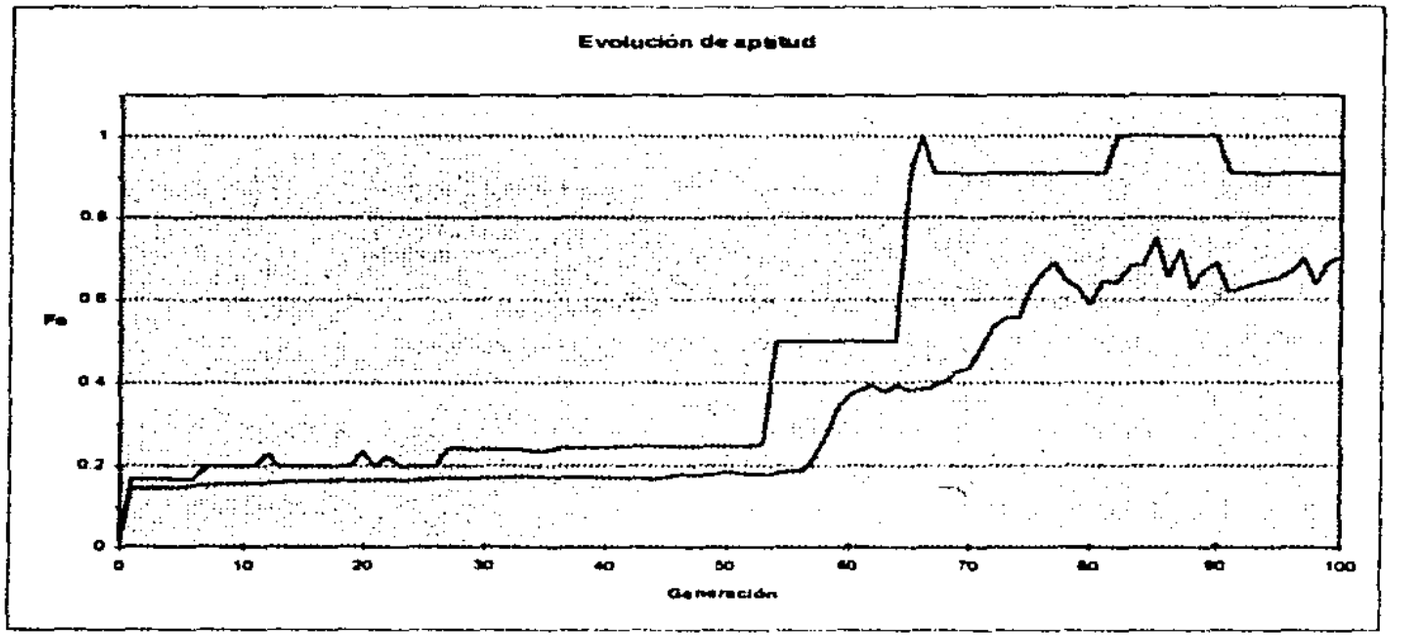


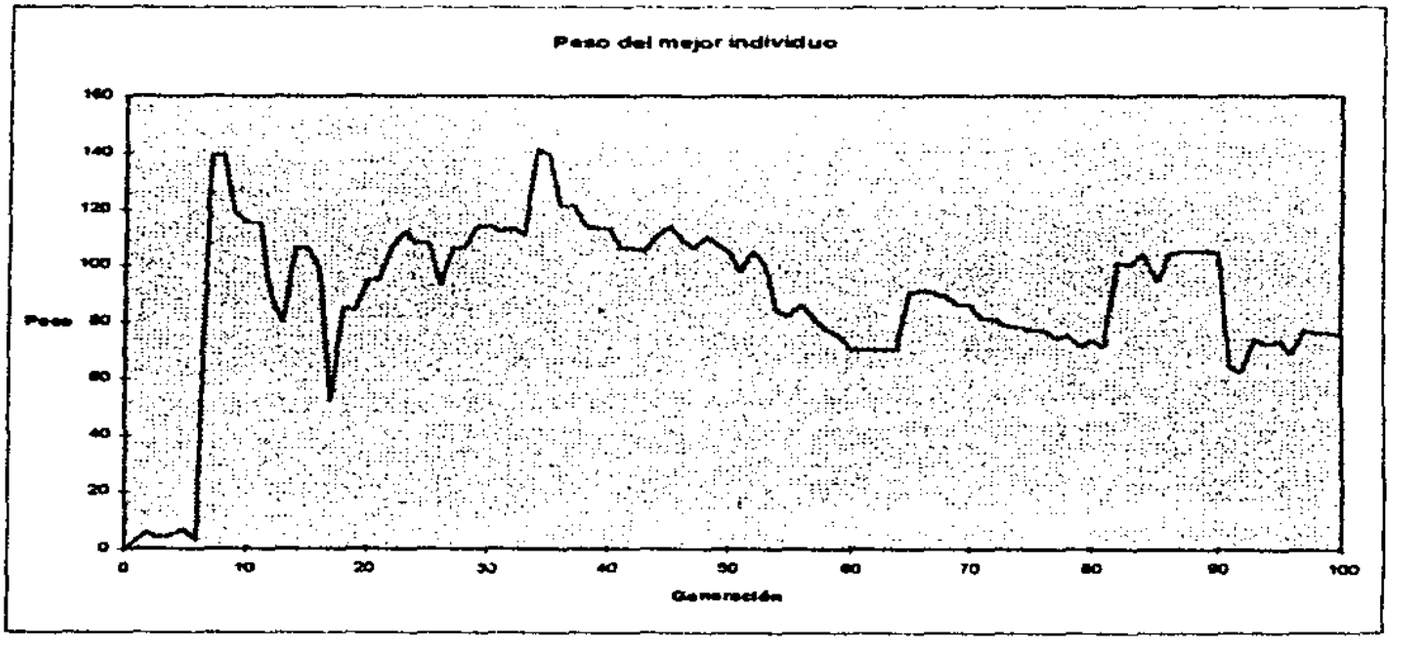
Corrida No. 43: aptitud y peso





Corrida No. 45: aptitud y peso





Corrida No. 48: aptitud y peso