AMPLIACIÓN DE UNGENÉTICO: UNA LIBRERÍA EN C++ DE ALGORITMOS GENÉTICOS CON CODIFICACIÓN HÍBRIDA

ANDRÉS RAMIRO DELGADILLO VEGA
JOHAN SEBASTIÁN MADRID ARROYO
JORGE MARIO VÉLEZ GUTIÉRREZ

UNIVERSIDAD NACIONAL DE COLOMBIA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
BOGOTÁ D.C.
2004

AMPLIACIÓN DE UNGENÉTICO: UNA LIBRERÍA EN C++ DE ALGORITMOS GENÉTICOS CON CODIFICACIÓN HÍBRIDA

ANDRÉS RAMIRO DELGADILLO VEGA
JOHAN SEBASTIÁN MADRID ARROYO
JORGE MARIO VÉLEZ GUTIÉRREZ

Trabajo realizado como requisito parcial
para optar al título de Ingeniero Electricista

Director
ÓSCAR GERMÁN DUARTE VELASCO M.Sc., Ph.D.

UNIVERSIDAD NACIONAL DE COLOMBIA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
BOGOTÁ D.C.
2004

Nota de aceptación

_____

_____

_____

_____

_____

Ing. Oscar Duarte Velasco M.Sc., Ph.D.

Director del proyecto

_____

Firma del presidente del jurado

_____

Firma del jurado

_____

Firma del jurado

Bogotá D.C., 15 de Abril de 2004

A mi padres Ramiro y Rosa,
a mis hermanas Liliana y Diana
y a Maria Paula
que me apoyaron durante estos cinco años
Andrés

A mis padres y hermanitas,
a mis abuelitos (ángeles de la guarda),
a mi música y a todos los que me han aportado
en la construcción de este corto e importante
periodo de vida universitaria.
Jorge Mario

A mis padres y hermanos
Gracias por brindarme su apoyo y comprensión
Sebastián

AGRADECIMIENTOS

- Al ingeniero *Óscar Duarte Velasco M.Sc., Ph.D.* quien nos guió durante el desarrollo de todo el proyecto.
- Al ingeniero *Pedro Nel Martínez* quien aportó la información y el conocimiento para el desarrollo del modelo de mantenimiento industrial.
- A *Julián Toro*, quien realizó el diseño de la presentación para la segunda versión de UNGenético.
- A las personas que colaboraron en el desarrollo de este proyecto.

CONTENIDO

LISTA DE TABLAS

LISTA DE FIGURAS

INTRODUCCIÓN

Los algoritmos genéticos constituyen un caso particular de las técnicas de búsqueda y optimización que simulan el proceso de evolución natural. En los últimos años este concepto ha llamado ampliamente la atención de los investigadores al punto de convertirse en una de las técnicas más implementadas en problemas de esta naturaleza, especialmente en casos en los cuales el espacio de búsqueda es de gran tamaño y variabilidad.

En el año 2002 se publicó UNGenético 1.0, una librería en lenguaje C++ que contiene las clases básicas para la optimización de modelos de este tipo utilizando algoritmos genéticos. Esta librería cuenta con dos características especiales que hacen de ella una herramienta importante en su área. La primera es la capacidad para almacenar dentro del mismo cromosoma genes de distinto tipo, lo que se conoce como codificación híbrida; la segunda es la posibilidad que ofrece de resolver diferentes tipos de problemas, desde funciones sencillas hasta modelos complejos.

UNGenético 1.0 representó un avance importante en el área de investigación de la computación flexible, sin embargo, presentaba algunas limitaciones. Su utilización no resultaba totalmente comprensible para usuarios que no contaran con un nivel considerable de conocimiento en programación en C++. Además, no existía una documentación completa para su uso, ya que no se contaba con un manual de apoyo al programador que brindara la información suficiente para realizar proyectos de optimización.

Con el desarrollo de este proyecto se pretende primordialmente implementar una nueva versión de la librería que supere las limitaciones encontradas en la primera. Con este fin se plantearon distintas estrategias que en conjunto pueden lograr que esta nueva versión, denominada

UNGenético 2.0, se posicione dentro del ámbito investigativo como una herramienta importante de búsqueda y optimización. Entre estas estrategias se encuentran la simplificación de la definición del modelo por parte del usuario, la ampliación de las posibilidades de manipular la información con que trabaja el algoritmo genético, la creación de una interfaz gráfica adaptable a distintos sistemas operativos, la elaboración de una documentación amplia y comprensible que proporcione al usuario un soporte bibliográfico útil al emplear la librería y finalmente, el desarrollo de ejemplos prácticos en los cuales se utilice UNGenético 2.0 como herramienta de optimización que demuestren su utilidad en problemas aplicables a diferentes áreas de la ciencia.

El punto de partida para el desarrollo de este proyecto fue la librería UNGenético 1.0, de la cual aún se mantienen su lenguaje y estructura básica. A partir de esta primera versión se crearon nuevas clases y se perfeccionaron las existentes. Se creó también una documentación descriptiva de la librería y un manual de referencia para el usuario de la nueva versión. Paralelamente a estas actividades se elaboraron herramientas gráficas que facilitan la elaboración de proyectos de optimización y permiten visualizar los resultados del proceso de ejecución del algoritmo genético manteniendo el propósito de funcionalidad de la librería en diferentes plataformas. Finalmente, se recurrió al conocimiento de profesionales en distintas áreas de la ingeniería quienes hicieron aportes importantes a este proyecto al adoptar a UNGenético 2.0 como herramienta para la solución de los modelos inherentes a sus investigaciones. Los resultados de estas implementaciones sirven como base para verificar la aplicabilidad de UNGenético 2.0.

La incursión en estrategias como los algoritmos genéticos y su correcta implementación en la búsqueda de soluciones óptimas podrían tener un significado importante tanto para el área de trabajo de la computación flexible como para las áreas donde sean aplicadas este tipo de estrategias. De cada implementación desarrollada con UNGenético 2.0 se podrían obtener resultados valiosos que consoliden a los algoritmos genéticos como una forma eficaz de optimización de modelos que supera a las herramientas tradicionales.

Capítulo 1

ALGORITMOS GENÉTICOS

Los algoritmos genéticos constituyen una técnica de búsqueda fundamentada en el proceso de evolución natural donde los individuos más adaptados tienen mayores probabilidades de sobrevivir y de transferir su material genético a las siguientes generaciones.

La idea fundamental de los algoritmos genéticos consiste en encontrar una solución aceptable a un problema por medio del mejoramiento de un conjunto de individuos cuya función de evaluación corresponde a una solución del problema. Esta optimización se realiza mediante procesos selectivos y de intercambio de información genética.

La información sobre la que trabaja el algoritmo está almacenada en genes. Cada individuo está constituido por una cadena de genes denominada genoma que representa un punto en el espacio de búsqueda del problema, de este modo, el objetivo del algoritmo genético es encontrar al individuo que contenga las mejores características posibles que solucionan el problema.

1.1. Codificación genética

La codificación genética indica la forma de representar la variables del problema a optimizar dentro del genoma. Cada individuo I de la población del algoritmo genético se compone de un arreglo de genes cuyo tamaño corresponde al tamaño del problema a optimizar, de este modo, el individuo C_x estará conformado por el arreglo $\{c_1^x, \ldots, c_n^x\}$ donde n representa el número de

variables del problema y cada c_i^x representa al gen del individuo C_x ubicado en la posición i del genoma.

Tradicionalmente, la codificación genética se ha desarrollado a través de cadenas de genes booleanos que representan los posibles valores que pueden tomar las variables del sistema. Sin embargo, este tipo de codificación, denominada *binaria*, presenta algunas dificultades en espacios de búsqueda continuos de grandes dimensiones y donde se requiere una alta precisión numérica.

Alternativamente a la codificación binaria, la literatura [7] plantea un sistema de codificación *real* donde cada variable del sistema a optimizar está representada por un gen que almacena un valor de tipo real en el genoma. El valor almacenado en cada gen se mantiene dentro de los límites del dominio de la variable a la cual representa, de este modo, todo proceso genético al que se someta el genoma debe preservar dichos límites. La codificación real se adapta en mejor manera a los espacios de búsqueda continuos y brinda mayor precisión numérica ya que representa de manera muy aproximada la formulación natural del problema.

Muchos de los operadores genéticos que operan individualmente sobre cada gen del genoma, basan su funcionamiento en esquemas de codificación real; por lo tanto, cuando se requiera, se hará la respectiva connotación a los operadores que pueden abordar otro tipo de codificación.

La figura 1.1 describe el proceso del algoritmo genético efectuado en una población $P(t)$, donde t indica la generación o iteración del algoritmo.

La creación de la población inicial $P(0)$ por lo general se realiza aleatoriamente mientras que el criterio de evaluación de cada población se basa en la función objetivo de sus individuos. En las siguientes secciones se detallan las demás fases propias del algoritmo genético, un análisis teórico más profundo se puede encontrar en [7] y [10].

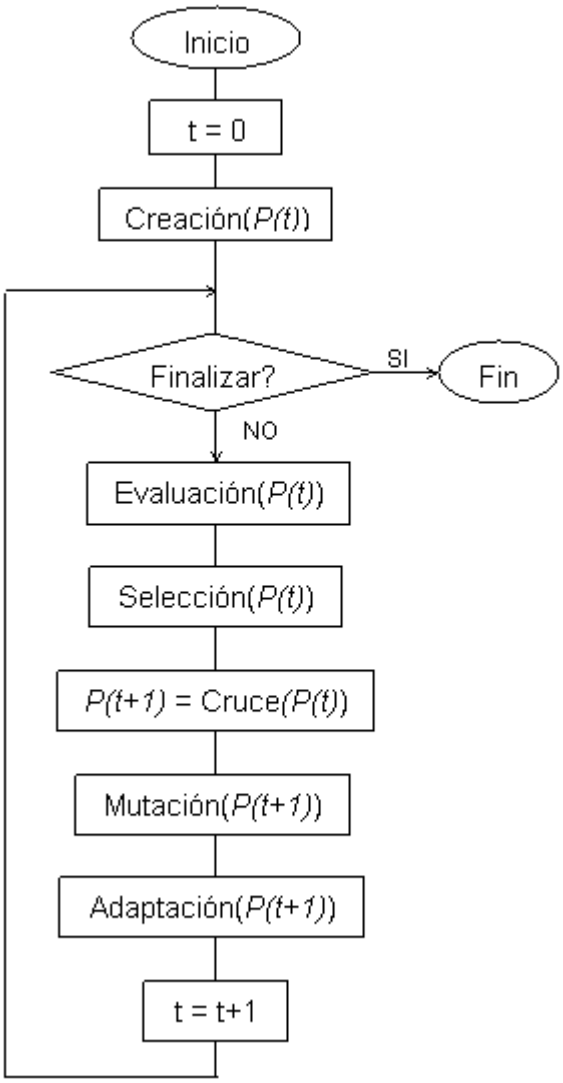


Figura 1.1: Diagrama de flujo generalizado de un algoritmo genético

1.2. Proceso de selección

El proceso de selección busca escoger ciertos individuos de la población quienes darán origen a las futuras generaciones. Por lo general, la selección depende del valor de la función de evaluación de cada individuo. Este proceso tiene dos etapas:

1.2.1. Asignación de probabilidad

Para una población de *N* individuos, se asigna una probabilidad de supervivencia *ps(i)* para el individuo ubicado en la posición *i* de la población, con este fin se emplea un operador de probabilidad. Algunos de los más usuales son:

1.2.1.1. Operador de probabilidad homogénea. Asigna a cada individuo de la población la misma probabilidad de supervivencia dada por:

$$ps(i) = \frac{1}{N}$$

1.2.1.2. Operador de probabilidad lineal. Ordena a los individuos de la población de mejor a peor y asigna a cada uno una probabilidad de supervivencia que depende de su posición en la generación, de esta manera:

$$ps(i) = \frac{\eta_{max} - \frac{(\eta_{max} - \eta_{min})i}{N+1}}{N}$$

Donde,

$$\eta_{max} = 2 - \eta_{min}$$

η_{min} se escoge del intervalo $[0, 2]$.

1.2.1.3. Operador de probabilidad proporcional. Asigna a cada individuo una probabilidad de supervivencia proporcional al valor de su función objetivo *FObj(i)*, así:

- Al minimizar

$$ps(i) = \frac{\frac{1}{FObjProp(i)}}{\Sigma_{i=1}^{N} \frac{1}{FObj(i)}}$$

- Al maximizar

$$ps(i) = \frac{FObjProp(i)}{\Sigma_{i=1}^{N} FObj(i)}$$

Donde

$$FObjProp(i) = FObj(i) + (1 - FObjmin)$$

$$FObjmin = \min\{1, FObj(i)\}, \quad i=1, \ldots, N.$$

1.2.2. Selección de individuos

La selección establece cuántas copias de cada individuo de la generación actual continuarán el proceso evolutivo, para ello se utiliza el operador de selección, uno de los más utilizados es:

1.2.2.1. Operador selección estocástica con reemplazo. Los individuos que componen la población son ordenados como segmentos de una línea, cada segmento corresponde a la probabilidad de supervivencia del individuo. La selección se produce al generarse un valor aleatorio que coincida con el segmento del individuo. El proceso se repite hasta obtener el número de individuos deseados, creando así la base para la nueva población.

1.3. Proceso de cruce

Consiste en mezclar la información genética de dos individuos con el fin de generar nuevos individuos. Este proceso se ejecuta en varias etapas:

1.3.1. Asignación de parejas

Por medio de un operador de asignación de parejas se decide entre cuáles individuos de la población actual se va a intercambiar la información genética para generar nuevos individuos, entre ellos se destacan:

1.3.1.1. Operador de asignacion de parejas aleatorias. La asignación de pareja para el individuo i se realiza aleatoriamente.

1.3.1.2. Operador de asignacion de parejas adyacentes. Al individuo i le corresponde como pareja el indivduo $i+1$ del arreglo de individuos de la población. $i \in [1, N]$ tal que i es impar.

1.3.1.3. Operador de asignacion de parejas extremos. Al primer individuo de la población le corresponde como pareja el último individuo en el arreglo, el segundo y penúltimo individuo conforman otra pareja. El proceso se repite hasta asignar parejas para todos los individuos de la población.

1.3.2. Estrategia general de reproducción

Empleando un operador de reproducción se definen los individuos que conformarán la siguiente generación del algortimo. Los más importantes son:

1.3.2.1. Operador de reproducción dos padres dos hijos. Por cada pareja que combina su información genética se generan dos hijos que reemplazarán a sus padres en la siguiente generación.

1.3.2.2. Operador de reproducción mejor padre mejor hijo. Por cada pareja que combina su información genética se generan dos hijos, el mejor hijo reemplazará al peor padre en la siguiente generación; la selección depende del valor de la función objetivo de los individuos.

1.3.2.3. Operador de reproducción mejores entre padres e hijos. Por cada pareja que combina su información genética se generan dos hijos, entre los padres y los hijos se escogen los dos mejores individuos y estos harán parte de la siguiente generación, la selección depende del valor de la función objetivo de los individuos.

1.3.3. Cruce

El proceso de cruce define la forma en que dos indviduos $C_1 = \{c_1^1, \ldots, c_n^1\}$ y $C_2 = \{c_1^2, \ldots, c_n^2\}$ con c_i restringido al intervalo $[a_i, b_i]$ comparten su información genética para generar un número determinado de individuos nuevos $H_x = \{h_1^x, \ldots, h_n^x\}$. Este proceso se realiza a través de operadores de cruce que trabajan sobre cada pareja de genes padres creando nuevos genes. El conjunto de posibles valores que pueden tomar estos nuevos genes se denomina *Intervalo de definición*, éste se puede dividir en dos clases de intervalos:

- *Intervalo de explotación:* sus puntos contienen infomación común a los dos genes padres, sus extremos son los valores de los genes padres.
- *Intervalo de exploración:* se define entre el extremo del intervalo de definición de cada cruce y el valor del gen padre más cercano al mismo. No existe garantía de que sus puntos contengan infomación común a los dos genes padres.

A continuación se detallan los operadores de cruce más comunes mientras que en el anexo A se describen gráficamente los intervalos de definición para algunos de ellos:

Operador de cruce simple. Dados dos individuos padres con genomas $C_1 = \{c_1^1, \ldots, c_n^1\}$ y $C_2 = \{c_1^2, \ldots, c_n^2\}$, se generan dos nuevos individuos hijos con genomas $H_1 = \left\{h_1^1, \ldots, h_i^1, h_{i+1}^2, \ldots, h_n^2\right\}$ y $H_2 = \left\{h_1^2, \ldots, h_i^2, h_{i+1}^1, \ldots, h_n^1\right\}$, donde i es un número entero aleatorio del intervalo $[1, n-1]$ denominado punto de cruce. Este operador se adapta a genes de cualquier tipo.

1.3.3.1. Operadores de cruce para genes de tipo binario

- **Operador de cruce discreto.** El valor de cada nuevo gen creado se escoge aleatoriamente del conjunto $\{c_i^1, c_i^2\}$. Este es el operador de cruce más usual en sistemas de codificación binaria.

1.3.3.2. Operadores de cruce para genes de tipo real

- **Operador de cruce aritmético.** Genera dos genes hijos dentro del intervalo de explotación cuyos valores dependen de un parámetro constante λ, así:

$$H_i^1 = \lambda c_i^1 + (1 - \lambda) c_i^2$$

$$H_i^2 = \lambda c_i^2 + (1 - \lambda) c_i^1$$

- **Operador de cruce BLX-α.** Con este operador es posible equilibrar las tendencias de explotación y exploración, gracias a que el parámetro $\alpha \in [0, 1]$ modifica el intervalo de creación para los nuevos genes. El valor de los nuevos genes se escoge aleatoriamente del intervalo dado por:

$$[c_{min} - I\alpha, c_{max} + I\alpha]$$

Donde,

$$c_{min} = min\left\{c_i^1, c_i^2\right\}$$

$$c_{max} = max\left\{c_i^1, c_i^2\right\}$$

$$I = c_{max} - c_{min}$$

- **Operador de cruce discreto.** El valor de cada nuevo gen creado se escoge aleatoriamente del conjunto $\{c_i^1, c_i^2\}$.

- **Operador de cruce heurístico.** Este operador da importancia al valor de la función objetivo de los individuos que realizan el cruce. Los genes producidos se localizan en el intervalo de exploración del lado del mejor padre de la siguiente manera:

$$h_i = r(c_i^1 - c_i^2) + c_i^1$$

donde C_1 es el padre con mejor función de evaluación y r es un número aleatorio perteneciente al intervalo $[0, 1]$.

- **Operador de cruce intermedio extendido.** Este operador incentiva en mayor proporción la explotación, el valor de cada nuevo gen creado obedece a:

$$h_i = c_i^1 + \alpha(c_i^2 - c_i^1)$$

el valor de α se escoge aleatoriamente del intervalo $[-0{,}25, 1{,}25]$.

- **Operador de cruce lineal.** Genera tres genes hijos de los cuales se escogen sólo dos para conformar los nuevos individuos. Cada uno de los genes creados se ubica en un punto central de los intervalos de explotación y exploración, de la siguiente manera:

$$h_i^1 = \frac{1}{2}c_i^1 + \frac{1}{2}c_i^2$$

$$h_i^2 = \frac{3}{2}c_i^1 - \frac{1}{2}c_i^2$$

$$h_i^3 = -\frac{1}{2}c_i^1 + \frac{3}{2}c_i^2$$

- **Operador de cruce lineal BGA.** Este operador da importancia al valor de la función objetivo de los individuos que realizan el cruce. Crea genes en los intervalos de explotación o exploración cercanos al mejor padre siendo mayor la probabilidad de incentivar la exploración, así:

$$h_i = c_i^1 \pm rango \cdot \gamma \cdot \Lambda$$

donde

$$\Lambda = \frac{c_i^2 - c_i^1}{\|FObj(C_1) - FObj(C_2)\|}$$

$$rango = factor(b_i - a_i)$$

$$\gamma = \sum_{k=0}^{15} \alpha_k 2^{-k}$$

C_1 es el padre con mejor función de evaluación, $factor \in [0, 1]$ y α_k se escoge aleatoriamente del conjunto $\{0, 1\}$ con probabilidad de 0.0625 de ser igual a 1.

- **Operador de cruce plano.** Este operador produce genes hijos en el intervalo de explotación cuyos valores se crean aleatoriamente en el intervalo $[c_i^1, c_i^2]$.

1.4. Proceso de mutación

Consiste en alterar las características genéticas de un individuo con el fin de aumentar la probabilidad de exploración del espacio de búsqueda y disminuir el riesgo de estancamiento del algoritmo en óptimos locales. Cada gen c_i tiene una probabilidad de mutación pm que determina la alteración del mismo a un nuevo gen c_i'. El nuevo valor del gen debe estar en el rango $[a_i, b_i]$. Los operadores de mutación más comunes son:

1.4.1. Operadores de mutación para genes de tipo binario

1.4.1.1. Operador de mutación uniforme. Cambia el valor del gen por su complemento booleano, de *verdadero* a *falso* o viceversa.

1.4.2. Operadores de mutación para genes de tipo real

1.4.2.1. Operador de mutación uniforme. El nuevo valor del gen c_i' es un número aleatorio del intervalo $[a_i, b_i]$.

1.4.2.2. Operador de mutación no uniforme. Con este operador, el intervalo de generación del nuevo valor disminuye conforme avanzan las iteraciones del algoritmo permitiendo una búsqueda local en las etapas finales. El nuevo valor del gen es creado de la siguiente manera:

$$c_i' = \begin{cases} c_i + \Delta(t, b_i - c_i) & \text{si } \beta = 0 \\ c_i - \Delta(t, b_i - c_i) & \text{si } \beta = 1 \end{cases}$$

Donde

$$\Delta(t, y) = y(1 - r^{(1-\frac{t}{T})^b})$$

t es el número de la generación actual, T es el máximo número de generaciones del algoritmo, β es un número aleatorio del conjunto $\{0, 1\}$, r es un número aleatorio del intervalo $[0, 1]$, y b es un parámetro que determina el grado de dependencia con el número de iteraciones.

1.4.2.3. Operador de mutación de Muhlenbein Este operador produce valores muy cercanos al valor original del gen, así:

$$c_i' = c_i \pm rango \cdot \gamma$$

Donde

$$rango = factor(b_i - a_i)$$

$$\gamma = \sum_{k=0}^{15} \alpha_k 2^{-k}$$

El signo $+$ o $-$ se escoge con igual probabilidad, $factor \in [0, 1]$ y α_k se escoge aleatoriamente del conjunto $\{0, 1\}$ con probabilidad de 0.0625 de ser igual a 1.

1.5. Proceso de adaptación

Un sistema adaptativo tiene la facilidad de modificar sus propiedades conforme va conociendo su entorno con el fin de adecuarse a las características del mismo. Los algoritmos genéticos usan operadores de adaptación que ajustan sus parámetros básicos con el fin de optimizar el proceso de búsqueda, entre ellos se encuentran:

1.5.1. Operador de adaptación elitismo

Obliga a que en cada nueva generación del algoritmo se incluya el mejor individuo presente en la generación anterior, asegurando que tienda a mejorar su respuesta.

1.5.2. Operador de adaptación tamaño de la población

Disminuye el número de individuos presente en cada generación conforme avanza la ejecución del algoritmo siguiendo un comportamiento lineal. En un número amplio de generaciones se tendrá un número mínimo de individuos que favorecerá el ajuste local.

1.5.3. Operador de adaptación probabilidad de mutación

Varía la probabilidad de mutación de cada gen. Tiene como propósito incentivar la explotación o exploración del espacio de búsqueda según sea el caso,

- Variación escalonada de la probabilidad de mutación. Favorece la exploración del espacio de búsqueda. La probabilidad de mutación de los genes aumenta escalonadamente cuando la medida offline del algoritmo no presenta una variación significativa durante un número determinado de generaciones.
- Variación exponencial decreciente de la probabilidad de mutación. Favorece la explotación del espacio de búsqueda. La probabilidad de mutación de los genes en la generación t,

$pm(t)$, disminuye simultáneamente con la ejecución del algoritmo siguiendo una curva exponencial así:

$$pm(t) = pm_{max}e^{-b\frac{t}{T}}$$

donde pm_{max} es la probabilidad de mutación máxima permitida, b es un parámetro que gradúa la curva, t es el número de la generación actual y T es el número máximo de generaciones.

1.6. Proceso de finalización

La finalización del algoritmo genético puede darse por medio de un truncamiento o por medio de técnicas de supervisión de su comportamiento. Generalmente, estas técnicas se basan en las medidas online y offline del algoritmo:

- Medida online: promedio de las funciones de evaluación de todos los individuos que han aparecido hasta ese momento en todas las generaciones.

- Medida offline: promedio de las funciones de evaluación de los mejores individuos de cada generación hasta ese momento.

Los encargados de interrumpir el proceso genético son los operadores de finalización, entre ellos se encuentran:

1.6.1. Operador de finalización número máximo de generaciones

Interrumpe el algoritmo genético cuando se han ejecutado un número máximo de generaciones. El límite se escoge aleatoriamente dependiendo del sistema a optimizar, sin embargo, la literatura sugiere que este límite no sea menor que 40.

1.6.2. Operador de finalización online

Interrumpe el algoritmo genético cuando éste no presenta una variación significativa en su medida online después de un número determinado de generaciones. De esta forma se asegura que los individuos de las generaciones finales del algoritmo tengan un comportamiento homogéneo.

1.6.3. Operador de finalización offline

Interrumpe el algoritmo genético cuando éste no presenta una variación significativa en su medida offline después de un número determinado de generaciones. Asegura la detención del algoritmo cuando no se encuentra una mejor respuesta a la actual.

Capítulo 2

CAMBIOS REALIZADOS A UNGenético 1.0

UNGenético 2.0 tiene como finalidad ser una librería de amplia aplicación en diversos tipos de problemas de optimización y búsqueda, de correcto funcionamiento y manejo de memoria, flexibilidad, de fácil comprensión y utilización. Tomando UNGenético 1.0 como punto de partida, se hicieron numerosas modificaciones que permitieran superar sus limitaciones y alcanzar los objetivos propuestos para esta segunda versión.

Los cambios realizados se pueden resumir en estructurales, de ampliación de la librería y de corrección de errores y optimización de funciones. Como cambios estructurales se tienen la modificación de la definición del modelo, la cual se simplificó considerablemente y el encapsulamiento de clases. Como cambios de ampliación se tienen la creación de un nuevo tipo de gen con sus respectivos operadores de mutación y cruce, la creación de nuevas clases correspondientes a diferentes operadores de probabilidad, asignación de parejas, reproducción, cruce, mutación, adaptación y finalización. También la creación de clases genéricas para uso de un entorno gráfico en aplicaciones realizadas utilizando la librería, y la definición de MACROS y tipos de datos útiles en la definición del algoritmo genético.

Una descripción de las clases de la primera versión, al igual que su código fuente y fundamentos de utilización se puede consultar en [10]. Los cambios realizados se describen a lo largo de este capítulo sin entrar en detalles del código fuente, el cual se encuentra en el CD adjunto a este volumen. Las modificaciones son expuestas a un nivel descriptivo, procurando dejar en claro las razones por las cuales fueron adoptadas y las diferencias entre las dos versiones.

2.1. Nomenclatura utilizada

En esta versión de UNGenético se adoptó una nomenclatura unificada para la denominación de variables con el fin de facilitar la depuración de cualquier aplicación que utilice la librería. La unificación se realizó para los siguientes ítems:

2.1.1. Miembros de clases

En los miembros o propiedades de todas las clases pertenecientes a UNGenético 2.0 se adoptó el prefijo *"m_"*. De esta forma, al depurar una función que pertenece a una clase, es posible diferenciar cuando se hace referencia a un miembro de la clase, o a una variable definida dentro de la función. Por ejemplo, en la clase *GenEntero*, el miembro que contiene el valor del gen, adoptó el nombre de *m_Valor*.

2.1.2. Apuntadores

Cualquier apuntador definido dentro de la librería lleva el prefijo *"p"*. Si el apuntador es además miembro de una clase, su prefijo es *"m_p"*. Por ejemplo, en la clase *Individuo*, el miembro que apunta a la pareja del individuo adoptó el nombre de *m_pPareja*.

2.1.3. MACROS

Todas las MACROS definidas en la librería tienen su nombre completo en mayúsculas para diferenciarse de las funciones. Por ejemplo, se definen MACROS como *ADICIONAR_GENENTERO, DEFINIR_OPERADOR_PROBABILIDAD*, etc. En la seccion 2.6 se expone esta nueva alternativa que ofrece la librería.

2.1.4. Constantes

Al igual que las macros, todas las constantes se nombran con mayúsculas. Esto incluye tanto las constantes definidas con directivas de preprocesador *(#define)*, como las definidas en enumeraciones (bloques *enum*). Por ejemplo, se definieron las constantes *USAR_VENTANA, ADAPTACION_PROBMUTACION_OFFLINE, ESTADO_DECODIFICAR*, etc. En la sección 2.8 se expone la definición de nuevas constantes.

2.2. Encapsulamiento de clases y ampliación de funcionalidad

Para hablar de encapsulamiento de clases, es conveniente definir los términos clase y encapsulamiento, ampliamente utilizados en programación orientada por objetos.

Una clase, en términos de programación orientada por objetos, se entiende como la abstracción de un objeto real para lograr su codificación en un lenguaje de programación. Al igual que el objeto al que representa, la clase consta de ciertas propiedades, miembros o datos que la caracterizan y de funciones, métodos u operaciones mediante los cuales interactúa con el entorno.

El encapsulamiento de una clase se puede definir como la capacidad que tiene ésta de interactuar con el entorno siendo "consciente" de la forma como éste afecta sus propiedades, logrando mantener su consistencia y asegurando que no se podrán violar ciertas reglas de integridad. Como su nombre lo indica, encapsular implica ocultar, diferenciando entre lo visible o accesible para el entorno, de lo invisible o inaccesible.

Para citar un ejemplo sencillo en UNGenético 2.0, se puede hablar de la clase *GenEntero*, cuya propiedad principal es el valor almacenado por el gen (*m_Valor*). Asimismo, tiene dos propiedades que determinan el rango dentro del cual puede variar este valor; éstas son el límite superior (*m_Maximo*) y el límite inferior (*m_Minimo*). Si la clase no se encontrara encapsulada, desde alguna parte del programa podría establecerse para un *GenEntero* un valor fuera del rango definido, provocando inconsistencias y resultados probablemente no deseados en el programa.

Al encapsular la clase, se puede asegurar que el valor que se asigne siempre esté dentro del rango definido, manteniendo su consistencia.

Para lograr encapsulamiento, el lenguaje C++ permite restringir el acceso a un miembro desde fuera de la clase, declarándolo como privado (*private*) o protegido (*protected*), en cuyo caso la única forma de acceder a este miembro es a través de funciones pertenecientes a la clase. En el caso citado, para mantener la integridad de *GenEntero*, se declara el miembro *m_Valor* como privado y se adicionan dos funciones a la clase: una para permitir leer el valor (*getVal()*) y una para permitir cambiar el valor (*setVal(nuevoValor)*). Esta última, antes de asignar a *m_Valor* el nuevo valor, realiza una comprobación de los límites, asegurando que no se tendrán inconsistencias. Es claro que esta comprobación se podría realizar desde fuera de la clase pero el programador que utilice la clase incurriría en un trabajo adicional cada vez que quisiera modificar el valor.

De igual manera, es posible que se requiera definir algún miembro como "de solo lectura", es decir, que se pueda conocer pero no alterar desde el exterior, en cuyo caso se define únicamente la primera función mencionada (*getVal()*) para obtener su valor.

En UNGenético 1.0 no se tenía restricción de miembros en ninguna de las clases, lo cual provocaba, como ya se explicó, un trabajo adicional y en algunas ocasiones inconsistencias en los valores. Visto desde este punto, el trabajo adicional en la ampliación de la funcionalidad para lograr encapsulamiento genera mejores resultados.

Una ventaja adicional del encapsulamiento es la capacidad que se logra de modificar la implementación de un objeto a nivel interno, sin tener que modificar el código donde éste es utilizado. Esta fue una de las mayores dificultades en la obtención de la versión 2.0 de UNGenético, ya que cada modificación al interior de una clase, implicaba cambiar el código en el resto de la librería.

Uno de los ejemplos clave de encapsulamiento para citar en UNGenético 2.0, es la clase *Individuo*. Esta contiene un arreglo de genes que almacena la información genética del individuo modelado. Una de sus variables o propiedades mas importantes es *m_Objetivo*, la cual almacena el valor de la función objetivo. Al presentarse un cambio en la información genética, es muy probable que varíe la evaluación de su función objetivo. Si la clase no se encuentra debidamente encapsulada, se da la posibilidad de hacer alguna variación a la información genética desde el

entorno sin actualizar la función objetivo, provocando inconsistencias a la hora de consultar este valor si no es actualizado previamente también desde el exterior de la clase.

Cuando se realiza el encapsulamiento completo de esta clase, en lugar de permitir el acceso directo al miembro *m_Objetivo*, se provee una función llamada *objetivo()*, mediante la cual se puede asegurar que cada vez que se consulte el valor de la función objetivo del individuo, se retorne el valor que realmente refleje la información genética actual, sin que el programador se preocupe por su actualización explícita.

Teniendo en cuenta que actualizar el valor de *m_Objetivo* implica decodificar la información genética y calcular la función objetivo con las variables decodificadas, también se asegura un menor tiempo de proceso si se mantiene una variable auxiliar booleana (*m_objetivoActualizado*), por supuesto privada, la cual indica si *m_Objetivo* se encuentra actualizado o no. De esta forma, cada vez que se invoca la función *objetivo()*, ésta comprueba el valor de *m_objetivoActualizado*, calculando el valor de la función objetivo únicamente cuando es necesario.

Lógicamente, para mantener la variable *m_objetivoActualizado* en un valor correcto, se debe tener encapsulada la información genética para registrar cada variación que se realice sobre la misma. Obsérvese que todos los métodos de *Individuo* que realizan alguna modificación sobre la información genética, asignan a esta variable el valor *false*.

Se han citado solo algunos ejemplos, pero en general, en UNGenético 2.0 las clases han sido encapsuladas y se ha procurado proveer consistencia en los datos que se administran.

En términos generales, para lograr encapsular las clases se realizaron principalmente los siguientes tipos de modificaciones:

- Restricción de acceso a la implementación de clases. Esto es:
  - Restricción de acceso a propiedades. Implica adición de funciones de lectura/escritura de los datos que se deben conocer o modificar desde el exterior de la clase.
  - Restricción de acceso a funciones. Funciones intermedias a las que sólo pueden acceder otras funciones pertenecientes a la misma clase.
- Definición de diferentes constructores para la misma clase:

    - Constructor por defecto.
    - Constructor con parámetros iniciales.
    - Constructor por copia de otro objeto de la misma clase.

- Definición de operadores de asignación, para la copia de objetos de la misma clase.

- Uso del modificador *const* en funciones que no deben alterar los miembros de la clase, especialmente en clases abstractas.

- Retorno de datos de solo lectura (*const*).

Cabe aclarar que cada una de las modificaciones mencionadas no se aplicó a todas las clases, ya que no se consideró necesario en todos los casos.

La clase *AlgoritmoGenetico* no se encapsuló fuertemente en razón a que es la clase que el usuario debe derivar para poder implementar su algoritmo. Sin embargo, muchas de las operaciones que realizaba esta clase accediendo a los miembros de los objetos que la conformaban, se delegaron a funciones de clases inferiores como *Poblacion, Individuo y Gen* respectivamente.

2.3. Definición del modelo

La librería se ha reestructurado de tal forma que la definición del modelo se ha simplificado de una manera significativa. Probablemente éste es el cambio más visible para el usuario de UNGenético, y, simultáneamente, es el resultado de cambios menos visibles implementados en el interior de la librería.

En lugar de crear una clase derivada de *Individuo*, la cual ya no es abstracta, ahora debe crearse una clase derivada de *AlgoritmoGenetico*, la cual fue convertida en abstracta. A continuación se presenta una comparación entre la forma de definir el modelo en las dos versiones.

2.3.1. Definición de las variables del problema

Es similar en ambas versiones, salvo que en UNGenético 1.0 se realizaba en el interior de la clase derivada de *Individuo*, de forma que cada individuo creado guardaba una instancia de cada variable del problema. En la presente versión, las variables del problema se definen en el interior de la clase derivada de *AlgoritmoGenetico*, lo que se traduce en menor gasto de memoria, pues únicamente se necesita una instancia de cada variable, la cual será administrada por la clase *AlgoritmoGenetico*.

Se podría pensar que esto implica un gasto mayor de tiempo por ser una sola variable que se va actualizando con el paso de los individuos, pero en realidad el tiempo empleado es el mismo, ya que en ambos casos se debe decodificar la información genética de cada individuo para el cálculo de la función objetivo.

2.3.2. Codificación del modelo

Al hablar de codificación del modelo se hace referencia a la forma que debe tomar el individuo a nivel genético para representar una solución del mismo, a la forma en que se realiza la conversión de la información genética en variables del modelo y a la conversión de las variables del modelo en información genética. Desde este punto de vista, se presentan tres estados o instancias diferentes por las que debe pasar cada individuo para representar fielmente al modelo durante el proceso de optimización, estos son: creación de individuos, codificación de la información genética y decodificación de la información genética, los cuales se describen brevemente a continuación.

2.3.2.1. Creación de individuos. En el momento de su construcción, se define la forma que adoptará el individuo, determinada por la constitución del genoma, es decir, el número y tipos de genes que contendrá.

2.3.2.2. Codificación de la información genética. Se refiere a la forma en que se traducen las variables del modelo en información genética del individuo.

2.3.2.3. Decodificación de la información genética. Se refiere a la forma en que se traduce la información genética del individuo en las variables del modelo.

En UNGenético 1.0, el usuario debía proveer obligatoriamente una función para la definición de cada estado del individuo, de la siguiente manera:

- Creación: se debían especificar los genes que contendría el individuo mediante el constructor de la clase derivada de *Individuo*.
- Codificación: se especificaba mediante la función *codificar()* de la clase derivada de *Individuo*.
- Decodificación: se especificaba mediante la función *decodificar()* de la clase derivada de *Individuo*.

En UNGenético 2.0 también deben definirse obligatoriamente los tres estados de codificación, pero todos integrados en una sola función perteneciente a la clase derivada de *AlgoritmoGenetico*. Esta función es de la forma *codificacion(pIndividuo, estado)*, la cual recibe como argumentos el individuo en proceso de codificación y el estado en que se encuentra (creación, codificación o decodificación). También se han creado MACROS que facilitan la adición de genes al individuo, y que proveen los tres estados del individuo en un solo paso. Estas se describen en la sección 2.6 de este capítulo.

Adicionalmente, para la creación de un nuevo *Individuo* por copia de otro, en la versión 1.0, el usuario tenía que definir las funciones *copiarDetalles()* y *crearCopia()*. Estas fueron abolidas para la presente versión, ya que se hicieron innecesarias al convertir *Individuo* en una clase no abstracta y mediante la implementación del operador de asignación y del constructor por copia para esta clase.

2.3.3. Definición de la función objetivo

Es muy similar en las dos versiones. Como es de esperarse, la definición de la función objetivo por parte del usuario es de carácter obligatorio, ya que es la característica principal a tomar en cuenta en el proceso de optimización del problema. En la primera versión, esta función hacía

parte de la clase derivada de *Individuo*, y en la versión actual hace parte de la clase derivada de *AlgoritmoGenetico*. En ambas versiones, la función objetivo hace uso de las variables del modelo y no de la información genética, la cual, de manera transparente para el usuario, ha sido decodificada previamente.

2.3.4. Definición de operadores a utilizar

En ambas versiones es opcional. En la primera versión de UNGenético, se realizaba mediante la función virtual *crearOperadores*, perteneciente a la clase *Individuo*, y en la versión actual se realiza mediante la función virtual *definirOperadores()*, perteneciente a la clase *AlgoritmoGenetico*. En ambos casos la función virtual puede sobrecargarse para definir los operadores que se prefieran y en caso de no definirse, se toman los operadores por defecto.

La diferencia radica en que en la versión 1.0, si no se definían los operadores de mutación y cruce para cada gen, el programa presentaba errores. UNGenético 2.0 permite cambiar todos o algunos de los operadores y se realiza una revisión y corrección de su definición para evitar errores en el programa del usuario. Adicionalmente, se crearon MACROS que facilitan la definición de los operadores a utilizar. Estas MACROS se describen en la sección 2.6 de este capítulo.

2.3.5. Definición de parámetros del algoritmo genético

En la presente versión se provee la función *inicializarParametros()*, la cual puede ser sobrecargada por el usuario para cambiar los parámetros que se han establecido por defecto para la clase *AlgoritmoGenetico*. Esto incluye el tamaño de la población, número máximo de generaciones, indicadores, etc. Estos parámetros son explicados en la sección 4.2.3.1. La versión anterior no suministraba esta función, pero permitía variar los parámetros desde fuera de la clase, ya que todos eran públicos.

Como complemento y según sus requerimientos, el usuario puede adicionar la funcionalidad que desee para mostrar, guardar, modificar resultados, o para incluir su *AlgoritmoGenetico* en una aplicación de gran escala. La forma de definir y utilizar la clase derivada de *AlgoritmoGenetico*

y sus funciones, se explica con detalle en el capítulo 4.

En esta instancia cabe aclarar que, en razón a que ya no es necesario crear una clase derivada de *Individuo*, se incurriría en un error al tratar de definir las funciones que en UNGenético 1.0 se debían implementar obligatoriamente para la clase derivada de *Individuo*.

2.4. Implementación de un nuevo tipo de Gen: Arreglo de genes de tamaño variable

En UNGenético 1.0, se habían implementado ya tres tipos de genes: *GenBool, GenEntero y GenReal*, los cuales manejan datos de tipo *bool, long y double*, respectivamente. Una de las nuevas características de la librería, consiste en la adición de un nuevo tipo de gen que maneja un arreglo de tamaño variable, el cual contiene genes de los tipos preestablecidos. De igual forma se implementaron sus respectivos operadores de mutación y cruce, y se definieron tipos de datos equivalentes, útiles para simplificar el uso de este gen.

La clase implementada se denomina *GenArreglo<G,T>*, la cual almacena y encapsula de forma flexible un arreglo de genes de tamaño variable. *GenArreglo<G,T>*, derivada de la clase *Gen*, es una clase genérica, lo cual implica que puede almacenar genes de cualquier tipo, con la restricción de que todos los genes del arreglo sean del mismo tipo, y lógicamente, derivados de la clase *Gen*.

La implementación de esta clase es compleja, pero su uso es sencillo y similar al de los genes definidos con anterioridad. El tamaño del arreglo es variable dentro de un rango definido por el usuario. De igual manera, los límites de los valores que pueden tomar los genes componentes del arreglo son definidos por el usuario, con la restricción de que todos los genes tendrán los mismos límites.

Su definición se realizó usando plantillas (*templates* en C++), que permiten la implementación de clases genéricas. De ahí la inclusión de los símbolos *<G,T>* en el nombre de la clase. Al crear un objeto de tipo *GenArreglo<G,T>*, deben definirse explícitamente los tipos de datos *G y T*, donde *G* indica el tipo de genes que contiene el arreglo y *T* indica el tipo de datos que

maneja cada gen de tipo *G*. Por ejemplo, puede crearse una instancia de un arreglo de genes enteros declarando un objeto de tipo *GenArreglo<GenEntero, long>*.

La clase *GenArreglo<G,T>*,ofrece la funcionalidad necesaria para realizar operaciones sobre los genes almacenados. La explicación detallada de su implementación puede verse en la documentación de UNGenético 2.0.

Igualmente, se implementaron los operadores de mutación y de cruce respectivos para este tipo de gen: *OperadorMutacionArreglo<G,T> y OperadorCruceArreglo<G,T>*, los cuales se describen brevemente en la sección 2.5.2 de este capítulo, y se explican con detalle en la documentación de UNGenético 2.0.

Para brindar facilidad en el uso de *GenArreglo<G,T>* y sus respectivos operadores de mutación y cruce, se definieron tipos de datos sinónimos, que se muestran en la sección 2.7 de este capítulo.

2.5. Modificaciones realizadas en operadores

En términos de los operadores de UNGenético 1.0, los cambios realizados se pueden clasificar en dos aspectos:

- Optimización y corrección de los operadores existentes.
- Creación de nuevos operadores.

2.5.1. Optimización y corrección de los operadores existentes

Los operadores ya incluidos en la versión 1.0 fueron mejorados en los siguientes aspectos.

- Adición de funcionalidad para variar los parámetros del operador.
- Forma de acceder al objeto sobre el que operan.

- Corrección de errores y optimización de funciones.

2.5.1.1. Adición de funcionalidad para variar los parámetros del operador. En todos los operadores que tienen parámetros de libre elección, se creó la opción de variarlos en el momento de su construcción cambiando los valores por defecto, o en algún momento durante el algoritmo genético mediante funciones de lectura/escritura con comprobación de límites.

2.5.1.2. Forma de acceder al objeto sobre el que opera. En esta versión los operadores de reproducción, selección, probabilidad y parejas, reciben el objeto sobre el que operan, es decir, un objeto de la clase *Poblacion*, como argumento en su función de operación. En la versión 1.0, éste no se recibía como argumento de la función, sino que se guardaba su apuntador como miembro de la clase del operador desde el momento de su construcción, de manera que sólo se podía operar sobre un objeto del que se podía perder la referencia en caso que éste cambiara de dirección en memoria, a menos que se hiciera el cambio de referencia explícitamente antes de invocar la función de operación.

Los operadores de mutación y cruce no fueron modificados en este aspecto, pues ya recibían como argumentos los objetos a operar; únicamente se eliminó el miembro que hacía referencia al objeto *AlgoritmoGenetico*, el cual se consideró innecesario.

2.5.1.3. Corrección de errores y optimización de funciones. Algunos de los operadores presentes en UNGenético 1.0 presentaban errores de implementación, razón por la cual no trabajaban adecuadamente. También se hicieron modificaciones para optimizar la realización de las operaciones. A continuación se describen los errores y se expone la forma como fueron solucionados.

- **Operador de Probabilidad Proporcional.** Este operador presentaba una desventaja importante, en razón a que la probabilidad de supervivencia asignada a los individuos tomaba valores incoherentes en casos en los cuales la función objetivo podía tomar valores positivos y negativos para diferentes individuos dentro de la misma generación. El error radicaba en que si la evaluación de la función objetivo de un individuo tenía valor negativo, se asignaba a éste una probabilidad de supervivencia con valor negativo.

Una desventaja adicional se presentaba si se estaba minimizando la función objetivo, pues se asumía que el mejor valor que podía alcanzar la función objetivo era cero, asignándole la mayor probabilidad de supervivencia a los individuos que la tuvieran, y probabilidad de supervivencia igual a cero en otro caso.

Estos inconvenientes fueron solucionados restando a todos los individuos, el menor valor de la función objetivo de la población disminuido en 1.0. De esta manera se asegura que al asignar la probabilidad, se hará proporcionalmente a valores mayores o iguales que 1.0, y que la suma de las probabilidades de supervivencia de todos los individuos será 1.0.

Una optimización adicional realizada a este operador fue la eliminación de la llamada a la función *ordenar()* del objeto *Poblacion*, en razón a que para asignar este tipo de probabilidad no se requiere ordenar los individuos de mejor a peor, generando un gran consumo innecesario de recursos y por consiguiente, de tiempo.

- **Operadores de Reproducción.** En razón a que algunos operadores de cruce requieren que se conozca de antemano cuál es el individuo padre con mejor función de evaluación, se realizó una modificación en todos los operadores de reproducción, la cual consiste en hallar al padre con mejor función de evaluación y pasar sus genes como primer parámetro al método *cruzar* del operador de cruce respectivo. De esta manera todos los operadores de cruce sabrán cuál de los genes recibidos corresponde al del mejor individuo entre los que se está realizando el cruce.

- **Operador de Reproducción Cruce Simple.** En la versión anterior llevaba el nombre de *OperadorReproduccionCrucePlano*. Sólo se modificó el nombre a la clase que representa este operador, ya que su funcionamiento realmente realiza un cruce simple de acuerdo con su definición en la teoría de algoritmos genéticos (1.3.3). El nuevo nombre es *OperadorReproduccionCruceSimple*.

- **Operador de Cruce Bool Discreto.** En la versión 1.0 llevaba el nombre de *OperadorCruceBoolPlano*. Sólo se modificó el nombre a la clase que representa este operador, ya que su funcionamiento representa de mejor forma un cruce discreto que un cruce plano, de acuerdo con su definición en la teoría de algoritmos genéticos (1.3.3.1), pues cada gen hijo, toma aleatoriamente el valor de uno de los genes padres. El nuevo nombre es *OperadorCruceBoolDiscreto*.

- **Operador de Mutación Bool Uniforme.** Se corrigió su implementación. En UN-

GENÉTICO 1.0 se asignaba al gen en proceso de mutación un valor aleatorio, pero la definición indica que su nuevo valor, debe ser la negación del valor original.

- **Operador de Mutación Entero No Uniforme y Operador de Mutación Real No Uniforme.** Presentaban un error en el cálculo del nuevo valor del gen, ya que efectuaban división de enteros en lugar de división de punto flotante entre los parámetros *t y T*. Ver sección 1.4.2.2.

- **Operador de Mutación Entero Muhlenbein y Operador de Mutación Real Muhlenbein.** En la versión anterior llevaban el nombre de *OperadorMutacionEnteroBGA y OperadorMutacionRealBGA*, respectivamente. Sólo se modificaron los nombres a las clases que representan estos operadores por *OperadorMutacionEnteroMuhlenbein y OperadorMutacionRealMuhlenbein*, respectivamente.

2.5.2. Creación de nuevos operadores

Una de las motivaciones más importantes para la realización de esta versión de UNGENÉTICO fue precisamente la ampliación de la gama de operadores disponibles, ya que se logra incrementar las posibilidades en la obtención de una respuesta óptima.

Se debe anotar que UNGENÉTICO es bastante flexible en la creación y utilización de operadores personalizados, ya que provee clases abstractas que sirven como base para la derivación de nuevos operadores. Haciendo uso de estas clases base, se derivaron operadores de probabilidad, de asignación de parejas, de reproducción, de cruce y de mutación.

Adicionalmente, se crearon dos nuevos operadores base y sus respectivos operadores derivados. Se trata de la conversión en operadores, de los criterios de adaptación y finalización del algoritmo genético. A continuación se describen los operadores implementados.

2.5.2.1. Nuevos operadores base. Se definieron dos nuevas clases abstractas, las cuales sirven de base para operadores de adaptación y finalización del algoritmo genético. Llevan los nombres de *OperadorAdaptacion y OperadorFinalizacion* respectivamente. Anteriormente no eran operadores sino funciones pertenecientes a la clase *AlgoritmoGenetico*.

En UNGenético 1.0 ya se tenía previsto el uso de criterios de adaptación, pero no se había implementado ninguno como tal. Cabe aclarar que se había implementado la estrategia de elitismo sin incluirla como criterio de adaptación, sino adicionándola como una función extra y opcional en el proceso de optimización. Como se explica más adelante, en esta versión se adoptó el elitismo como un operador de adaptación. En cuanto a criterios de finalización, únicamente se había implementado uno, basado en el número máximo de generaciones, el cual en la actualidad es siempre evaluado, antes de invocar los nuevos criterios implementados por operadores de finalización.

Ambos criterios, adaptación y finalización, eran tratados como funciones dentro de la clase *AlgoritmoGenetico*, pero se considera que al convertirlos en operadores, se obtiene mayor flexibilidad, ya que se brinda al usuario la posibilidad de crear y utilizar sus propios criterios sin entrar a modificar el código fuente de la librería. Una ventaja adicional se obtiene al poder incluir varios criterios de adaptación o finalización simultáneamente dentro del algoritmo.

Actualmente, las funciones *adaptacion() y finalizar()*, pertenecientes a la clase *AlgorimoGenetico*, invocan, respectivamente, los operadores de adaptación y finalización incluidos por el usuario en su algoritmo genético.

2.5.2.2. Nuevos operadores derivados de operadores base. A continuación se enumeran y se describen brevemente los operadores implementados para UNGenético 2.0. La explicación detallada de los mismos, puede verse en la documentación de UNGenético 2.0.

- **Operador de Probabilidad Homogénea.** Asigna la misma probabilidad de supervivencia a todos los individuos de la población. La clase creada se denomina *OperadorProbabilidadHomogenea*, derivada de *OperadorProbabilidad*.
- **Operador de Probabilidad Lineal.** Asigna a los individuos de la población una probabilidad de supervivencia linealmente decreciente según su función objetivo. La pendiente se determina con un parámetro que puede ser elegido por el usuario. La clase creada se denomina *OperadorProbabilidadLineal*, derivada de *OperadorProbabilidad*.
- **Operador de Parejas Adyacentes.** Asigna parejas de individuos para el cruce, dependiendo de su ubicación dentro de la población. Se forman parejas tomando el primer

individuo con el segundo, el tercero con el cuarto, y así sucesivamente, hasta asignar parejas a todos los individuos de la población. La clase creada se denomina *OperadorParejasAdyacentes*, derivada de *OperadorParejas*.

- **Operador de Parejas Extremos.** Asigna parejas de individuos para el cruce, dependiendo de su ubicación dentro de la población. Se forman parejas tomando del primer individuo con el último, el segundo con el penúltimo, y así sucesivamente, hasta asignar parejas a todos los individuos de la población. La clase creada se denomina *OperadorParejasExtremos*, derivada de *OperadorParejas*.

- **Operador de Reproducción Mejor Padre Mejor Hijo.** Se crean dos individuos hijos, producto del cruce de cada pareja de individuos asignada por el operador de parejas respectivo. El mejor individuo hijo de cada pareja, remplaza a su peor padre en su posición dentro de la población. La clase creada se denomina *OperadorReproduccionMejorPadreMejorHijo*, derivada de *OperadorReproduccion*.

- **Operador de Reproducción Mejores Entre Padres e Hijos.** Se crean dos individuos hijos, producto del cruce entre cada pareja de individuos asignada por el operador de parejas respectivo. De cada pareja y sus dos hijos, se seleccionan los dos mejores y pasan a remplazar en la población a los dos individuos padres. La clase creada se denomina *OperadorReproduccionMejoresEntrePadresEHijos*, derivada de *OperadorReproduccion*.

- **Operador de Cruce Entero Discreto y Operador de Cruce Real Discreto.** Cada gen hijo producido por un cruce discreto, toma aleatoriamente el valor de uno de los genes padres. Las clases correspondientes se denominan *OperadorCruceEnteroDiscreto y OperadorCruceRealDiscreto*, derivadas de *OperadorCruce*.

- **Operador de Cruce Entero BLX y Operador de Cruce Real BLX.** Cada gen hijo toma un valor aleatorio, contenido en un rango limitado por el intervalo de explotación extendido superior e inferiormente en un factor alfa, parámetro que puede ser elegido por el usuario. Las clases correspondientes se denominan *OperadorCruceEnteroBLX y OperadorCruceRealBLX*, derivadas de *OperadorCruce*.

- **Operador de Cruce Entero Intermedio Extendido y Operador de Cruce Real Intermedio Extendido.** Equivalente al cruce BLX, aplicando un factor $alfa = 0.25$. Las clases correspondientes se denominan *OperadorCruceEnteroIntermedioExtendido y OperadorCruceRealIntermedioExtendido*, derivadas de *OperadorCruce*.

- **Operador de Cruce Entero Heurístico y Operador de Cruce Real Heurístico.** Cada gen hijo generado toma un valor aleatorio, en el intervalo de exploración cercano al valor del gen padre perteneciente al individuo con mejor función objetivo. Las clases correspondientes se denominan *OperadorCruceEnteroHeuristico y OperadorCruceRealHeuristico*, derivadas de *OperadorCruce*.

- **Operador de Cruce Entero Lineal y Operador de Cruce Real Lineal.** Se producen tres tipos de genes hijos, dependiendo del número de hijos solicitado por el operador de reproducción. El valor del primer hijo es un promedio ponderado de los valores de los padres, dando un factor de 1.5 al valor del gen padre perteneciente al individuo con mejor función objetivo y un factor de -0.5 al gen perteneciente al peor padre. El valor del segundo es el promedio de los valores de los padres, y el valor del tercero es un promedio ponderado de los valores de los padres, dando un factor de -0.5 al valor del gen padre perteneciente al individuo con mejor función objetivo y un factor de 1.5 al gen perteneciente al peor padre. Las clases correspondientes se denominan *OperadorCruceEnteroLineal y OperadorCruceRealLineal*, derivadas de *OperadorCruce*.

- **Operador de Cruce Entero Lineal BGA y Operador de Cruce Real Lineal BGA.** Cada gen hijo toma un valor aleatorio en los intervalos de explotación o exploración cercanos al valor del gen perteneciente al mejor individuo padre, con probabilidad mayor de visitar el intervalo de exploración que el de explotación. Las clases correspondientes se denominan **OperadorCruceEnteroLinealBGA y OperadorCruceRealLinealBGA**, derivadas de **OperadorCruce**.

- **Operador de Cruce para arreglos de genes de tamaño variable.** La clase creada se denomina *OperadorCruceArreglo<G,T>*, la cual fue implementada para efectuar el cruce entre dos genes de tipo *GenArreglo<G,T>*, donde *G* indica el tipo de genes que contiene el arreglo y *T* indica el tipo de datos que maneja el gen de tipo *G*. Cada gen hijo se obtiene de la siguiente manera:

  El tamaño del nuevo arreglo se determina por medio del cruce de los tamaños de los genes padres usando el operador de cruce entero por defecto.

  Cada uno de los genes que hacen parte del nuevo arreglo, se obtiene cruzando los genes correspondientes a la misma posición en el arreglo de los padres, usando el operador de cruce por defecto para el tipo de genes *G*. Si el nuevo tamaño es mayor que el de alguno

de los padres, los genes faltantes se copian del otro padre. Si el nuevo tamaño es mayor que el de ambos padres, se crean los genes faltantes aleatoriamente.

- **Operador de Mutación para arreglos de genes de tamaño variable.** La clase creada se denomina *OperadorMutacionArreglo<G,T>*, la cual fue implementada para efectuar la mutación de genes de tipo *GenArreglo<G,T>*, donde *G* indica el tipo de genes que contiene el arreglo y *T* indica el tipo de datos que maneja el gen de tipo *G*.

  El gen cambia de tamaño con una probabilidad igual a la probabilidad de mutación establecida para el operador. El nuevo tamaño toma un valor aleatorio en el intervalo establecido por el usuario. Si el nuevo tamaño es mayor que el anterior, los nuevos genes toman valores aleatorios; si es menor, se eliminan los genes sobrantes.

  Independientemente del cambio tamaño, todos los genes del arreglo son mutados por el operador de mutación por defecto respectivo para el tipo de gen G, usando la probabilidad de mutación establecida por el usuario.

- **Operador de Adaptación Elitismo.** Como se mencionó en la sección 2.5.2.1, la estrategia de elitismo existente en UNGenético 1.0, fue convertida en operador de adaptación. Anteriormente se trabajaba con funciones y miembros pertenecientes a las clases *AlgoritmoGenetico y Poblacion*, los cuales fueron eliminados.

  La clase implementada se denomina *OperadorAdaptacionElitismo*, derivada de *OperadorAdaptacion*, y su función de operación remplaza un individuo aleatorio dentro de la generación actual del algoritmo genético, por una copia del mejor individuo de todas las generaciones anteriores.

- **Operador de Adaptación Número de Individuos.** Varía el número de individuos de la población de forma linealmente decreciente entre los valores inicial y final, establecidos por el usuario. La clase creada se denomina *OperadorAdaptacionNumIndividuos*, derivada de *OperadorAdaptacion*.

- **Operador de Adaptación Probabilidad de Mutación.** Varía la probabilidad de mutación de todos los operadores de probabilidad incluidos en el algoritmo genético. La variación de la probabilidad puede ser de dos formas: por escalones, dependiendo de la medida offline del algoritmo genético, o decreciente exponencialmente. La forma de la variación es definida por el usuario en el momento de adicionar el operador a su algoritmo. Los parámetros con que se realiza la variación pueden modificarse desde su construcción, o

en algún momento del algoritmo. La clase creada se denomina *OperadorAdaptacionProbMutacion*, derivada de *OperadorAdaptacion*, y se explica con detalle en la documentación de UNGenético 2.0.

- **Operador de Finalización Offline.** Ordena finalizar el algoritmo genético cuando después de cierto número de iteraciones no se obtiene una variación significativa de la medida offline. El número de iteraciones límite y el factor mínimo de variación pueden ser ajustados por el usuario. La clase creada se denomina *OperadorfinalizacionOffline*, derivada de *OperadorFinalización*.

- **Operador de Finalización Online.** Ordena finalizar el algoritmo genético cuando después de cierto número de iteraciones no se obtiene una variación significativa de la medida online. El número de iteraciones límite y el factor mínimo de variación pueden ser ajustados por el usuario. La clase creada se denomina *OperadorfinalizacionOnline*, derivada de *OperadorFinalización*.

2.6. Creación de MACROS para la definición del algoritmo

Uno de los objetivos de UNGenético 2.0 es brindar al usuario una mayor facilidad en la implementación de su algoritmo genético. Para tal fin, se crearon algunas macro-instrucciones que disminuyen en gran medida la cantidad y complejidad de instrucciones a utilizar en la definición del modelo a optimizar.

Se definieron macros para las siguientes tareas:

- Declaración de la clase derivada de *AlgoritmoGenetico*.
- Declaración de la clase para la aplicación.
- Adición de genes al genoma de un individuo.
- Definición de operadores a utilizar en el algoritmo genético.

Cabe aclarar, que el uso de estas MACROS no es obligatorio, pero puede ser de gran utilidad. A continuación se describen brevemente. Para un mejor entendimiento de su uso puede verse el capítulo 4.

2.6.1. Declaración de la clase derivada de AlgoritmoGenetico

La declaración de la clase que el usuario debe derivar de la clase AlgoritmoGenetico para poder implementar su algoritmo genético, se simplifica utilizando dos macros:

- *DECLARAR_ALGORITMO(nombreAlgoritmo)*
- *FIN_DECLARAR_ALGORITMO*

En el intermedio de éstas, deben declararse las variables y métodos que se requieran. Al utilizarlas, implícitamente se está declarando la clase, su constructor por defecto, y la sobrecarga de las funciones puramente virtuales, las cuales deben implementarse posteriormente.

2.6.2. Declaración de la clase para la aplicación gráfica (usando wxWindows)

En caso de requerir el uso de un entorno gráfico para la aplicación, UNGenético 2.0 incorpora la posibilidad de hacerlo mediante la librería de libre distribución wxWindows (La licencia de wxWindows se encuentra en el anexo H). La utilización de esta nueva característica se explica en el capítulo 4. En tal caso, se debe crear una clase para la aplicación, lo cual se simplifica usando la MACRO:

- *DECLARAR_APLICACION(nombreAplicacion)*

Con ésta única instrucción, implícitamente se está declarando la clase para la aplicación, el apuntador a la ventana principal que se utilizará y la función *OnInit()*, la cual debe implementarse posteriormente y es la que recibe el control del programa una vez se ha creado la aplicación (similar a la función *main()* en programas de consola).

2.6.3. Adición de genes al genoma de un individuo

Como se explicó en la sección 2.3.2, referente a la codificación del modelo, en la función *codificacion(pIndividuo, estado)*, perteneciente a la clase *AlgoritmoGenetico*, deben definirse los tres estados de codificación del individuo. Desde este punto de vista, la tarea de adicionar un gen al individuo puede ser algo engorrosa, y tal vez no muy clara para un usuario inexperto, pues para cada gen del individuo deben definirse sus estados de creación, codificación y decodificación.

Como alternativa para la realización de esta tarea, se definieron MACROS que adicionan un gen al individuo, proporcionando sus tres estados de codificación en una sola instrucción. La MACRO a utilizar para adicionar cada gen, depende del tipo de gen, y puede seleccionarse de las siguientes posibilidades:

- *ADICIONAR_GENBOOL(...)*
- *ADICIONAR_GENENTERO(...)*
- *ADICIONAR_GENREAL(...)*
- *ADICIONAR_GENARREGLO_BOOL(...)*
- *ADICIONAR_GENARREGLO_ENTERO(...)*
- *ADICIONAR_GENARREGLO_REAL(...)*

Los parámetros que toman, expresados por ahora como (...), involucran al individuo que se está codificando, la posición del gen respectivo dentro del individuo, la variable asociada al gen, la cual debe pertenecer a la clase derivada de *AlgoritmoGenetico*, y el rango y valor inicial de esta variable. La forma de uso de estas MACROS puede verse la sección 4.2.3.3.

2.6.4. Definición de operadores a utilizar en el algoritmo genético

Como se explicó en la sección 2.3.4, el uso de operadores diferentes a los establecidos por defecto es opcional y su definición se realiza en la función virtual *definirOperadores()* de la clase derivada

de *AlgoritmoGenetico*. En tal caso, la definición de los operadores a utilizar puede no ser muy clara para un usuario inexperto, por lo cual se definieron las siguientes MACROS:

- *DEFINIR_OPERADOR_PROBABILIDAD(tipoOperador)*
- *DEFINIR_OPERADOR_SELECCION(tipoOperador)*
- *DEFINIR_OPERADOR_PAREJAS(tipoOperador)*
- *DEFINIR_OPERADOR_REPRODUCCION(tipoOperador)*
- *ADICIONAR_OPERADOR_ADAPTACION(tipoOperador)*
- *ADICIONAR_OPERADOR_FINALIZACION(tipoOperador)*
- *ADICIONAR_OPERADOR_MUTACION(tipoOperador)*
- *ADICIONAR_OPERADOR_CRUCE(tipoOperador)*

En estas macro-instrucciones, el parámetro *tipoOperador* especifica el tipo específico de operador a utilizar en cada caso y sus parámetros iniciales. Obsérvese en los nombres de estas MACROS , que algunas tienen el prefijo *DEFINIR_* y otras el prefijo *ADICIONAR_*. La diferencia radica en que las primeras hacen referencia a operadores de cuyo tipo solo se debe definir uno para operar en el algoritmo genético, mientras que las restantes se refieren a operadores que pueden actuar simultáneamente en el algoritmo, como los operadores de adaptación y finalización, o los operadores que corresponden a cada gen del modelo (operadores de cruce y mutación).

2.7. Definición de tipos de datos equivalentes

En vista de la dificultad que puede acarrear el uso de plantillas (*templates* de C++) en la implementación de UNGenético 2.0 para un usuario no experimentado en programación en lenguaje C++, se definieron tipos de datos equivalentes a los necesarios para crear instancias de clases genéricas. Haciendo uso de dichos tipos, el usuario no está obligado a saber qué son *templates* de C++ para utilizar las clases genéricas.

En UNGenético existen cuatro clases genéricas diferentes que usan plantillas (*templates*):

- *Arreglo<T>*
- *GenArreglo<G,T>*
- *OperadorMutacionArreglo<G,T>*
- *OperadorCruceArreglo<G,T>*

Para facilitar el uso de arreglos de datos booleanos, enteros y reales, se definieron respectivamente los siguientes tipos:

- *ArregloBool*, equivalente a *Arreglo<bool>*
- *ArregloEntero*, equivalente a *Arreglo<long>*
- *ArregloReal*, equivalente a *Arreglo<double>*

Para facilitar el uso de GenArreglo con genes de tipo *GenBool, GenEntero y GenReal*, se definieron respectivamente los siguientes tipos:

- *GenArregloEntero*, equivalente a *GenArreglo<GenEntero, long>*
- *GenArregloReal*, equivalente a *GenArreglo<GenReal, double>*
- *GenArregloBool*, equivalente a *GenArreglo<GenBool,bool>*

Para facilitar el uso de los operadores de cruce y mutación para *GenArreglo* con genes de tipo *GenBool, GenEntero y GenReal*, se definieron respectivamente los siguientes tipos:

- *OperadorMutacionArregloBool*, equivalente a *OperadorMutacionArreglo<GenBool,bool>*
- *OperadorMutacionArregloEntero*, equivalente a *OperadorMutacionArreglo<GenEntero,long>*

- *OperadorMutacionArregloReal*, equivalente a *OperadorMutacionArreglo<GenReal,double>*
- *OperadorCruceArregloBool*, equivalente a *OperadorCruceArreglo<GenBool,bool>*
- *OperadorCruceArregloEntero*, equivalente a *OperadorCruceArreglo<GenEntero,long>*
- *OperadorCruceArregloReal*, equivalente a *OperadorCruceArreglo<GenReal,double>*

2.8. Definición de constantes

Se definen dos tipos de constantes diferentes:

2.8.1. Constantes de preprocesador

Su fin es realizar compilación condicional. La única constante de este estilo que utiliza UNGenético 2.0, que debe conocer el usuario, es la constante *USAR_VENTANA*, la cual debe ser definida por el mismo antes de incluir la librería en su proyecto, para indicar que desea hacer uso del entorno gráfico que provee UNGenético 2.0 haciendo uso de wxWindows. El valor que se dé a la constante es ignorado, únicamente se verifica si fue definida o no.

Existen otras constantes de preprocesador definidas internamente, cuyo fin es evitar la compilación de algún archivo de la librería en más de una ocasión.

2.8.2. Constantes definidas en enumeraciones (bloques enum)

Su fin es facilitar la modificación de la librería y proveer una mejor comprensión del significado de ciertos valores usando identificadores. Por ejemplo, es de más fácil comprensión la instrucción

SetPagina(ID_CONSOLA);

que la instrucción

SetPagina(2000);

la cual el usuario debe especificar para poner el foco en la página de salida cuando hace uso del entorno gráfico de wxWindows.

Al modificar la librería no importa el valor que tomen estas constantes, el resultado será el mismo al emplearlas como indicadores.

En UNGenético 2.0 se definieron las siguientes enumeraciones:

```
enum Paginas
{
        ID_CONSOLA=2000,
        ID_GRAFICA,
        ID_OTRA
};


enum CriteriosDeAdaptacion
{
        ADAPTACION_PROBMUTACION_OFFLINE = 1,
        ADAPTACION_PROBMUTACION_EXPONENCIAL
};


enum EstadosIndividuo
{
        ESTADO_CODIFICAR = 1,
        ESTADO_DECODIFICAR,
        ESTADO_CREAR
};


enum menus
{
        MENU_FILE_SAVE=1000,
        MENU_FILE_QUIT,
        MENU_INFO_ABOUT,
        MENU_GRAFICA_FONDO,
        MENU_GRAFICA_GRILLA,
        MENU_GRAFICA_LINEA,
        MENU_GRAFICA_GRAFICAR,
        MENU_CONGELAR,
        NOTEBOOK
 };
```

2.9. Creación de clases genéricas de interfaz gráfica

Con el fin de permitir al usuario visualizar el proceso de ejecución del problema planteado, UNGenético 2.0 incluye una nueva característica que consiste en la creación de dos clases que facilitan el uso de una interfaz gráfica. Estas clases hacen uso de la librería de libre distribución wxWindows , la cual se asume, el usuario debe tener instalada en su computador. La utilización de dichas clases y del entorno gráfico que suministran se explica en el capítulo 4. Las dos nuevas clases se denominan:

- *AGFrame*
- *AGVentana*

2.9.1. Clase AGFrame

La clase *AGFrame* es derivada de la clase *wxFrame* de la librería wxWindows.
Un objeto de esta clase contiene una barra de menú, donde el usuario puede seleccionar varias opciones como guardar los archivos creados, crear gráficas, cambiar los colores de las gráficas, detener y restablecer la visualización de la ejecución del algoritmo y obtener información de la aplicación. También contiene ventanas adicionales en las que aparecen el proceso del algoritmo y las gráficas con las medidas de desempeño del mismo.

2.9.2. Clase AGVentana

La clase *AGVentana* es derivada de la clase *wxScrolledWindow* de la librería wxWindows.
Al establecer la utilización del ambiente gráfico se crean dos tipos de ventana, una que contiene un control de texto en el que aparece la ejecución del algoritmo, y otras que contienen las gráficas de desempeño del algoritmo genético. Sin embargo, el usuario puede crear una clase derivada de *AGVentana* y adicionar cualquier control que pertenezca a la librería wxWindows.

2.10. Creación de un Wizard para UNGenético 2.0

En razón a que la utilización de la librería UNGenético requiere que el usuario tenga suficientes conocimientos en el lenguaje de programación C++, y que existen usuarios que no tienen dichos conocimientos, se desarrolló una aplicación independiente llamada *UNGenético Wizard*, la cual crea un archivo de código fuente de C++ (*.cpp*) que incluye la estructura básica que debe tener un programa basado en UNGenético 2.0.

Con *UNGenético Wizard* el usuario solamente debe seleccionar el número y tipo de variables, los límites de las mismas, los operadores y los parámetros del algoritmo genético. Una explicación detallada sobre la utilización de *UNGenético Wizard* se encuentra en la sección 4.4.

2.11. Optimización general de las funciones de la librería

En general se optimizaron las funciones de UNGenético en aspectos como:

Uso de funciones en línea (inline) de C++: Las funciones con pocas instrucciones se hicieron *inline*, con el fin de reducir el tiempo de procesamiento. Estas producen un ejecutable de mayor tamaño, pero para la librería se consideró mas importante el factor tiempo que el factor tamaño.

Prevención de desborde de límites en arreglos: Se incluyeron restricciones y precauciones necesarias para no incurrir en errores de programas basados en la librería, por acceso a posiciones inexistentes en arreglos de C++.

Prevención de división por cero: Se incluyeron las restricciones necesarias para evitar divisiones por cero, las cuales no generan errores de aserción, pero pasan inadvertidas generando datos erróneos en el programa.

Corrección de divisiones entre datos enteros: Al realizar una división entre dos datos enteros en C++, el resultado es un dato entero, por lo cual sus decimales son recortados, produciendo datos diferentes a los requeridos. Por esta razón las divisiones de este tipo fueron sustituidas por división de punto flotante en los casos que podían producir errores.

Redondeo de números: Se creó una función global para realizar el redondeo de números. Esto fue necesario, puesto que en la versión anterior no se realizaba redondeo, sino que se utilizaba la conversión implícita del lenguaje C++, la cual simplemente recorta los decimales del dato real, proporcionando valores diferentes a los requeridos. La función creada se denomina *redondear()*.

Asignación de números aleatorios enteros: La asignación de números aleatorios enteros se realizaba con la misma operación que con datos reales, lo cual ocasionaba el mismo problema de conversión mencionado en el ítem anterior. Esto se corrigió utilizando la función *redondear()*, aplicada al valor aleatorio producido.

Reducción de ciclos iterativos: Con el fin de disminuir el tiempo de procesamiento, se procuró disminuir la cantidad de instrucciones incluidas en ciclos iterativos, cambiando en ocasiones la lógica utilizada en los procesos pero produciendo los mismos resultados.

Uso del tipo bool en lugar de int en GenBool: El uso de datos de tipo *bool* con valores *true* o *false* es equivalente al uso de datos de tipo *int* con valores 1 o 0. La diferencia de la cual se pretende tomar ventaja en esta modificación es la disminución del uso de memoria, pues en algunos compiladores como Visual C++ (a partir de su versión 5.0), el tipo *bool* ocupa 1 byte mientras el tipo *int* ocupa 4 bytes. En compiladores donde esto no aplica, no hay problemas de compatibilidad, pues la palabra clave *bool* es tomada como *int*.

Capítulo 3

DOCUMENTACIÓN DE UNGenético 2.0

La librería UNGenético 2.0 está conformada por un conjunto amplio de clases, cada una con miembros propios que cumplen una función específica en la ejecución del algoritmo genético. Al realizar una documentación para la librería se busca principalmente dar una orientación básica al usuario para que éste logre adquirir una visión clara de la estructura jerárquica y funcionalidad de los elementos que la componen.

Como soporte básico para la elaboración de la documentación de UNGenético 2.0 se utilizó Doxygen[1], una herramienta computacional de libre distribución especializada en la documentación de proyectos escritos en lenguaje C++. Con ayuda de esta herramienta se logró desarrollar una documentación de la librería y un manual de usuario que brinda un soporte importante en el momento de utilizar UNGenético 2.0 en un proyecto de optimización. El compendio de toda la documentación generada para la librería se encuentra en el CD que acompaña a este documento en la carpeta *Documentación*.

[1]Doxygen versión 1.3.6: Generador de documentación para proyectos en C++, su distribución es libre y se puede descargar en el sitio www.doxygen.org.

3.1. Documentación de la librería

La documentación de clases y otros elementos que hacen parte de UNGenético 2.0 se realizó procurando establecer una base explicativa clara acerca de la labor que desempeña cada uno de los elementos que conforman la librería y la forma en que éstos interactúan para desarrollar el algoritmo genético. Con este propósito, se utilizó ampliamente la herramienta Doxygen con la que fué posible desarrollar una documentación con las siguientes características:

- En primera instancia, muestra una descripción de UNGenético 2.0 que brinda al usuario una idea general de la librería, sus caraterísticas sobresalientes, utilidades y formas de utilización.
- Contiene listas completas de las clases que componen a UNGenético 2.0. La primera de ellas está organizada de manera tal que permite visualizar la estructura jerárquica que cumple cada una de las clases, mientras que la segunda lista organiza la totalidad de las clases de manera alfabética y presenta una breve descripción de cada una. Cada ítem presente en estas listas contiene un vínculo de enlace al sitio principal de la clase donde se describe detalladamente.
- Cada clase de la librería cuenta con un sitio principal, en el que se exponen organizadamente algunas propiedades de la clase que apoyan la labor explicativa tales como:
    1. *Diagrama de herencias:* muestra de qué clase se deriva y qué clases se derivan de la clase actual cuando se dé alguno de estos casos.
    2. *Lista de todos los miembros:* enlace a una lista alfabética que contiene la totalidad de los miembros de la clase, diferenciando en cada uno de ellos su nivel de acceso como público, privado o protegido. Cada ítem contiene enlaces a los lugares donde se documenta cada miembro.
    3. *Descripción detallada:* definición general de la clase que indica su aplicación en la librería y sus características destacadas.
    4. *Clasificación de los miembros:* lista donde se clasifican los miembros de la clase de acuerdo a su nivel de acceso. Cada uno cuenta con una breve descripción.
    5. *Documentación de los miembros:* los miembros de las clases tales como constructores, destructores, métodos, variables, enumeraciones, apuntadores y otros, cuentan

con un espacio donde se encuentra una explicación en detalle acerca de sus características sobresalientes. Para los métodos propios de las clases, también están detallados los parámetros de entrada y los valores que retornan.

- La documentación muestra además una lista con los nombres de los archivos que hacen parte de la librería donde están implementadas las clases y demás elementos que conforman a UNGenético 2.0.

- Presenta una lista general en orden alfabético de todos los miembros de las clases que conforman a UNGenético 2.0. Esta lista puede clasificarse de acuerdo al tipo de elemento como funciones, datos, enumeraciones y demás tipos presentes en la librería. En cada ítem de la lista se especifica a qué clase o clases pertenece el objeto.

- Además de contener una documentación de las clases, posee un lista con breves descripciones de los distintos elementos que hacen parte de UNGenético 2.0 que no pertenecen a una clase en particular como definiciones, enumeraciones y funciones globales.

- Como complemento, se cuenta con vínculos de enlace a sitios de interés para el usuario que facilitan la utilización de la librería.

La documentación de UNGenético 2.0 con las características mencionadas se ha creado en formatos compatibles con diferentes sistemas operativos como documentos html, documentos pdf con vínculos interactivos y archivos de ayuda de Windows. La variedad de formatos representa una ventaja importante para la distribución de la librería ya que brinda mayores oportunidades de divulgación y uso.

3.2. Manual del usuario

Como complemento al trabajo realizado en la documentación, se ha desarrollado un manual de usuario para UNGenético 2.0 que consta de un compendio de documentos cuyo propósito es exponer claramente las distintas formas en que el usuario puede emplear la librería para implementar sus proyectos de optimización.

Es importante destacar que el manual del usuario para UNGenético 2.0 está dirigido a personas con conocimientos tanto en el área de los algoritmos genéticos como en el lenguaje C++. Aunque se procuró que los documentos ilustraran de la forma más clara y sencilla posible los temas tratados, éstos contienen conceptos técnicos que podrían no ser comprendidos enteramente por algunos usuarios.

También cabe destacar que en el manual de usuario se presentan las formas básicas en que se puede utilizar UNGenético 2.0. Sin embargo, dependiendo de las habilidades del usuario en el lenguaje C++, es posible generar nuevas formas de utilización de la librería para proyectos de optimización de mayor complejidad y/o capacidad.

Para esta labor también se utilizó la herramienta Doxygen que posibilitó insertar el manual del usuario en la documentación de la librería con el fin de conformar un documento maestro que contuviera la totalidad de la documentación realizada para UNGenético 2.0. El manual de usuario cuenta con las siguientes características:

- En principio, hace una referencia a las herramientas de soporte necesarias para realizar un proyecto de optimización utilizando UNGenético 2.0. En este espacio se recomiendan algunos productos en los que se han realizado pruebas satisfactorias de la librería junto con algunos consejos útiles para el mismo propósito.
- Seguidamente, el manual del usuario muestra un primer acercamiento a la construcción de un proyecto de optimización en UNGenético 2.0 mediante la ilustración de un ejemplo sencillo que contiene las instrucciones mínimas necesarias para ejecutar el algoritmo genético que encuentra el valor mínimo de una función matemática simple.
- Provee además un tutorial que guía al usuario paso a paso en la realización de proyectos de optimización de mayor complejidad. En este tutorial se desarrolla un ejemplo de optimización con codificación híbrida a medida que se van abordando cada una de sus etapas. Paralelamente, se destacan los segmentos donde se hace necesaria la distinción del uso de consola o ventana como entorno de salida para el proyecto, esta selección corre por cuenta del usuario.
- Muestra la manera de utilizar la aplicación UNGenético Wizard para generar el archivo principal que contiene las instrucciones básicas de un proyecto de optimización

con UNGenético 2.0. En esta sección se desarrolla el mismo ejemplo híbrido usado en el tutorial de construcción manual del proyecto.

- Finaliza exponiendo la forma en que se crea la aplicación donde se presentan el proceso de ejecución y/o los resultados de la optimización. Estos pueden ser visualizados en consola o en un entorno gráfico, de cualquier manera, el manual del usuario expone cómo se pueden crear gráficas representativas de la ejecución del algoritmo.

El manual de usuario para UNGenético 2.0 posee las mismas características funcionales de la documentación, por lo tanto, su uso es sencillo y se adecúa al sistema operativo en que se utilice.

Capítulo 4

UTILIZACIÓN DE UNGenético 2.0

En este capítulo se tratarán los puntos básicos para la construcción de un proyecto de optimización usando UNGenético 2.0. Es importante aclarar que estos proyectos pueden ser construidos para ser visualizados en dos formas distintas: la primera y más sencilla corresponde a la aplicación estándar de los compiladores de lenguaje C++ conocida como *consola*, la segunda corresponde al entorno gráfico ofrecido por la librería wxWindows, donde se pueden crear aplicaciones en diferentes plataformas siempre y cuando se implementen un conjunto de instrucciones propias de esta librería. En las siguientes secciones se explicará el método de construcción para los dos tipos de aplicaciones, cuando así se requiera.

En la sección 4.1 se da un primer acercamiento a la construcción de un proyecto de optimización usando UNGenético 2.0, ésta muestra un ejemplo que contiene las instrucciones mínimas necesarias para optimizar una función simple. En la sección 4.2 se expone cada etapa del proceso de construcción de un proyecto de optimización más complejo, que utiliza codificación híbrida.

4.1. Ejemplo sencillo

Esta sección tiene el propósito de mostrar al usuario las etapas mínimas necesarias para elaborar un proyecto de optimización en UNGenético 2.0, para este fin se ha planteado una función matemática que servirá de modelo para la creación del proyecto.

4.1.1. Función propuesta

Se requiere encontrar el valor mínimo de la función:

$$f(x) = \sin(x) + \sin(4x)$$

Donde, $x \in [0, 10]$.

Como se observa en la figura 4.1, esta función presenta varios mínimos locales en el intervalo indicado. Sin embargo el mínimo global de $f(x)$ se encuentra en el punto *(4.342, -1.928)*.

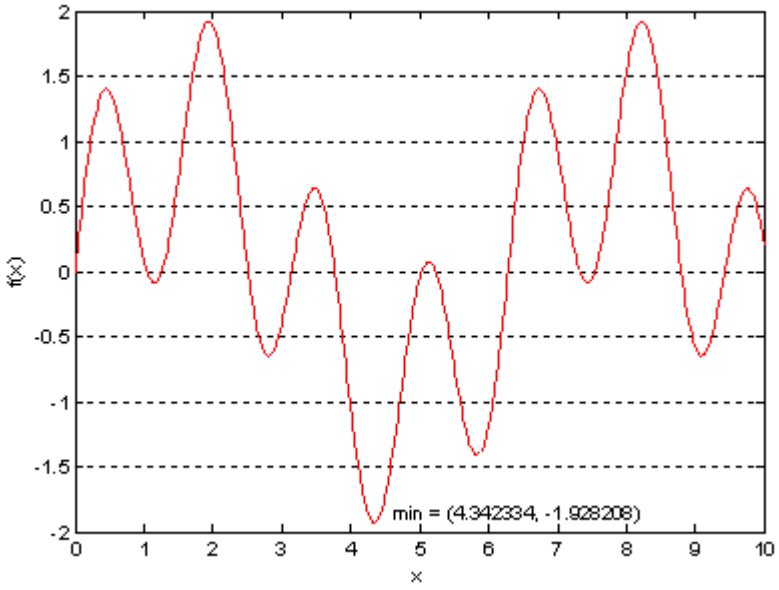


Figura 4.1: Función matemática simple

4.1.2. Construcción del proyecto

Las optimizaciones implementadas con UNGenético 2.0 se ejecutan dentro de un proyecto propio de un compilador de lenguaje C++. Para incluir UNGenético 2.0, en la carpeta del proyecto deben copiarse todos los archivos que hacen parte de la librería, tanto los encabezados como los fuentes (extensiones *.h* y *.cpp*); en caso contrario, se debe especificar en el proyecto la

ruta donde se encuentran dichos archivos. El proyecto debe contener por lo menos un archivo fuente de C++ (extensión *.cpp*) donde se especificarán todas las características del sistema a optimizar, su nombre es de libre escogencia.

En este archivo se fijarán las características propias del modelo a optimizar, para este caso se deben definir:

- El individuo modelo, en este caso estará conformado por un sólo gen de tipo real quien almacenará el valor de la variable x.
- La función objetivo, que representa fielmente la función a minimizar.

A continuación se presenta el código fuente del archivo principal del proyecto:

```
#include "UNGenetico.h"

DECLARAR_ALGORITMO(MiAG)
    double x;
FIN_DECLARAR_ALGORITMO

void MiAG::codificacion(Individuo* modelo, int estado)
{
    ADICIONAR_GENREAL(modelo, 0, x, 0, 10, 0);
}

double MiAG::objetivo()
{
    return sin(x) + sin(4*x);
}

int main()
{
    cout << "EJEMPLO MINIMO DE APLICACION DE UNGenético 2.0\n\n";
    cout << "Encuentra el mínimo de la función\n";
    cout << "f(x)=sin(x)+sin(4x) en el intervalo [0, 10]";
    cout << "\n\nOptimizando...\n\n\n\n";
    cout.flush();
    MiAG AG;
    AG.optimizar();
    cout << "Solución:";
    cout << "\n\nF(Xmin)= " << AG.m_pMejorEnLaHistoria->objetivo(true);
    cout << " Xmin= "<< AG.x << endl;
    return 0;
}
```

El anterior fragmento resume las condiciones mínimas que debe contener el archivo, éstas son:

- Al iniciar, es necesario incluir el archivo *UNGenetico.h* encargado de vincular a los demás archivos de la librería.
- Se debe declarar una clase derivada de *AlgortimoGenetico* que contiene todas las variables y métodos que requiere la optimización en particular. Las macros *DECLARAR_ALGORITMO(NombreAlgoritmo)* y *FIN_DECLARAR_ALGORITMO* son útiles para este propósito, en medio de ellas se declara la variable x de tipo *double*.
- La construcción del individuo modelo debe realizarse dentro del método *codificacion(Individuo* modelo, int estado)* quien transfiere el dato presente en cada variable del modelo al gen respectivo dentro del individuo. La implementación de este método utiliza la macro *ADICIONAR_GENREAL(individuo, pos, var, valormin, valormax, valorinicial)* especilizada en insertar dentro del individuo un gen de tipo real que simultáneamente se relaciona con la respectiva variable del sistema.
- Para definir la función a maximizar se debe sobrecargar el método *objetivo()* de la clase *AlgoritmoGenetico*. Este método debe retornar un número real de tipo *double* resultado de la relación entre la variable x del sistema y la función objetivo.
- El procedimiento de optimización se realiza dentro de la función principal del archivo. Dentro de ella, obligatoriamente debe crearse un objeto de la clase derivada de *AlgoritmoGenético* que se declaró; además debe invocarse el método *optimizar()* el cual realiza el proceso de optimización utilizando todos los valores establecidos por defecto en la librería.

Las instrucciones de visualización son complementarias para el ejemplo y permiten detallar la respuesta del algoritmo genético.

4.2. Ejemplo híbrido

Esta sección tiene el propósito de exponer cada etapa del proceso de construcción de un proyecto de optimización usando UNGenético. Con el fin de ilustrar este proceso plenamente, en el apartado 4.2.1 se plantea una función prototipo para la cual se buscará el valor del óptimo global, mientras que en las demás secciones se explica cada etapa de la construcción del proyecto.

4.2.1. Sistema propuesto

Como ejemplo se propone la maximización de la función

$$f(x, y, z) = \sum_{i=1}^{k} x_i + 4y + \sin(15z)e^{-z}$$

Donde,

$$x_i \in \{0, 1\}\text{, con } k = 1, \ldots 10.$$

$$y \in \mathbb{Z}[-10, 40] \text{ tal que } y \text{ es impar}$$

$$z \in \mathbb{R}[0, 6]$$

La optimización de esta función sugiere la utilización de codificación híbrida ya que cada una de sus variables es de distinto tipo; además la función cuenta con una restricción para una de ellas, por lo tanto se justifica el uso de UNGenético 2.0 como herramienta de optimización para esta función.

4.2.2. Creación del proyecto

Como se mencionó en la sección 4.1.2, las optimizaciones implementadas con UNGenético 2.0 se ejecutan dentro de un proyecto propio de un compilador de lenguaje C++ que debe incluir todos los archivos que hacen parte de la librería. El proyecto debe contener por lo menos un archivo fuente de C++ (extensión *.cpp*) donde se especificarán todas las características del sistema a optimizar.

4.2.2.1. Creación del proyecto en consola. En la mayoría de compiladores de C++ es posible crear proyectos especiales para aplicaciones en consola. Por lo tanto, dependiendo del compilador utilizado se debe crear un proyecto que construya este tipo de aplicación.

En el archivo principal del proyecto, es necesario emplear una instrucción que incluya el archivo *UNGenetico.h*, encargado de vincular a los demás archivos de la librería. El siguiente fragmento muestra cómo debe iniciarse el archivo mencionado para el sistema propuesto.

```
#include "UNGenetico.h"
```

4.2.2.2. Creación del proyecto usando entorno gráfico. Para crear una aplicación gráfica es necesario crear un proyecto que incluya la librería wxWindows, estos proyectos cambian su forma de creación y configuración en cada compilador donde se implemente, por lo tanto es recomendable consultar la documentación de esta librería para crear este tipo de proyectos.

En el archivo principal del proyecto, debe realizarse la definición de la constante *USAR_VENTANA*, con el fin de utilizar la interfaz gráfica diseñada para UNGenético 2.0. Adicionalmente, es necesario incluir el archivo *UNGenetico.h*. El siguiente fragmento muestra cómo debe iniciar el archivo principal para el sistema propuesto usando entorno gráfico.

```
#define USAR_VENTANA
#include "UNGenetico.h"
```

4.2.3. Definición de las propiedades del algoritmo genético

Para todo proyecto de optimización es necesario definir una clase derivada de *AlgoritmoGenetico* en la que se establecen las propiedades particulares del sistema a optimizar tales como los operadores genéticos, las características de los individuos de la población, la función objetivo, entre otras.

Las macros *DECLARAR_ALGORITMO(NombreAlgoritmo)* y *FIN_DECLARAR_ALGORITMO* son útiles para definir una clase derivada de AlgoritmoGenetico cuyo nombre se puede asignar libremente. Al utilizarlas, implícitamente se está declarando la clase y su constructor por defecto.

Dentro de esta definición es posible, si se requiere, sobrecargar los métodos que cambian las propiedades por defecto de la clase, estos son:

- Método *inicializarParametros()*: modifica los valores establecidos por defecto de los miembros de la clase AlgoritmoGenetico.
- Método *definirOperadores()*: define los operadores genéticos que se usarán en el proyecto.

Dentro de la definición de esta clase se deben declarar las variables del sistema que coincidirán con los genes que conformarán a los individuos de la población; opcionalmente para este ejemplo se declara también el método *mostrarIndividuo(Individuo& Ind)* que servirá para visualizar la información genética presente en un objeto de la clase *Individuo*.

En el siguiente fragmento se muestra el uso de las macros mencionadas para definir una clase derivada de *AlgoritmoGenetico* llamada *MiAlgoritmoGenetico*, también se declaran las variables *x*, *y*, y *z* del tipo adecuado para cumplir con las condiciones establecidas en el sistema propuesto.

```
DECLARAR_ALGORITMO(MiAlgoritmoGenetico)
    void inicializarParametros();
    void definirOperadores();
    ArregloBool x;
    long y;
    double z;
    void mostrarIndividuo(Individuo& Ind);
FIN_DECLARAR_ALGORITMO
```

4.2.3.1. Modificación de las propiedades de la clase AlgoritmoGenetico. La clase *AlgoritmoGenetico* posee por defecto un conjunto de propiedades típicas para procesos de optimización. Estas propiedades pueden ser observadas en el anexo B. Al implementar el método *inicializarParametros()* declarado anteriormente es posible redefinir los parámetros de la clase *AlgoritmoGenetico* de los cuales se desee cambiar su valor por defecto.

Para el sistema actual se desea:

- Maximizar la función objetivo.

- Ampliar la cantidad de individuos por generación del algoritmo genético a 200.
- Asignar valores iniciales no aleatorios a los genes de la primera generación del algoritmo.

El siguiente fragmento muestra cómo se realizan los cambios mencionados:

```
void MiAlgoritmoGenetico::inicializarParametros()
{
    m_IndicadorMaximizar=true;
    m_TamanoPoblacion=200;
    m_IndicadorInicializarPoblacionAleatoria=false;
}
```

4.2.3.2. Modificación de los operadores del algoritmo genético. UNGenético 2.0 posee un conjunto de operadores de probabilidad, selección, asignación de parejas, reproducción, cruce, mutación, adaptación y finalización para ejecutar el algortimo genético. El anexo C señala cuáles son los operadores establecidos por defecto. Para utilizar un conjunto diferente de operadores debe implementarse el método *definirOperadores()* de la clase *AlgoritmoGenetico*. En la implementación de este método es recomendable utilizar las macros que definen los operadores que utilizará el algoritmo. Los dos tipos de macros diseñadas para este fin son:

- Macros con el prefijo *DEFINIR_*: hacen referencia a operadores de los que sólo debe definirse uno para operar en el algoritmo genético, es decir, los operadores de probabilidad, selección, asignación de parejas y reproducción. Estas son:
  DEFINIR_OPERADOR_PROBABILIDAD(tipoOperador)
  DEFINIR_OPERADOR_SELECCION(tipoOperador)
  DEFINIR_OPERADOR_PAREJAS(tipoOperador)
  DEFINIR_OPERADOR_REPRODUCCION(tipoOperador)

- Macros con el prefijo *ADICIONAR_*: hacen referencia a operadores que pueden actuar en conjunto en el algoritmo, como los operadores de adaptación y finalización, o los correspondientes a cada gen del modelo como los operadores de cruce y mutación. Estas son:
  ADICIONAR_OPERADOR_ADAPTACION(tipoOperador)
  ADICIONAR_OPERADOR_FINALIZACION(tipoOperador)

ADICIONAR_OPERADOR_MUTACION(tipoOperador)
ADICIONAR_OPERADOR_CRUCE(tipoOperador)

En todas ellas, el parámetro *tipoOperador* especifica el tipo de operador a utilizar en cada caso junto con sus parámetros iniciales.

El siguiente fragmento muestra cómo definir un nuevo conjunto de operadores para el algoritmo, mientras que en el anexo D se puede observar una lista general de los operadores que hacen parte de la librería y que pueden reemplazar a los operadores asignados por defecto.

```
void MiAlgoritmoGenetico::definirOperadores()
{
    DEFINIR_OPERADOR_PROBABILIDAD(OperadorProbabilidadProporcional)
    DEFINIR_OPERADOR_SELECCION( OperadorSeleccionEstocasticaRemplazo)
    DEFINIR_OPERADOR_PAREJAS(OperadorParejasAdyacentes)
    DEFINIR_OPERADOR_REPRODUCCION(OperadorReproduccionMejorPadreMejorHijo)

    ADICIONAR_OPERADOR_ADAPTACION(OperadorAdaptacionProbMutacion
     (this,ADAPTACION_PROBMUTACION_OFFLINE))
    ADICIONAR_OPERADOR_FINALIZACION(OperadorTerminacionOffline)

    ADICIONAR_OPERADOR_MUTACION(OperadorMutacionArregloBool)
    ADICIONAR_OPERADOR_CRUCE(OperadorCruceArregloBool)
    ADICIONAR_OPERADOR_MUTACION(OperadorMutacionEnteroNoUniforme(this))
    ADICIONAR_OPERADOR_CRUCE(OperadorCruceEnteroHeuristico)
    ADICIONAR_OPERADOR_MUTACION(OperadorMutacionRealMuhlenbein)
    ADICIONAR_OPERADOR_CRUCE(OperadorCruceRealBLX)
}
```

En algunas de las anteriores definiciones se especifica la pablabra clave *this* como parámetro del operador. Esto es necesario en operadores que requieren una referencia al objeto *AlgoritmoGenetico* sobre el que operan.

Cuando se requiera cambiar algún operador de cruce y/o mutación por defecto para algún gen, es indispensable adicionar operadores de cruce y/o mutación para todos los genes que conforman al individuo teniendo en cuenta que el orden en que se adicionen estos operadores debe coincidir con el orden y tipo de los genes del individuo.

4.2.3.3. Definición de las propiedades de los individuos. Los individuos del algoritmo deben ser definidos de acuerdo con las variables del sistema a optimizar. Con este fin se implementa el método *codificacion(Individuo * Ind, int estado)*. Adicionalmente, para este ejemplo en particular se ha definido el método *mostrarIndividuo(Individuo & Ind)* que se utilizará para visualizar la información genética de los individuos durante la ejecución del algoritmo.

- *Método codificacion(Individuo * Ind, int estado).* La codificación consiste en transferir el dato presente en cada variable del modelo hacia su gen respectivo dentro del individuo. La implementación de este método utiliza las macros especilizadas en insertar genes del tipo adecuado dentro del individuo que simultáneamente se relacionarán con una variable del sistema.

  Las macros mencionadas llevan el prefijo *ADICIONAR_GEN*, y continúan con el tipo de gen que insertan en el individuo, todas ellas reciben los siguientes parámetros, en su orden:

  - Individuo: referencia al individuo en el que se adiciona el gen.
  - Posición: posición del gen dentro del individuo.
  - Variable: variable del sistema asociada con el gen.
  - Valor mínimo: valor mínimo que puede tomar el gen. No aplica para arreglos de tipo *bool* ni para arreglos de genes de tipo *bool*.
  - Valor máximo: valor máximo que puede tomar el gen. No aplica para arreglos de tipo *bool* ni para arreglos de genes de tipo *bool*.
  - Longitud mínima: longitud mínima del arreglo de genes. Sólo aplica para genes de tipo arreglo.
  - Longitud máxima: longitud máxima del arreglo de genes. Sólo aplica para genes de tipo arreglo.
  - Valor inicial: valor que toma el gen en la primera generación. Siempre debe especificarse, pero sólo se usa cuando no se generan valores iniciales aleatorios en los genes ($m\_InidicadorInicializarPoblacionAleatoria = false$).

Este procedimiento se observa en el siguiente fragmento:

```
void MiAlgoritmoGenetico::codificacion(Individuo * Ind , int estado)
{
    ADICIONAR_GENARREGLO_BOOL(Ind, 0, x, 1, 10 )
    ADICIONAR_GENENTERO(Ind, 1, y,  -10, 40, 20)
    ADICIONAR_GENREAL(Ind, 2, z, 0.00, 6.00, 1.0)
}
```

- *Método mostrarIndividuo(Individuo & Ind) en consola.* Este método se emplea para visualizar el valor de cada gen presente en el individuo apuntado por *Ind* junto con su función objetivo. El siguiente fragmento muestra el procedimiento realizado para el sistema planteado:

```
void MiAlgoritmoGenetico::mostrarIndividuo(Individuo & Ind)
{
    cout <<"\nFuncion Objetivo:\t" << Ind.objetivo(true);
    cout << "\nx: ";
    int tam = x.getSize();
    for(int i=0; i<tam; i++)
    {
        cout << x[i];
    }
    cout << "\ny: " << y;
    cout << "\nz: " << z;
}
```

  En la primera línea de esta función, la instrucción *Ind.objetivo(true)* recibe el parámetro *true* para indicar que deben actualizarse las variables del sistema, puesto que se van a imprimir. Si no se utilizara esta instrucción, o no se quisiera acceder a la función objetivo, sería necesario extraer los valores presentes dentro de cada gen por medio del método *codificacion(&Ind, ESTADO_DECODIFICAR)* quien ejecuta la decodificación, es decir, transfiere la información presente en los genes del individuo a las variables del sistema.

- *Método mostrarIndividuo(Individuo * Ind) usando entorno gráfico.* Para la utilización de la interfaz gráfica, los valores de las variables y la función objetivo deben ser almacenados en una cadena de caracteres de tipo *wxstring* propia de la librería wxWindows, que en este caso se ha llamado *Cad*. Gracias a la función *AppendText(Cad)*, esta cadena se agregará a un objeto de control de texto apuntado por *m_pTextCtrl* que pertenece a la ventana donde se muestra el proceso del algoritmo. Este procedimiento puede observarse en el siguiente fragmento:

```
void MiAlgoritmoGenetico::mostrarIndividuo(Individuo & Ind)
{
    wxString Cad;
    Cad <<"\nFuncion Objetivo:\t" << Ind.objetivo(true);
    Cad << "\nx: ";
    int tam = x.getSize();
    for(int i=0; i<tam; i++)
    {
        Cad << x[i];
    }
    Cad << "\ny: " << y;
    Cad << "\nz: " << z;
    m_pFrame->m_pTextCtrl->AppendText(Cad);
}
```

4.2.3.4. Definición de la función objetivo. En el método *objetivo()* de la clase *AlgoritmoGenetico* se define la función a optimizar, este método retorna un valor real (de tipo *double*) que debe estar relacionado por medio de la función objetivo con las variables del sistema previamente declaradas. Cuando existan restricciones para esta función, debe penalizarse a los individuos que las incumplan, la penalización debe reflejarse en su función objetivo.

Para el sistema planteado en este caso, el método *objetivo()* retorna la variable *FO* correspondiente al valor de la función objetivo para un individuo. En este caso, se requiere maximizar la función objetivo, por lo tanto, los individuos que incumplan la restricción de valores pares para la variable *y*, serán penalizados disminuyendo su función objetivo en 100. Lo anterior puede resumirse en el siguiente fragmento:

```
double MiAlgoritmoGenetico::objetivo()
{
    double FO=0.0;
    int tam = x.getSize();
    for(int i=0; i<tam; i++)
    {
        FO = FO + x[i];
    }
    FO = FO + 4*y + sin(15*z)*exp(-z);
    if(y%2==0)
        FO = FO - 100;
    return (FO);
}
```

4.2.4. Definición del procedimiento principal en consola

El procedimiento principal que enmarca al algoritmo genético se establece en la función *main()* propia del lenguaje C++. Dentro de este procedimiento es necesario realizar varias actividades, entre ellas:

- Se debe crear una instancia de la clase derivada de *AlgoritmoGenetico* que ha sido definida. Para este ejemplo se ha creado un objeto de la clase *MiAlgoritmoGenetico* llamado *MiAg*.
- El algoritmo genético puede ejecutarse en un sólo paso o mediante un ciclo que recorra cada una de sus generaciones. La primera opción es la más sencilla y se implementa al invocar el método *optimizar()* de la clase *AlgoritmoGenetico*. La optimización paso a paso debe iniciar invocando al método *iniciarOptimizacion()* y continúa con un ciclo que repite la ejecución del método *iterarOptimizacion()* hasta que la función *finalizar()* retorne *true*.
- Cuando el algoritmo es ejecutado paso a paso, es posible obtener información del algoritmo genético en cada generación. En este caso se ha utilizado el método *mostrarIndividuo(Individuo & Ind)* creado para este ejemplo, el cual recibirá como parámetro una referencia al mejor individuo de cada generación del algoritmo.
- Si se requiere, se puede invocar la función *mostrarMedidas()* para obtener los valores de las medidas propias del algoritmo.

El siguiente fragmento muestra la implementación del procedimiento principal en consola para el sistema tratado.

```
void main()
{
    MiAlgoritmoGenetico MiAg;
    int i=0;
    MiAg.iniciarOptimizacion();
    do
    {
        MiAg.iterarOptimizacion();
        cout << "\n\nGENERACION "<< i;
        MiAg.mostrarIndividuo(*MiAg.m_pMejorEnEstaGeneracion);
        i++;
    }while(!MiAg.finalizar());
    MiAg.mostrarMedidas();
}
```

4.2.5. Definición del procedimiento principal usando entorno gráfico

Con el fin de crear una ventana para la aplicación del proyecto de optimización, se debe utilizar la macro *DECLARAR_APLICACION(NombreAplicacion)* la cual define a una clase derivada de la clase *wxApp* propia de la librería wxWindows. Con esta única instrucción, implícitamente se está declarando la clase para la aplicación, el apuntador a la ventana principal que se utilizará y el método *OnInit()*, el cual debe implementarse posteriormente y es quien recibe el control del programa una vez se ha creado la aplicación.

El método *OnInit()* es equivalente a la función principal de la aplicación, dentro de éste es necesario realizar varias actividades adicionales a las menciondas en el caso anterior, estas son:

- A través del apuntador *m_pframe* que pertenece a la clase de la aplicación, se debe asignar espacio en memoria para un objeto de la clase *AGFrame* que corresponde a la ventana principal; en la misma instrucción se asignan su título, coordenadas y tamaño.
- Se requiere invocar a los métodos *SetTopWindow()* y *Show()* para que la ventana de aplicación sea visible.
- La macro *EJECUTAR_EVENTOS* ejecuta los eventos pendientes de la aplicación, debe usarse en medio de procesos largos para dar una orden de ejecución a las tareas y evitar bloqueos.
- El método *OnInit()* debe retornar el booleano *true* para evitar inconvenientes en la aplicación.

El siguiente fragmento muestra la implementación de las tareas descritas anteriormente para una aplicación denominada *AGApp*.

```
DECLARAR_APLICACION(AGApp)
bool AGApp::OnInit()
{
    m_pFrame = new AGFrame("UNGenético 2.0", 50, 50, 700, 500);
    SetTopWindow(m_pFrame);
    m_pFrame->Show(true);
    MiAlgoritmoGenetico MiAg(m_pFrame);
    MiAg.iniciarOptimizacion();
```

```
    int i=0;
    do
    {
        MiAg.iterarOptimizacion();
        wxString Cad;
        Cad << "\n\nGENERACION "<< i;
        m_pFrame->m_pTextCtrl->AppendText(Cad);
        MiAg.mostrarIndividuo(*MiAg.m_pMejorEnEstaGeneracion);
        i++;
        EJECUTAR_EVENTOS
    }while(!MiAg.finalizar());
    MiAg.mostrarMedidas();
    return true;
}
```

4.2.6. Ejecución del proyecto

El paso siguiente al terminar la construcción del archivo principal del proyecto, es su ejecución. Este consiste en compilar el proyecto para verificar su correcta estructura de lenguaje y construir finalmente la aplicación donde se ejecuta el algoritmo genético. Estas actividades son realizadas en distintas formas dependiendo del compilador de lenguaje C++ donde se haya creado el proyecto.

Si se siguieron correctamente las secciones anteriores, entonces en la aplicación escogida se presentará la información genética del mejor individuo para cada generación del algoritmo y al finalizar se mostrarán las medidas propias del algoritmo.

Si se definió el uso de la interfaz gráfica, se obtendrá una ventana como la que se muestra en la figura 4.2. La información que se obtiene en esta ventana, puede ser modificada y salvada como un archivo de texto (extensión *.txt*) escogiendo la opción *Guardar* del menú *Archivo*.

4.3. Resumen gráfico del desempeño del algoritmo

Por defecto UNGenético 2.0 crea un archivo de texto con los resultados del algoritmo genético llamado *salidas.txt*, este archivo contiene los valores de las medidas propias del algoritmo en las generaciones que han sido seleccionadas.

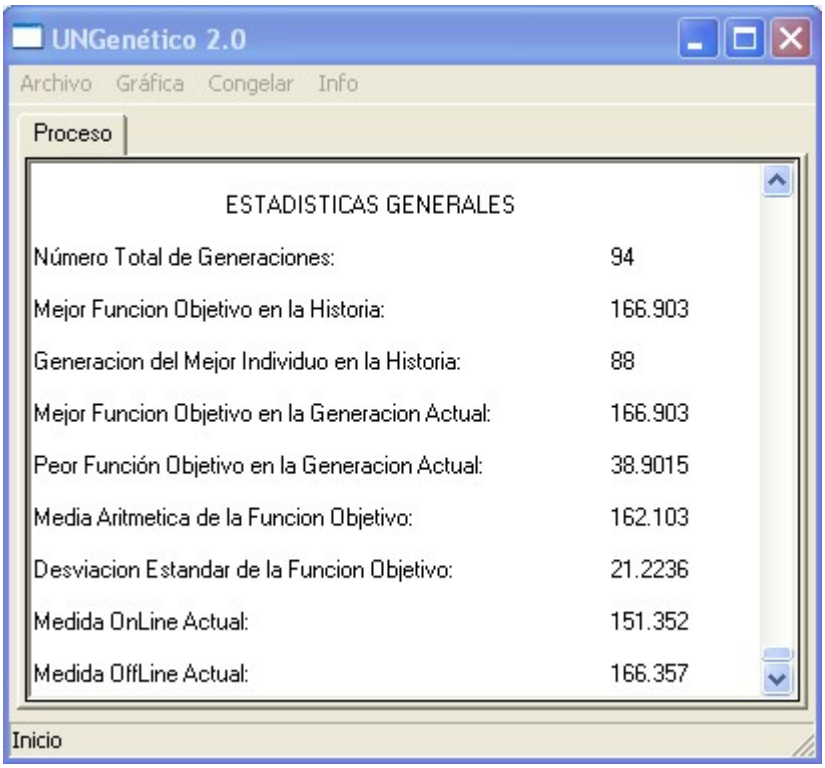


Figura 4.2: Aplicación gráfica de UNGenético 2.0

Como complemento de la librería, se ha creado UNGenético Graphics, una aplicación diseñada especialmente para describir gráficamente el proceso de optimización. Esta herramienta obtiene sus datos de entrada del archivo de texto creado al ejecutar un proyecto que utiliza UNGenético 2.0. De este modo es posible obtener un registro gráfico de la ejecución del algoritmo sin importar si su ejecución se realizó en consola o en un entorno gráfico.

Si el proyecto se ejecutó usando entorno gráfico, UNGenético Graphics estará incluido en la aplicación del proyecto y obtendrá sus datos de entrada automáticamente del archivo *salidas.txt* que se encuentre en la misma carpeta donde está ubicado el proyecto. Si éste se ejecutó en consola, se deberá abrir de manera independiente la aplicación UNGenético Graphics e indicar la ruta del archivo de salida del proyecto.

Una vez dentro de la aplicación, al escogerse la opción *Graficar* del menú *Gráfica* se mostrarán las gráficas de las medidas representativas del algoritmo genético en distintas páginas con el siguiente orden:

- *Mejor en la historia*: Gráfica del valor de la función objetivo del mejor individuo que ha

aparecido en la historia del algoritmo contra la generación en la que apareció.

- *Generación del mejor en la historia*: Gráfica de la generación en la que apareció el mejor individuo en la historia del algoritmo.
- *Mejor en generación actual*: Gráfica del valor de la función objetivo del mejor individuo en cada generación.
- *Peor en generación actual*: Gráfica del valor de la función objetivo del peor individuo en cada generación.
- *Media*: Gráfica del valor de la media aritmética de la función objetivo de los individuos en cada generación.
- *Desviación estándar*: Gráfica del valor de la desviación estándar de la media aritmética de la función objetivo de los individuos en cada generación.
- *Medida online*: Gráfica del valor de la medida online del algoritmo en cada generación.
- *Medida offline*: Gráfica del valor de la medida offline del algoritmo en cada generación.

La figura 4.3 ilustra un ejemplo de los resultados gráficos obtenidos para el sistema propuesto. Cada una de estas gráficas puede ser editada cambiando el color del fondo, de los ejes y de la curva al escoger la opción correspondiente en el menú *Gráfica*; complementariamente, estas gráficas pueden ser salvadas como archivos de imagen en formato *png* al seleccionar la opción *Guardar* del menú *Archivo*.

4.4. Utilización de UNGenético Wizard

UNGenético Wizard es una aplicación diseñada con el fin de ayudar al usuario en la creación del archivo principal de un proyecto que utiliza UNGenético 2.0, donde se especifican las características más importantes del sistema a optimizar.

Las siguientes secciones describen el procedimiento de creación del archivo principal del proyecto de optimización para el sistema propuesto en la sección 4.2.1 utilizando UNGenético Wizard.

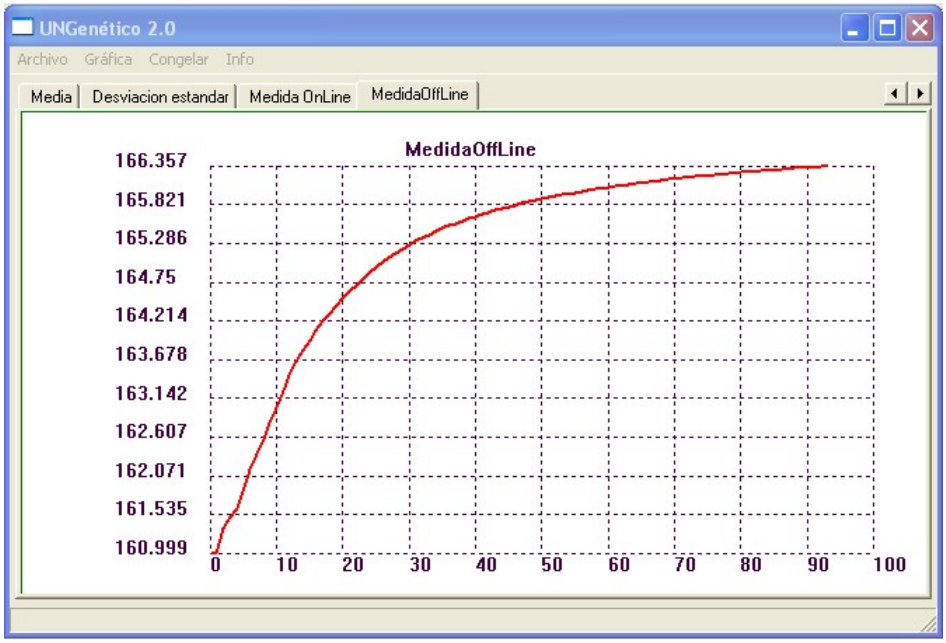


Figura 4.3: Resultados de la optimización obtenidos con UNGenético Graphics

4.4.1. Adición de variables

UNGenético Wizard permite adicionar al modelo variables de los tipos que permite manejar la librería. La página *Variables* de la aplicación se especializa en esta labor. Al escoger la opción *Adicionar* se mostrará una ventana que permite especificar las características de las variables del modelo y relacionarlas con un gen, estas son:

- *Tipo de variable*: especifica el tipo de variable del modelo. Puede ser booleana, entera, real o arreglos de estos mismos tipos.
- *Nombre*: identifica a la variable del modelo.
- *Operador de mutación*: define el operador de mutación para el gen relacionado con la variable.
- *Operador de cruce*: define el operador de cruce para el gen relacionado con la variable.
- *Valor máximo*: valor máximo permitido que puede tomar la variable.
- *Valor mínimo*: valor mínimo permitido que puede tomar la variable.

- *Dimensión máxima*: dimensión máxima permitida que puede tomar el arreglo, sólo se puede establecer para variables de tipo arreglo.
- *Dimensión mínima*: dimensión mínima permitida que puede tomar el arreglo, sólo se puede establecer para variables de tipo arreglo.

La figura 4.4 describe la selección de parámetros para una variable del sistema, se ha escogido una variable de tipo *Entera* cuyos valores se han restringido en el rango $[-10, 40]$; también se han escogido los operadores *No Uniforme* y *Heurístico* como operadores de mutación y cruce respectivamente para el gen relacionado con esta variable.

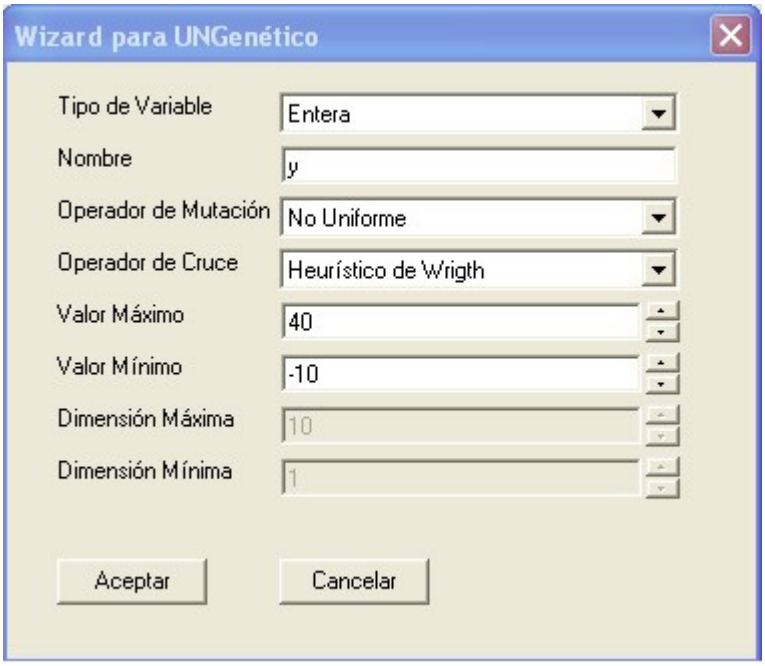


Figura 4.4: Adición de una variable con UNGenético Wizard

Al finalizar la adición para todas las variables del sistema, éstas quedarán almacenadas junto con todas sus propiedades en la página *Variables* de la aplicación. La figura 4.5 representa la adición de las variables para el sistema planteado.

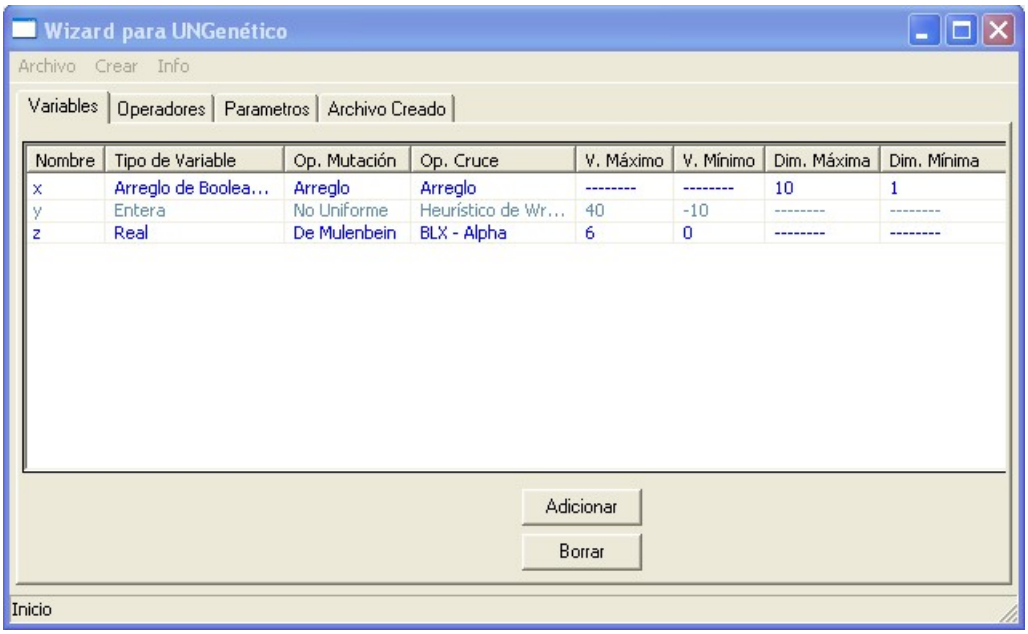


Figura 4.5: Adición de variables en UNGenético Wizard

4.4.2. Definición de operadores genéticos

Con UNGenético Wizard también es posible definir los operadores genéticos que se utilizarán en el proyecto de optimización. La página *Operadores* ofrece distintas posibilidades para este propósito.

Al activar la opción *Operadores por Defecto*, el proyecto utilizará todos los operadores genéticos establecidos por defecto para la librería (ver Anexo C), incluyendo los operadores de cruce y mutación establecidos por defecto para cada tipo de gen; al desactivar esta opción, es posible escoger individualmente cada operador genético que se desee utilizar en el proyecto. Para cada caso, UNGenético Wizard mostrará todas las posibles opciones.

La figura 4.6 muestra la página *Operadores* donde se establecen los operadores genéticos para el sistema tratado.

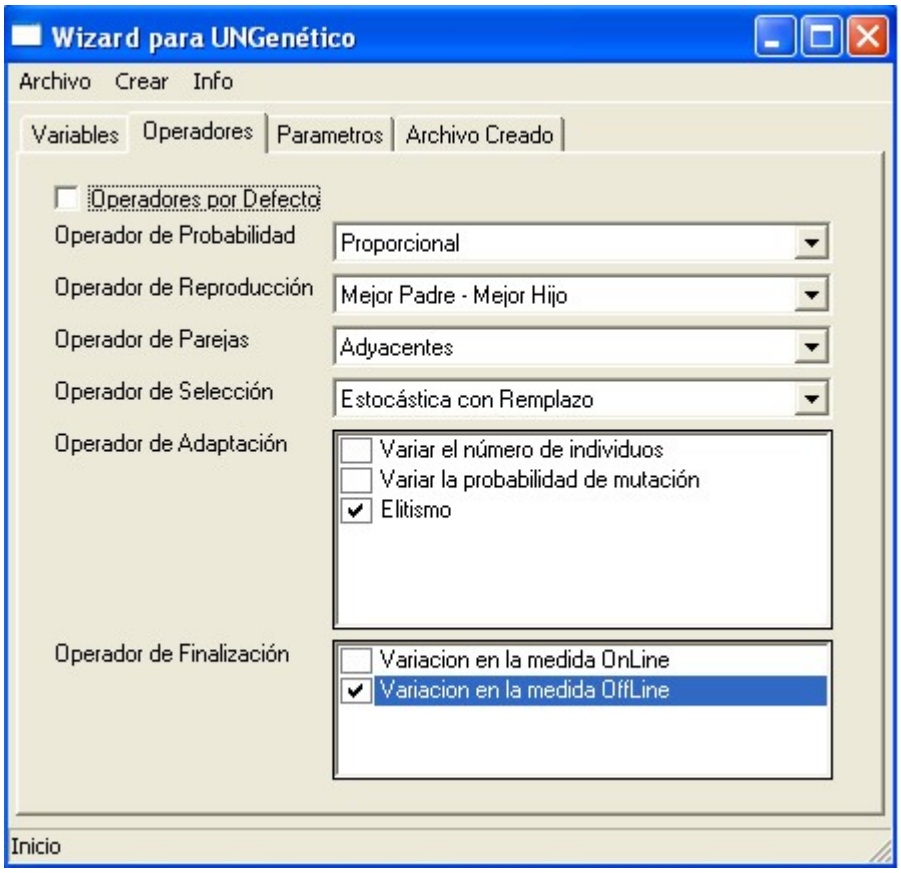


Figura 4.6: Definición de operadores genéticos con UNGenético Wizard

4.4.3. Definición de parámetros para la optimización

UNGenético Wizard permite modificar los parámetros del algoritmo genético que definen el número de individuos por generación y el número máximo de iteraciones del algoritmo; también permite establecer si se debe maximizar o minimizar la función objetivo. Esto es posible desde la página *Parámetros* de la aplicación; si en esta página, la opción *Parámetros por Defecto* se encuentra activa, el proyecto utilizará los valores por defecto establecidos en la clase AlgoritmoGenetico (ver anexo B). En esta página también es posible seleccionar algunos parámetros propios de la librería como el nombre del archivo donde se almacenarán las medidas del desempeño del algoritmo y el intervalo de generaciones en que deben ser guardadas estas medidas.

La figura 4.7 muestra la página de elección de parámetros para el sistema propuesto, para el sistema propuesto, se han establecido 200 individuos por generación del algoritmo y un máximo

de 200 iteraciones, también se ha escogido la opción *Maximizar* para encontrar el valor máximo de la función de evaluación.

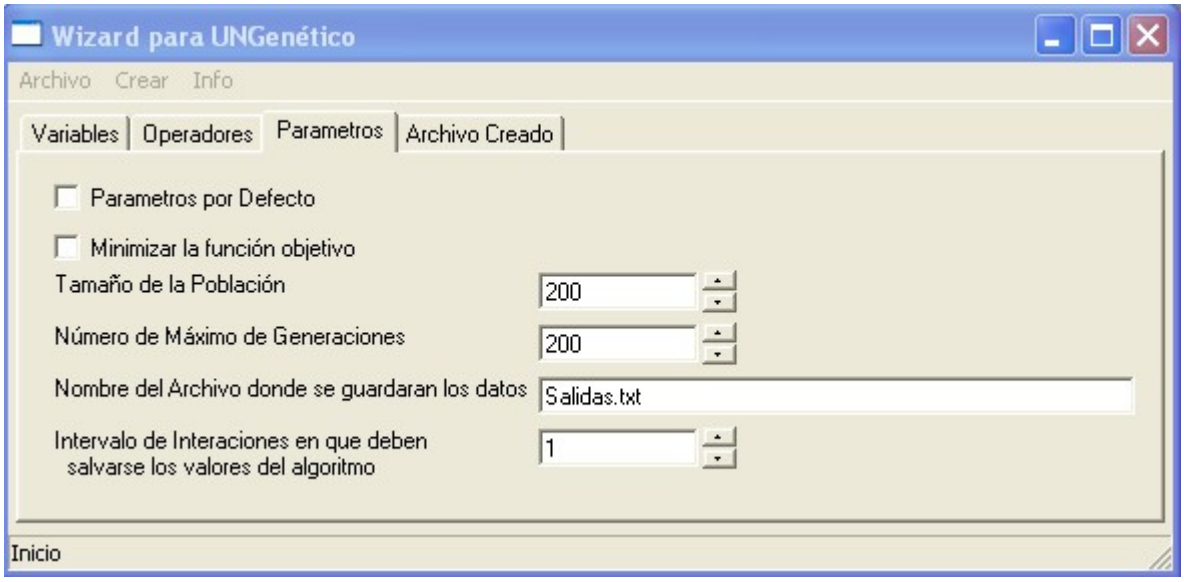


Figura 4.7: Definición de parámetros con UNGenético Wizard

4.4.4. Creación del archivo principal

El siguiente paso después de definir las propiedades del proyecto de optimización es crear su archivo principal. Esta acción se realiza mediante la opción *Crear* del menú *Crear*; al ejecutarse, la aplicación generará el código en lenguaje C++ del archivo principal del proyecto y lo mostrará en la página *Archivo creado* de la aplicación. Este código puede ser guardado en un archivo fuente de C++ (extensión *.cpp*) seleccionando la opción *Guardar* del menú *Archivo*.

Es importante resaltar que este código solamente contiene información básica acerca del proyecto, el usuario debe complementar y/o modificar este archivo con una función objetivo acorde a su problema a optimizar; si es necesario debe añadir las restricciones pertinentes y las instrucciones propias que se requieran.

Capítulo 5

EJEMPLO DE APLICACIÓN FUNCIONES DE PRUEBA

En este capítulo se presenta la implementación de una aplicación que utiliza la librería UNGenético 2.0, para encontrar el valor óptimo global de algunas funciones, que se han denominado *funciones de prueba* debido a que cuentan con características especiales que permiten verificar el funcionamiento de los algoritmos de búsqueda, entre ellos los algoritmos genéticos. Para estas funciones, extraídas de [7], se conocen los valores que definen su punto óptimo.

Una breve descripción de cada una de las funciones de prueba se realiza en la sección 5.1. La sección 5.2 presenta la implementación de la optimización de estas funciones utilizando UNGenético 2.0 y los resultados obtenidos se resumen en la sección 5.3.

5.1. Funciones de Prueba

5.1.1. Modelo Esférico

$$f(\vec{x}) = \sum_{i=1}^{n} x_i^2 \tag{5.1}$$
$$\text{mín}(f) = f(0, \ldots, 0) = 0$$

Es una función continua y es considerada sencilla para optimizar debido a que presenta un solo óptimo.

5.1.2. Función de Rosenbrock Generalizada

$$f(\vec{x}) = \sum_{i=1}^{n-1} (100 \cdot (x_{i+1} - x_i^2)^2 + (x_i - 1)^2) \tag{5.2}$$
$$\text{mín}(f) = f(1, \ldots, 1) = 0$$

Es una función continua y es considerada una función compleja debido a que el óptimo global está localizado en un valle parabólico con fuertes desniveles y con un fondo llano.

5.1.3. Función de Schwefel 1.2

$$f(\vec{x}) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right)^2 \tag{5.3}$$
$$\text{mín}(f) = f(0, \ldots, 0) = 0$$

Es una función continua y posee un solo óptimo. Su dificultad radica en que la búsqueda a lo largo de los ejes de coordenadas proporciona una convergencia ineficaz, ya que la pendiente de esta función no está orientada a lo largo de los mismos.

5.1.4. Función de Rastrigin Generalizada

$$f(\vec{x}) = n \cdot a + \sum_{i=1}^{n} \left(x_i^2 - a \cdot \cos(\omega \cdot x_i)\right) \tag{5.4}$$
$$\text{mín}(f) = f(0, \ldots, 0) = 0$$

Es una función continua que presenta varios óptimos locales. Su superficie está determinada por los parámetros a y ω que controlan la amplitud y frecuencia de la modulación, respectivamente. Al alejarse del origen, es similar al modelo esférico (ecuación (5.1)), sin embargo, con valores pequeños de x_i, el efecto de la modulación se hace mayor.

5.1.5. Función de Griewangk

$$f(\vec{x}) = \frac{1}{d} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \tag{5.5}$$
$$\text{mín}(f) = f(0, \ldots, 0) = 0$$

Es una función continua que posee cuatro óptimos locales en los puntos $(\pm\pi, \pm\pi\sqrt{2}, 0, \ldots, 0)$ con función objetivo de *0.0074*.

5.1.6. Función de Ackley

$$f(\vec{x}) = -a \cdot \exp\left(-b \cdot \sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n} \cdot \sum_{i=1}^{n} \cos(c \cdot x_i)\right) + a + e \tag{5.6}$$
$$\text{mín}(f) = f(0, \ldots, 0) = 0$$

Es una función continua y su superficie posee muchos óptimos locales. Además presenta un amplio hueco donde está situado el óptimo global. En este espacio existen algunos óptimos locales situados en pequeñas llanuras.

5.1.7. Función de Schwefel

$$f(\vec{x}) = 418{,}9829 \cdot n - \sum_{i=1}^{n} x_i \sin\sqrt{|x_i|} \tag{5.7}$$

$$\text{mín}(f) = f(420{,}9687, \ldots, 420{,}9687) = 0$$

Es una función continua que tiene muchos óptimos locales. Los valores $x_i = 420{,}9687,\ x_j = -302{,}5232$, para $i = 1, \ldots, n,\ i \neq j$ representan el mejor óptimo local pero se encuentran muy alejados del óptimo global.

5.1.8. Función de Michalewicz

$$f(\vec{x}) = -\sum_{i=1}^{n} \sin(x_i) \cdot \sin^{2m}\left(\frac{i \cdot x_i^2}{\pi}\right) \tag{5.8}$$

$$\text{Para } 0 \leq x_i \leq \pi \text{ y } n = 10$$

$$\text{mín}(f) = -9{,}70413$$

Posee $n!$ óptimos locales y cuanto mayor es m, la búsqueda del óptimo global es más difícil.

5.1.9. Función de Katsuura

$$f(\vec{x}) = \prod_{i=1}^{n} \left(1 + i \cdot \sum_{k=1}^{\beta} \frac{|2^k x_i - |2^i x_i||}{2^k}\right) \tag{5.9}$$

$$\text{mín}(f) = f(0, \ldots, 0) = 1$$

Es una función continua. Su característica principal es que para $\beta \to \infty$ no es diferenciable, y posee un número infinito de óptimos locales.

5.2. Implementación utilizando UNGenético 2.0

Para encontrar los valores óptimos de las funciones de prueba planteadas en la sección anterior utilizando la librería UNGenético 2.0 se creó una aplicación llamada *funciones*, en la cual se utiliza el entorno gráfico de wxWindows.

La aplicación muestra el progreso de la función objetivo de la mejor solución encontrada, las medidas online y offline, y los valores de cada variable del sistema en cada iteración del algoritmo. Al finalizar la ejecución son creados los siguientes archivos:

- *Valores.txt:* guarda los valores de cada variable en el trascurso de las iteraciones del algoritmo, en el formato reconocido por UNGenético Graphics para realizar su visualización.
- *Salidas.txt:* contiene el resumen de las medidas de desempeño de todas las generaciones del algoritmo.
- *Iteraciones.txt:* guarda el resumen de cada ejecución del algoritmo genético adicionando los siguientes datos: número de individuos, número de generaciones y mejor función objetivo obtenida.

5.2.1. Clase AGFunciones

La clase que define el algoritmo genético, derivada de *AlgoritmoGenetico*, indispensable en cualquier proyecto de optimización con UNGenético 2.0, se denominó *AGFunciones*, la cual se implementó con los siguientes miembros:

- Se sobrecargó la función virtual **inicializarParametros()**, donde se modifican los valores de *m_TamanoPoblacion* y *m_GeneracionMaxima* que son establecidos por el usuario desde el entorno gráfico.
- Se sobrecargó la función virtual **definirOperadores()**, donde se asignan los operadores del algoritmo. Se utilizaron los operadores que se muestran en la tabla 5.1.

Tipo de Operador	Operador Escogido
Operador de Probabilidad	*OperadorProbabilidadLineal*
Operador de Selección	*OperadorSeleccionEstocasticaRemplazo*
Operador de Parejas	*OperadorParejasAleatorias*
Operador de Reproducción	*OperadorReproduccionMejoresEntrePadresEHijos*
Operador de Adaptación	*OperadorAdaptacionElitismo*
Operador de Finalización	Seleccionable por el usuario
Operador de Mutación	*OperadorMutacionArregloReal* con probabilidad de mutación igual a 0.01

Tabla 5.1: Operadores utilizados en la aplicación *funciones*

- Se sobrecargó la función virtual **codificacion()**. Para realizar la codificación de este problema el individuo utilizado tiene un sólo gen de tipo arreglo de valores reales, como se muestra en la figura 5.1. Por lo tanto su implementación se hace utilizando la macro *ADICIONAR_GENARREGLO_REAL*.

  Tanto la dimensión de este arreglo correspondiente a la dimensión del problema, como los límites de las variables x_i para $i = 1, 2, \ldots, dimension$, son establecidos por el usuario.

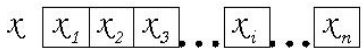


Figura 5.1: Individuo utilizado en la aplicación *funciones*

- Se sobrecargó la función virtual **objetivo()**, donde se calcula el valor de la función seleccionada por el usuario.

- Se definió la función **mostrar(Individuo& Ind, int generacion)**, para desplegar en la página *Proceso* los valores contenidos en cada gen del individuo *Ind* y el valor de su función objetivo.

- Se definió la función **GuardarValores(Individuo& Ind, int generacion)**, que guarda en el archivo de texto *valores.txt* los valores almacenados en cada gen del individuo *Ind*.

- Se definieron las variables enteras *Dimension, Emax, Emin* y *func*. *Dimension* corresponde al número de variables de la función, *Emax, Emin* corresponden al valor máximo y mínimo de las variables de cada gen, y *func* especifica cuál función está siendo utilizada. Estos valores son modificados por el usuario.

- Se definieron las variables enteras *aRAS, dGRI, aACK, mMIC*, que corresponden al valor de a en la funcion de Rastrigin (5.4), al valor de d en la función de Griewangk (5.5), al valor de a en la función de Ackley (5.6), y al valor de m en la función de Michalewicz (5.8), respectivamente. Estos valores son modificados por el usuario.
- La variable *x* de tipo *ArregloReal*, la cual contiene los valores decodificados de la información genética de un individuo.
- El apuntador *pValores* al archivo de texto *valores.txt*

5.2.2. Clase MiVentana

Se creó la clase *MiVentana*, derivada de *AGVentana*, que construye una ventana adicional cuando se ejecuta el programa y permite al usuario seleccionar la función a optimizar, la dimensión de la función, los valores máximo y mínimo de las variables, parámetros específicos de cada función, y parámetros del algoritmo genético como el operador de finalización a utilizar, el número máximo de generaciones y el número de individuos de cada generación.

5.3. Resultados

En esta sección se presentan los resultados obtenidos al ejecutar el algoritmo genético para cada una de las funciones mencionadas. En la tabla 5.2 se muestra un resumen de la ejecución del algoritmo, mientras que en cada subsección se presenta un análisis más detallado de la solución para cada función.

Función	Dimensión	Número de Individuos	Número de Generaciones	Solución Obtenida
Modelo Esférico	25	200	400	0.001592
		500	400	4×10^{-6}
Rosenbrock	25	200	400	19.680764
		1000	400	22.046754
Schwefel 1.2	25	200	1000	16.859768
		1000	1000	0.537747
Rastrigin	25	200	500	$3{,}28 \times 10^{-4}$
		1000	500	1×10^{-6}
Griewangk	25	200	400	$1{,}324 \times 10^{-3}$
		1000	400	1×10^{-6}
Ackley	25	100	200	0.849190
		500	200	$2{,}706 \times 10^{-3}$
Schwefel	25	1000	600	1×10^{-8}
Michalewicz $m = 1$	10	100	500	-9.704127
Michalewicz $m = 100$	10	100	500	-9.654648
Katsuura	20	1000	300	1.003301

Tabla 5.2: Resumen de la ejecución del algoritmo genético para las funciones de prueba

5.3.1. Modelo Esférico

Se tomó la dimensión de la función igual a 25 y los valores de las variables se especificaron en el intervalo $[-1000, 1000]$.

Debido a que esta función no presenta óptimos locales, para el algoritmo genético llegar al valor óptimo es relativamente sencillo. Con 200 individuos y 400 generaciones las soluciones encontradas son del orden de 10^{-3}, y con 500 individuos y el mismo número de generaciones las soluciones encontradas son del orden de 10^{-6}.

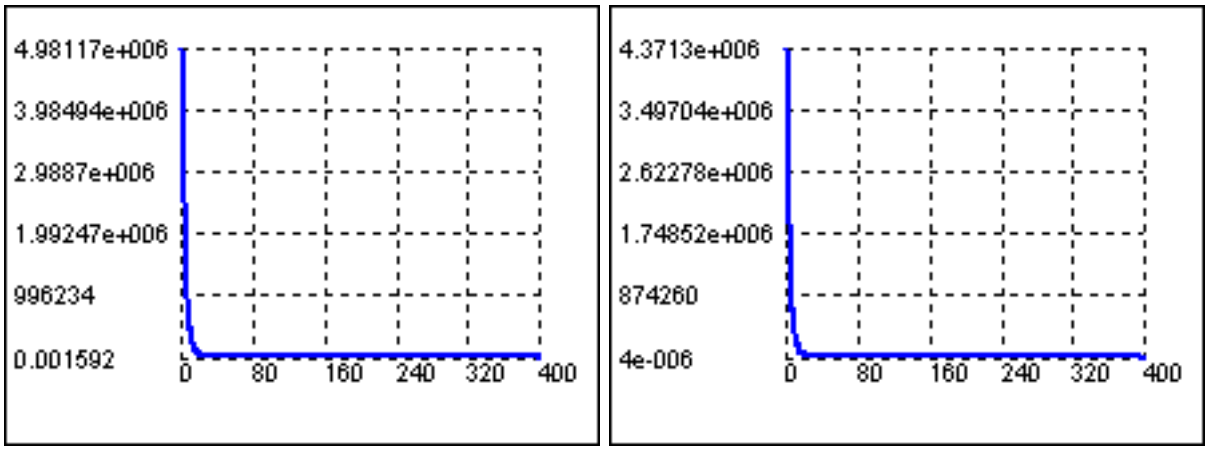


Figura 5.2: Gráficas del Mejor en la Historia para la función Modelo Esférico usando poblaciones de 200 y 500 individuos

5.3.2. Función de Rosenbrock Generalizada

Es complejo para los algoritmos genéticos encontrar la respuesta óptima de esta función en el intervalo $[-5, 5]$ y dimensión igual a 25. Con 200 individuos la mejor solución lograda fue 19,6808; al aumentar el número de individuos no se obtiene una mejora significativa.

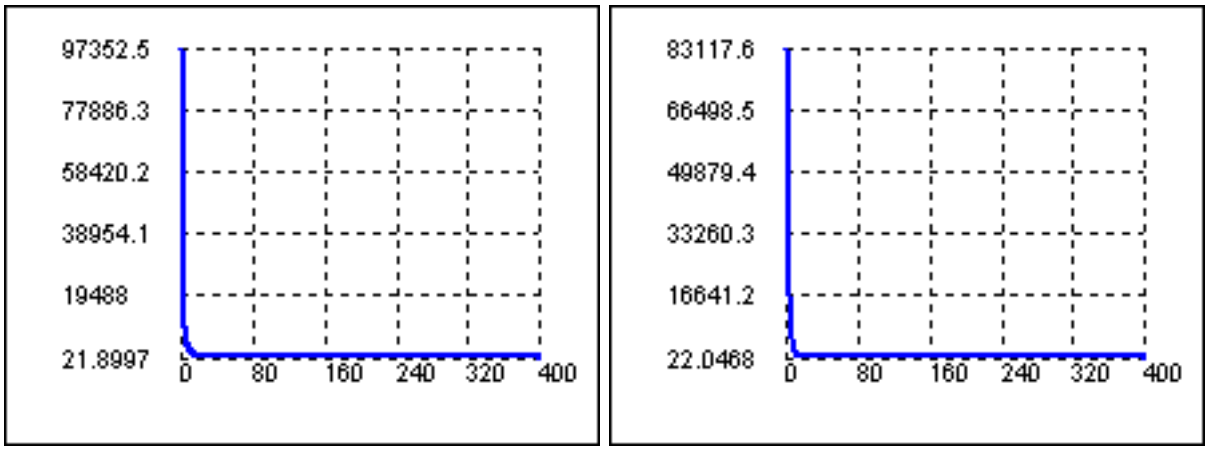


Figura 5.3: Gráficas del Mejor en la Historia para la función de Rosenbrock usando poblaciones de 200 y 1000 individuos

5.3.3. Función de Schwefel 1.2

Para el intervalo $[-65, 65]$ y dimensión igual a 25 encontrar la solución óptima de esta ecuación requiere de un tiempo considerable. Con una población de 200 individuos y 1000 generaciones la mejor solución encontrada fue $16{,}86$, pero al aumentar el número de individuos la solución mejora notablemente. Con 1000 individuos en la generación 1000 las soluciones son del orden de 10^{-1}.

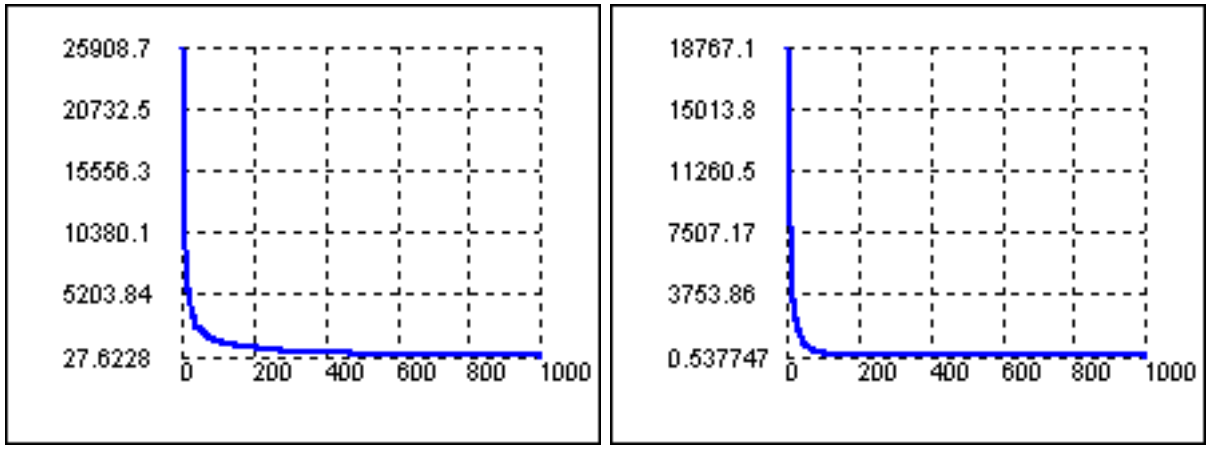


Figura 5.4: Gráficas del Mejor en la Historia para la función de Schwefel 1.2 usando poblaciones de 200 y 1000 individuos

5.3.4. Función de Rastrigin Generalizada

Los parámetros utilizados fueron $a = 10$ y $\omega = 2\pi$. Los valores de las variables se encuentran en el intervalo $[-5, 5]$ y la dimensión es igual a 25.

En esta función el factor $a \cdot \cos(\omega \cdot x_i),\ \omega = 2\pi$, se vuelve importante y hace que algunos x_i sean iguales a 1 ó -1. El poseer muchos óptimos locales cerca del óptimo global provoca que el algoritmo algunas veces se estanque en dichos valores.

A pesar de lo anterior, al ejecutar el algoritmo genético usando una población de 200 individuos y 500 generaciones las soluciones son del orden de 10^{-4}. Al aumentar el número de individuos a 1000 las soluciones son del orden de 10^{-6}.

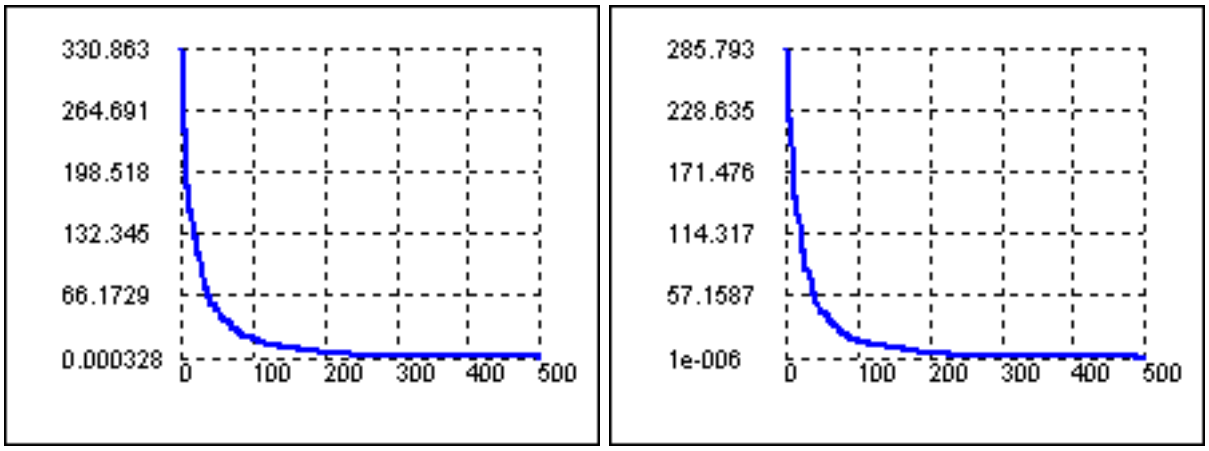


Figura 5.5: Gráficas del Mejor en la Historia para la función de Rastrigin usando poblaciones de 200 y 1000 individuos

5.3.5. Función de Griewangk

En el intervalo $[-600, 600]$, con $d = 1$ y dimensión igual a 25 se obtiene fácilmente el valor óptimo de esta función. Con tan sólo 200 individuos y 400 generaciones las soluciones son del orden de 10^{-3}. Con 1000 individuos y el mismo número de generaciones las soluciones son del orden de 10^{-8}.

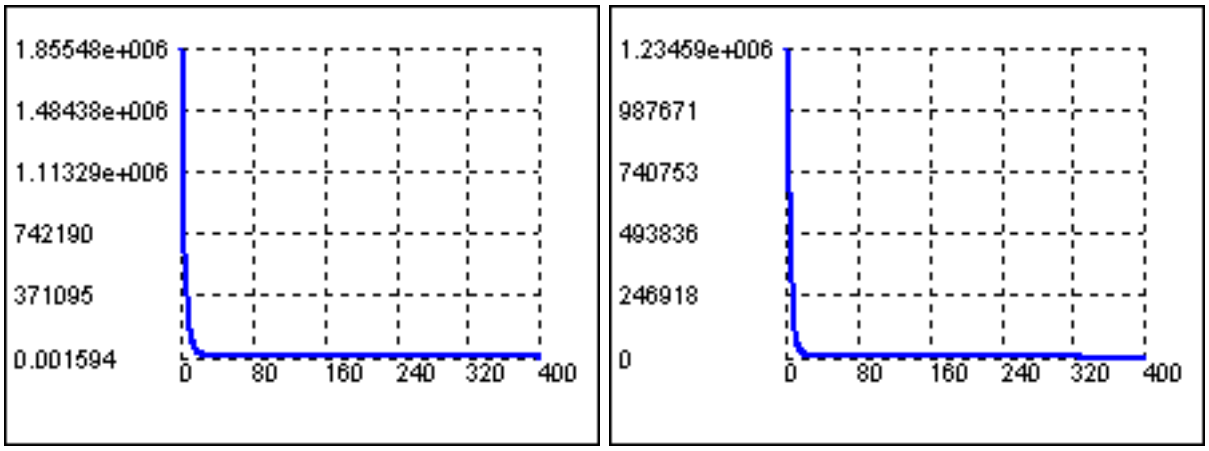


Figura 5.6: Gráficas del Mejor en la Historia para la función de Griewangk usando poblaciones de 200 y 1000 individuos

5.3.6. Función de Ackley

Para la ejecución del algoritmo genético, el valor de los parámetros utilizados fueron: $a = 20, b = 0.2, c = 2\pi$, y la dimensión igual a 25.

Encontrar la solución óptima en el intervalo $[-50, 50]$ de esta función es sencillo y rápido. Con 100 individuos en 200 generaciones las soluciones son del orden de 10^{-1}, al aumentar el número de individuos a 500 el orden de las soluciones es de 10^{-3}.

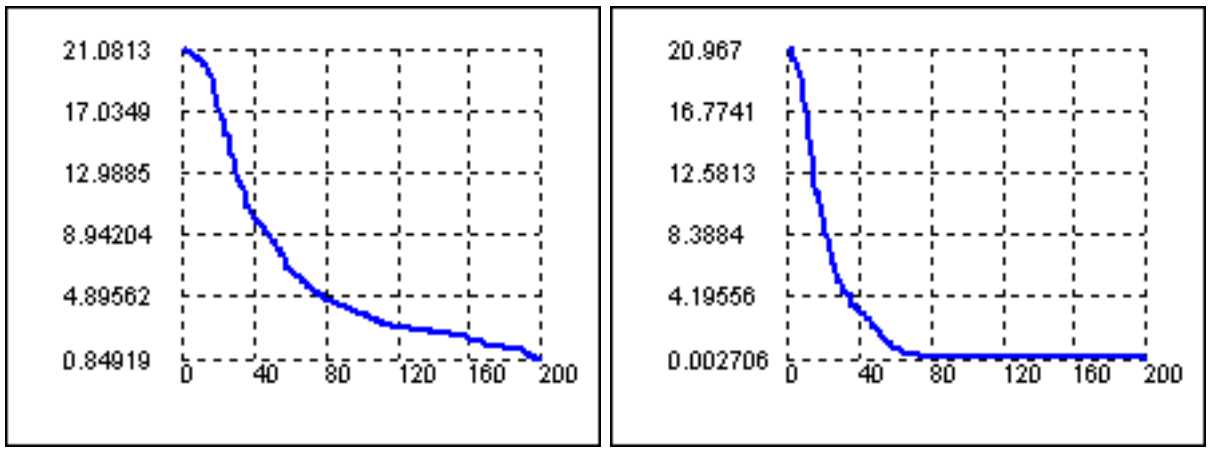


Figura 5.7: Gráficas del Mejor en la Historia para la función de Ackley usando poblaciones de 100 y 500 individuos

5.3.7. Función de Schwefel

En el intervalo $[-500, 500]$ y dimensión igual a 25, el algoritmo genético al principio tiende a encontrar valores de x_i iguales a $-302{,}5232$, $203{,}803$ y $-124{,}829$ que representan un buen óptimo local pero se encuentran muy alejados de $x_i = 420{,}9687$ donde se encuentra el óptimo global. Sin embargo el algoritmo sale de estos óptimos locales y en la generación 600 utilizando 1000 individuos, las soluciones son del orden de 10^{-8}.

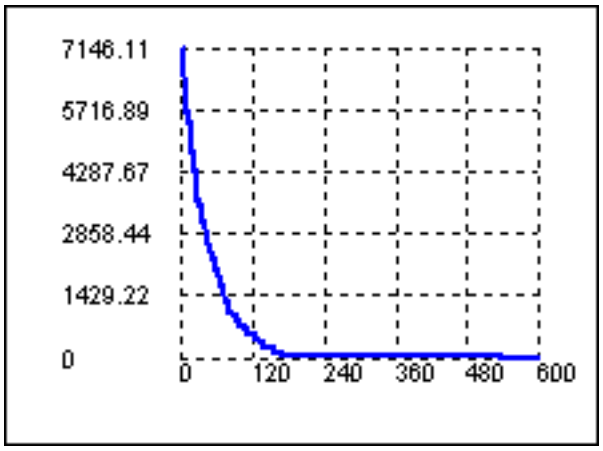


Figura 5.8: Gráfica del Mejor en la Historia para la función de Schwefel usando una población 1000 individuos

5.3.8. Función de Michalewicz

En el intervalo $[0, \pi]$ y con tan sólo 100 individuos el algoritmo encuentra la solución fácilmente para $m = 1$, a medida que el valor de *m* aumenta, el algoritmo tarda más tiempo en encontar el óptimo global.

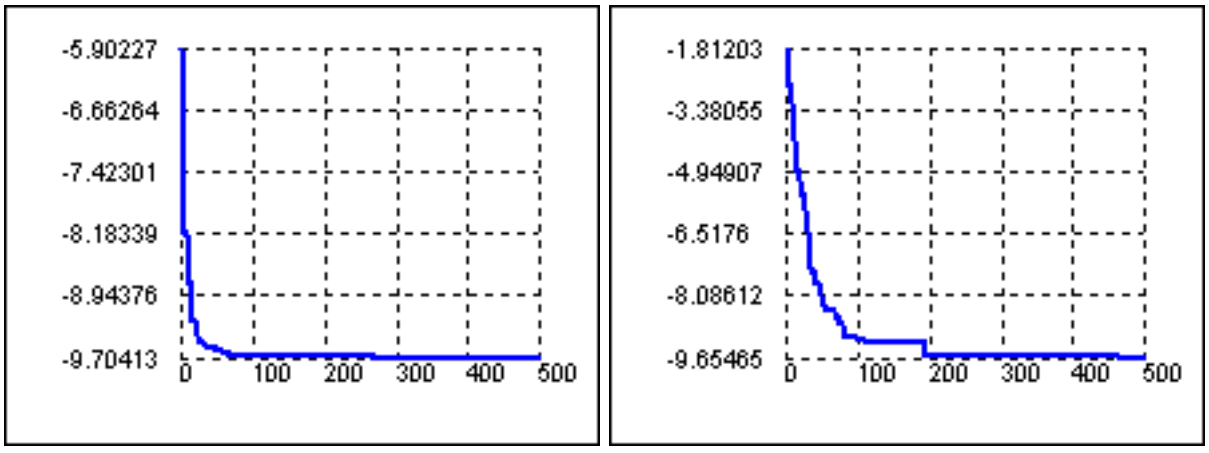


Figura 5.9: Gráficas del Mejor en la Historia para la función de Michalewicz con $m = 1$ y $m = 100$

5.3.9. Función de Katsuura

Para la ejecución del algoritmo genético, el valor seleccionado para β fue 30, se tomó como dimensión 20 y los valores de las variables se encuentran entre -1000 y 1000.

La solución de esta función se encuentra fácilmente, y aunque con valores de las variables muy cercanos a 0, el valor de la función puede ser muy grande, del orden de 10^{40}. Es posible llegar a la respuesta óptima, con 1000 individuos y 300 generaciones.

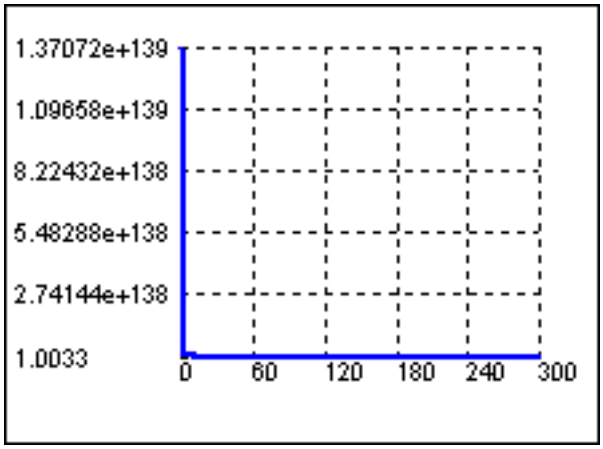


Figura 5.10: Gráfica del Mejor en la Historia para la función de Katsuura

5.3.10. Análisis

Como se puede observar en los resultados anteriores, los algoritmos genéticos son una buena alternativa en la búsqueda de óptimos para funciones que cuentan con un alto número de variables y presentan muchos óptimos locales.

Para funciones con un solo óptimo como el modelo esférico (5.1) y la función de Schwefel 1.2 (5.3), la aplicación implementada utilizando UNGenético 2.0 encuentra la solución fácilmente. Sin embargo, para funciones con muchos óptimos locales y que se encuentren alejados del óptimo global como la función de Rastrigin (5.4), la función de Schwefel (5.7) y la función de Michalewicz (5.8), la aplicación tiende a encontrar primero los óptimos locales, lo que ocasiona un mayor tiempo de proceso en la búsqueda de la solución.

Capítulo 6

EJEMPLO DE APLICACIÓN CALIBRACIÓN DE UN MODELO HIDROLÓGICO

Una de las aplicaciones de los algoritmos genéticos radica en la búsqueda de parámetros para un modelo, este proceso se denomina *calibración del modelo*. En este tipo de aplicaciones la función a minimizar es el error entre los valores obtenidos por la simulación del modelo y los valores medidos o reales del sistema.

Como ejemplo de aplicación de UNGenético 2.0 se presenta la calibración de un modelo de lluvia-escorrentía para una cuenca hidrológica definida, conocido como modelo de Thomas[1]. La sección 6.1 hace una breve reseña de este modelo, una presentación más detallada puede obtenerse en [12]. La descripción de los datos de la cuenca utilizada para la simulación se hace en la sección 6.2, mientras que en las secciones 6.3 y 6.4 se muestra la utilización de la librería y los resultados obtenidos, respectivamente.

[1]Esta aplicación hace parte del desarrollo de la tesis de maestría en Recursos Hídricos en la Universidad Nacional de Colombia, del Ing. Mario Castro

6.1. Modelo de Thomas

Uno de los objetivos de la hidrología es modelar el comportamiento de una cuenca hidrográfica en función de sus características climáticas, geomorfológicas y cobertura vegetal, y determinar los caudales de sus ríos en un periodo de tiempo. Para este propósito se utilizan modelos de lluvia-escorrentía, que relacionan la precipitación, evapotranspiración y características de la cuenca, con el caudal de la misma.

Un modelo de lluvia-escorrentía existente es el modelo de Thomas, el cual utiliza cuatro parámetros que representan algunas características de la cuenca, estos son:

- a: tendencia de que ocurra escorrentía antes de que el suelo se encuentre completamente saturado. Se encuentra en el intervalo $(0, 1]$
- b: límite superior de la suma de la evapotranspiración y el contenido de humedad del suelo. Se encuentra en el intervalo $[10, 500]$
- c: fracción de escorrentía proveniente del agua subterránea. Se encuentra en el intervalo $[0, 0{,}9]$
- d: valor recíproco del tiempo de residencia del agua subterránea. Se encuentra en el intervalo $[0, 1]$

También utiliza dos condiciones iniciales:

- Sw_0: contenido inicial de humedad en el suelo. Se encuentra en el intervalo $[0, 500]$
- Sg_0: almacenamiento inicial de agua subterránea. Se encuentra en el intervalo $[0, 500]$

A continuación se presenta la ecuación del modelo que relaciona el caudal Q, en un período de tiempo determinado, con la precipitación P, la evapotranspiración E, los parámetros a, b, c, d, y las condiciones iniciales Sw_0 y Sg_0:

$$Q = (1-c)\left[P + Sw_0 - \frac{P + Sw_0 + b}{2a} + \sqrt{\left(\frac{P + Sw_0 + b}{2a}\right)^2 - \frac{(P + Sw_0)}{a}}\right] + \frac{d}{d+1}\left\{c\left[P + Sw_0 - \frac{P + Sw_0 + b}{2a} + \sqrt{\left(\frac{P + Sw_0 + b}{2a}\right)^2 - \frac{(P + Sw_0)}{a}}\right] + Sg_0\right\} \tag{6.1}$$

Esta ecuación se puede simplificar en pequeñas ecuaciones que brindan datos adicionales de la cuenca:

- Cálculo del agua disponible

$$W = P + Sw_0 \tag{6.2}$$

- Cálculo de la variable Y

$$Y = \frac{W+b}{2a} - \sqrt{\left(\frac{W+b}{2a}\right)^2 - \frac{W \cdot b}{a}} \tag{6.3}$$

- Cálculo del contenido de humedad del suelo

$$Sw = Y \cdot e^{-E/b} \tag{6.4}$$

- Cálculo de la escorrentía directa

$$Ro = (1-c)(W-Y) \tag{6.5}$$

- Cálculo de la recarga de agua subterránea

$$Rg = c(W-Y) \tag{6.6}$$

- Cálculo del almacenamiento de agua subterránea

$$Sg = \frac{Rg + Sg_0}{d+1} \tag{6.7}$$

- Cálculo del caudal subterráneo

$$Qg = d \cdot Sg \tag{6.8}$$

- Cálculo del caudal a la salida de la cuenca

$$Q = Ro + Qg \tag{6.9}$$

La calibración del modelo de Thomas se logra cuando se lleguen a establecer los valores de a, b, c, d, Sw_0 y Sg_0 tales que al ingresar la precipitación medida *P* y la evapotranspiración medida *E*, se obtenga un caudal simulado por el modelo *Qfinal* igual al caudal observado en la cuenca *Qreal*.

6.2. Descripción de los datos utilizados

Los datos de entrada al modelo son las series de los valores medios mensuales de caudales, valores totales mensuales de precipitación y valores totales mensuales de evapotranspiración registrados en la estación Berlín del IDEAM entre los años 1981 y 1996, para la cuenca del río Jordán, ubicada en el departamento de Santander, que tiene un área de $50km^2$.

En la tabla 6.1 se encuentran los promedios mensuales en los 15 años de cada una de las variables mencionadas, mientras que en el anexo E se encuentran los archivos originales suministrados por el IDEAM. En la figura 6.1 se puede ver el hidrograma de caudales medios mensuales de la estación Berlín.

Mes	Precipitación $[mm]$	Evapotranspiración $[mm]$	Caudal $[m^3/s]$	Caudal $[mm]$
Enero	13.96	113.80625	0.6693125	34.69716
Febrero	28.4375	94.478125	0.6238125	32.33844
Marzo	39.0875	104.509375	0.6534375	33.8742
Abril	97.15625	88.9125	0.860625	44.6148
Mayo	92.025	85.36875	1.2293125	63.72756
Junio	55.26875	86.6125	0.9376875	48.60972
Julio	56.8625	83.6625	0.7615625	39.4794
Agosto	74.19375	91.85625	0.8229375	42.66108
Septiembre	94.41625	90.806875	1.0508125	54.47412
Octubre	81.87625	96.026875	1.3018125	67.48596
Noviembre	42.74875	91.88375	1.207375	62.59032
Diciembre	21.0025	97.9375	0.831125	43.08552

Tabla 6.1: Valores medios mensuales de precipitación, evapotranspiración y caudal de la cuenca Jordán

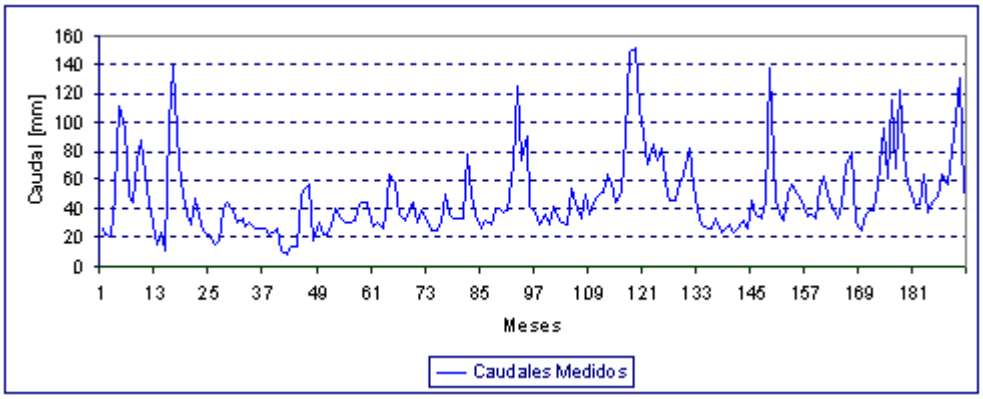


Figura 6.1: Hidrograma de caudales medios mensuales

6.3. Implementación del modelo utilizando UNGenético 2.0

Se desarrolló una aplicación llamada *thomas* para realizar la calibración del modelo de Thomas utilizando librería UNGenético 2.0 y el entorno gráfico de wxWindows. En los siguientes apartados se describe el proceso de elaboración de dicha aplicación.

6.3.1. Clase ModeloThomas

La clase que define el algoritmo genético, derivada de *AlgoritmoGenetico*, indispensable en cualquier proyecto de optimización con UNGenético 2.0, se denominó *ModeloThomas*. Los miembros de esta clase fueron definidos para cumplir distintas funciones, entre ellas:

- Se sobrecargó la función virtual **inicializarParametros()**, donde se modifican los valores de *m_TamanoPoblacion* y *m_GeneracionMaxima* con el fin de establecer el número de individuos por generación en 300 y el número máximo de generaciones del algoritmo en 1000.
- Se sobrecargó la función virtual **definirOperadores()**, donde se asignan los operadores del algoritmo. La tabla 6.2 resume los operadores asignados para el modelo
- Se sobrecargó la función virtual **codificacion()** para realizar la codificación de este mode-

Tipo de Operador	Operador Escogido
Operador de Probabilidad	*OperadorProbabilidadLineal*
Operador de Selección	*OperadorSeleccionEstocasticaRemplazo*
Operador de Parejas	*OperadorParejasAdyacentes*
Operador de Reproducción	*OperadorReproduccionMejoresEntrePadresEHijos*
Operador de Adaptación	*OperadorAdaptacionElitismo*
Operador de Finalización	*OperadorFinalizacionOffline*
Operador de Mutación	*OperadorMutacionRealUniforme* con probabilidad de mutación igual a 0.07
Operador de cruce	*OperadorCruceRealIntermedioExtendido*

Tabla 6.2: Operadores utilizados en la aplicación *thomas*

lo. El individuo utilizado contiene seis genes reales que representan los cuatro parámetros del modelo y los dos valores iniciales del mismo, como se muestra en la figura 6.2. Su implementación se realizó utilizando la macro *ADICIONAR_GENREAL*.

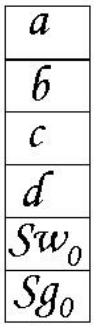


Figura 6.2: Individuo utilizado en la aplicación *thomas*

- Se sobrecargó la función virtual **objetivo()**, donde se calcula el valor de los caudales de la cuenca utilizando los parámetros obtenidos por el algoritmo genético. El valor de la función objetivo corresponde al error cuadrático medio entre los caudales medidos *(Qreal[i])* y los calculados por el modelo *(Qfinal[i])* para cada periodo de tiempo i, $i \in [1, n]$.

$$FO = \frac{1}{n}\sqrt{\sum_{i=1}^{n}(Qreal_i - Qfinal_i)^2} \tag{6.10}$$

- Se definió la función **mostrar(Individuo& Ind, int generacion)**, para desplegar en la página *"Proceso"* la información genética de cada individuo *Ind*, correspondiente a los parámetros del modelo y al valor de su función de evaluación.

- Se definió la función **GuardarParametros(Individuo& Ind, int generacion)**, que guarda en el archivo de texto *"parametros.txt"* los parámetros del mejor individuo de cada generación del algoritmo encontrados por UNGenético 2.0, en el formato requerido para visualizar dichos datos en UNGenético Graphics.

- Se definió la función **GuardarCaudales(Individuo& Ind, int generacion)**, que guarda en el archivo de texto *"caudales.txt"* los caudales calculados por UNGenético 2.0 para cada periodo en cada generación del algoritmo.

- Las variables *a, b, c, d, swo, sgo* de tipo *double*, que corresponden a los parámetros del modelo, y son los valores decodificados de cada uno de los genes del individuo.

- Los apuntadores *caudal* y *par* hacen referencia a los archivos de texto *"caudales.txt"* y *"parametros.txt"* respectivamente.

6.3.2. Clase MiVentana

Se creó la clase *MiVentana* derivada de *AGVentana*, que construye una ventana adicional cuando se ejecuta el programa y permite al usuario indicar la ruta y el nombre del archivo de texto donde se encuentran los datos de la cuenca que alimentan al algoritmo.

El archivo de texto debe tener la siguiente configuración: en la primera fila se debe indicar el número de periodos para los que se tienen datos, en la segunda fila deben especificarse los valores de caudal medido, en la tercera los valores de precipitación y en la cuarta los valores de evapotranspiración para cada periodo. Cada valor debe estar separado por el caracter tabulación.

6.4. Resultados

El programa se ejecutó utilizando todos los valores mensuales medidos y los valores promedio mensuales en los 15 años.

6.4.1. Ejecución empleando los valores medios mensuales

El algoritmo tiende a encontrar una solución óptima rápidamente; el valor de la función objetivo no disminuye significativamente después de varias generaciones, esto se puede notar en la figura 6.3. Cabe destacar que algunos valores de los parámetros para el mejor individuo de cada generación varían considerablemente durante la ejecución del algoritmo. La razón es que se pueden tener muchas combinaciones de parámetros con funciones objetivo similares como se puede observar en la figura 6.4 y en la tabla 6.3. Además los parámetros con más variaciones *(a, Swo)* no afectan en gran medida la respuesta del modelo, es decir, el modelo es menos sensible a variaciones de estos parámetros que a variaciones de *b*, *c*, *d* y *Sgo*.

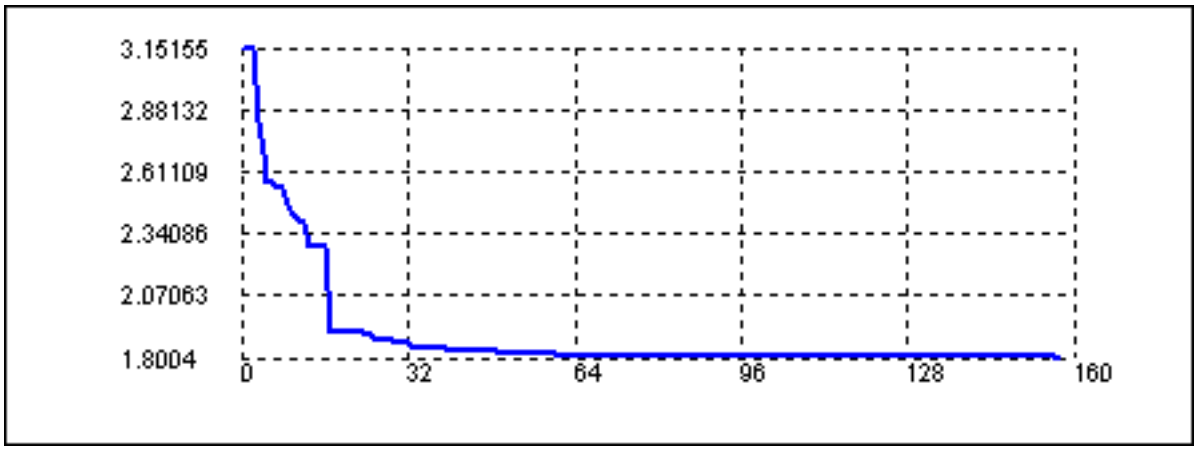


Figura 6.3: Gráfica del Mejor en la Historia en la calibración del modelo de Thomas utilizando los valores promedios mensuales con 300 Individuos

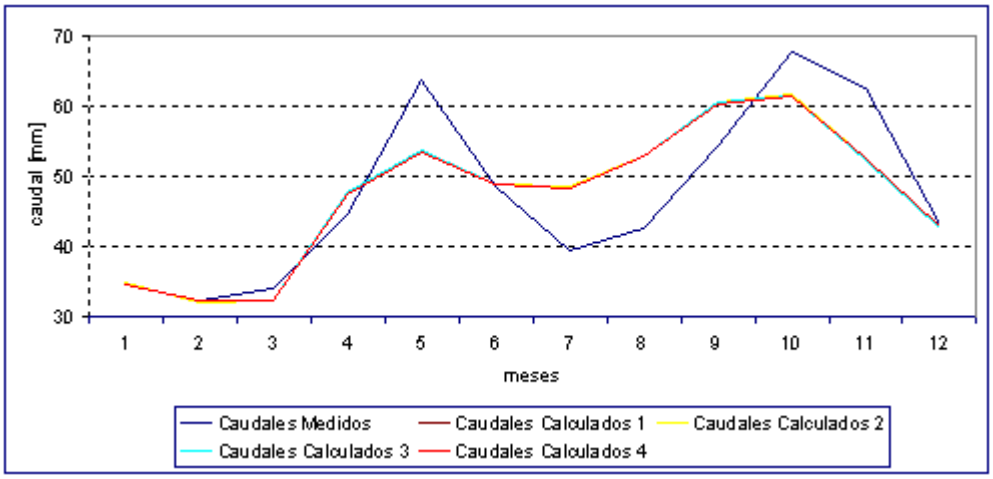


Figura 6.4: Gráfica de comparación entre el caudal medido y los caudales calculados utilizando los valores promedios mensuales

a	b	c	d	Swo	Sgo	Función Objetivo
0.240356	12.8816	0.899992	0.252891	11.5621	149.755	1.80107
0.484136	12.6345	0.899999	0.25561	1.82814	159.793	1.80018
0.60883	12.0472	0.9	0.250937	12.4764	152.168	1.80118
0.191053	13.5839	0.899995	0.262816	8.57773	149.712	1.80059

Tabla 6.3: Parámetros de la calibración del modelo de Thomas utilizando los valores promedios mensuales

6.4.2. Ejecución empleando todos los valores mensuales

Al ejecutar el algoritmo genético utilizando todos los valores mensuales, al igual que en el caso anterior la respuesta se obtiene rápidamente como se puede observar en la figura 6.5.

Se podría pensar que la búsqueda de los parámetros utilizando un mayor número de datos debería proporcionar una mejor calibración del modelo, sin embargo estos parámetros no deberían cambiar considerablemente respecto a los encontrados utilizando los valores promedio mensuales.

Al observar en la tabla 6.4, los parámetros *b, c* y *d* son muy similares a los calculados en el caso anterior. En este caso es interesante notar que el valor de *a* no varía, manteniéndose igual

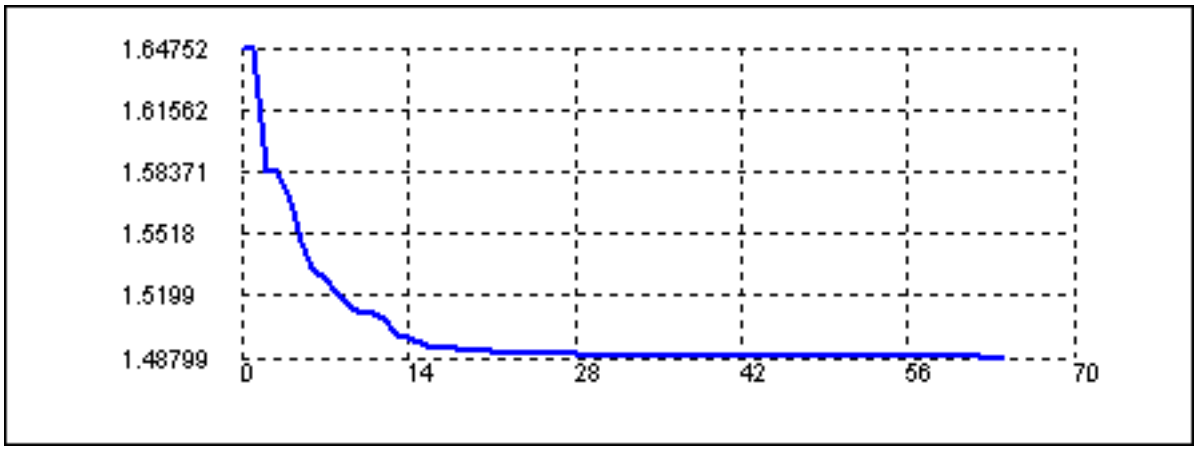


Figura 6.5: Gráfica del Mejor en la Historia en la calibración del modelo de Thomas utilizando todos los valores mensuales con 300 Individuos

a 1, mientras que los valores de *Swo* y *Sgo* presentan variaciones considerables.

a	b	c	d	Swo	Sgo	Función Objetivo
1	10.9849	0.9	0.256967	41.2895	68.2737	1.48855
0.999967	10.8379	0.899792	0.267996	9.19845	107.842	1.48796
0.999995	10.8391	0.899903	0.268329	46.2356	64.6287	1.4881
0.999999	10.8206	0.9	0.268214	18.3902	96.7262	1.48797
0.984003	10.9517	0.9	0.274216	39.3241	66.2058	1.48846
1	10.8264	0.899894	0.26843	1.37322	116.386	1.48795

Tabla 6.4: Parámetros de la calibración del modelo de Thomas utilizando todos los valores mensuales

En la figura 6.6 se puede ver que los caudales calculados utilizando los parámetros obtenidos por el algoritmo genético para el modelo son una buena aproximación de los caudales reales. El problema radica en que al existir datos muy alejados del valor promedio, el modelo de Thomas es incapaz de simular dichos valores.

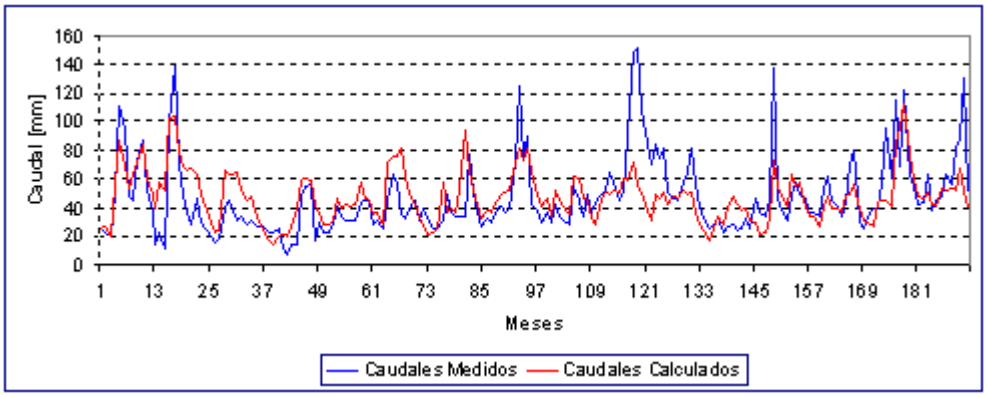


Figura 6.6: Gráfica de comparación entre el caudal medido y los caudales calculados

6.4.3. Análisis

Al existir diferencias entre los valores calculados con el modelo implementado con UNGenético 2.0, y los valores reales del sistema, no se puede afirmar que el método de calibración sea la única fuente de error. Es conveniente aclarar que un modelo es sólo la representación de un sistema real y por lo tanto no puede incluir todas las variables que lo afectan. También puede pensarse que el modelo empleado no sea el mejor para el caso escogido, o que en el proceso de toma de mediciones en el sistema ocurran eventos que afecten la medida y por tanto ésta contenga errores que la alejen de la realidad.

Capítulo 7

EJEMPLO DE APLICACIÓN PROGRAMACIÓN DE MANTENIMIENTO DE MÁQUINAS

Uno de los problemas con los que se enfrenta la industria es definir la programación de mantenimiento de sus máquinas, de tal forma que ésta afecte en la menor medida posible la producción de la fábrica y su costo sea óptimo.

En este capítulo se presenta la utilización de UNGenético 2.0 para encontrar la programación de mantenimiento de la fábrica INDUPLAST S.A.[1]. En esta aplicación se busca hallar una programación óptima en términos de simultaneidad de mantenimientos y distribución adecuada a lo largo de un periodo de tiempo de un año, sin tener en cuenta los costos de la misma[2].

En la sección 7.1 se realiza una descripción de los datos de entrada a la aplicación para realizar la programación de mantenimiento. En la sección 7.2 se describen las restricciones que se tuvieron en cuenta para encontrar una programación de mantenimiento óptima y en la sección 7.3 se especifican los datos de salida de la aplicación. La implementación de la aplicación utilizando UNGenético 2.0 se encuentra en la sección 7.4 y los resultados en la sección 7.5.

[1]Los datos utilizados fueron suministrados por el Ing. Pedro Nel Martínez y hacen parte de su tesis de maestría en Automatización Industrial

[2]La relación entre la programación de mantenimiento y los costos de la misma se está realizando en la tesis de maestría en Automatización Industrial del Ing. Pedro Nel Martínez

7.1. Descripción de los datos de entrada

Los datos empleados fueron suministrados por el Ing. Pedro Nel Martínez y hacen parte del desarrollo de su tesis de maestría en Automatización Industrial en la Universidad Nacional de Colombia. Sin embargo, la aplicación se puede utilizar con datos de cualquier empresa siguiendo los parámetros establecidos.

Los datos deben encontrarse en un archivo con formato de texto plano el cual debe contener en la primera fila el número de máquinas, el número de individuos, el número máximo de generaciones y la probabilidad de mutación a utilizar en el algoritmo, separados por tabulación. En las filas siguientes deben encontrarse los datos de cada máquina. Los datos utilizados se pueden ver en el anexo F. En cada fila correspondiente a una máquina deben especificarse el código del equipo, el código de importancia, el código de trabajo y la clasificación del tipo de mantenimiento de la máquina[3]. Éstos se explican a continuación:

7.1.1. Código del Equipo

La codificación es un problema propio de cada planta, sin embargo existen principios universales para su diseño. En el sistema de codificación adoptado, la primera letra del código representa el tipo de máquina y los demás caracteres corresponden a la numeración respectiva dentro de la empresa. Para el problema planteado únicamente es tenido en cuenta el primer caracter, sin importar la longitud del código.

7.1.2. Código de importancia

Es un valor entero entre 1 y 10. Define la importancia de la máquina dentro del proceso de producción. Entre mayor sea su valor, implica una mayor importancia.

[3]Sistema de codificación adoptado en el proyecto de grado de la maestría en Automatización Industrial del Ing. Pedro Nel Martínez

7.1.3. Código de trabajo

Es un valor entero entre 1 y 6. Define la "'cantidad"' de trabajo realizado por la máquina dentro del proceso de producción. A mayor valor, mayor cantidad de trabajo es la realizada por la máquina.

7.1.4. Clasificación del tipo de mantenimiento

En el problema planteado, existen 5 tipos de mantenimiento que se pueden realizar en una máquina. Estos son:

- Mantenimiento Anual (A).
- Mantenimiento Semestral (S).
- Mantenimiento Trimestral (T).
- Mantenimiento Mensual (M).
- Mantenimiento Semanal (W).

En razón a que cada máquina no requiere los mismos tipos de mantenimiento, su clasificación se puede representar en un sistema binario, donde "1"significa que el mantenimiento se debe realizar y "0"que no. En la figura 7.1 se encuentra la representación binaria para la clasificación de mantenimiento de una máquina. Según esta codificación, la clasificación de mantenimiento puede variar entre 0 y 31.

7.2. Restricciones

- En lo posible, no se deben presentar mantenimientos anuales y semestrales simultáneos para dos o más maquinas de la misma categoría.

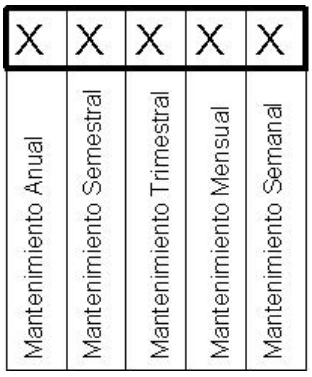


Figura 7.1: Representación binaria de la clasificación de mantenimiento

- En lo posible, no se deben presentar mantenimientos anuales y semestrales de las máquinas en una misma semana, deben distribuirse uniformemente en las 48 semanas del año[4].
- Si una máquina presenta varios tipo de mantenimiento, estos deben realizarse todos en la misma semana del mes.
- Tiempo entre mantenimientos:
  - Los mantenimientos semestrales deben estar separados 24 semanas.
  - Los mantenimientos trimestrales deben estar separados 12 semanas.
  - Los mantenimientos mensuales deben estar separados 4 semanas.

7.3. Datos de salida de la aplicación

Se requiere obtener un archivo en formato de hoja de cálculo de excel (.xls), en el cual se pueda visualizar la programación de mantenimiento obtenida por el algoritmo genético. Cada fila contiene la programación de una máquina. La primera columna contiene el código de la máquina, la segunda, el tipo de mantenimiento a realizar en la primera semana para la máquina, la tercera, el tipo de mantenimiento a realizar en la segunda semana, y así sucesivamente hasta completar las 48 semanas del año. El tipo de mantenimiento se especifica con los códigos presentados en la sección 7.1.4.

[4]Un año tiene 52 semanas pero para efectos de la programación del mantenimiento, un mes tiene 4 semanas, y el año 48 semanas

7.4. Implementación utilizando UNGenético 2.0

Para encontrar una programación de mantenimiento utilizando la librería UNGenético 2.0 se creó una aplicación llamada *mantenimiento*, en la cual se utiliza el entorno gráfico de wxWindows.

La aplicación muestra el progreso de la función objetivo de la mejor solución encontrada cada cinco iteraciones del algoritmo. Al finalizar la ejecución son guardados los siguientes archivos:

- ProgrSemanal.xls: Guarda la mejor programación encontrada por el algoritmo genético en el formato especificado en la sección 7.3.
- Individuos.txt: Guarda el gen completo y la función objetivo del mejor individuo encontrado cada cinco iteraciones.
- Salidas.txt: Contiene el resumen de las medidas de desempaño de todas las generaciones del algoritmo.
- General.txt: Guarda el resumen de cada ejecución del algoritmo genético adicionando los siguientes datos: número de individuos, número de generaciones, número de máquinas, mejor función objetivo obtenida, generación en que se obtuvo la mejor función objetivo, probabilidad de mutación utilizada y tiempo de proceso.

7.4.1. Clase AGMantenimiento

La clase que define el algoritmo genético, derivada de *AlgoritmoGenetico*, indispensable en cualquier proyecto de optimización en UNGenético 2.0, se denominó *AGMantenimiento*, la cual se implementó con los siguientes miembros:

- Se sobrecargó la función virtual **inicializarParametros()**, donde se modifican los valores de *m_TamanoPoblacion, m_GeneracionMaxima* que son establecidos por el usuario desde el archivo de entrada a la aplicación.

- Se sobrecargó la función virtual **definirOperadores()**, donde se asignan los operadores del algoritmo. Se utilizaron los operadores que se muestran en la tabla 7.1. La probabilidad de mutación puede ser definida por el usuario desde el archivo de entrada.

Tipo de Operador	Operador Escogido
Operador de Probabilidad	*OperadorProbabilidadProporcional*
Operador de Selección	*OperadorSeleccionEstocasticaRemplazo*
Operador de Parejas	*OperadorParejasAdyacentes*
Operador de Reproducción	*OperadorReproduccionMejoresEntrePadresEHijos*
Operador de Adaptación	*OperadorAdaptacionElitismo*
Operador de Finalización	Ninguno
Operador de Mutación	*OperadorMutacionArregloEntero*. La probabilidad de mutación es seleccionable por el usuario

Tabla 7.1: Operadores utilizados en la aplicación *mantenimiento*

- Se sobrecargó la función virtual **codificacion()**. Para realizar la codificación de este problema, el individuo utilizado tiene un solo gen denominado *SemMantenimiento* de tipo arreglo de valores enteros y su dimensión es el número de máquinas que se van a utilizar, como se muestra en la figura 7.2. Cada valor del arreglo se encuentra en el intervalo $[0, 47]$, que corresponde a la semana del año donde se realizarán los mantenimientos anuales, semestrales, trimestrales y mensuales. La adición de este gen se hace utilizando la macro *ADICIONAR_GENARREGLO_ENTERO*.

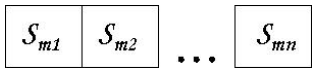


Figura 7.2: Individuo utilizado en la aplicación *mantenimiento*

Como los valores del gen solo informan una semana del año donde se van a presentar todos los tipos de mantenimiento para una máquina, los mantenimientos que se deben realizar varias veces en el año van a estar distribuidos uniformemente, es decir, el otro mantenimiento semestral se va a encontrar separado exactamente 24 semanas de la semana que indique el gen, los trimestrales separados 12 semanas y los mensuales 4 semanas.

Con la anterior codificación las restricciones nombradas en la sección 7.2 se van a respetar y el tiempo de búsqueda es considerablemte menor que si se tomaran genes para cada

tipo de mantenimiento.

- Se sobrecargó la función virtual **objetivo()**. Calcula la función objetivo de un individuo. Se inicia encontrando el número máximo de mantenimientos que pueden presentarse en una semanana. Para cada semana que exceda este número, se incrementa la función objetivo proporcionalmente a la cantidad en que excedió el máximo. Si se presentan mantenimientos anuales y/o semestrales en la misma semana para máquinas de la misma categoría, la función objetivo es incrementada proporcionalmente a la suma de la prioridad de las dos máquinas, definida como la multiplicación del código de importancia y el código de trabajo.

- Se definió la función **mostrar(Individuo& Ind)**, para desplegar en la página *"Proceso"* el valor de la función objetivo del Individuo *Ind*.

- Se definió la funcion **guardar(Individuo& Ind, FILE *pArch, int generacion)**, que guarda en el archivo de texto *"Individuos.txt"* el gen completo y la función objetivo del individuo *Ind*.

- Se definió la función **guardarProgramacion()**, que guarda en el archivo *"ProgrSemanal.xls"* la mejor programación encontrada en un formato tabular.

- La variable *SemMantenimiento* de tipo *ArregloEntero*, la cual contiene los valores decodificados de la información genética de un Individuo y corresponde a la semana del año cuando se van a presentar todos los tipos de mantenimiento.

7.4.2. Clase MiVentana

Se creó la clase *MiVentana*, derivada de *AGVentana*, la cual construye una ventana adicional cuando se ejecuta el programa y permite al usuario indicar la ruta y el nombre del archivo de texto donde se encuentran los datos de las máquinas de la fábrica, cuyo formato se presentó en la sección 7.1.

7.5. Resultados

Con los datos suministrados se buscó una programación de mantenimiento óptima para 48 y 67 máquinas. A continuación se detallan los resultados obtenidos para los dos casos.

7.5.1. Programación de mantenimiento para 48 máquinas

Con 100 individuos el algortimo genético tarda aproximadamente 80 generaciones en encontrar la solución óptima cuya función objetivo es igual a cero. Al aumentar el número de individuos a 400, la mejor solución se encuentra en 50 generaciones (figura 7.3).

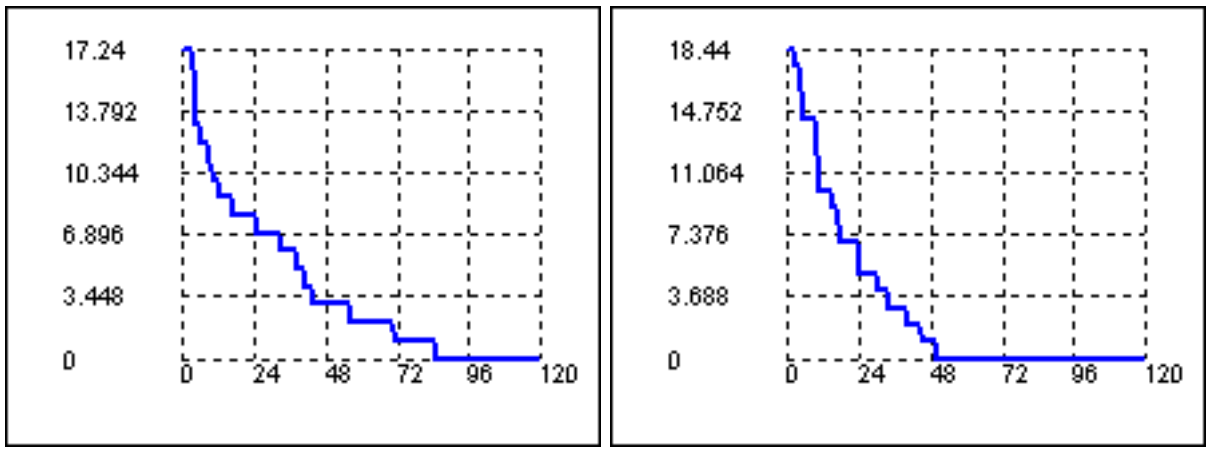


Figura 7.3: Gráfica del mejor individuo en la historia para 48 máquinas utilizando 100 y 400 individuos

7.5.2. Programación de mantenimiento para 67 máquinas

Al utilizar todas las máquinas que se encuentran en el archivo suministrado, con 100 individuos el algortimo genético tarda aproximadamente 80 generaciones en encontrar la mejor solución obtenida cuya función objetivo es igual a 1.8, este valor se debe a dos máquinas de la misma clase con mantenimiento semestral en la misma semana, cuyos códigos de importancia son muy bajos lo cual es una situación admisible. Al aumentar el número de individuos, la mejor solución se encuentra en un menor número de generaciones, pero su función objetivo no disminuye (ver

figura 7.4). La programación obtenida por el algoritmo genético se puede observar en el anexo G

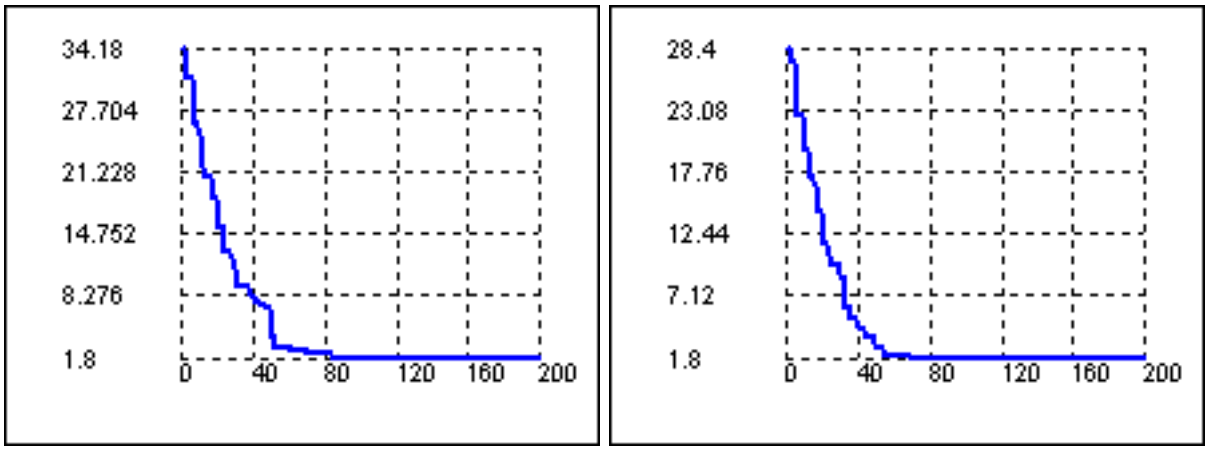


Figura 7.4: Gráfica del mejor individuo en la historia para 67 máquinas utilizando 100 y 1000 individuos

7.5.3. Análisis

Al utilizar UNGenético 2.0 para la obtención de una programación de mantenimiento óptima, se obtienen buenos resultados en una manera rápida. Con la codificación utilizada es fácil encontrar una buena programación, donde los mantenimientos se deben realizar de una forma distribuida a lo largo del año, teniendo en cuenta la no simultaneidad de mantenimientos considerados complicados, es decir, anuales o semestrales, en máquinas de alta prioridad para la empresa. En este ejemplo de aplicación no se definió el costo de llevar a cabo cada programación de mantenimiento, pues esa parte del problema se está resolviendo en la tesis de maestría en Automatización Industrial del Ing. Pedro Nel Martínez.

Capítulo 8

CONCLUSIONES Y TRABAJOS FUTUROS

- UNGenético 2.0 está enfocada hacia el usuario, es decir, las modificaciones realizadas a la primera versión se llevaron a cabo con el fin de facilitar su utilización. Con este propósito se crearon distintos medios opcionales de ayuda tanto al interior de la librería como en herramientas externas. Todas ellas evitan al usuario no experimentado el manejo directo de las clases de la librería y sus componentes procurando que éste sólo deba conocer a fondo su modelo a optimizar, sin tener conocimientos muy profundos en programación en C++ y en algoritmos genéticos.
- Se aumentaron las opciones en la forma de codificar y de manejar la información de un algoritmo genético por medio de la implementación de nuevas clases correspondientes a nuevos tipos de genes y operadores.
- El usuario de UNGenético 2.0 puede obtener un soporte al momento de utilizar la librería, gracias a la creación de una documentación amplia que describe todos sus componentes y un manual de usuario que expone las distintas formas de hacer uso de la misma.
- Se facilita el uso del entorno gráfico suministrado por la librería wxWindows, por medio de la creación de clases genéricas que permiten construir una interfaz gráfica compatible con diferentes sistemas operativos para las aplicaciones realizadas utilizando UNGenético 2.0.
- Se demostró la utilidad de UNGenético 2.0 en problemas de diferentes áreas de la ciencia mediante la implementación de ejemplos en los que se utiliza la librería obteniendo

un buen desempeño medido en la calidad de la solución y en el tiempo requerido para su obtención.

8.1. Trabajos futuros

- La librería UNGenético 2.0 amplía la cantidad de operadores genéticos y tipos de genes con respecto a la primera versión. Sin embargo, debido a la cantidad de operadores y tipos de genes existentes en la literatura de algoritmos genéticos, esta colección aún puede ser ampliada de modo que ofrezca nuevas posibilidades en la solución de problemas específicos.
- UNGenético 2.0 es una base útil en el desarrollo de aplicaciones utilizando algoritmos genéticos. Ésta ofrece una buena perspectiva para la investigación de soluciones a problemas complejos.
- UNGenético 2.0 puede utilizarse como herramienta de investigación en algoritmos genéticos. Mediante esta librería se puede realizar la determinación de los parámetros de los algoritmos y de cada operador genético que mejoran el desempeño del algoritmo.

BIBLIOGRAFÍA

[1] HEESCH Dimitri van, *Doxygen 1.3.6 manual [en línea]*, <http://www.doxygen.org>, 2004.

[2] BRAEM Franky, *wxWindows 2, programming cross-platform GUI applications in C++ [en línea]*, <http://www.wxwindows.org>, 2002.

[3] MARTÍNEZ José J. and ROJAS Sergio, *Introducción a la informática evolutiva: Un nuevo enfoque para resolver problemas de ingeniería*, primera edición ed., Universidad Nacional de Colombia. Facultad de Ingeniería. Unidad de Publicaciones - Ingeniería, Bogotá D.C., Colombia, 1999.

[4] SOULIÉ Juan, *The cplusplus.com tutorial, complete C++ language tutorial [en línea]*, The C++ Resources Network, <http://www.cplusplus.com>, 2002.

[5] SMART Julian, ROEBLING Robert, ZEITLIN Vadim, and DUNN Robin, *wxWindows 2.4.0: A portable C++ and python GUI toolkit [en línea]*, <http://www.wxwindows.org>, 2003.

[6] REFSON Keith, *teTEX, a documentation guide [en línea]*, <http://www.tug.org/tetex>, 1999.

[7] LOZANO Manuel, *Aplicación de técnicas basadas en lógica difusa para la mejora del comportamiento de los algoritmos genéticos con codificación real*, Ph.D. thesis, Universidad de Granada. E.T.S. de Ingeniería Informática. Departamento de Ciencias de la Computación e Inteligencia Artificial, Granada, España, 1996.

[8] LESLIE Martin, *C programming reference introduction [en línea]*, <http://www.kdevelop.org>, 1996.

[9] Microsoft Corporation, *MSDN library visual studio 6.0 [CD]*, 1998.

[10] DUARTE Oscar Germán, *UNGENÉTICO: Una librería en C++ de algoritmos genéticos con codificación híbrida*, Universidad Nacional de Colombia. Facultad de Ingeniería, Bogotá D.C., Colombia, 2002.

[11] BRACHET Pascal, *The kile handbook*, 2003.

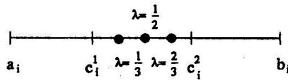
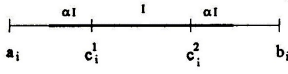
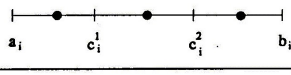
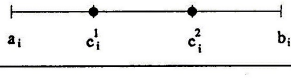
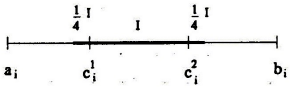
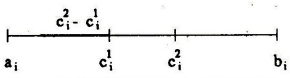
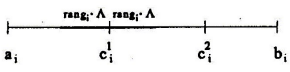
[12] OLARTE Rafael, *Herramientas para la implementación de algoritmos genéticos en ingeniería civil con enfasis en hidroinformática*, pp. 70–100, Pontificia Universidad Javeriana, Bogotá D.C., Colombia, 2003.

[13] NOLDEN Ralf, *The kDevelop programming handbook [en línea]*, <http://www.kdevelop.org>, 2001.

[14] DE CASTRO Rodrigo, *El universo latex*, segunda edición ed., Universidad Nacional de Colombia, Bogotá D.C., Colombia, 2003.

ANEXO A

Intervalos de definición para operadores de cruce de genes de tipo real

Tipo de Cruce	*Gráfica*
Aritmético	
BLX-α	
Lineal	
Discreto	
Extendido	
Heurístico	
Lineal BGA	
Plano	

ANEXO B

Valores por defecto de la clase AlgoritmoGenetico

Miembro	Valor
m_GeneracionMaxima	100
m_TamanoPoblacion	10
m_IndicadorUsarAdaptacion	true
m_IndicadorInicializarPoblacionAleatoria	true
m_IndicadorMaximizar	false
m_IndicadorArchivo	true
m_IntervaloSalvar	1
m_NombreArchivo	"salidas.txt"
m_IndicadorMostrar	false
m_IndicadorMostrarMejorEnHistoria	true
m_IndicadorMostrarGeneracionMejorHistorico	true
m_IndicadorMostrarMejorEnGeneracion	true
m_IndicadorMostrarPeorEnGeneracion	true
m_IndicadorMostrarMedia	true
m_IndicadorMostrarDesviacion	true
m_IndicadorMostrarOnLine	true
m_IndicadorMostrarOffLine	true

ANEXO C

Operadores por defecto de UNGenético 2.0

Operador	Operador por defecto
OperadorProbabilidad	OperadorProbabilidadLineal
OperadorSeleccion	OperadorSeleccionEstocasticaRemplazo
OperadorParejas	OperadorParejasAdyacentes
OperadorReproduccion	OperadorReproduccionMejorPadreMejorHijo
OperadorAdaptacion	OperadorAdaptacionElitismo
OperadorFinalizacion	

Tipo de Gen	Operador	Operador por defecto
GenBool	OperadorMutacion	OperadorMutacionBoolUniforme
	OperadorCruce	OperadorCruceBoolDiscreto
GenEntero	OperadorMutacion	OperadorMutacionEnteroUniforme
	OperadorCruce	OperadorCruceEnteroPlano
GenReal	OperadorMutacion	OperadorMutacionRealUniforme
	OperadorCruce	OperadorCruceRealBLX
GenArreglo	OperadorMutacion	OperadorMutacionArreglo
	OperadorCruce	OperadorCruceArreglo

ANEXO D

Operadores de UNGenético 2.0

	Operador	**Parámetros por defecto**
OperadorProbabilidad	OperadorProbabilidadProporcional	
	OperadorProbabilidadLineal	nmin=0.5
	OperadorProbabilidadHomogenea	
OperadorSeleccion	OperadorSeleccionEstocasticaRemplazo	
OperadorParejas	OperadorParejasAleatoria	
	OperadorParejasSiguiente	
	OperadorParejasExtremos	
OperadorReproduccion	OperadorReproduccionCruceSimple	
	OperadorReproduccionDosPadresDosHijos	
	OperadorReproduccionMejorPadreMejorHijo	
	OperadorReproduccionMejoresEntrePadresEHijos	
OperadorAdaptacion	OperadorAdaptacionElitismo	
	OperadorAdaptacionNumIndividuos	NumIndivInicio = TamañoPoblacion
		NumIndivFin = TamañoPoblacion/4
	OperadorAdaptacionProbMutacion ADAPTACION_PROBMUTACION_OFFLINE	MaxProb=0.5
		FactorVariacion=0.001
		Escalon = 0.1
		MaxCont = 10
	OperadorAdaptacionProbMutacion	MaxProb=0.5

	ADAPTACION_PROBMUTACION_EXPONENCIAL	T = GeneraciónMaxima/2
OperadorFinalizacion	OperadorFinalizacionOnline	FactorVariacion=0.001
		MaxCont=30
	OperadorFinalizacionOffline	FactorVariacion=0.0005
		MaxCont=30
GenBool	OperadorMutacionBoolUniforme	
	OperadorCruceBoolPlano	
GenEntero	OperadorMutacionEnteroUniforme	
	OperadorMutacionEnteroNoUniforme	b=0.5
	OperadorMutacionEnteroMuhlenbein	Factor=0.1
	OperadorCruceEnteroPlano	
	OperadorCruceEnteroAritmetico	Lambda=0.7
	OperadorCruceEnteroBLX	Alfa=0.3
	OperadorCruceEnteroLineal	
	OperadorCruceEnteroDiscreto	
	OperadorCruceEnteroIntermedioExtendido	
	OperadorCruceEnteroHeuristico	
	OperadorCruceEnteroLinealBGA	
GenReal	OperadorMutacionRealUniforme	
	OperadorMutacionRealNoUniforme	b=0.5
	OperadorMutacionRealMuhlenbein	Factor=0.1
	OperadorCruceRealPlano	
	OperadorCruceRealAritmetico	Lambda=0.7
	OperadorCruceRealBLX	Alfa=0.3
	OperadorCruceRealLineal	
	OperadorCruceRealDiscreto	
	OperadorCruceRealIntermedioExtendido	
	OperadorCruceRealHeuristico	
	OperadorCruceRealLinealBGA	
GenArreglo	OperadorMutacionArreglo	
	OperadorCruceArreglo	

ANEXO E

Datos Hidrológicos de la cuenca Jordán

E.1. Valores medios mensuales de caudales

I D E A M - INSTITUTO DE HIDROLOGIA, METEOROLOGIA Y ESTUDIOS AMBIENTALES
SISTEMA DE INFORMACION
VALORES MEDIOS MENSUALES DE CAUDALES (m3/seg)
NACIONAL AMBIENTAL
FECHA DE PROCESO : 2004/03/23
ESTACION : 3701701 BERLIN

LATITUD 0711 N
TIPO EST LM
DEPTO SANTANDER
FECHA-INSTALACION 1972-MAY
LONGITUD 7255 W
ENTIDAD 01 IDEAM
MUNICIPIO TONA
FECHA-SUSPENSION 2001-NOV
ELEVACION 3330 m.s.n.m
REGIONAL 08 SANTANDERES
CORRIENTE JORDAN

A#O	EST	ENT	ENERO *	FEBRE *	MARZO *	ABRIL *	MAYO *	JUNIO *
1981	2	01	.486	.419	.394	.976 8	2.147 8	1.837 8
1982	2	01	.290	.448 8	.216	1.908 8	2.682 8	1.384 8
1983	2	01	.385	.297	.349	.756 7	.846 8	.736
1984	2	01	.495	.429	.434	.500	.206	.150
1985	2	01	.574	.423 6	.420	.529 6	.776 6	.633
1986	2	01	.525	.602 7	.505 6	.876 8	1.216 8	1.118 8
1987	2	01	.617	.479 6	.483	.598	.952	.702
1988	2	01	.511	.617	.563	.749	.777	.718
1989	2	01	.720	.573	.707	.563	.816	.662
1990	2	01	.713	.838	.965	.991	1.242 8	1.119

1991	2	01	1.796		1.350		1.622	8	1.417	1.565	.974
1992	2	01	.783		.572		.503		.507	.638	.449
1993	2	01	.880		.710		.660		.870	2.650 8	.880
1994	2	01	.669		.696		.645		1.036	1.192	.864
1995	1	01	.474		.698		.758		.764	1.096	1.846
1996	1	01	.791		.830		1.231		.730	.868	.931
1997	1	01	.934		.770		.716		.797	1.241 8	.780
2000	1	01	.804		.737		.604				
MEDIOS			0.692		0.638		0.654		0.857	1.230	0.928
MAXIMOS			1.796		1.350		1.622		1.908	2.682	1.846
MINIMOS			0.290		0.297		0.216		0.500	0.206	0.150

JULIO	*	AGOST	*	SEPTI	*	OCTUB	*	NOVIE	*	DICIE	*	VR ANUAL
.931		.863		1.403 8		1.678 8		.999 6		.719 6		1.07
.964		.687		.553		.907 8		.565		.457		0.92
.607		.640		.530		.606		.514		.509		0.57
.261		.262		.930 8		1.047 7		1.089 8		.340		0.51
.583		.577		.610		.830 6		.845 6		.871 6		0.64
.681		.613		.764		.869		.603		.741		0.76
.660		.630		.632		1.497 8		.973		.698		0.74
.766		1.367		2.405 8		1.435		1.736 8		.795		1.04
.585		.547		1.042		.859		.633		.954		0.72
.871		.994		1.720		2.866 8		2.922		2.116		1.45
.884		.875		1.102		1.236		1.572		1.084		1.29
.507		.544		.460		.501		.623		.504		0.55
.720		.610		.960		1.100		.950		.890		0.99
.779		.654		.833		1.342		1.542		.605		0.91
1.175		2.215		1.313		2.346		1.226		1.043		1.25
1.211		1.089		1.556		1.710 8		2.526 8		.972		1.20
.875		.556		.782		1.105 8		1.286		.432		0.86
.472	8	.385	8									0.60 3
0.752		0.784		1.035		1.290		1.212		0.808		0.91
1.211		2.215		2.405		2.866		2.922		2.116		2.92
0.261		0.262		0.460		0.501		0.514		0.340		0.15

E.2. Valores totales mensuales de precipitacion

I D E A M - INSTITUTO DE HIDROLOGIA, METEOROLOGIA Y ESTUDIOS AMBIENTALES
SISTEMA DE INFORMACION
VALORES TOTALES MENSUALES DE PRECIPITACION (mms)

NACIONAL AMBIENTAL
FECHA DE PROCESO : 2004/03/23
ESTACION : 3701502 BERLIN

LATITUD 0711 N
TIPO EST CO
DEPTO SANTANDER
FECHA-INSTALACION 1968-MAY
LONGITUD 7252 W
ENTIDAD 01 IDEAM
MUNICIPIO TONA
FECHA-SUSPENSION
ELEVACION 3214 m.s.n.m
REGIONAL 08 SANTANDERES
CORRIENTE JORDAN

A#O	EST	ENT	ENERO *	FEBRE *	MARZO *	ABRIL *	MAYO *	JUNIO *
1981	2	01	.0	32.6	13.5 3	151.9	192.6	65.0
1982	2	01	7.5	107.1	56.7	234.6	150.8	35.6
1983	2	01	1.9	8.2	29.7	179.0	94.4	82.1
1984	2	01	9.6 3	6.5 3	2.1 3	44.7	36.4	29.7 3
1985	2	01	25.0	13.7	29.5 3	46.8 3	97.4 3	43.0
1986	2	01	19.6 3	42.4 3	17.6	179.9	125.2	95.8
1987	2	01		10.0 3	28.6 3	40.3	152.0	14.7
1988	2	01	12.0	52.3	43.8	77.9	77.3	67.2
1989	1	01	11.1	17.0	62.9	14.3	101.0	45.8
1990	1	01	16.4	18.6	87.8	87.3	62.6	67.2
1991	2	01	23.6	8.3	96.1	59.8	72.5	35.4
1992	1	01	6.5	3.3	1.3	48.5	67.1	34.2
1993	2	01	30.6	10.5	29.1	67.5	189.5	47.3
1994	2	01	13.7	37.3	19.6	81.6	82.4	33.6
1995	2	01	14.2	26.1	27.8	93.8	68.2	56.2
1996	2	01	7.7	42.5	67.6	34.9	40.3	93.8
1997	2	01	7.8	15.7	33.9	38.5	86.9	53.0
1998	1	01	15.9	1.5	33.1	136.5	102.5	93.1
1999	1	01	15.7	32.5	16.0	108.7	25.9	62.2
2000	1	01	23.5	55.1	16.4	34.1	32.1	41.1
MEDIOS			13.8	27.1	35.7	88.0	92.9	54.8
MAXIMOS			30.6	107.1	96.1	234.6	192.6	95.8
MINIMOS			0.0	1.5	1.3	14.3	25.9	14.7

JULIO *	AGOST *	SEPTI *	OCTUB *	NOVIE *	DICIE *	VR ANUAL *
21.8 3	88.5	106.8	131.6 3	20.3	23.0	847.6 3
42.8	59.7	75.5	65.5	20.8	10.4	867.0
79.2	32.0	29.0	60.4	5.7 3	15.1	616.7 3
52.9	84.4 3	128.2 3	88.1 3	73.4	5.9	561.9 3
56.0 3	50.1	56.3	108.9	39.2 3	44.3 3	610.2 3
			107.9 3		28.7 3	617.1 3
52.4	38.1	101.8	216.4	38.0	16.6	708.9 3
67.1	134.8	130.4	72.5	107.3	45.8	888.4

```
39.4     29.6    132.8     84.5     23.7     41.9       604.0
62.9     96.1     71.0    114.9     38.7     39.0       762.5
66.3     55.2     75.5     59.3     61.9     15.5       629.4
80.1     82.6     43.5     38.7     47.5     24.2       477.5
41.8     27.6    118.9     60.6     70.7     12.6       706.7
44.4     41.7     83.9     73.4     87.1      9.9       608.6
45.2    203.5    156.0    179.1     17.6     14.3       902.0
63.9     70.8     61.5    113.3     36.0      6.1       638.4
51.8 3   42.0     78.6    105.0     23.8      7.8       544.8 3
33.6     60.0    106.2    112.8               46.6       741.8 3
48.1     87.4    107.0     97.6    118.3     15.2       734.6
49.4     33.0    120.2     82.8     81.9     14.4       584.0
52.6     69.3     93.8     98.7     50.7     21.9       699.1
80.1    203.5    156.0    216.4    118.3     46.6       234.6
21.8     27.6     29.0     38.7      5.7      5.9         0.0
```

E.3. Valores totales mensuales de evaporación

```
I D E A M  -  INSTITUTO DE HIDROLOGIA, METEOROLOGIA Y ESTUDIOS AMBIENTALES
SISTEMA DE INFORMACION
VALORES TOTALES MENSUALES DE EVAPORACION (mms)
NACIONAL AMBIENTAL
FECHA DE PROCESO :  2004/03/23
ESTACION : 3701502   BERLIN

    LATITUD    0711 N
    TIPO EST    CO
    DEPTO      SANTANDER
    FECHA-INSTALACION   1968-MAY
    LONGITUD   7252 W
    ENTIDAD     01  IDEAM
    MUNICIPIO  TONA
    FECHA-SUSPENSION
    ELEVACION  3214 m.s.n.m
    REGIONAL    08  SANTANDERES
    CORRIENTE  JORDAN
A#O EST ENT ENERO *  FEBRE *  MARZO *  ABRIL *  MAYO  *  JUNIO *
1981  2  01  162.7    103.1 3  148.4 3   85.4 3   82.7     71.8 3
1982  2  01  135.3     86.0 3   99.5 3   73.0     73.4 3   71.6
1983  2  01  131.6    119.0 3  129.2     78.2 3   71.9 3   87.5
1984  2  01  117.9    110.0 3  140.9    121.6    100.2 3   77.8 3
1985  2  01   99.6 3                     90.2 3   67.3 3   58.3
1986  2  01  129.2 3   93.9 3  103.2 3   50.1 3   74.0 3   63.8 3
1987  2  01                    122.6    105.8     87.6     86.4
1988  2  01   91.5 3   72.6 3  100.7 3   84.7 3   72.6 3  101.6
1989  2  01   97.4     91.2    110.4    105.5     73.3     89.5
```

1990	2	01	85.1	*	90.2	97.0	83.8	76.6			
1991	2	01	89.2	96.8	72.2	90.1 3	108.3	95.7			
1992	2	01	112.8	93.7	*	89.8	99.7	100.4			
1993	2	01	117.5	89.9	126.6	95.1	87.9	88.6			
1994	2	01	111.9	80.1	112.0	96.3	88.7	81.0			
1995	1	01	85.3 3		109.3 3	86.0 3	99.8 3	86.3 3			
1996	1	01	131.3 3	82.4 3	106.2 3	111.3	110.5	58.7 3			
1997	1	01									
1998	1	01	136.2	106.9	125.9	110.3	94.5	91.6			
1999	1	01	111.8 3	79.5 3	122.1 3	95.1 3	85.3 3	83.0 3			
2000	1	01	103.4	98.0	105.4	102.6	101.7	89.0			
MEDIOS			113.9	93.5	113.2	93.1	87.5	82.1			
MAXIMOS			162.7	119.0	148.4	121.6	110.5	101.6			
MINIMOS			85.1	72.6	72.2	50.1	67.3	58.3			

JULIO *	AGOST *	SEPTI *	OCTUB *	NOVIE *	DICIE *	VR ANUAL *
96.5 3	84.2 3	95.9 3	91.7 3	111.2 3	105.4	1239.0 3
66.5 3	104.8 3	73.6 3	79.4	86.2	95.2	1044.5 3
84.9 3	95.6 3	91.5 3	95.6	105.6 3	95.2 3	1185.8 3
94.9 3	92.2	90.6 3	74.0 3	82.6 3		1102.7 3
61.5 3	105.7	88.7 3	72.6 3	82.1	72.0 3	798.0 3
			61.1 3		78.8 3	654.1 3
91.1	112.5	116.4	81.5 3	76.3 3	108.0	988.2 3
87.0 3	77.4 3	77.2 3	100.3 3	81.5 3	88.9 3	1036.0 3
69.5	107.2	77.4	97.7	78.4	95.0	1092.5
85.9	87.3 3	94.1	96.3	89.2	80.4 3	965.9 3
81.4	80.5	112.0	97.5	80.8	101.5	1106.0 3
70.4	93.8	110.7	117.3	99.8	119.5	1107.9 3
92.7 3	108.5	98.0 3	94.3	92.4	94.3	1185.8 3
83.8	101.1	106.2	104.2	101.1	93.0	1159.4
103.0 3	82.5 3	103.5 3	97.2 3	105.7	102.1 3	1060.7 3
76.4 3	100.7	79.3 3	106.4	83.6 3	108.9	1155.7 3
			90.1	111.4	136.8	338.3 3
102.9	118.9	106.4	102.5		89.0	1185.1 3
109.3	77.4 3	96.1 3	89.8 3	100.9 3	104.0	1154.3 3
98.3	108.8	101.5	92.5	101.3	95.5	1198.0
86.4	96.6	95.5	92.1	92.8	98.1	1144.8
109.3	118.9	116.4	117.3	111.4	136.8	162.7
61.5	77.4	73.6	61.1	76.3	72.0	50.1

ANEXO F

Datos utilizados en la aplicación de mantenimiento

```
48 100 120 0.009

E010301 10 6 27
E010501 4 6 22
E020301 10 6 27
E020501 4 6 22
E030301 10 6 27
E040301 10 6 27
E050301 10 6 27
E060301 10 6 27
E070401 3 6 28
E080501 4 6 10
I010101 10 6 27
I020101 10 6 27
I030101 10 6 27
I040101 10 6 27
I050101 10 6 27
I060101 10 6 27
I070101 10 6 27
I080101 10 6 27
I090101 10 6 27
I100101 10 6 27
M021901 9 6 10
M032001 6 6 10
M042101 9 6 11
M051402 10 6 19
M062101 9 6 11
M072101 9 6 11
S010202 10 6 27
S020202 10 6 27
S030202 10 6 27
S040202 10 6 27
```

```
S050202 10 6 27
S060201 10 6 27
S070202 10 6 27
S080202 10 6 27
S090202 10 6 27
T021702 3 6 20
T031702 3 6 20
T041702 3 6 20
T051702 3 6 20
T061702 3 6 20
T071702 3 6 20
V030601 6 6 21
V040601 6 6 25
V050701 10 6 27
V060701 10 6 27
V070702 6 6 11
V080702 6 6 11
V090901 3 6 4
V101001 3 6 10
V111002 3 6 10
V121101 9 6 24
V131201 5 6 3
V141201 5 6 3
V151301 6 6 10
V160801 10 6 25
V170801 10 6 25
V180801 10 6 25
V191401 10 6 27
V201401 10 6 27
V211402 10 6 27
V221501 2 6 11
V231601 3 6 11
V241102 9 6 14
V301601 3 6 11
V311501 2 6 11
V321201 5 6 3
V332201 3 6 9
```

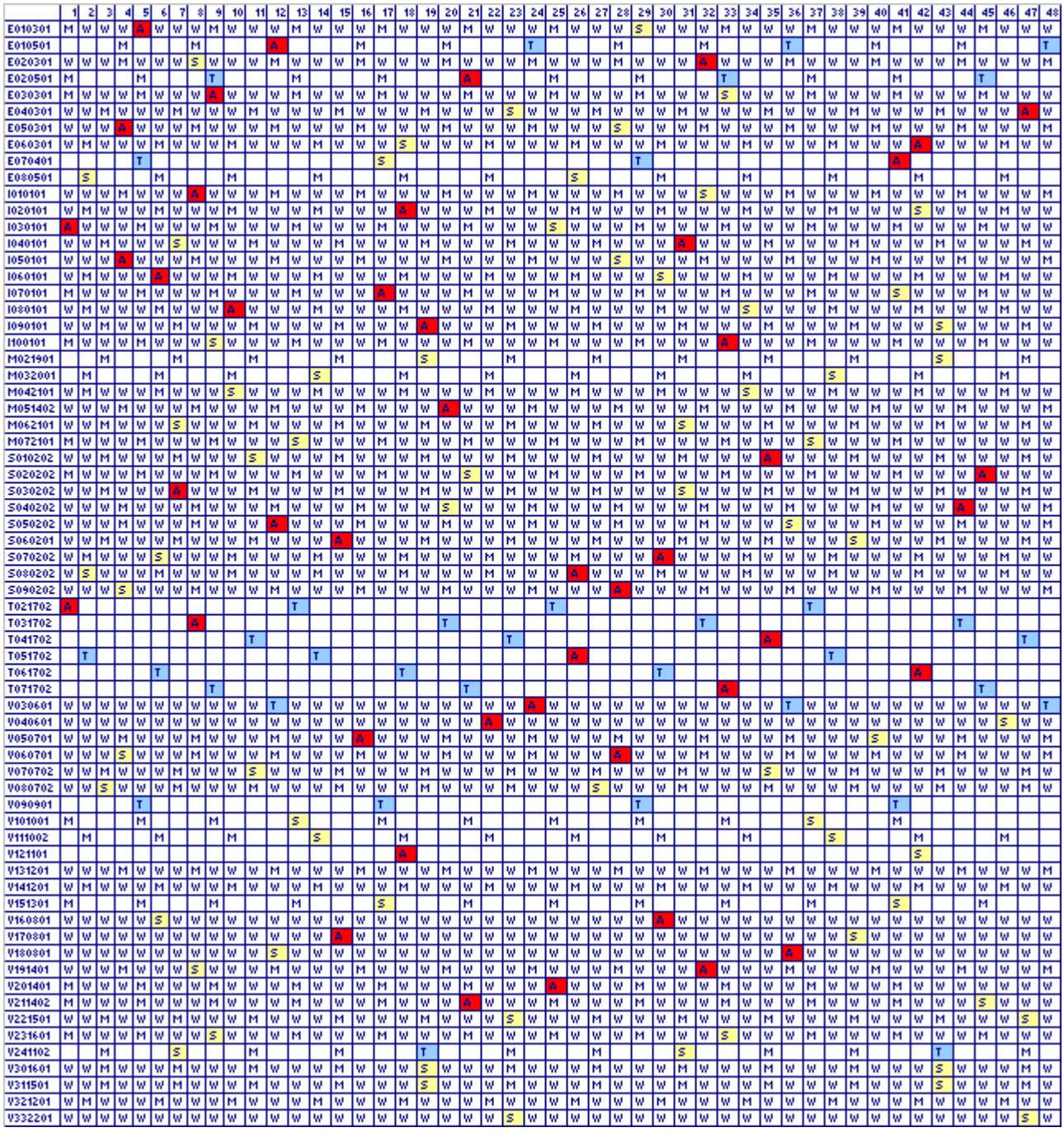
ANEXO G

Programacion de Mantenimiento

En este anexo se encuentra el resultado de ejecutar el algoritmo genético para encontrar la programación de Mantenimiento de la fabrica INDUPLAST S.A.

La letra que aparece en cada semana significa el mantenimiento más importante que se debe llevar a cabo para la máquina en esta semana, quedando implícito que los otros mantenimientos también se deben realizar. De esta forma la letra "A "es para el mantenimiento anual, la "S"para el semestral, la "T"para el trimestral, la "M"para el mensual y la "W"para el semanal.

El color significa los mantenimientos que se deben realizar en la máquina. En la semana con color rojo se deben llevar a cabo mantenimiento anual, semestral, trimestral y mensual, en la semana con color amarillo existirá mantenimiento semestral, trimestral y mensual y en la semana con color azul se realizarán los mantenimientos trimestral y mensual.



ANEXO H

wxWindows Library License, Version 3

wxWindows Library License, Version 3

Copyright (c) 1992-2002 Julian Smart, Robert Roebling, Vadim Zeitlin et al.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

WXWINDOWS LIBRARY LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this software, usually in a file named COPYING.LIB. If not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

EXCEPTION NOTICE

1. As a special exception, the copyright holders of this library give permission for additional uses of the text contained in this release of the library as licensed under the wxWindows Library License, applying either version 3 of the License, or (at your option) any later version of the License as published by the copyright holders of version 3 of the License document.

2. The exception is that you may create binary object code versions of any works using this library or based on this library, and use, copy, modify, link and distribute such binary object code files unrestricted under terms of your choice.

3. If you copy code from files distributed under the terms of the GNU General Public License or the GNU Library General Public License into a copy of this library, as this license permits, the exception does not apply to the code that you add in this way. To avoid misleading anyone as to the status of such modified files, you must delete this exception notice from such code and/or adjust the licensing conditions notice accordingly.

4. If you write modifications of your own for this library, it is your choice whether to permit this exception to apply to your modifications. If you do not wish that, you must delete the exception notice from such code and/or adjust the licensing conditions notice accordingly.