



Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Desarrollo de una aplicación para la implementación de Algoritmos Genéticos

Trabajo Especial de Grado
Presentado ante la Ilustre
Universidad Central de Venezuela

Por el bachiller
Alejandro R. Monascal Caso

Para optar al Título de
Licenciado de Computación

Tutor:
Eugenio Scalise

Laboratorio de Métodos Formales en
Ingeniería de Software

Tutora:
Haydemar Núñez

Laboratorio de Inteligencia Artificial

Caracas, Mayo 2013



Universidad Central de Venezuela

Facultad de Ciencias
Escuela de Computación

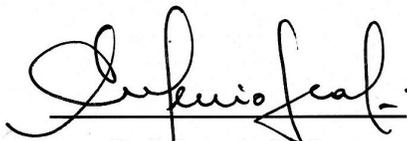
ACTA DEL VEREDICTO

Quienes suscriben, miembros del jurado designado por el Consejo de la Escuela de Computación, para dictaminar sobre el Trabajo Especial de Grado titulado: **Desarrollo de una aplicación para la implementación de Algoritmos Genéticos** y presentado por el bachiller *Alejandro Rafael Monascal Caso*, cédula de identidad V-19.672.195, para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

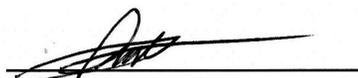
Leído como fue, dicho trabajo por cada uno de los miembros del jurado, se fijó el día 24 de mayo de 2013 a las 11:00 a.m., para que su autor lo defendiera en forma pública, lo que hizo en la Sala 3 del Postgrado de la Escuela de Computación, mediante una presentación oral del contenido del Trabajo Especial de Grado, luego de lo cual respondió a las preguntas formuladas. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió **APROBARLO**.

En fe de lo cual se levanta la presente Acta, en la Ciudad Universitaria de Caracas a los 24 días del mes de mayo del año dos mil trece.

Jurado Principal


Prof. Eugenio Scalise
(Tutor)


Prof. Haydemar Núñez
(Tutora)


Prof. Ivan Flores
(Jurado)


Prof. Rhadames Carmona
(Jurado)

Resumen

Los algoritmos genéticos (AG), son muy usados en la actualidad en la resolución de problemas de optimización en diferentes dominios. Para el desarrollo de aplicaciones basadas en AG se han creado diversas herramientas las cuales, en general, comparten una característica: requieren que el usuario tenga conocimientos del lenguaje de programación en el cual fueron implementadas, restringiendo así su uso a usuarios expertos. En este trabajo se propone el desarrollo de una aplicación para la resolución de problemas con AG, que provea herramientas útiles para la implementación de algoritmos genéticos, sin la necesidad de tener conocimientos en programación y haciendo uso de una interfaz gráfica. Para su desarrollo, se propuso dividir el problema en una Biblioteca y una Interfaz, las cuales se comunican mediante un módulo de comunicación. Además, la biblioteca puede ser usada junto o separado a la interfaz, con la finalidad de que se cuente con esta biblioteca para el desarrollo de futuros proyectos, así como material de apoyo en materias dictadas en el área de Inteligencia Artificial o su distribución. Se obtuvo como resultado una aplicación que cumple con los requerimientos que fueron pautados.

Palabras Clave. Algoritmos Genéticos, Algoritmos Evolutivos, Herramientas de desarrollo, Biblioteca, Interfaz Gráfica.

Dedicatoria

A mis amados padres,
mi adorado hermano,
mis queridos amigos,
mi casa de estudio la Universidad Central de Venezuela
y mi Facultad de Ciencias.

Agradecimientos

Le agradezco a mis padres y a mi hermano, por contar siempre con su cariño y apoyo, por mostrar siempre interés y preocupación en mí, por aguantar todas mis locuras y por tantas cosas más. A ustedes dedico todos mis logros, pues ustedes son la fortaleza que me ayuda a seguir adelante en mis metas. Gracias por todo lo que me han dado, no sería la persona en que me he convertido hoy de no ser por ustedes.

Índice General

Introducción	1
1. Marco Teórico	2
1.1. Algoritmos Evolutivos	2
1.1.1. Terminología Biológica	2
1.1.2. Historia	3
1.2. Algoritmos Genéticos	3
1.2.1. Función de Aptitud	4
1.2.2. Parametros de entrada para un AG simple	4
1.2.3. Operadores Genéticos	4
1.2.4. Estructura de un AG	6
1.3. Variaciones de los AG	7
1.3.1. Mecanismo de Toma de Muestra	7
1.3.2. AG modificado	9
1.3.3. Variaciones del operador de Cruce	10
1.3.4. Otros Operadores Genéticos	12
1.4. Bibliotecas para el desarrollo de Algoritmos Genéticos	13
1.4.1. GAlib	14
1.4.1.1. Características Generales	14
1.4.1.2. Diagrama de Clases	15
1.4.1.3. Desarrollo de AG usando GAlib	15
1.4.1.4. Definición de operadores	16
1.4.2. MATLAB GA Toolbox	17
1.4.2.1. Características Generales	17
1.4.2.2. Desarrollo de AG usando GA Toolbox	17
1.4.3. JGAP	18
1.4.3.1. Características Generales	19
1.4.3.2. Diagrama de Clases	19
1.4.3.3. Desarrollo de AG usando JGAP	20
1.4.4. Comentarios del uso de bibliotecas para AG	21
2. Marco Aplicativo	22
2.1. Objetivo General	22
2.2. Objetivos Específicos	22
2.3. Desarrollo de la Solución	22
2.3.1. Tecnologías Utilizadas	23
2.3.2. Biblioteca	23

2.3.2.1. Explicación detallada de las clases	24
2.3.2.2. Pruebas realizadas sobre la biblioteca	34
2.3.3. Interfaz	41
2.3.3.1. Diseño y funcionamiento de la interfaz	44
2.3.3.2. Conexión entre la biblioteca y la interfaz	60
2.3.3.3. Pruebas realizadas sobre la interfaz	60
Conclusiones y Recomendaciones	69
Referencias	70

Índice de Figuras

Figura 1.1. Cruce Clásico	6
Figura 1.2. Mutación	6
Figura 1.3. Estructura general de AG	6
Figura 1.4. Estructura general de un Agmod	9
Figura 1.5. Cruce de 2 Puntos	11
Figura 1.6. Cruce Uniforme	12
Figura 1.7. Cruce Ordenado	12
Figura 1.8. Inversión	13
Figura 1.9. Barajeo	13
Figura 1.10. Intercambio	13
Figura 1.11. Diagrama de Clases de GALib	15
Figura 1.12. Ejemplo de un programa desarrollado en GALib	16
Figura 1.13. Ejemplo de definición de operadores propios en GALib	16
Figura 1.14. Función objetivo del ejemplo en GA Toolbox	18
Figura 1.15. Ejemplo de un programa desarrollado en MATLAB usando GA Toolbox	18
Figura 1.16. Diagrama de Clases de JGAP	19
Figura 1.17. Ejemplo de un programa desarrollado en JGAP - Parte 1	20
Figura 1.18. Ejemplo de un programa desarrollado en JGAP - Parte 2	21
Figura 2.1. Diseño de la aplicación	23
Figura 2.2. Diagrama De Clases 1 – Diagrama General	24
Figura 2.3. Diagrama de Clases 2 – Manejadores	25
Figura 2.4. Diagrama de Clases 3 – Genes	28
Figura 2.5. Diagrama de Clases 4 – Selector Natural	30
Figura 2.6. Diagrama de Clases 5 – Operadores Genéticos	31
Figura 2.7. Diagrama de Clases 6 – Operadores de Cruce	31
Figura 2.8. Diagrama de Clases 7 – Excepciones	34
Figura 2.9. Clase principal del primer caso de prueba	35
Figura 2.10. Función de Aptitud del primer caso de prueba	36
Tabal 2.1. Resultados del primer caso de prueba	36
Figura 2.11. Condición de Terminación del segundo caso de prueba	37
Figura 2.12. Función de Aptitud del segundo caso de prueba	37
Figura 2.13. Clase Principal del segundo caso de prueba	38
Tabla 2.2. Resultados del segundo caso de prueba	39
Figura 2.14. Grafo TSP	39
Figura 2.15. Clase Principal del tercer caso de prueba	40
Figura 2.16. Función de Aptitud del tercer caso de prueba	41

Tabla 2.3. Resultados del segundo caso de prueba	41
Figura 2.17. Casos de uso principales para el desarrollo de un AG	42
Figura 2.18. Casos de uso para Definir Cromosoma	42
Figura 2.19. Casos de uso para Definir Función	42
Figura 2.20. Casos de uso para Configurar Algoritmo	43
Figura 2.21. Casos de uso para Ejecutar Algoritmo	43
Figura 2.22. Casos de uso para Observar Progreso de Trabajo	44
Figura 2.23. Interfaz Diseñada por Gustavo Torres	44
Figura 2.24. Interfaz Diseñada para aplicación de AG	45
Figura 2.25. Interfaz para Definir Cromosoma	46
Figura 2.26. Interfaz para Definir Genes	46
Figura 2.27. Interfaz para Definir Función	47
Figura 2.28. Interfaz para Crear Función de Aptitud	48
Figura 2.29. Interfaz para Crear Función de Terminación	49
Figura 2.30. Código escrito en GAL_Parser	51
Figura 2.31. Árbol de un código en GAL_Parser	52
Figura 2.32. Interfaz para Configurar Algoritmo	52
Figura 2.33. Interfaz para el Selector	53
Figura 2.34. Interfaz para los Operadores	53
Figura 2.35. Interfaz para los Parámetros	54
Figura 2.36. Interfaz para Ejecutar Algoritmo	54
Figura 2.37. Interfaz para Resultados	55
Figura 2.38. Interfaz para Progreso de Trabajo 1	56
Figura 2.39. Interfaz para Progreso de Trabajo 2	56
Figura 2.40. Interfaz para Progreso de Trabajo 3	56
Figura 2.41. Interfaz para Ver Cromosoma	56
Figura 2.42. Interfaz para Ver Función	57
Figura 2.43. Interfaz para Ver Configuración	57
Figura 2.44. Interfaz para el Sistema de Ayuda	58
Figura 2.45. Caso de Prueba 1 – Definir Genes 1	61
Figura 2.46. Caso de Prueba 1 – Definir Genes 2	61
Figura 2.47. Caso de Prueba 1 – Crear Función de Aptitud	62
Figura 2.48. Caso de Prueba 1 – Selector	62
Figura 2.49. Caso de Prueba 1 – Operadores	63
Figura 2.50. Caso de Prueba 1 – Parámetros	63
Figura 2.51. Caso de Prueba 1 – Resultados	64
Figura 2.52. Caso de Prueba 2 – Archivo	65
Figura 2.53. Caso de Prueba 2 – Resultados	66
Figura 2.54. Caso de Prueba 3 – Archivo	67
Figura 2.55. Caso de Prueba 3 – Resultados	68

Introducción

Actualmente existen muchos problemas de optimización que por su complejidad, no pueden ser resueltos por algoritmos deterministas, debido al tiempo que tardarían en ofrecer resultados. Sin embargo, en la mayoría de estos problemas no es necesario obtener un resultado exacto, sino que basta con tener un aproximado al óptimo. Los algoritmos que buscan aproximados al resultado se basan en probabilidades y forman parte de los algoritmos probabilísticos. Los Algoritmos Genéticos son usados para problemas de optimización y forman parte de estos algoritmos probabilísticos.

El Algoritmo Genético (AG), toma sus ideas del proceso evolutivo de los seres vivos y fue publicado por primera vez por John Holland [3], quien describe a un conjunto de posibles soluciones al problema como una población de individuos que irán evolucionando de generación en generación a través de los operadores genéticos de selección, cruce y mutación; hasta un número de generaciones donde se tomará el mejor individuo, el cual teóricamente deberá estar cerca a la solución óptima.

Los Algoritmos Genéticos se presentan como una solución elegante a muchos problemas de optimización, muchos de éstos están ligados a áreas de estudio no relacionadas a la computación. Para el desarrollo de soluciones basadas en AG se han creado muchas herramientas; sin embargo, éstas tienen un problema en común; requieren que el usuario tenga conocimientos del lenguaje de programación en el cual fueron implementadas, restringiendo su uso a solamente programadores o requiriendo de un tiempo adicional para adiestrarse en éstas

Este trabajo se enfoca en el desarrollo de una aplicación para la implementación de Algoritmos Genéticos, que permita a cualquier usuario sin competencias en programación, hacer uso de las herramientas que éstos ofrecen. Para su desarrollo, se propuso dividir el problema en una Interfaz y una Biblioteca las cuales se comunican mediante un módulo de comunicación.

A continuación se especifica la estructura del documento:

Capítulo 1: Marco Teórico. En este capítulo se explican los conceptos básicos necesarios para comprender en qué consiste un AG, así como se describe la historia detrás del desarrollo de éstos. Además, se muestran diversas variaciones al algoritmo genético clásico, que dependiendo del problema a ser solucionado, pueden obtener mejores resultados. Finalmente, se describen tres bibliotecas para la implementación de Algoritmos Genéticos (GALib, GA Toolbox y JGAP), donde por cada una se presentan sus características generales y un ejemplo de su uso.

Capítulo 2: Marco Aplicativo. En este capítulo se describen la motivación, el objetivo general y los objetivos específicos. Además, se explica en detalle los pasos que fueron realizados para desarrollar la solución, dividiendo la explicación en Biblioteca e Interfaz.

Finalmente, se presentan las conclusiones y algunas recomendaciones, seguido de las referencias utilizadas para el desarrollo de este documento.

Capítulo 1. Marco Teórico

La computación evolutiva es una rama de la Inteligencia Artificial, que comprende un conjunto de técnicas basadas en los principios de la evolución biológica, para la resolución de diversos problemas. Estas técnicas son usadas en una amplia variedad de problemas, desde problemas prácticos en las industrias hasta investigaciones científicas. Diversos enfoques a la computación evolutiva han sido propuestos y se les denominan de manera colectiva como Algoritmos Evolutivos, entre los cuales uno de los más utilizados son los Algoritmos Genéticos. [10]

En este Capítulo se realiza una breve explicación de los Algoritmos Evolutivos, especificando las terminologías biológicas utilizadas, así como la historia detrás de éstos. Además, se explica con mayor detalle los Algoritmos Genéticos, haciendo énfasis en la función de aptitud, los operadores genéticos, estructura y variaciones del algoritmo.

1.1. Algoritmos Evolutivos

Los Algoritmos Evolutivos (AE) son algoritmos metaheurísticos y probabilísticos para la resolución de problemas de optimización. Los AE se basan en el uso de poblaciones de individuos, uno por cada iteración del algoritmo. Cada individuo representa una posible solución al problema y debe ser evaluado para hallar una medida de su “aptitud”. Posteriormente, se construye una nueva población a partir de la selección de los individuos más aptos, la cual será utilizada en la siguiente iteración (etapa de selección). Algunos de los miembros de la nueva población sufrirán transformaciones inspiradas en la evolución, con la finalidad de hallar nuevas soluciones (etapa de alteración). Después de un número de iteraciones el programa converge, se espera que el mejor individuo represente una solución cercana al óptimo. [1]

1.1.1. Terminología Biológica

En los AEs se utilizan muchos términos inspirados en la biología, aunque las entidades a las que se refieren son mucho más simples que su equivalente biológico.

Todos los organismos vivos consisten de células y cada célula contiene el mismo conjunto de uno o más *cromosomas* (cadenas de ADN) que sirven como modelo para el organismo. Un cromosoma puede ser dividido en *genes* y cada uno codifica una proteína en particular. Se puede pensar que los genes son codificadores de *rasgos*, como el color del ojo.

Las diferentes configuraciones para un rasgo son conocidos como *alelos*, por ejemplo para el color de los ojos los alelos serían azul, marrón, negro, verde, entre otros. Cada gen se encuentra en una posición particular del cromosoma.

Muchos organismos poseen múltiples cromosomas en cada célula. La colección completa de material genético (todos los cromosomas de una célula) se llama *genoma* del organismo. El término *genotipo* es utilizado para referirse al conjunto de genes que están contenidos en un genoma. Dos individuos que compartan el mismo genoma, se dice que comparten el mismo

genotipo.

Durante la reproducción sexual, ocurre la *recombinación* (o *cruce*). En las reproducciones sexuales, los genes son intercambiados entre los cromosomas de los padres. Los descendientes pueden estar sujetos a la *mutación*, donde cada nucleótido (bit elemental del ADN) puede sufrir un cambio.

La aptitud de un organismo es usualmente definido como la probabilidad de que un organismo vivirá para reproducirse (*viabilidad*) o como una función del número de descendientes que el organismo tiene (*fertilidad*).

En los AE, el término cromosoma es usado para referirse a las soluciones candidatas al problema. Los genes son bits individuales o bloques cortos de bits adyacentes que codifican el valor de un elemento particular del cromosoma. Los alelos son los valores que puede tener cada gen, por ejemplo en una cadena de bits serían los valores 0 y 1. El cruce usualmente consiste en intercambiar material genético entre dos padres. La mutación consiste en remplazar el alelo en una posición seleccionada de manera aleatoria por otro alelo seleccionado también de manera aleatoria.

En los AE, a cada iteración se conoce como *generación*. Por cada generación se forma un genoma nombrado usualmente *población*. El conjunto total de generaciones se llama *corrida*.

1.1.2. Historia

En las décadas de los 50 y 60 muchos científicos computacionales estudiaron de manera independiente los sistemas evolutivos, con la idea de que la evolución podría ser usado como una herramienta de optimización en diversos problemas.[6]

En la década de los 60, Renchemberg (1965, 1973) introduce su trabajo “Estrategias Evolutivas” (*evolutions strategies* en su idioma original Alemán), un método que utilizó para optimizar valores reales para dispositivos como airfoils. Esta idea fue luego desarrollada por Schwefel (1975, 1977). Fowel, Owens y Walsh (1966), quienes desarrollaron los Programas Evolutivos (PE), una técnica en donde las soluciones candidatas fueron representadas por máquinas de estado finitas.

Los Algoritmos Genéticos (AG) fueron propuestos por John Holland en la década de los 60 y fueron desarrollados por Holland, sus estudiantes y colegas de la Universidad de Michigan entre las décadas de 1960 y 1970. En 1975, Holland publica el libro “Adaptación en Sistemas Naturales y Artificiales” (*Adaptation in Natural and Artificial Systems*) [3], donde presenta los Algoritmos Genéticos como una abstracción de la evolución biológica.

1.2. Algoritmos Genéticos

Los AG comparten junto con los otros AE la misma estructura de programa y las mismas terminologías biológicas. Las diferencias entre los diferentes AE vienen dadas por lo general por la representación de los cromosomas y el proceso interno durante las etapas de selección y

alteración.

El AG propuesto por Holland [3], describe que la población está formada por cromosomas cuya estructura es una cadena de bits de tamaño fijo. Se cuenta con los operadores genéticos *Selección* para la etapa de selección; y *Cruce* y *Mutación* para la etapa de alteración.

En 1985, De Jong mencionó su interés en el uso de estructuras de datos más complejas para representar los individuos [4]. Los investigadores en el área decidieron prestar atención en la sugerencia que De Jong proponía y poco a poco el término “Algoritmo Genético” fue recibiendo un significado muy distinto al que le dio Holland. Hoy en día los AG utilizan una población formada por individuos cuya estructura de dato sea natural para el problema, manteniéndose el uso de los operadores genéticos para las etapas de selección y alteración.

1.2.1. Función de Aptitud

La Función de Aptitud es una función que calcula qué tan apto es un cromosoma, es decir qué tan bien soluciona el problema que se está resolviendo. También se le conoce como función de evaluación.

La función se puede representar como:

eval: $S \rightarrow N$.

Donde S es un cromosoma y N un valor numérico.

Usualmente, mientras mayor sea el valor de N más apto será el cromosoma.

1.2.2. Parámetros de entrada para un AG simple

Existen ciertos valores que deben ser ingresados antes de ejecutar cualquier AG, éstos son:

- Cantidad de Generaciones: Denota la cantidad máxima de generaciones que se realizarán si no se llega a cumplir la condición de terminación. Usualmente se encuentra entre 50 y 500.
- Tamaño de la Población: Define cuántos cromosomas se tienen por generación.
- Probabilidades de ocurrencia de los operadores genéticos: La probabilidad de aplicar cada uno de los operadores genéticos, exceptuando la selección, el cual es de carácter obligatorio.

Existen otras versiones de AGs que requieren de otros parámetros adicionales, éstos serán explicados más adelante.

1.2.3. Operadores Genéticos

a) Selección

Este operador consiste en la selección de cromosomas de la población para reproducirse y formar nuevos cromosomas en la siguiente generación.

El método de selección propuesto por Holland [3] se denomina selección por ruleta, donde el tamaño de las casillas de la ruleta es proporcional a la aptitud de los cromosomas de una generación (para la ruleta se asume que los valores que retorna la función de aptitud serán siempre positivos, de manera que si se retorna valores negativos se deben realizar otras operaciones previas).

Se definen los siguientes pasos para construir la ruleta: [1]

- Calcular la aptitud de cada cromosoma v_i ($i = 1..tam_poblacion$).
- Hallar la aptitud total de la población.

$$F = \sum_{i=1}^{tam.poblacion} eval(v_i)$$

- Calcular la probabilidad de selección p_i por cada cromosoma v_i ($i = 1..tam_poblacion$).
- Calcular la probabilidad acumulativa q_i por cada cromosoma v_i ($i = 1..tam_poblacion$).

$$p_i = eval(v_i) / F$$

$$q_i = \sum_{j=1}^i p_j$$

Este método se basa en hacer “girar” la ruleta tantas veces como la cantidad de cromosomas en la población ($tam_poblacion$); cada vez que se gira la ruleta se selecciona un cromosoma para la nueva población de la siguiente manera:

- Generar un número real aleatorio r en el rango $[0,1]$.
- Si $r < q_1$ entonces se elige el primer cromosoma (v_1); si no se selecciona el i -ésimo cromosoma v_i ($2 \leq i \leq tam_poblacion$) tal que $q_{i-1} < r \leq q_i$.

Es evidente que con este método algunos cromosomas podrán ser seleccionados más de una vez. De esta manera, los mejores cromosomas tendrán más probabilidades de recibir más copias y los peores de morir en esa generación.

b) Cruce

Todos los operadores genéticos exceptuando la selección requieren de una probabilidad, la cual corresponde a la probabilidad de que dicho operador sea aplicado. Para el cruce, a esa probabilidad se le denota p_c . El cruce es una operación que se realiza a nivel del cromosoma; por lo tanto, se espera que a $p_c * tam_poblacion$ cromosomas se les aplique esta operación.

Por cada cromosoma en la nueva población se debe:

- Generar un número real aleatorio r en el rango $[0,1]$.
- Si $r < p_c$ entonces se elige ese cromosoma para hacer la operación de cruce.

Existen diversos tipos de cruce. El propuesto por Holland [3] es el cruce simple, en donde una vez seleccionado los cromosomas a ser afectados por la operación de cruce, se colocan en parejas seleccionadas de manera aleatoria. Si la cantidad de cromosomas a los que se les aplica cruce es impar, se debe agregar o eliminar de la lista un cromosoma. Posteriormente, por cada par de cromosomas se selecciona de manera aleatoria un lugar p , el cual indica el punto de cruce

entre los dos cromosomas. En la Figura 1.1 se muestra como se aplica el cruce entre dos cromosomas.

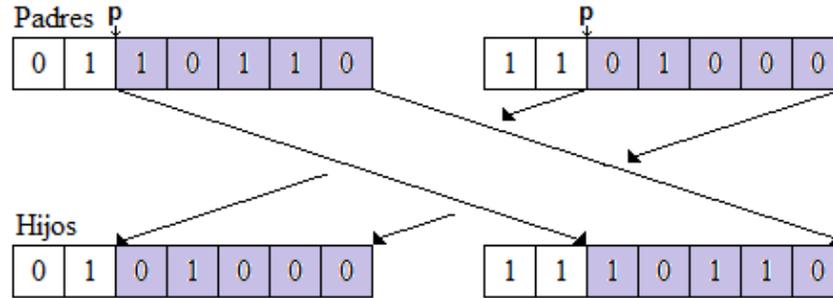


Figura 1.1. Cruce Clásico

c) Mutación

Al igual que con el cruce, requiere de una probabilidad. Se espera que hayan $p_m * m * \text{tam_poblacion}$ genes a los que se les aplique esta operación, donde p_m representa la probabilidad de que ocurra una mutación en cualquier gen y m es la cantidad de genes dentro de un cromosoma.

La mutación cambia el alelo que tiene un gen, por ejemplo en el caso de una cadena de bits, si un gen es seleccionado para aplicar mutación sobre él, se cambiaría de 0 a 1 o viceversa, como se muestra en la Figura 1.2.

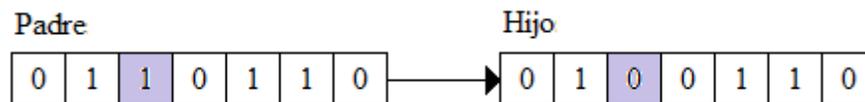


Figura 1.2. Mutación

1.2.4 Estructura de un AG

Como se mencionó anteriormente, los AG siguen la misma estructura que cualquier otro AE. En la Figura 1.3 se muestra una estructura general de un AG.

```

procedimiento Algoritmo Genético
  comenzar
    t = 0;
    inicializar P(t);
    evaluar P(t);
    mientras (no condición de terminación) hacer
      t = t+1;
      seleccionar P(t) desde P(t - 1);
      alterar P(t);
      evaluar P(t);
    fin;
  retornar;
fin;
  
```

Figura 1.3. Estructura general de un AG

Las primeras tres líneas son parte de la inicialización del AG. La variable t representa la generación actual; se coloca su valor en 0 representando la primera generación. La función inicializar genera la primera población de manera aleatoria, mientras que evaluar calcula el valor de la función de aptitud por cada cromosoma en la población actual.

La siguiente línea consiste en la condición de terminación del AG. Por ejemplo, que el valor de aptitud de uno de los cromosomas de la población exeda un umbral. Es recomendable siempre agregar a la condición de terminación un límite de generaciones para evitar que el programa se ejecute indefinidamente.

Si la condición de terminación no se cumple, se ejecuta lo que se encuentra interno al ciclo. Primero, se incrementa la generación en uno. Se prosigue a la etapa de selección, donde la función seleccionar aplica el operador genético de selección sobre la población $t-1$ y lo retorna en la población t . La etapa de alteración es manejada por la función alterar, donde se aplican los operadores genéticos que fueron seleccionados para la mutación y cruce. Finalmente se debe evaluar la nueva población a ser usada durante el próximo ciclo.

Una vez terminado el AG se retorna el valor que se deseaba calcular, éste suele ser la mejor solución encontrada, pero para ciertos casos se podría desear otros valores, como la mejor solución de todas las generaciones.

1.3. Variaciones de los AG

La teoría de Algoritmos Genéticos provee algunas explicaciones de por qué para un problema se puede conseguir una convergencia al punto óptimo. Lamentablemente, no siempre se siguen estas teorías, siendo las razones principales:

- La codificación del problema usualmente mueve el AG a un espacio diferente del problema mismo.
- Hay un límite en el número de iteraciones.
- Hay un límite en el tamaño de la población.

Una de las implicaciones de estas observaciones es la incapacidad de los AG para hallar el óptimo ante ciertas condiciones; estos fallos son causados usualmente por convergencias prematuras a óptimos locales.

Muchas variaciones han sido creadas por los investigadores con la finalidad de reducir estas fallas, algunas de ellas relacionadas con los operadores genéticos, otras con la función de evaluación.

1.3.1. Mecanismo de Toma de Muestra

Existen dos problemas importantes en el proceso de evolución: diversidad de la población y presión selectiva. Estos factores están fuertemente relacionados, un incremento en la presión selectiva decrementa la diversidad de la población, y viceversa. Es decir, una presión selectiva fuerte puede traer como consecuencia una convergencia prematura, mientras que una

convergencia débil puede hacer que la búsqueda se vuelva ineficaz. Es por esto que se debe mantener un equilibrio entre estos dos factores; los mecanismos de toma de muestra buscan precisamente conseguir dicho equilibrio.

El primer trabajo en mecanismo de toma de muestra fue realizado por De Jong[5] en 1975. De Jong diseñó el Modelo Elitista, el cual permite que los k mejores cromosomas pasen directamente sin alteraciones a la siguiente generación. Posterior a esto, se realiza la técnica de la ruleta para seleccionar a los tam_poblacion – k cromosomas que faltan.

Otros métodos para tomar muestras de una población se basan en introducir pesos artificiales para los cromosomas; En la mayoría de estos casos el peso de cada cromosoma está basado en su ranking en vez de en su valor de evaluación. Estos métodos están basados en la creencia de que la razón más común por la que ocurre la convergencia prematura es la presencia de súper individuos, los cuales son mucho mejor que el resto de individuos en promedio. Dichos súper individuos tienden a tener una gran cantidad de descendientes y no permiten que otros individuos con material deseable obtengan también descendientes, causando una convergencia rápida posiblemente a un óptimo local.

Existen diversos métodos para asignar el número de descendientes basado en el ranking. Uno de éstos desarrollado por Baker [8], en el cual se toma un valor definido por el usuario (MAX), como el límite superior para la cantidad esperada de descendientes. Se forma entonces una curva basado en el valor de MAX donde el área bajo la curva sea igual al tamaño de la población. De esta manera se puede conocer fácilmente la cantidad de descendientes esperados entre dos individuos *adyacentes* en el ranking. Por ejemplo, para MAX= 2 y tam_poblacion= 50, entonces la diferencia en la cantidad esperada de descendientes de dos individuos adyacentes será 0.04.

Otra posibilidad es tomar un parámetro q y definir una función lineal como:
 $prob(rank) = q - (rank - 1)r$

O una función no lineal como:
 $prob(rank) = q(1 - q)^{rank - 1}$

Ambas funciones retornan la probabilidad de que un individuo en la posición rank (rank= 1 significa el mejor individuo, rank= tam_poblacion el peor) sea seleccionado. Ambos esquemas permiten que el usuario manipule la presión selectiva del algoritmo.

Para ambos casos se debe cumplir que:

$$\sum_{i=1}^{tam.poblacion} prob(i) = 1$$

Esto implica para la primera opción que:

$$\sum_{i=1}^{tam.poblacion} prob(i) = \sum_{i=1}^{tam.poblacion} q - (i - 1)r = q * tam.poblacion - r(tam.poblacion - 1)(tam.poblacion) / 2 = 1$$

Si se despeja q, se obtiene entonces:

$$q = r(tam_poblacion - 1) / 2 + 1 / tam_poblacion$$

Por lo tanto, si $r=0$ entonces $q=1/\text{tam_poblacion}$ y no existe presión selectiva (todos los individuos tienen la misma probabilidad de ser seleccionado). Por otra parte, si $q=r(\text{tam_poblacion} - 1)$, se obtiene que:

$$q=2/\text{tam_poblacion}, \text{ y } r=2/(\text{tam_poblacion}(\text{tam_poblacion} - 1))$$

Obteniendo así la mayor presión selectiva posible. Es decir, para la función lineal descrita anteriormente, la variable q se debe encontrar entre $1/\text{tam_poblacion}$ y $2/\text{tam_poblacion}$.

Para la función no lineal, el valor q se encuentra en el rango $(0,1)$ y no depende del tamaño de la población. Valores más grandes de q implican mayor presión selectiva. Nótese que:

$$\sum_{i=1}^{\text{tam.poblacion}} \text{prob}(i) = \sum_{i=1}^{\text{tam.poblacion}} q(1-q)^{\text{rank}-1} \approx 1$$

Otro método de toma de muestra es la selección por torneo, combina la idea del ranking de una manera interesante y eficiente. Este método (en una sola iteración), selecciona un número k de individuos y selecciona el mejor del conjunto. Este proceso es repetido tam_poblacion veces. Está claro que el valor de k permite manipular la presión selectiva, típicamente el valor aceptado para k es 2.

Estos algoritmos muestran mejoras en el comportamiento de AG en algunos casos, más tiene ciertas desventajas el utilizarlas:

- La responsabilidad de elegir cuando usar estos mecanismos recae en el usuario.
- Ignoran la información sobre la evaluación relativa de diferentes cromosomas.
- Tratan todos los casos de manera uniforme, sin importar la magnitud del problema.

Pero por otra parte, controlan de mejor manera la presión selectiva y le dan mayor importancia a la búsqueda.

1.3.2. AG modificado

El AG modificado (AGmod) es una variación del AG. Fue diseñado con el propósito de reducir la influencia indeseada que puede generar las características de la función.

```

procedimiento AGmod
  comenzar
    t = 0;
    inicializar P(t);
    evaluar P(t);
    mientras (no condición de terminación) hacer
      t = t+1;
      seleccionar-padres desde P(t - 1);
      seleccionar-muertos desde P(t - 1);
      formar P(t): reproducir los padres;
      evaluar P(t);
    fin;
fin;

```

Figura 1.4. Estructura general de un Agmod

Como se muestra en la Figura 1.4, la selección se subdivide en dos pasos:

- Paso 1 – Seleccionar Padres: Se seleccionan de manera independiente r cromosomas (No necesariamente distintos) para reproducirse.
- Paso 2 – Seleccionar Muertos: Se seleccionan r cromosomas distintos para morir.

Después de realizar estos dos pasos, se forman 3 grupos no necesariamente disjuntos:

- r cromosomas para reproducirse.
- r cromosomas para morir.
- El resto de los cromosomas, llamados cromosomas neutrales.

La nueva población será formado entonces por:

- Los r descendientes formados a partir de los r padres.
- Los $tam_poblacion - r$ cromosomas que no fueron elegidos para morir.

Una de las ideas del AGmod es una mejor utilización del espacio disponible para la población. Este algoritmo evita la formación de múltiples copias exactas de los mismos cromosomas en la nueva población (el cual puede ocurrir aún por accidente).

Existen cambios importantes entre este método y los otros métodos de selección explicado con anterioridad:

- Padres y descendientes tienen una buena posibilidad de salir en la siguiente generación.
- Todos los operadores genéticos se aplican directamente sobre todo el individuo en vez de por genes individuales. Esto provee un tratamiento uniforme de los operadores genéticos utilizados. De manera que si existen dos operadores genéticos (Ej. Mutación y Cruce), algunos de los padres serán operados con mutación y el resto con cruce.

1.3.3. Variaciones del operador de Cruce

El operador de cruce tradicional tiene varios inconvenientes. Por ejemplo, supongamos dos esquemas con un alto desempeño:

$$S_1 = (0 \ 1 \ * \ * \ * \ * \ 0) \text{ y}$$

$$S_2 = (* \ * \ * \ 1 \ 0 \ * \ *).$$

Existen además 2 cromosomas de la población V_1 y V_2 que concuerdan con S_1 y S_2 respectivamente.

$$V_1 = (0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0) \text{ y}$$

$$V_2 = (1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0).$$

Es imposible conseguir un cruce entre los cromosomas V_1 y V_2 de manera que concuerde con el esquema:

$$S_3 = (0 \ 1 \ * \ 1 \ 0 \ * \ 0).$$

El cruce tradicional tiene además la desventaja de que a diferencia de la mutación no depende del tamaño del cromosoma. Por ejemplo, si $p_m = 0.1$, entonces para cromosomas de tamaño 100, el promedio de bits mutados es 1, pero si el tamaño del cromosoma es 1000, el promedio de bits mutados es 10. El operador de cruce sólo realiza la operación en un único punto de cruce sin importar si el cromosoma mide 100 o 1000.

Algunos investigadores (ej. [11] y [12]), han experimentado con otras técnicas de cruce. Por ejemplo el cruce de 2 puntos, donde se eligen dos puntos dentro de los cromosomas y el material que se encuentra entre estos dos puntos es la parte que será intercambiada entre éstos. Claramente, para V_1 y V_2 este método puede producir descendientes que satisfagan el esquema S_3 , como se aprecia en la Figura 1.5.

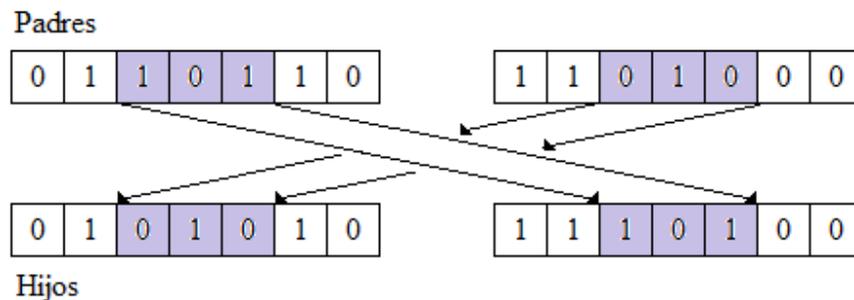


Figura 1.5. Cruce de 2 Puntos

De igual manera, existen esquemas que el cruce de dos puntos no puede resolver. Se puede experimentar con cruces multi-puntos [11], donde se tienen 2 puntos o más de cruce y se sigue la misma lógica que el cruce de 2 puntos.

Algunos investigadores [11], experimentaron con otros métodos de cruce, conocidos como cruce por segmentos y cruce aleatorio:

- El cruce por segmentos es una variante del cruce multi-puntos, donde la cantidad de puntos de cruce puede variar. La cantidad fija de cruces es cambiado por una probabilidad de cambiar de segmento en cualquier gen. Por ejemplo, si la probabilidad de cambiar de segmento es 0.2, entonces la cantidad esperada de segmentos será $0.2 * m$.
- El cruce aleatorio consiste en mezclar de manera aleatoria los valores de los cromosomas a ser afectados por el cruce. Posterior a esto, se cruzan ambos cromosomas con cualquiera de las técnicas mencionadas anteriormente. Finalmente, se ordenan los valores de los descendientes.

Una mayor generalización del cruce multi-puntos es el cruce uniforme [13], [12]: Por cada bit en el descendiente se decide con una probabilidad p , cual padre va a colaborar con esa posición. El segundo descendiente obtendrá los bits del otro padre. Por ejemplo, para $p = 0.5$ (0.5 – Cruce Uniforme), se podría obtener un resultado como el de la Figura 1.6.

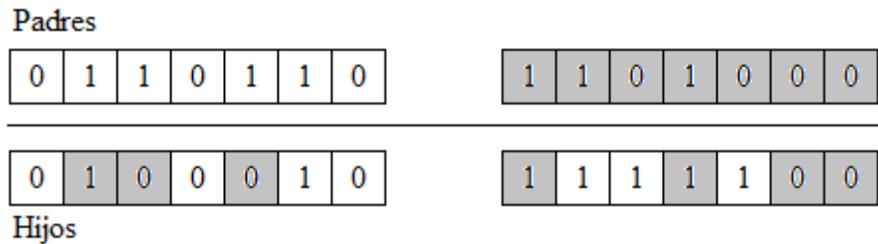


Figura 1.6. Cruce Uniforme

Dado que el cruce uniforme intercambia genes en vez de segmentos, puede combinar características sin importar su localización relativa.

Finalmente, el cruce ordenado como su nombre lo indica, realiza un cruce manteniendo un orden en las posiciones que se cruzan, como se observa en la Figura 1.7; es un cruce diseñado para problemas de permutación, aunque puede ser usado para otros problemas donde los genes comparten la misma configuración.

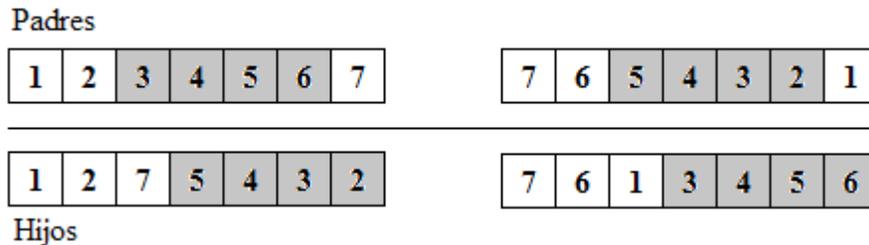


Figura 1.7. Cruce Ordenado

Eshelman, Caruana y Schaffer [11] reportaron diversos experimentos para varios operadores de cruce. Los peores resultados se encontraron con el cruce clásico, sin embargo no se logró definir un ganador, ya que cada una de las técnicas de cruces son útiles para ciertos problemas y para otros no.

1.3.4. Otros Operadores Genéticos

Existen otros operadores además de los operadores de cruce y mutación, éstos son utilizados generalmente para problemas específicos. Entre estos se encuentra los operadores de Inversión, Barajeo e Intercambio.

a) Inversión

Forma parte de los operadores que definió originalmente Holland[3], pero no es muy usado en la actualidad dado que no se conoce los efectos que tiene sobre un AG. Se espera que hayan $p_i \cdot \text{tam_poblacion}$ cromosomas a los que se les aplique esta operación, donde p_i representa la probabilidad de inversión por cromosoma.

Si un cromosoma es seleccionado para ser invertido se debe:

- Seleccionar 2 posiciones p_1 y p_2 de manera aleatoria tal que $p_1 < p_2$.

- Invertir los genes que se encuentran entre p_1 y p_2 como se muestra en la Figura 1.8.

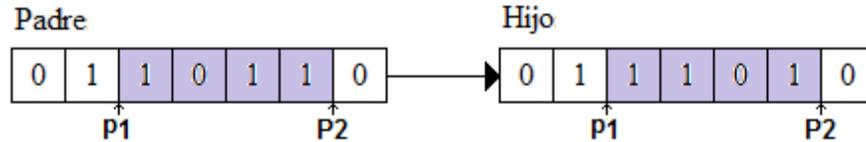


Figura 1.8. Inversión

b) Barajeo

También conocido como Mutación por Barajeo, se espera que hayan $p_b \cdot \text{tam_poblacion}$ cromosomas a los que se les aplique esta operación, donde p_b representa la probabilidad de barajeo por cromosoma.

Si un cromosoma es seleccionado para ser barajeado se debe:

- Seleccionar 2 posiciones p_1 y p_2 de manera aleatoria tal que $p_1 < p_2$.
- Barajar los genes que se encuentran entre p_1 y p_2 como se observa en la Figura 1.9.

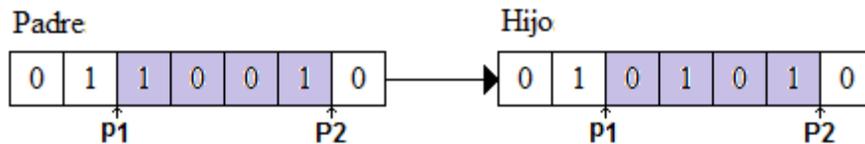


Figura 1.9. Barajeo

c) Intercambio

También conocido como Mutación por Intercambio, se espera que hayan $p_s \cdot \text{tam_poblacion}$ cromosomas a los que se les aplique esta operación, donde p_s representa la probabilidad de intercambio por cromosoma.

Si un cromosoma es seleccionado para realizar un intercambio se debe:

- Seleccionar 2 posiciones p_1 y p_2 de manera aleatoria tal que $p_1 < p_2$.
- Se intercambian los genes que se encuentran en p_1 y p_2 como se aprecia en la Figura 1.10.

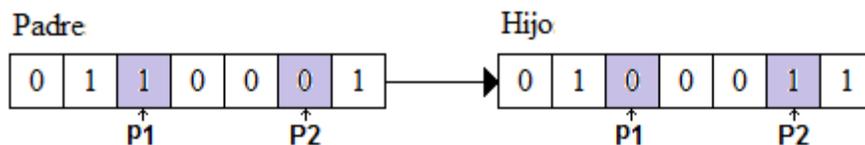


Figura 1.10. Intercambio

1.4. Bibliotecas para el desarrollo de Algoritmos Genéticos

Existen muchas bibliotecas que han sido implementadas para ser usadas en el desarrollo de Algoritmos Genéticos. En esta sección se describen tres bibliotecas que sirven de antecedente

para el desarrollo de la aplicación que se realizó. Por cada biblioteca se describen las características generales y los diagramas de clases. Además, se desarrolla un ejemplo del uso de cada una de éstas, usando un mismo problema, para ver las diferencias que presentan al nivel del código.

1.4.1. GALib

GALib es una biblioteca de C++ para Algoritmos Genéticos. La biblioteca incluye herramientas para desarrollar AG en cualquier programa en C++, usando cualquier representación y cualquier operador genético. [14]

1.4.1.1. Características Generales

- Algunas clases para los genomas hacen uso de Plantillas, pero GALib puede ser usado sin necesidad de usarlos si el compilador no los entiende.
- Cuatro generadores de números aleatorios están incluidos en la biblioteca.
- GALib puede ser usado con PVM (Parallel Virtual Machine) para evolucionar poblaciones y/o individuos en paralelo en múltiples CPUs.
- Los parámetros del AG pueden ser configurados desde un archivo, línea de comandos y/o por código.
- Nuevos AG pueden ser rápidamente implementados, derivando desde la clase base de Algoritmos Genéticos. En muchos casos, solo se necesita sobrescribir una función virtual.
- Los métodos de terminación ya incorporados a la biblioteca son convergencia y número de generaciones. Los métodos de terminación pueden ser personalizados para cualquier AG.
- Los métodos de selección incorporados a la biblioteca son rank, ruleta, torneo, muestreo restante estocástico, muestreo uniforme estocástico y muestreo determinista. El operador de selección puede ser personalizado.
- Las estadísticas del AG son grabadas en un archivo. Las estadísticas guardadas son: máximo, mínimo, media, desviación estándar y diversidad. Se puede especificar cuales estadísticas deben ser grabadas y cuán seguido deben ser enviadas al archivo.
- Los cromosomas pueden ser construidos de cualquier tipo de dato de C++. Se pueden usar los tipos ya incorporados a la biblioteca (bit-string, array, list, tree) o derivar un cromosoma basado en un objeto creado por el programador.
- Los cromosomas ya incorporados en la biblioteca son arreglos, listas, árboles, arreglos de hasta tres dimensiones y strings binarios de hasta tres dimensiones. Los strings binarios, strings y arreglos pueden tener una longitud variable. Las listas y árboles pueden contener cualquier objeto en sus nodos.
- Los métodos de inicialización (*initialization*), mutación (*mutation*), cruce (*crossover*) y comparación (*comparison*), pueden ser personalizados para cualquier cromosoma.
- Los métodos de inicialización ya incorporados son aleatorio uniforme, aleatorio basado en el orden e inicialización en cero.
- Los métodos de mutación ya incorporados son simple, intercambio aleatorio, gaussiano, destructivo, intercambio de sub-árboles, intercambio de nodos.
- Los métodos de cruce ya incorporados son coincidencia parcial, ordenado, ciclo, un punto, dos puntos, pares, impares, uniforme, un punto para nodos y árboles.

En la Figura 1.12 se muestra el código que resuelve el problema propuesto usando GALib.

```
float Objective(GAGenome&); //Función objetivo implementada posteriormente

main(){
    //Inicialización de variables
    int length= 200, pop_size= 50, ngen= 100;
    float pmut= 0.0175, pcross= 0.7;
    //Creación del genoma
    GALDBinaryStringGenome genome(length, Objective);
    //Creación del algoritmo genético
    GASimpleGA ga(genome);
    //Configuración del tamaño de la población
    ga.populationSize(pop_size);
    //Configuración de la cantidad máxima de generaciones
    ga.nGenerations(ngen);
    //Configuración de la probabilidad de mutar un bit
    ga.pMutation(pmut);
    //Configuración de la probabilidad de cruzar dos cromosomas
    ga.pCrossover(pcross);
    // Realiza el proceso de evolución
    ga.evolve();
    //Imprime los resultados
    cout << ga.statistics() << endl;
}

//Implementación de la función objetivo
float Objective(GAGenome&) {
    GALDBinaryStringGenome & genome = (GALDBinaryStringGenome &) g;
    int i, j;
    float values[20], score= 0.0;

    //Conversión de bits a decimales en el rango [-512,512)
    for (i=0;i<20;i++){
        values[i]= -512;
        for(j=0;j<10;j++){
            values[i]+= genome.gene(i*10+j)*pow(2,j);
        }
    }

    //Cálculo de la función objetivo
    for(i=0; i<20; i++)
        score += values[i]*values[i];
    return 5242880 - score; //Máximo valor posible 512^2*20= 5242880
}
```

Figura 1.12. Ejemplo de un programa desarrollado en GALib

1.4.1.4. Definición de operadores

Para asignar un operador a un genoma, basta con usar la función de la clase apropiada, por ejemplo, el siguiente pedazo de código en la Figura 1.13 asigna “MyInitializer” como la función de inicialización y “MyCrossover” como la función de cruce.

```
GALDBinaryStringGenome genome(20);
genome.initializer(MyInitializer);
genome.crossover(MyCrossover);
```

Figura 1.13. Ejemplo de definición de operadores propios en GALib

1.4.2. MATLAB GA Toolbox

GA Toolbox es un paquete de dominio público para MATLAB, cuya finalidad es hacer más accesible el desarrollo de Algoritmos Genéticos mediante bibliotecas de paquetes CACSD (Computer Aided Control System Design). [15]

MATLAB soporta esencialmente un solo tipo de estructura de datos, matrices de números reales o complejos. Por lo tanto, este paquete no está orientado a objetos.

1.4.2.1. Características Generales

- Soporta cromosomas con representación binaria, entera y real.
- Las estructuras principales para GA Toolbox son cromosomas, fenotipos, valores de la función objetivo y valores de aptitud.
- La estructura de los cromosomas almacena una población completa en una sola matriz de $N_i \times L_i$, donde N_i es la cantidad de individuos y L_i es el tamaño de un cromosoma.
- Los fenotipos son almacenados en una matriz de $N_i \times N_v$ donde N_v es la cantidad de variables de decisión.
- Los valores de la función objetivo se almacenan en una matriz de $N_i \times N_o$, donde N_o es la cantidad de objetivos.
- Los valores de aptitud se almacenan en un vector de longitud N_i .
- Cuenta con el método de escalamiento de Goldberg [16] mediante la función *scaling* y el algoritmo de ranking lineal de Baker [8] mediante la función *ranking*.
- La función *ranking* soporta de manera opcional el ranking no lineal.
- Las funciones de selección incluidas en el paquete son *rws* (*Roulette Wheel Selection*) y *sus* (*stochastic universal sampling*).
- Los cruces de un solo punto, dos puntos y barajeado están implementados respectivamente en las funciones *xovsp* (*crossover single point*), *xovdp* (*crossover double point*) y *xovsh* (*crossover shuffle*).
- Cruces con sustituto reducido son soportados para cruces de un solo punto, dos puntos y barajeado mediante las funciones *xovsprs*, *xovdprs*, *xovshrs* (*rs – reduced surrogate*).
- Una función general para cruce multipuntos es implementada bajo la función *xovmp* (*crossover multipoint*) y soporta cruce uniforme.
- La mutación binaria y entera vienen dadas por la función *mut* (*mutation*).
- Para representación real se utiliza la función de mutación para un tipo de AG llamado *breeder* (criador) mediante la función *mutbga* (*mutation breeder genetic algorithm*).

1.4.2.2. Desarrollo de un AG usando GA Toolbox

Como se mencionó anteriormente, se utilizará el mismo problema que fue propuesto en la Sección 1.4.1.3, por lo tanto la representación del cromosoma será la misma.

Al utilizar GA Toolbox, es necesario implementar la función objetivo separado al código del AG. La función objetivo viene dado por el segmento de código mostrado en la Figura 1.14.

```
function ObjVal = objfun1( Phen )
ObjVal = sum((Phen .* Phen)')';
```

Figura 1.14. Función objetivo del ejemplo en GA Toolbox

GA Toolbox cuenta con un convertidor sencillo del cromosoma de binario a real con la función *bs2rv*, se supone para la función objetivo que los parámetros son pasados ya como valores reales. Se muestra en la Figura 1.15 el código que resuelve el problema propuesto usando GA Toolbox.

```
NIND = 50; % Numero de individuos
MAXGEN = 100; % Maximo no. de generaciones
NVAR = 20; % No. de variables
PRECI = 10; % Precision de variables
GGAP = 0.9; % Brecha por Generación
PMUT = 0.0175; %Probabilidad de Mutación
PXOV = 0.7; %Probabilidad de Cruce

% Construcción del Esquema de Representación
FieldD = [rep([PRECI],[1,NVAR]);
          rep([-512;512],[1,NVAR]);
          rep([1;0;1;1],[1,NVAR])];

% Inicializar la población
Chrom = crtbp(NIND, NVAR*PRECI);
gen = 0; % Contador de Generaciones
% Evaluar la población inicial
ObjV = objfun1(bs2rv(Chrom,FieldD));
% Ciclo del Algoritmo Genético
while gen < MAXGEN,
    % Assignar el valor de aptitud a toda la población
    FitnV = ranking(ObjV);
    % Seleccionar individuos para reproducirse
    SelCh = select('sus', Chrom, FitnV, GGAP);
    % Aplicar Cruce
    SelCh = recomb('xovsp', SelCh, PXOV);
    % Aplicar Mutación
    SelCh = mut(SelCh, PMUT);
    % Evaluar descendientes mediante la funcion objetivo
    ObjVSel = objfun1(bs2rv(SelCh,FieldD));
    % Reinsertar descendientes a la población
    [Chrom ObjV]=reins(Chrom, SelCh, 1, 1, ObjV, ObjVSel);
    % Incrementar contador
    gen = gen+1;
end
% Convertir Cromosomas a valores reales
Phen = bs2rv(Chrom, FieldD);
```

Figura 1.15. Ejemplo de un programa desarrollado en MATLAB usando GA Toolbox

1.4.3. JGAP

JGAP (Java Genetic Algorithm Package) es una biblioteca de Java para Algoritmos Genéticos y programación genética, que provee mecanismos genéticos básicos que pueden ser usados fácilmente en la resolución de problemas. [17]

JGAP fue diseñado para ser fácil de usar por programadores principiantes y modular para

que programadores experimentados puedan fácilmente incorporar sus propios operadores genéticos y otros sub-componentes.

1.4.3.1. Características Generales

- Se cuenta con la clase Configuration que permite a JGAP ser flexible a las peticiones del usuario.
- Los cromosomas pueden estar formados por genes de distintos tipos.
- Los genes de los cromosomas pueden ser contruidos de cualquier tipo de dato de Java. Se pueden usar los tipos ya incorporados a la biblioteca o crear nuevos tipos de genes.
- Los genes ya incorporados en el biblioteca son enteros, reales, booleanos, binarios, compuestos y strings.
- Los métodos de selección incorporados a la biblioteca son ruleta, torneo y mejor cromosoma. El operador de selección puede ser personalizado.
- Los métodos de mutación y cruce pueden ser personalizados para cualquier cromosoma.
- Los métodos de mutación ya incorporados son simple y gaussiano.
- Los métodos de cruce ya incorporados son simple, promedio y codicioso.

1.4.3.2 Diagrama de Clases

En la Figura 1.16 se muestra el diagrama de clases de JGAP.

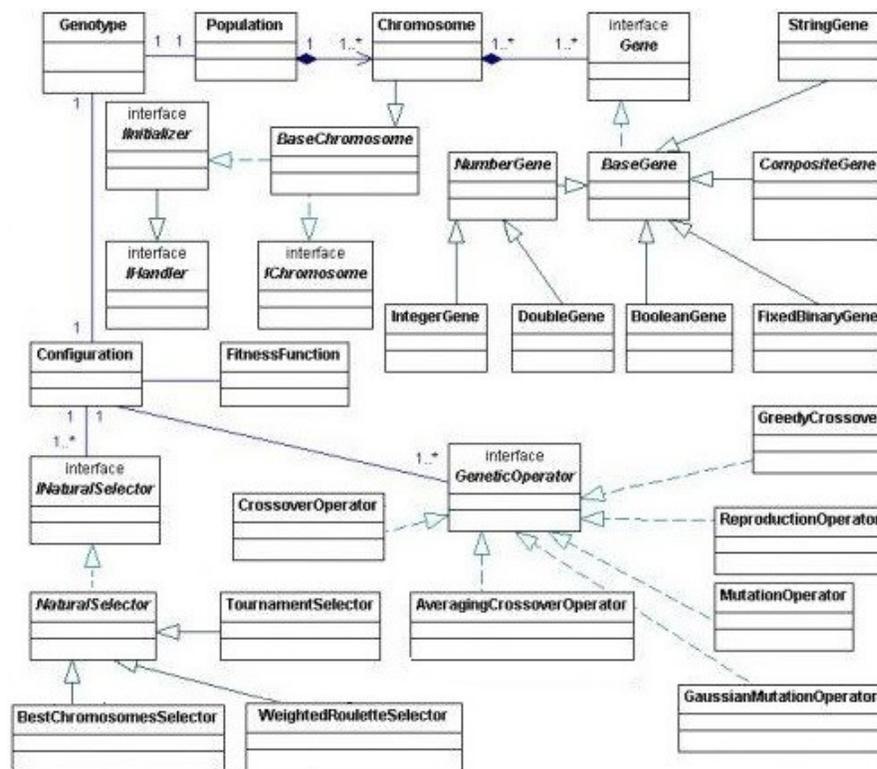


Figura 1.16. Diagrama de Clases de JGAP

1.4.3.3. Desarrollo de un AG usando JGAP

Al igual que con GAlib y GA Toolbox, se usará el problema propuesto en la Sección 1.4.1.3.

JGAP divide la resolución de un AG en dos grandes partes, la primera se muestra en la Figura 1.17 y consiste en el desarrollo de la clase principal, la cual se encargará de la corrida del algoritmo, así como de las configuraciones que sean necesarias para la resolución del problema.

```
Class Principal{
    public static void main(String[] args){
        int MAXGEN= 100;

        //Crear un objeto de Configuration
        Configuration conf = new DefaultConfiguration();

        //Crear un objeto de la función aptitud
        FitnessFunction myFunc = new MiFuncionDeMinimizacion()

        //Configuración de la función de aptitud a ser usada
        conf.setFitnessFunction( myFunc );

        //Se crea un arreglo de genes que formaran al cromosoma
        Gene[] sampleGenes = new Gene[20];

        //Se inicializan los genes como genes enteros en el rango [-512,512)
        for(i=0; i<20; i++)
            sampleGenes[i] = new IntegerGene(-512, 511);

        //Se crea un objeto del tipo Chromosome, al cual le pasamos
        //los genes por parámetro
        Chromosome sampleChromosome = new Chromosome(sampleGenes);

        //Configuración del tipo de cromosoma a usarse.
        conf.setSampleChromosome( sampleChromosome );

        //Configuración del tamaño de la población
        conf.setPopulationSize( 50 );

        //Crear la población utilizando la configuraciones en conf
        Genotype population = Genotype.randomInitialGenotype(conf);

        for(int i = 0; i < MAXGEN; i++)
            population.evolve();

        Chromosome mejorSolucion = population.getFittestChromosome();

        System.out.println("La mejor solución es:");

        for( i=0; i < 20; i++){
            System.out.print("Cromosoma " + i + ": " +
                mejorSolucion.getAllele(i).getValue());
        }
    }
}
```

Figura 1.17. Ejemplo de un programa desarrollado en JGAP – Parte 1

La segunda consiste en la implementación de la función de aptitud, que debe ser implementado en una clase separada a la principal, como se muestra en la Figura 1.18.

```
public class MiFuncionDeMinimizacion extends FitnessFunction{
    //Para este problema no se necesita inicializar nada en el construct-
or
    public MiFuncionDeMinimizacion(){}

    //Determina la aptitud de la instancia del cromosoma.
    public double evaluate(Chromosome cromosoma){
        double aptitud = -512;
        int value;
        for (i=0; i<20; i++){
            //Asignamos el valor del gen en la posición i a value
            value= cromosoma.getAllele(i).getValue();
            fitness+= value*value;
        }

        //Máximo valor posible  $512^2 \cdot 20 = 5242880$ 
        return 5242880 - fitness;
    }
}
```

Figura 1.18. Ejemplo de un programa desarrollado en JGAP – Parte 2

1.4.4. Comentarios del uso de bibliotecas para AG

Como se observó en el ejemplo con las bibliotecas GALib, GA Toolbox y JGAP; no es necesario que el usuario escriba más líneas de código que las necesarias para definir la función de aptitud y la configuración del AG; siendo una gran ventaja frente a no usar una, pues se salta el proceso de la implementación de los operadores genéticos y los cromosomas.

Hacer uso de una biblioteca para la resolución de un AG conlleva además otras ventajas:

- Permiten analizar resultados con diferentes operadores genéticos sobre un mismo problema con solo cambiar unas pocas líneas de código.
- Algunas bibliotecas, como GALib y JGAP, poseen la facilidad de poder implementar nuevos operadores genéticos sin complicaciones.

Sin embargo, estas bibliotecas tienen también desventajas, siendo la más clara que éstas tienen restringido su uso únicamente a programadores experimentados en el lenguaje en que fue desarrollado la herramienta.

Capítulo 2. Marco Aplicativo

Los AG son muy usados en la actualidad, debido a que éstos pueden conseguir soluciones cercanas al óptimo para problemas de optimización combinatoria en el mismo tiempo en que consigue una solución para cualquier otro problema. Debido a esto, los AG tienden a ser más rápidos para problemas de optimización combinatoria que otros algoritmos conocidos.

Para el desarrollo de AG se han creado muchas herramientas. Sin embargo, como se mencionó en la Sección 1.4.4, estas herramientas tienen un problema en común; requieren que el usuario tenga conocimientos del lenguaje de programación en el cual fueron implementadas, restringiendo su uso a solamente programadores o requiriendo de un tiempo adicional para adiestrarse en éstas.

Es por esto que se decidió desarrollar una herramienta para la resolución de problemas mediante AG, que permita a usuarios no experimentados en programación utilizar la herramienta, permitiendo que los AG puedan ser usados con mayor facilidad.

La creación de esta aplicación para el desarrollo de AG permite:

- Expandir el uso de AG en áreas de investigación no relacionadas a la computación.
- Incrementar el uso de AG en el ámbito empresarial.
- Facilitar el estudio de los AG en las universidades.
- Reducir el tiempo de desarrollo de un AG.

2.1. Objetivo General

Desarrollar una aplicación que provea herramientas útiles para la implementación de Algoritmos Genéticos, minimizando la programación a través de una interfaz gráfica, con la finalidad de ser utilizada primordialmente en pruebas y como material de apoyo.

2.2. Objetivos Específicos

- Implementar una biblioteca que permita el desarrollo de AG.
- Diseñar una interfaz que permita utilizar la biblioteca a través de una aplicación.
- Acoplar mediante un módulo de comunicación la biblioteca a la interfaz.
- Realizar pruebas sobre la aplicación con problemas conocidos para verificar su desempeño.

2.3. Desarrollo de la Solución

Para el desarrollo de la solución se utilizó una técnica Ad Hoc ágil, la cual consistió en la

repetición de los siguientes dos pasos:

- Implementar un sub-problema de la solución.
- Realizar Pruebas.

Estos pasos se realizaron desde los problemas más específicos hasta los problemas más generales, hasta finalmente llegar al resultado final.

La aplicación se divide en dos problemas: una interfaz y una biblioteca. Posteriormente se acoplaron estas partes mediante un módulo de comunicación como se muestra en la Figura 2.1.

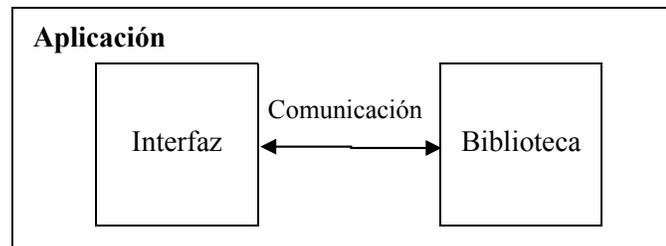


Figura 2.1. Diseño de la aplicación

En esta sección se explica en detalle la información relacionada a las tecnologías utilizadas, la biblioteca y la interfaz.

2.3.1. Tecnologías Utilizadas

Para el desarrollo de la aplicación se hizo uso del lenguaje de programación Java utilizando programación orientada a objetos.

Se utilizó eclipse como IDE para el desarrollo de la interfaz, junto con una extensión para diseño de Interfaces en Java llamada *WindowBuilder*¹.

2.3.2. Biblioteca

Para la biblioteca de AG se propone un diagrama de clases, donde se debe tomar en cuenta que:

- Sólo se muestran las relaciones entre las clases. Los métodos y atributos se explicarán luego.
- Las clases cuyo nombre se encuentre en cursiva son clases abstractas.

En la Figura 2.2 se observa un diagrama general de la biblioteca sin incluir las subclases. Se muestra además la conexión entre la biblioteca y la interfaz mediante la clase *GAL_Interface* (se realizó una abstracción de la interfaz mediante la representación de ésta con la clase *GAL_GUI* la cual es la clase principal de la interfaz).

¹ WindowBuilder: <http://www.eclipse.org/windowbuilder/>

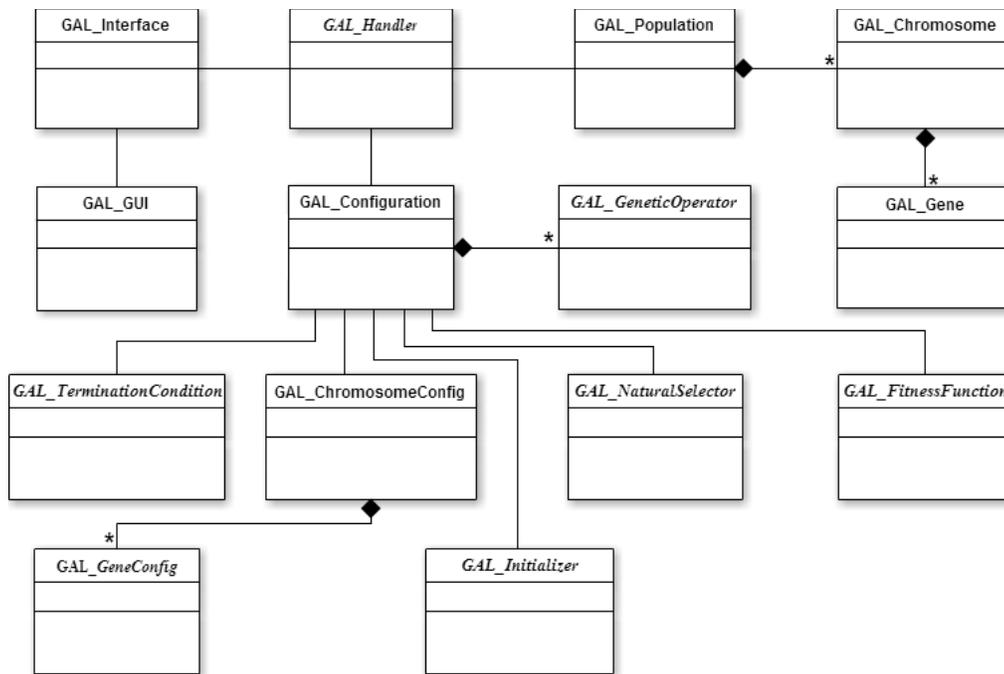


Figura 2.2. Diagrama De Clases 1 – Diagrama General

La biblioteca fue liberada como un proyecto de Software Libre con el nombre JGAL(Java Genetic Algorithm)². Se decidió utilizar inglés para los nombres de las clases, métodos y atributos, debido a que se acostumbra que el Software Libre esté en este lenguaje.

2.3.2.1. Explicación detallada de clases

A continuación se muestra una explicación detallada de las clases más importantes de la biblioteca.

GAL_Handler

Es una clase abstracta utilizada para definir manejadores de AG. Toda subclase del *GAL_Handler* debe definir un método para *runGAL*, el cual se encarga de ejecutar el AG con la configuración dada.

Como se muestra en la Figura 2.3, de esta clase heredan *GAL_ClassicHandler* y *GAL_ModHandler*, los cuales implementan dos variantes del AG descritos previamente en las Secciones 1.2.4 y 1.3.2 respectivamente.

² JGAL: <https://github.com/chango1611/JGAL>

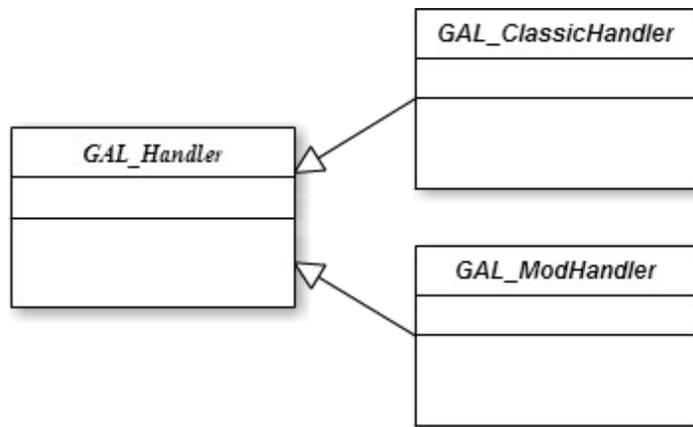


Figura 2.3. Diagrama de Clases 2 – Manejadores

Atributos:

- protected double[] averageFitnessFromGeneration: utilizado para almacenar la aptitud promedio de cada generación.
- protected GAL_Chromosome[] bestChromosomeFromGeneration: utilizado para almacenar el mejor cromosoma de cada generación.
- protected GAL_Configuration configuration: La configuración a ser utilizada por el Manejador.
- protected int lastGeneration: La última generación que se ejecuto.
- protected int maxGeneration: La máxima cantidad de generaciones permitidas.
- protected int populationSize: El tamaño de la población.
- protected java.util.LinkedList<GAL_Population> window: Almacena la ventana de poblaciones a ser usada en la condición de terminación.
- protected int windowSize: El tamaño máximo de la ventana.

Métodos:

- double getAverageFitnessFrom(int pos): Retorna la aptitud promedio para la generación descrita por el parámetro.
- double getAverageFitnessFromAll(): Retorna la aptitud promedio de todas las generaciones.
- double[] getAverageFitnessFromEach(): Retorna un arreglo de doubles donde cada elemento representa la aptitud promedio de cada generación.
- GAL_Chromosome getBestFrom(int pos): Retorna el mejor cromosoma de la generación descrita por el parámetro.
- GAL_Chromosome getBestFromAll(): Retorna el mejor cromosoma de todas las generaciones.
- GAL_Chromosome[] getBestFromEach(): Retorna un arreglo donde cada elemento representa el mejor cromosoma de cada generación.
- int getLastGenerationNumber(): Obtiene el número de la última generación que fue ejecutada + 1.

- GAL_Population getPopulationFromWindow(int pos): Obtiene la población de la ventana que se encuentra en la posición pasada por parámetro.
- GAL_Population[] getWindow(): Obtiene la ventana de poblaciones.
- abstract void runGAL(): corre el Algoritmo Genético.
- protected void saveData(GAL_Population population): Guarda la información relacionada al problema. Debe ser usado en cada ciclo del runGAL para que funcione correctamente.

GAL_Configuration

Contiene toda la información de configuración necesaria para correr un AG. Además posee muchos métodos útiles para el manejador.

Atributos:

- protected GAL_ChromosomeConfig chromosomeConfig: la configuración para los cromosomas a ser usada por el AG.
- protected GAL_TerminationCondition condition: La condición de terminación a ser usada por el AG.
- protected GAL_FitnessFunction fitnessFunction: La función de aptitud a ser usada por el AG.
- protected GAL_GeneticOperator[] operators: Los operadores genéticos a ser usados por el AG.
- protected GAL_NaturalSelector selector: El selector natural a ser usado por el AG.

Métodos:

- double changeProbTo(int pos, double prob): Cambia la probabilidad de ocurrencia del operador genético en la posición pasada por parámetro.
- void computeAllFitness(GAL_Population population): Calcula la aptitud de una población haciendo uso del campo fitnessFunction.
- GAL_Population createNewPopulation(GAL_Chromosome[] chrom): Crea una nueva población bajo la configuración dada por chromosomeConfig y un arreglo de cromosomas pasado por parámetro.
- GAL_Population createNewPopulation(int size): Crea una nueva población bajo la configuración dada por chromosomeConfig.
- GAL_Population operatePopulation(GAL_Population origin): Aplica todos los operadores genéticos en el orden en que se encuentren dentro del arreglo de operadores.
- GAL_Population operatePopulation(GAL_Population origin, int pos): Aplica el operador genético que se encuentre en la posición pasada por parámetro.
- int operatorsArraySize(): Retorna el tamaño del arreglo de operadores genéticos.
- GAL_Population selectNewPopulation(GAL_Population origin): Selecciona una nueva población a partir del selector natural.
- boolean verifyTerminationCondition(GAL_Population[] window): Verifica si la

condición de terminación se cumple.

GAL_ChromosomeConfig

Es una clase utilizada para configurar los cromosomas. Incluye métodos para la creación de nuevos cromosomas.

Atributos:

- protected GAL_GeneConfig[] configuration: un arreglo con la configuración de cada uno de los genes del cromosoma.

Métodos:

- GAL_ChromosomeConfig clone(): Retorna una copia de este GAL_ChromosomeConfig.
- GAL_Chromosome createNewChromosome(): Crea y retorna un nuevo GAL_Chromosome usando el método createNewGene en GAL_GeneConfig.
- GAL_GeneConfig[] getConfiguration(): Retorna el arreglo de configuraciones de genes.
- GAL_GeneConfig<?> getGeneConfiguration(int pos): Retorna la configuración del gen que se encuentre en la posición pasada por parámetro.
- void modifyChromosome(GAL_Chromosome chrom, GAL_Chromosome chrom2, int pos): Modifica un gen del primer cromosoma a partir del segundo cromosoma en la posición pasada por parámetro.
- void modifyChromosome(GAL_Chromosome chrom, GAL_Chromosome chrom2, int min, int max): Modifica los genes del primer cromosoma a partir del segundo cromosoma en las posiciones que se encuentren entre min y max-1.
- void modifyChromosome(GAL_Chromosome chromosome, int pos): Modifica aleatoriamente el gen de un cromosoma en la posición pasada por parámetro.
- void modifyChromosome(GAL_Chromosome chromosome, int min, int max): Modifica aleatoriamente el gen de un cromosoma en las posiciones que se encuentren entre min y max-1.
- int size(): Retorna el tamaño del arreglo de configuración de genes.

GAL_GeneConfig<T>

Es una clase abstracta utilizada para definir Genes cuyo rasgo es de tipo T. Toda subclase de *GAL_GeneConfig* debe implementar los métodos *setRandomValueTo* y *setValueTo*.

Las subclases implementadas para *GAL_GeneConfig* pueden ser observadas en la Figura 2.4.

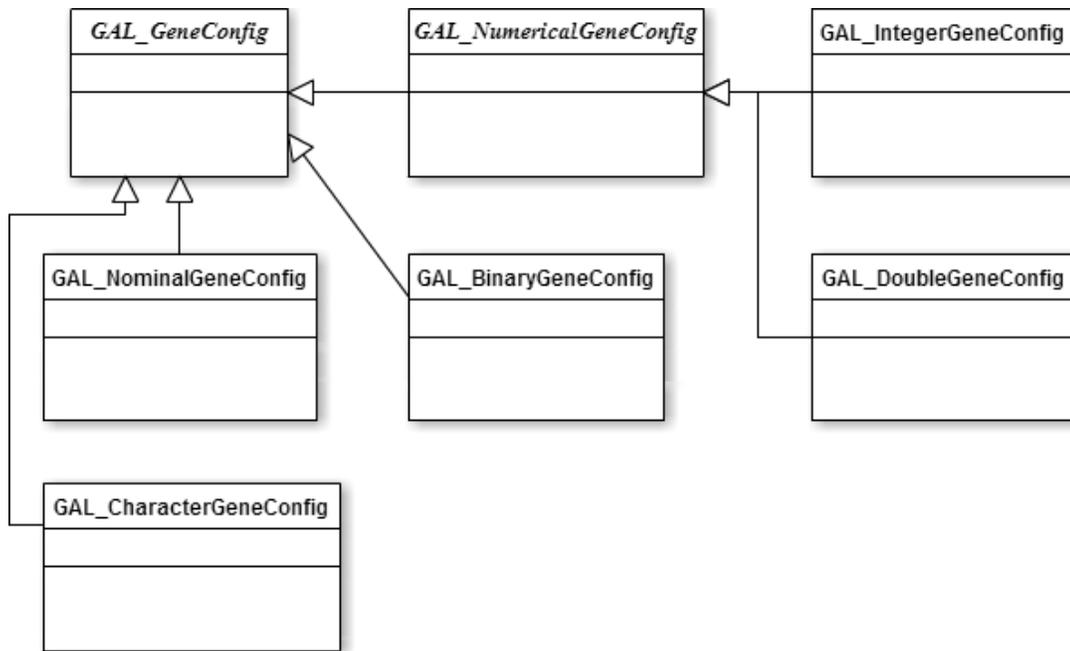


Figura 2.4. Diagrama de Clases 3 – Genes

Los genes numéricos y de caracter agregan los campos min y max que permiten limitar el rango, mientras que los nominales agregan un arreglo que posee los posibles valores que puede tener el gen.

Atributos:

- protected java.lang.String name: Nombre opcional del gen.
- protected java.util.Random rand: Objeto Random que puede ser usado para definir el método setRandomValueTo.

Métodos:

- abstract void changeValueTo(GAL_Gene gene): Cambia el rasgo de un gen a otro que sea permitido en los alelos.
- GAL_Gene createNewGene(): Crea y retorna un nuevo GAL_Gene usando el método setRandomValueTo.
- java.lang.String getName(): Retorna el nombre del gen.
- void setName(java.lang.String name): Establece un nuevo valor al nombre del gen.
- abstract void setRandomValueTo(GAL_Gene gene): Asigna un valor aleatorio al rasgo de un GAL_Gene manteniendo la configuración.
- abstract void setValueTo(GAL_Gene gene, T val): Asigna un valor al rasgo de un gen.
- void setValueToFrom(GAL_Gene gene, GAL_Gene gene2): Asigna el rasgo del segundo GAL_Gene al valor del rasgo del primer GAL_Gene.

GAL_FitnessFunction

Es una clase abstracta utilizada para definir la función de aptitud. Toda subclase de *GAL_FitnessFunction* debe implementar el método *computeFitness*.

De usar la biblioteca, el usuario deberá crear una clase que herede de *GAL_FitnessFunction* para ser usado como la función de aptitud del problema que se esté resolviendo.

Métodos:

- abstract double computeFitness(GAL_Chromosome chromosome): Calcula y guarda la aptitud del cromosoma pasado por parámetro.

GAL_TerminationCondition

Es una clase abstracta utilizada para definir la condición de terminación. Toda subclase de *GAL_TerminationCondition* debe implementar la función *evaluateTerminationCondition*.

De usar la biblioteca, el usuario puede crear una clase que herede de *GAL_TerminationCondition* para ser usado como la condición de terminación del problema que se esté resolviendo, sino se utilizara la condición de terminación por defecto *GAL_DefaultTerminationCondition*.

Métodos:

- abstract boolean evaluateTerminationCondition(GAL_Population[] window): Verifica si se cumple la condición de terminación.

GAL_Initializer

Es una clase abstracta utilizada para definir la inicialización de la población. Toda subclase de *GAL_Initializer* debe implementar la función *initialize*.

De usar la biblioteca, el usuario puede crear una clase que herede de *GAL_Initializer* para ser usado como la inicialización de la población, sino se utilizara la inicialización por defecto *GAL_DefaultInitializer*.

La biblioteca cuenta con la inicialización para problemas de permutación *GAL_PermutationsInitializer* el cual se puede usar bajo 3 condiciones:

1. Todos los genes deben tener la misma configuración.
2. Todos los genes deben ser de tipo Entero.
3. El valor máximo menos el valor mínimo de cada gen debe ser mayor o igual que el tamaño del cromosoma.

Métodos:

- abstract boolean initialize(GAL_ChromosomeConfig chromosomeConfig, int size): Crea una nueva población de tamaño size siguiendo la configuración pasada por parámetro.

GAL_NaturalSelector

Es una clase abstracta utilizada para definir el operador de selección. Toda subclase de *GAL_NaturalSelector* debe implementar la función *selectNewPopulation*.

Las subclases implementadas para *GAL_NaturalSelector* se muestran en la Figura 2.5. La información sobre el comportamiento de los selectores implementados puede ser hallada en la Sección 1.3.1.

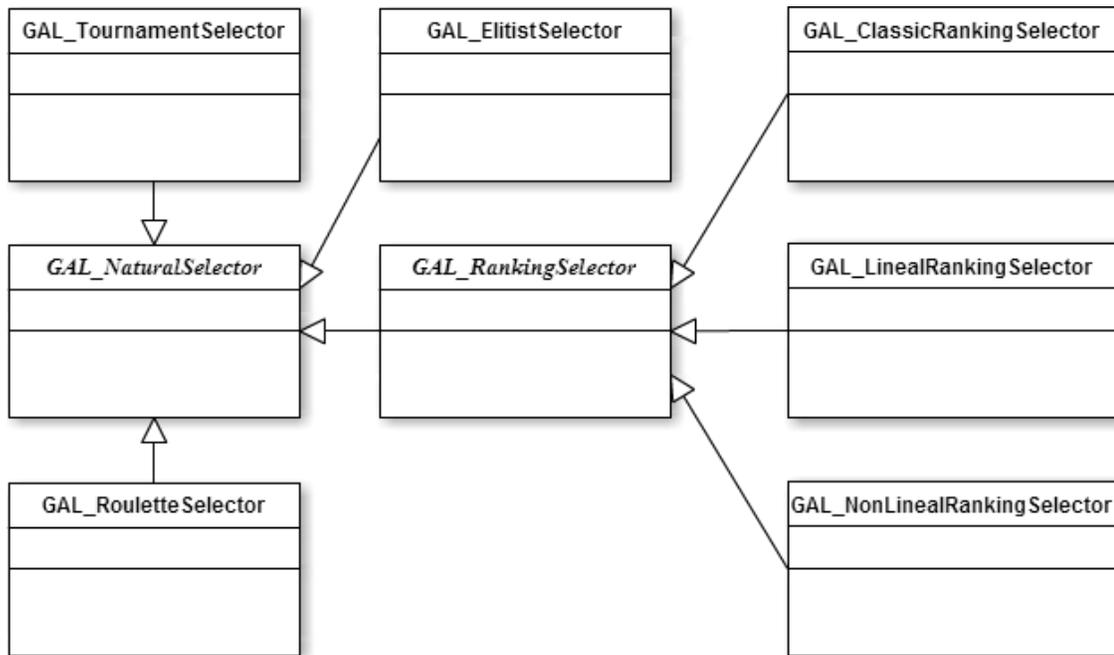


Figura 2.5. Diagrama de Clases 4 – Selector Natural

Atributos:

- `protected java.util.Random rand`: Objeto aleatorio que se puede utilizar dentro de la función *selectNewPopulation*.

Métodos:

- `abstract GAL_Population selectNewPopulation(GAL_Population origin, GAL_Chromosome Config config)`: Selecciona la nueva población a ser usada en la siguiente generación.

GAL_GeneticOperator

Es una clase abstracta utilizada para definir los operadores genéticos. Toda subclase de *GAL_GeneticOperator* debe implementar la función *applyOperator*.

Las subclases implementadas para *GAL_GeneticOperator* se reflejan en la Figura 2.6.

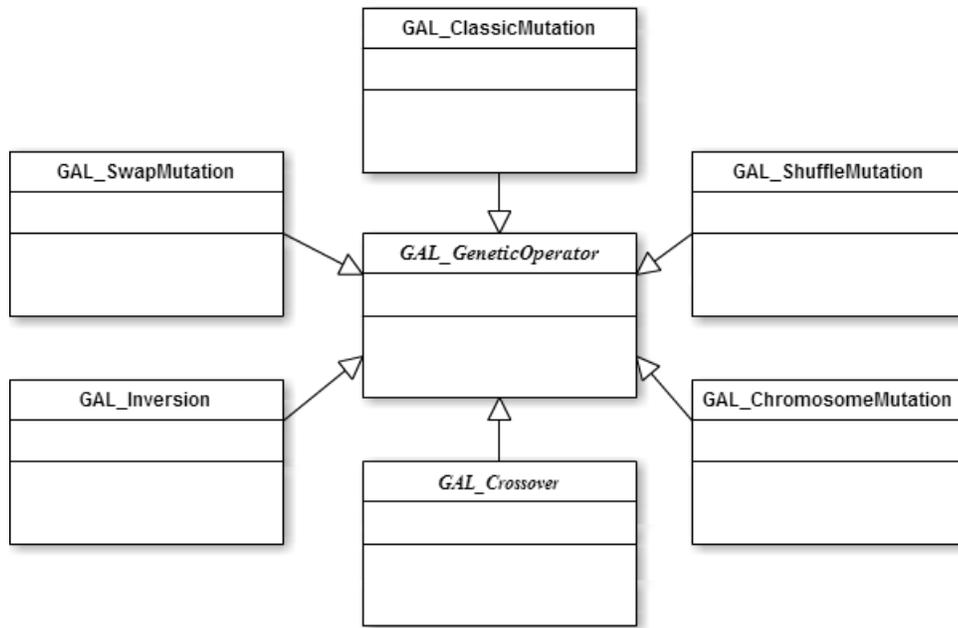


Figura 2.6. Diagrama de Clases 5 – Operadores Genéticos

Además, *GAL_Crossover* se subdivide en las clases que se muestran en la Figura 2.7.

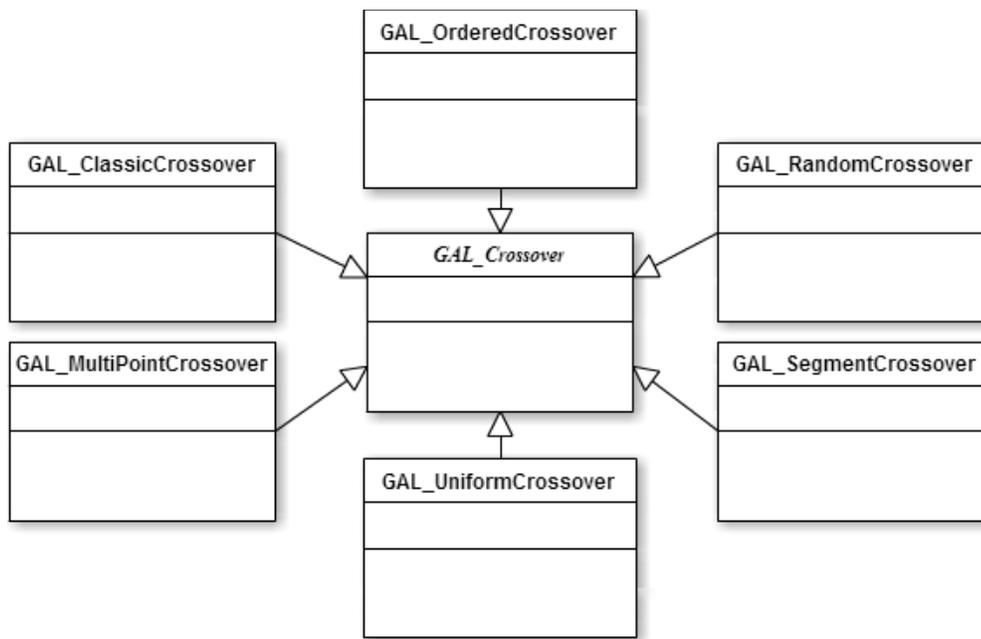


Figura 2.7. Diagrama de Clases 6 – Operadores de Cruce

El comportamiento de los cruces pueden ser observadas en la Sección 1.3.3. El comportamiento de *GAL_Inversion*, *GAL_SwapMutation* y *GAL_ShuffleMutation* se observan en la sección 1.3.4. Finalmente, *GAL_ChromosomeMutation* es una variante de la Mutación Clásica para poder ser usada con el AG Modificado.

Atributos:

- protected double prob: Probabilidad de ocurrencia del operador genético.
- protected java.util.Random rand: Objeto aleatorio que se puede utilizar dentro de los métodos.

Métodos:

- abstract GAL_Population applyOperator(GAL_Population fathers, GAL_Chromosome someConfig config): Aplica la operation sobre una población bajo las restricciones dadas en la configuración del cromosoma.
- double getProb(): Retorna la probabilidad de ocurrencia del operador genético.
- void setProb(double prob): Asigna una nueva probabilidad de ocurrencia.

GAL_Population

Representa una población de cromosomas para una generación. Los constructores públicos de esta clase requieren que se pase por parámetro la configuración del cromosoma, con la finalidad de evitar futuras incongruencias.

Atributos:

- protected GAL_Chromosome[] chromosomes: Los cromosomas que representan a la población.

Métodos:

- GAL_Population clone(): Retorna una copia de este GAL_Population.
- GAL_Chromosome getBestChromosome(): Retorna el cromosoma con la mayor aptitud en la población.
- GAL_Chromosome getChromosome(int pos): Retorna el cromosoma en la posición pasada por parámetro.
- GAL_Chromosome[] getChromosomes(): Retorna un arreglo con todos los cromosomas de la población.
- double getTotalFitness(): Retorna la sumatoria de todas las aptitudes de una población.
- int size(): Retorna el tamaño de la población.
- java.lang.String toString(): Retorna un String que representa a este GAL_Population.

GAL_Chromosome

Representa cualquier tipo de cromosoma. Utiliza *GAL_ChromosomeConfig* para su configuración.

Atributos:

- protected double fitness: La aptitud del cromosoma actual.

- protected GAL_Gene[] genes: Los genes que representan al cromosoma.

Métodos:

- GAL_Chromosome clone(): Retorna una copia de este GAL_Chromosome.
- int compareTo(GAL_Chromosome other): Compara dos GAL_Chromosomes por su aptitud.
- double getFitness(): Retorna la aptitud del cromosoma.
- GAL_Gene getGene(int pos): Retorna el gen en la posición pasada por parámetro.
- GAL_Gene[] getGenes(): Retorna un arreglo con todos los genes del cromosoma.
- java.lang.Object getTrait(int pos): Retorna el rasgo del gen que se encuentra en la posición pasada por parámetro.
- void setFitness(double fitness): Asigna una aptitud al cromosoma.
- void setGene(GAL_Gene gene, int pos): Asigna un gen a la posición pasada por parámetro.
- int size(): Retorna el tamaño del cromosoma.
- java.lang.String toString(): Retorna un String que representa a este GAL_Chromosome.

GAL_Gene

Representa cualquier tipo de gen. Utiliza *GAL_GeneConfig* para su configuración.

Atributos:

- protected java.lang.Object trait: Un objeto que representa el rasgo del gen.

Métodos:

- GAL_Gene clone(): Retorna una copia de este GAL_Gene.
- java.lang.Object getTrait(): Retorna el rasgo del gen.
- void setTrait(java.lang.Object trait): Asigna un valor al rasgo del gen.
- java.lang.String toString(): Retorna un String que representa a este GAL_Gene.

GAL_Util

Grupo de métodos que son útiles para el desarrollo de un AG, especialmente para la creación de operadores genéticos y selectores naturales.

Atributos:

- protected static java.util.Random rand: Un objeto aleatorio que es usado dentro de algunos métodos de la clase GAL_Util.

Métodos:

- static <T> T[] concatArrays(T[][] multiArray, T[] type): Concatena muchos arreglos

- del mismo tipo y lo retorna como un nuevo arreglo.
- static <T> T[] concatArrays(T[] array1, T[] array2, T[] type): Concatena dos arreglos del mismo tipo y lo retorna como un nuevo arreglo.
- static <T> T[][] divideAt(T[] array, int pos, T[][] type): Divide un arreglo de cualquier tipo en la posición pasada por parámetro.
- static <T> T[][] extractFrom(T[] array, int number, T[][] type): Divide un arreglo de cualquier tipo, de manera que se elige un número de elementos aleatoriamente para formar parte del segundo grupo, mientras que el resto forma parte del primer grupo.
- static <T> void orderBy(T[] array, T[] array2, int[] order): Ordena el primer arreglo y lo coloca sobre el segundo, haciendo uso de un arreglo de ordenamiento.
- static <T> void shuffle(T[] array): Mezcla los valores de un arreglo.

Excepciones

La biblioteca maneja cuatro tipos de excepciones como se muestra en la Figura 2.8

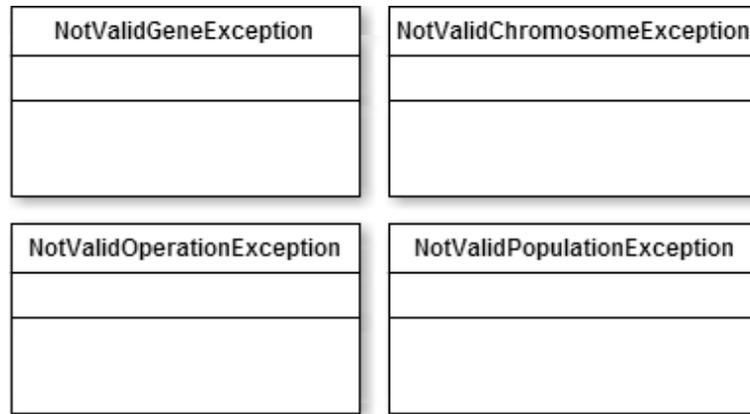


Figura 2.8. Diagrama de Clases 7 – Excepciones

A continuación se explica bajo que condiciones pueden ocurrir cada una de estas excepciones.

- NotValidGeneException: Esta excepción se arroja cuando se intenta crear un gen que no cumple con la configuración o cuando la configuración del gen no es válida.
- NotValidChromosomeException: Esta excepción se arroja cuando se intenta crear un cromosoma que no cumple con la configuración o cuando la configuración del cromosoma no es válida.
- NotValidPopulationException: Esta excepción se arroja cuando se intenta crear una población que no cumple con la configuración.
- NotValidOperationException: Esta excepción se arroja cuando no es posible realizar alguna operación durante la ejecución del AG; incluyendo aplicar Operadores Genéticos y seleccionar la nueva población.

2.3.2.2. Pruebas realizadas sobre la biblioteca

Se realizaron tres casos de prueba con la finalidad de verificar el funcionamiento de la

biblioteca y chequeo de errores.

Caso de Prueba 1

El primer caso de prueba es el mismo que se propuso en el capítulo uno para usar con las otras bibliotecas. Se propone minimizar el valor retornado por la función:

$$f(x) = \sum_{i=1}^n x_i^2, \quad -512 \leq x_i < 512$$

Donde n representa la cantidad de variables. Para este ejemplo, n= 20. Nótese que para esta función, el mínimo es $x_i=0$ para todo i en el rango [1,20]

Número de Generaciones= 100

Tam. Población= 50

Todo problema que se resuelva utilizando esta biblioteca requiere que el usuario diseñe al menos dos clases (Clase Principal y Función de Aptitud).

El código que se muestra en la Figura 2.9 corresponde a la Clase Principal.

```
class Casol{
    public static void main(String[] args) throws Exception{
        //Creo los genes
        GAL_GeneConfig[] geneConfig= new GAL_GeneConfig[20];
        for(int i=0;i<20;i++)
            geneConfig[i]= new GAL_IntegerGeneConfig(-512,512);

        //Creo la configuracion
        GAL_Configuration config= new GAL_Configuration(
            new GAL_ChromosomeConfig(geneConfig),
            new miFuncionAptitud(),
            new GAL_RouletteSelector(),
            new GAL_GeneticOperator[] {
                new GAL_ClassicCrossover(0.7),
                new GAL_Mutation(0.0175)
            }
        );

        //Creo el handler
        GAL_Handler handler= new GAL_ClassicHandler(config,100,50,1);
        handler.runGAL();

        //Imprimo los resultados
        System.out.println("Mejor cromosoma:\n" +
            handler.getBestFromAll());
        System.out.println("Aptitud: " +
            (5242880 - handler.getBestFromAll().getFitness()));
        System.out.println("\nPromedio de las aptitudes:\n" +
            handler.getAverageFitnessFromAll());
    }
}
```

Figura 2.9. Clase principal del primer caso de prueba

Y la Figura 2.10 corresponde a la Función de Aptitud que se diseño para este problema.

```
public class miFuncionAptitud extends GAL_FitnessFunction{

    public miFuncionAptitud(){}

    public double computeFitness(GAL_Chromosome chromosome){
        double fitness= 0; //La aptitud que se retornará
        for(int i=0;i<20;i++)
            fitness+= (int)chromosome.getGene(i).getTrait()*
                (int)chromosome.getGene(i).getTrait();

        //Maximo valor posible 512^2 * 20 = 5242880
        return 5242880 - fitness;
    }
}
```

Figura 2.10. Función de Aptitud del primer caso de prueba

Si se desea utilizar una condición de terminación se debe implementar una clase para ésta. Sin embargo para este problema se decidió utilizar la condición de terminación por defecto (*GAL_DefaultTerminationCondition*).

Se realizaron pruebas con diferentes configuraciones como se muestra en la Tabla 2.1, obteniendo los siguientes resultados.

Manejador	Selección	Operadores Genéticos	Mejor Cromosoma	Valor Total
Clásico	Ruleta	C.Clásico(0.7), Mutación(0.0175)	76;-8;-186;252;167;29;204;75;20;221; 128;-148;132;-275;7;116;55;-130;167;7;	421857.0
Clásico	Ranking Clásico(2)	C.Segmentos(0.7,0.5), Mutación(0.0175)	44;-29;43;-5;36;12;32;30;-16;-21; -17;31;-13;-8;-22;-8;21;-10;-18;13;	11777.0
Clásico	Torneo(2)	C.Clásico(0.7), Mutación(0.0175)	8;-5;-19;-75;-25;15;-9;33;63;-9; -51;-45;75;-11;-71;30;10;9;32;64;	33759.0
Clásico	Torneo(2)	C.Uniforme(0.7,0,5), Mutación(0.0175)	24;38;-5;-10;-5;20;0;15;-28;-11; -11;29;28;8;18;-10;37;19;-21;-35;	9330.0
Modificado(30)	Torneo(2)	C.Uniforme(0.7,0,5), Mutación Chrom(0.3,0.06)	10;-24;3;-11;12;-19;23;16;-57; -7;27;-2;-16;4;43;54;11;62;9;-2	15214.0

Tabla 2.1. Resultados del primer caso de prueba

El Selector de Torneo obtuvo mejores resultados que el Selector de Ruleta. Además, los Cruces Uniforme y por Segmento obtuvieron mejores resultados que el Cruce Clásico.

Caso de Prueba 2

El problema de la mochila, comúnmente abreviado por KP (del inglés *knapsack problem*) es un problema de optimización combinatoria. Modela una situación análoga al llenar una mochila, incapaz de soportar más de un peso determinado, con todo o parte de un conjunto de objetos, cada uno con un peso y valor específicos. Los objetos colocados en la mochila deben maximizar el valor total sin exceder el peso máximo.

Dado los siguientes objetos:

Valor= {45,10,71,49,41,26,32,18,23,15}

Peso= {12,30,54,23,56,42,23,60,54,10}

Y una mochila que aguanta 250 de peso como máximo. Resolver el Problema usando AG.

Número de Generaciones= 20

Tam. Población= 50

Para este caso se decidió utilizar una condición de terminación. Se conoce previamente que 289 es el valor total de la mochila para la mejor solución, pero no se conoce cual es la distribución de la mochila para dicha solución, de manera que si se consigue una aptitud igual a 289, se puede detener la ejecución, dado que se llegó al óptimo. Con la información anterior se crea una condición de terminación como la que se muestra en la Figura 2.11

```
public class miCondicionTerminacion extends GAL_TerminationCondition{
    public miCondicionTerminacion() {}
    public boolean verifyTerminationCondition(GAL_Population[] window){
        //Si la aptitud de la ultima poblacion en evaluarse es mayor que 288
        return window[0].getBestChromosome().getFitness()>288;
    }
}
```

Figura 2.11. Condición de Terminación del segundo caso de prueba

La función de aptitud sería como se muestra en la Figura 2.12.

```
public class miFuncionAptitud extends GAL_FitnessFunction{
    double[] points, weight; //valor y peso
    double max_weight; //maximo peso
    public miFuncionAptitud(double[] points, double[] weight,
    double max_weight){
        this.points= points;
        this.weight= weight;
        this.max_weight= max_weight;
    }
    public double computeFitness(GAL_Chromosome chromosome){
        double p= 0, w=0;
        for(int i=0;i<10;i++){
            if((byte)chromosome.getGene(i).getTrait() == (byte)1){
                p+=points[i]; w+= weight[i];
            }
        }
        if(w>max_weight) //Si el peso se pasa del máximo retorno 0
            return 0;
        return p;
    }
}
```

Figura 2.12. Función de Aptitud del segundo caso de prueba

Finalmente, la clase principal sería como la que se muestra en la Figura 2.13.

```

class Caso2{
    public static void main(String[] args) throws Exception{
        GAL_GeneConfig[] geneConfig= new GAL_GeneConfig[9];

        //Creo los genes
        for(int i=0;i<10;i++)
            geneConfig[i]= new GAL_BinaryGeneConfig();

        double[] p= {45,10,71,49,41,26,32,18,23,15},
            w={12,30,54,23,56,42,23,60,54,10};
        double mw= 250;

        //Creo la configuracion
        GAL_Configuration config= new GAL_Configuration(
            new GAL_ChromosomeConfig(geneConfig),
            new miCondicionTerminacion(),
            new miFuncionAptitud(p,w,mw),
            new GAL_ClassicRankingSelector(2),
            new GAL_GeneticOperator[] {
                new GAL_SegmentCrossover(0.7,0.5),
                new GAL_Mutation(0.0175)
            }
        );

        //Creo el handler
        GAL_Handler handler= new GAL_ClassicHandler(config,20,50,1);
        handler.runGAL();

        GAL_Chromosome c= handler.getBestFromAll();
        double tw=0;
        for(int i=0;i<10;i++)
            if((byte)c.getGene(i).getTrait() == (byte) 1)
                tw+= w[i];

        System.out.println("Number of Generations: "+
            handler.getLastGenerationNumber());

        System.out.println("Best from all generations:\n"+c);
        System.out.println("Valor: "+c.getFitness());
        System.out.println("Peso: "+tw);

        System.out.println("\nAverage fitness from "+
            "all generations:\n"+
            handler.getAverageFitnessFromAll());
    }
}

```

Figura 2.13. Clase Principal del segundo caso de prueba

Se realizaron pruebas con diferentes configuraciones como se muestra en la Tabla 2.2, obteniendo los siguientes resultados.

Manejador	Selección	Operadores Genéticos	Mejor Cromosoma	Número de Generaciones	Valor Total	Peso Total
Clásico	Ruleta	C.Clasico(0.7), Mutacion(0.0175)	1;1;1;1;1;1;1;0;0;1;	15	289.0	250
Clásico	Ruleta	C.MultiPuntos(0.7,2), Mutacion(0.0175)	1;1;1;1;1;1;1;0;0;1;	12	289.0	250
Clásico	Torneo(2)	C.Clasico(0.7), Mutacion(0.0175)	1;1;1;1;1;1;1;0;0;1;	9	289.0	250
Clásico	Torneo(2)	C.Uniforme(0.7,0.5), Mutacion(0.0175)	1;1;1;1;1;1;1;0;0;1;	5	289.0	250
Clásico	Ranking Clásico(2)	C.Segmentos(0.7,0.5), Mutacion(0.0175)	1;1;1;1;1;1;1;0;0;1;	8	289.0	250

Tabla 2.2. Resultados del segundo caso de prueba

Se observa que todas las pruebas lograron llegar al óptimo antes de alcanzar el número máximo de generaciones.

Caso de Prueba 3

El problema del vendedor viajero, conocido comunmente como TSP (del inglés *Travel Salesman Problem*), consiste en encontrar una ruta que comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades y minimice la distancia recorrida por el viajero.

Dado el grafo representado en la matriz que se observa en la Figura 2.14, donde la posición $[i][j]$ representa la distancia que se debe recorrer desde la ciudad i hasta la j . Se desea conseguir el camino más corto que comience y termine en la ciudad A, pasando por todas las ciudades una sola vez.

	A	B	C	D	E	F	G
A	0	500	200	185	205	104	232
B	500	0	305	360	340	225	154
C	200	305	0	320	165	300	250
D	185	360	320	0	302	205	213
E	205	340	165	302	0	100	198
F	104	225	300	205	100	0	345
G	232	154	250	213	198	345	0

Figura 2.14. Grafo de TSP

Una forma sencilla de representar los posibles camino es mediante permutaciones, donde cada posición representa la i -ésima ciudad que se visita. JGAP permite trabajar con permutaciones de enteros, por lo que se modifican los nombres de las ciudades a 0, 1, 2, 3, 4, 5 y

6 respectivamente.

Número de Generaciones= 20
Tam. Población= 20

El código que se muestra en la Figura 2.15 corresponde a la clase principal del problema

```
class Caso3{

    public static void main(String[] args) throws Exception{
        //costos[i][j] representa el costo de ir de la ciudad i a la j
        int[][] costos=
            {{0,500,200,185,205},
            {500,0,305,360,340},
            {200,305,0,320,165},
            {185,360,320,0,302},
            {205,340,165,302,0}};

        //Creo los genes
        GAL_GeneConfig[] geneConfig= new GAL_GeneConfig[4];
        for(int i=0;i<4;i++)
            geneConfig[i]= new GAL_IntegerGeneConfig(1,5);

        //Creo la configuracion
        GAL_Configuration config= new GAL_Configuration(
            new GAL_ChromosomeConfig(geneConfig),
            new GAL_PermutationsInitializer(),
            new miFuncionAptitud(costos),
            new GAL_TournamentSelector(2),
            new GAL_GeneticOperator[] {
                new GAL_OrderedCrossover(0.7),
                new GAL_SwapMutation(0.1)
            }
        );

        //Creo el handler
        GAL_Handler handler= new GAL_ClassicHandler(config,100,50,1);
        handler.runGAL();

        //Imprimo los resultados
        System.out.println("Best from all generations:\n0;" +
            handler.getBestFromAll() + "0;");
        System.out.println("Valor: " +
            (10000 - handler.getBestFromAll().getFitness()));
        System.out.println("\nAverage fitness from all generations:\n" +
            (10000-handler.getAverageFitnessFromAll()));
    }
}
```

Figura 2.15. Clase principal del tercer caso de prueba

Y en la Figura 2.16 se observa la Función de Aptitud que fue diseñada para este problema.

```

public class miFuncionAptitud extends GAL_FitnessFunction{

    int[][] costos;

    public miFuncionAptitud(int[][] costos){
        this.costos= costos;
    }

    public double computeFitness(GAL_Chromosome chrom){
        double fitness= costos[0][ (int)chromosome.getTrait(0) ];
        for(int i=1;i<chromosome.size();i++)
            fitness+=
                costos[ (int)chrom.getTrait(i-1) ][ (int)chrom.getTrait(i) ];
        fitness+= costos[ (int)chromosome.getTrait(chromosome.size()-1) ][0];
        return 10000 - fitness; //La suma de todas las aristas < 10000
    }
}

```

Figura 2.16. Función de Aptitud del tercer caso de prueba

Como se observa en la Tabla 2.3, se realizaron pruebas con diferentes configuraciones y se obtuvieron los siguientes resultados.

Manejador	Selección	Operadores Genéticos	Recorrido	Costo Total
Clásico	Torneo(2)	C.Ordenado(0.7), Intercambio(0.1)	0;3;6;1;5;4;2;0	1242.0
Clásico	Torneo(2)	C.Ordenado(0.7), Barajeo(0.1)	0;3;6;1;2;4;5;0	1226.0
Modificado(10)	Torneo(2)	C.Ordenado(0.7), Brajeo(0.3)	0;5;4;2;1;6;3;0	1226.0
Modificado(10)	Ruleta	C.Ordenado(0.7), Brajeo(0.3)	0;3;6;1;5;4;2;0	1242.0
Modificado(10)	Ranking No Lineal(0.5)	C.Ordenado(0.7), Brajeo(0.3)	0;5;4;2;1;6;3;0	1226.0

Tabla 2.3. Resultados del tercer caso de pruebas

Las pruebas indican que el mejor recorrido que pudo encontrar el AG fue 0;5;4;2;1;6;3;0, cuyo costo es de 1226.

2.3.3. Interfaz

Antes de implementar la interfaz, se definieron los casos de uso de la aplicación. En la Figura 2.17 se aprecian los casos de uso principales.

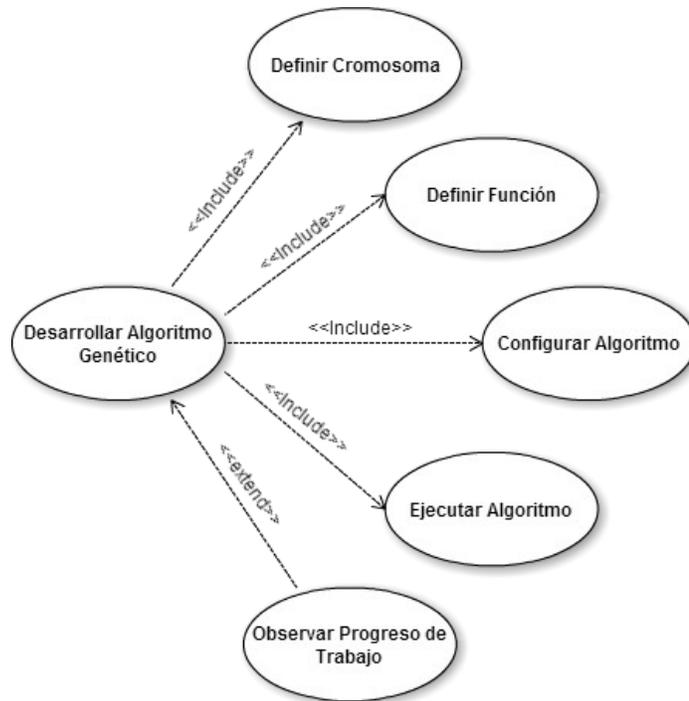


Figura 2.17. Casos de uso principales para el desarrollo de un AG

En las Figuras 2.18, 2.19, 2.20, 2.21 y 2.22 se observan los casos de uso específicos relacionados a los casos de uso de segundo nivel.

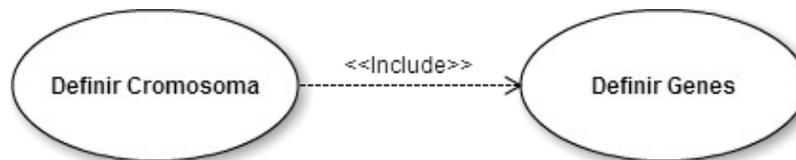


Figura 2.18. Casos de uso para Definir Cromosoma

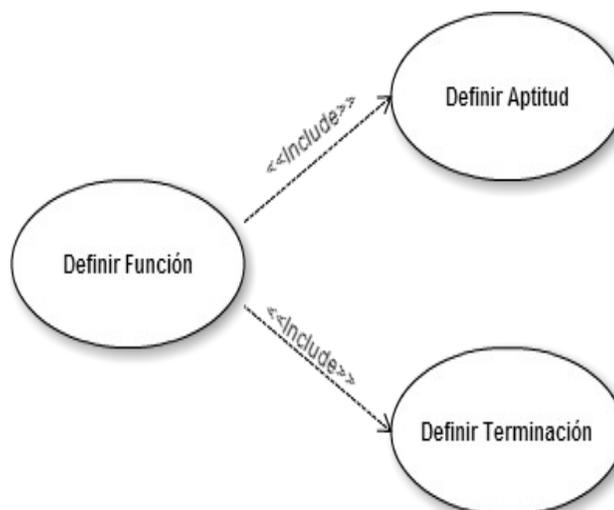


Figura 2.19. Casos de uso para Definir Función

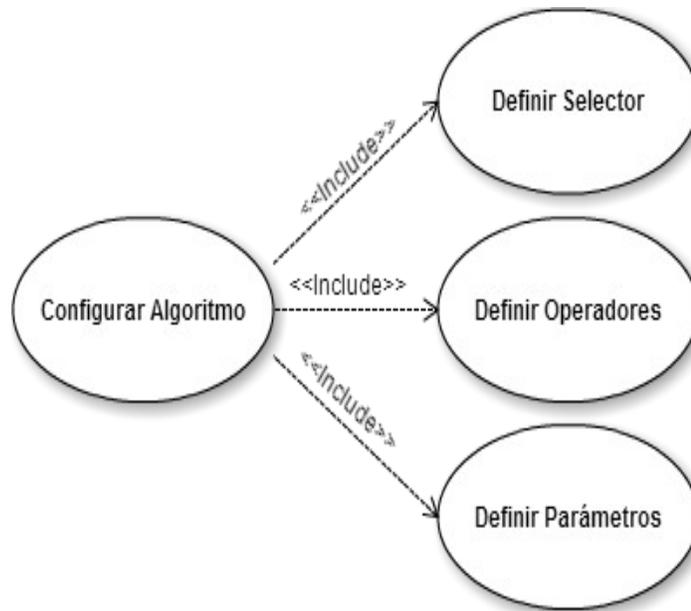


Figura 2.20. Casos de uso para Configurar Algoritmo

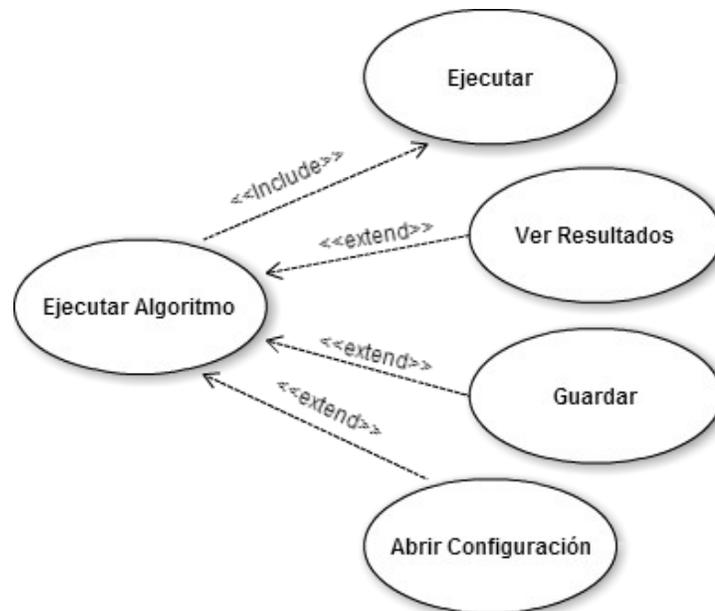


Figura 2.21. Casos de uso para Ejecutar Algoritmo

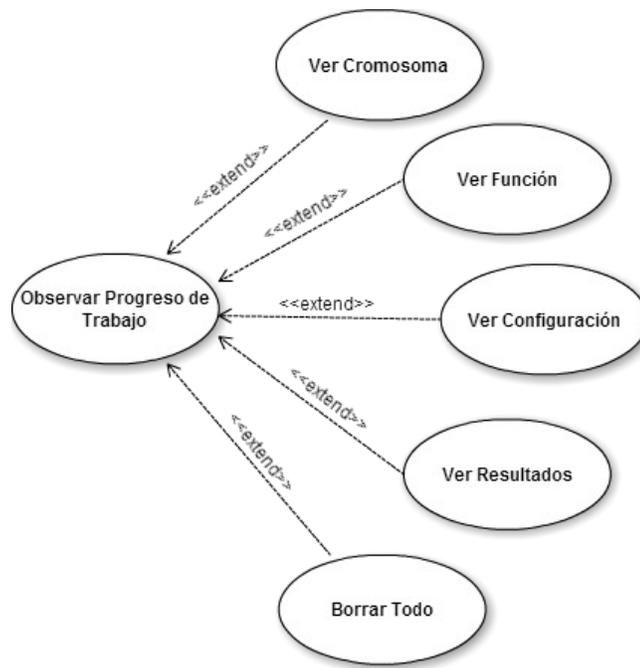


Figura 2.22. Casos de uso para Observar Progreso de Trabajo

2.3.3.1. Diseño y funcionamiento de la interfaz

Se utilizó como base la interfaz desarrollada previamente por el licenciado Gustavo Torres como Trabajo Especial de Grado [18]. A esta interfaz se le realizaron pruebas de usabilidad (evaluaciones heurísticas y pruebas de aceptación del usuario), por lo que no es necesario repetirlas sobre la nueva interfaz. La página de inicio de la interfaz utilizada como base se muestra en la Figura 2.23, mientras que la asociada a la interfaz diseñada para esta aplicación se muestra en la Figura 2.24.

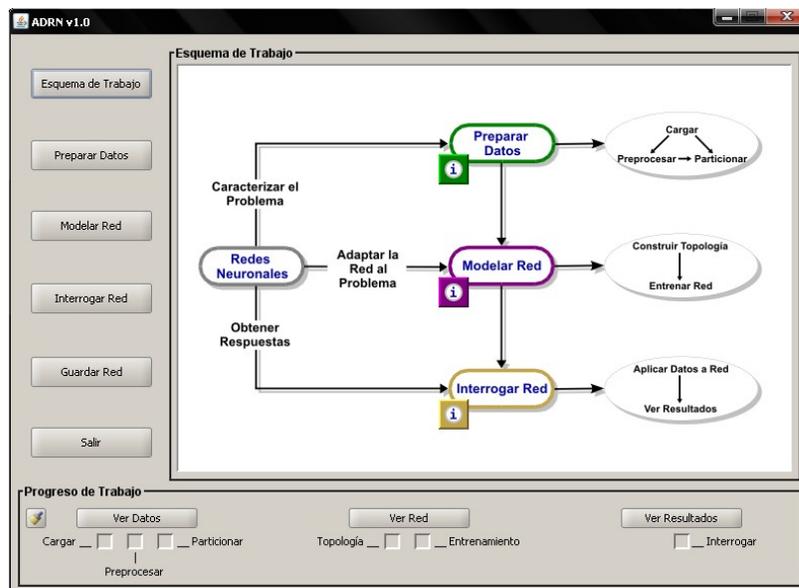


Figura 2.23. Interfaz Diseñada por Gustavo Torres

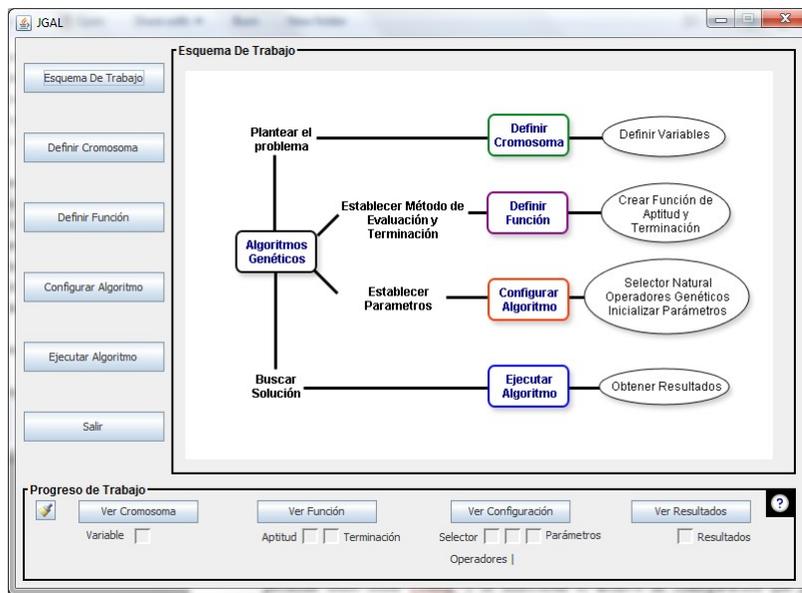


Figura 2.24. Interfaz Diseñada para aplicación de AG

Como se observa en la Figura 2.24, los casos de uso principales que se mostraron en la Figura 2.17 se ven reflejados en los botones laterales de la interfaz, el progreso de trabajo y el mapa cognitivo que se aprecia en el centro de ésta.

En esta sección se explican las modificaciones realizadas sobre los lineamientos descritos en [18], además del funcionamiento general de la interfaz.

a) Interfaz General

Como se observa en la Figura 2.24, la interfaz se subdivide en tres áreas: área de botones, área de trabajo y progreso de trabajo.

El área de botones consta de seis botones, donde cuatro de ellos corresponden a los casos de uso principales que se muestran en la Figura 2.17. Al pisar sobre cualquiera de estos botones (exceptuando Salir), se cambiará el área de trabajo en base al botón que fue presionado.

El área de trabajo consta de un recuadro grande mostrando el mapa cognitivo relacionado al trabajo activo en el momento, acompañado de botones utilizados para propósitos específicos al problema, los cuales corresponden a los casos de uso específicos mostrados en las Figuras 2.18, 2.19, 2.20 y 2.21 respectivamente. El área de trabajo incluye un botón en la esquina superior derecha, que al presionarlo abre el archivo de ayuda de la aplicación.

El Progreso de Trabajo consta de múltiples casillas que permiten observar el progreso a través del desarrollo de un AG y corresponde al caso de uso Observar Progreso de Trabajo que se muestra en la Figura 2.17. Será descrito en detalle en el apartado f.

b) Definir Cromosoma

Al presionar sobre el botón Definir Cromosomas, el área de trabajo se debe observar como

en la Figura 2.25.

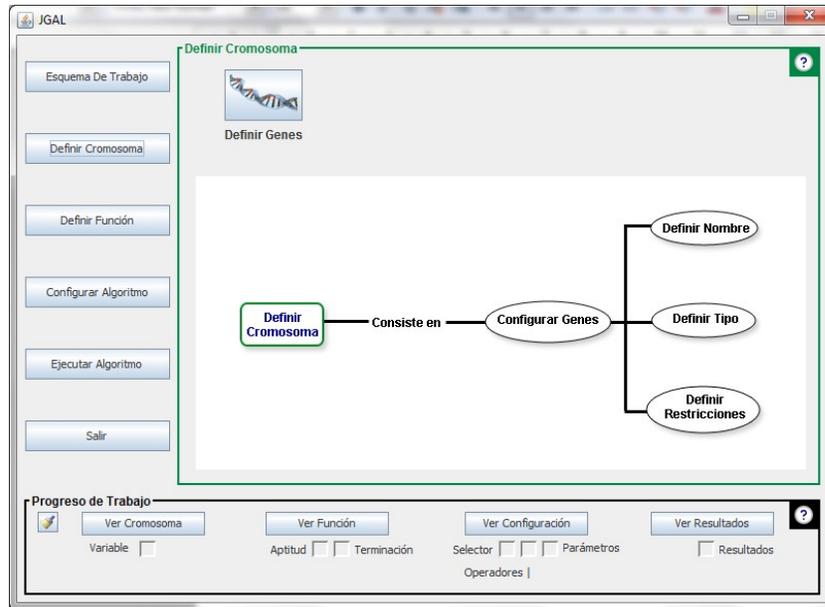


Figura 2.25. Interfaz para Definir Cromosoma

Es representado por el color verde (0,128,64) y permite definir la configuración que será utilizada para el cromosoma y específicamente los genes.

Al presionar sobre el botón Definir Genes, se abrirá una ventana como la que se muestra en la Figura 2.26.

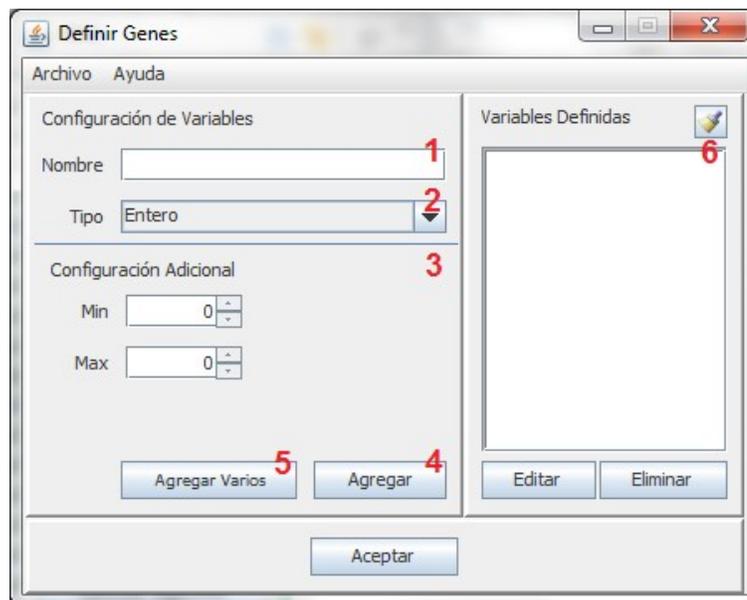


Figura 2.26. Interfaz para Definir Genes

Cada número implica un paso en la definición de los genes:

1. Definir Nombre del Gen
2. Definir Tipo del Gen
3. Configuración Adicional
4. Agregar Gen
5. Variables Definidas

Si se necesita crear múltiples genes con la misma configuración, se sugiere utilizar la opción Agregar Varios.

c) Definir Función

Definir Función es representado por el color morado (128,0,128) y se muestra en el área de trabajo al presionar sobre el botón Definir Función. El área de trabajo se observa como se aprecia en la Figura 2.27.

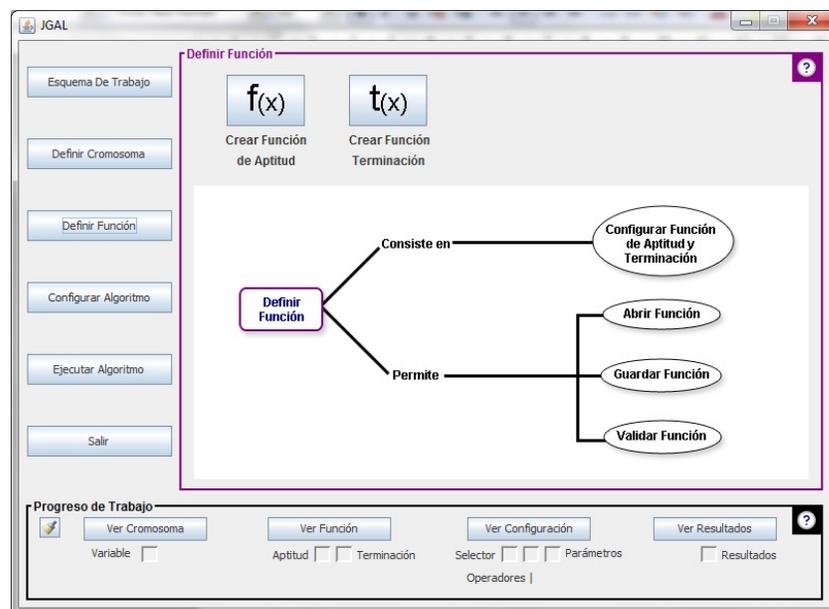


Figura 2.27. Interfaz para Definir Función

Permite crear las funciones de aptitud y de terminación. Aunque la función de aptitud es obligatoria, la de terminación no es necesaria para el desarrollo del AG.

Para definir la función de Aptitud se debe presionar sobre el botón Crear Función de Aptitud, abriéndose una ventana como la que se observa en la Figura 2.28.

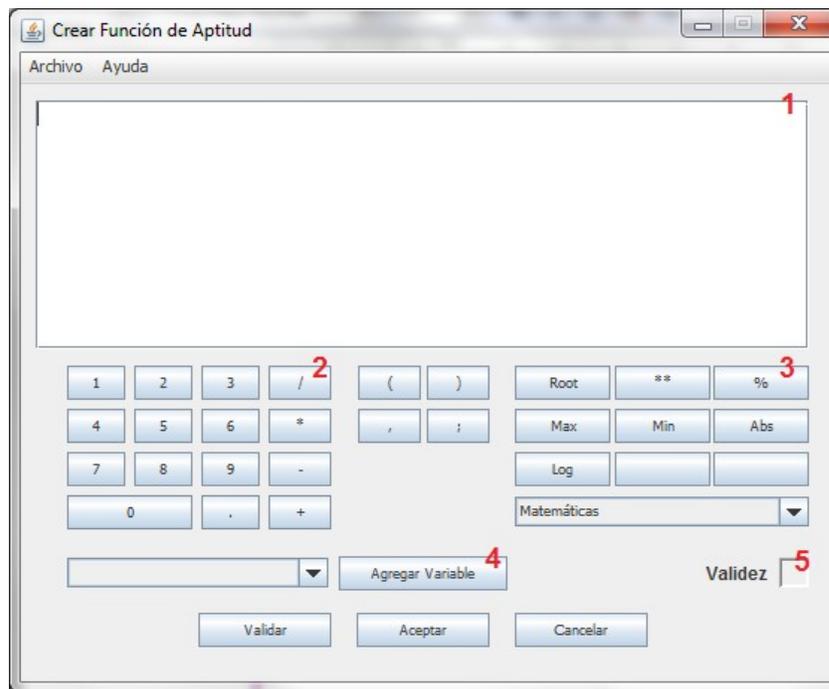


Figura 2.28 Interfaz para Crear Función de Aptitud

Cada número es definido como:

1. Espacio de Código
2. Calculadora
3. Botones Complejos
4. Agregar Variable
5. Validar

La función de aptitud no se puede definir hasta que el cromosoma sea definido y si se realizan cambios sobre el cromosoma, se debe volver a validar.

Para definir la función de Terminación se debe presionar sobre el botón Crear Función Terminación, abriéndose una ventana similar a la de Crear Función de Aptitud, como se muestra en la Figura 2.29.

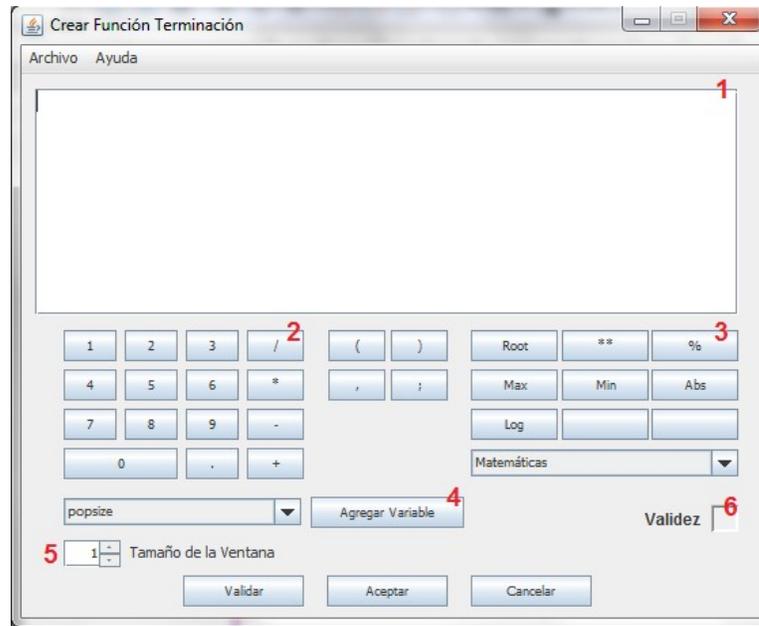


Figura 2.29. Interfaz para Crear Función de Terminación

Cada número representa:

1. Espacio de Código
2. Calculadora
3. Botones Complejos
4. Agregar Variable
5. Tamaño de la Ventana
6. Validar

Para ambas funciones fue necesario crear un Lenguaje de Programación parecido a Pascal al cual se le denominó *GAL_Parser*. Para esto, se utilizó una librería Open Source llamada Jparsec³, la cual permite desarrollar analizadores sintácticos de tipo combinatorio de manera rápida y sencilla en el lenguaje de programación Java.

A continuación se puede chequear la sintaxis general de *GAL_Parser*:

```

<Function> ::=
  ( <While Statement>;
  | <If Statement>;
  | <Assign Statement>;
  | <Logical Statement>;
  | <Arithmetic Statement>;
  | <Identifier>; )+

<While Statement> ::=
  While <Logical Statement> Do <Function> End

<If Statement> ::=
  If <Logical Statement> Then <Function> (Else <Function>)? End

```

³ Jparsec: <http://jparsec.codehaus.org/>

```

<Assign Statement> ::=
  <Identifier> "!="
    ( <Logical Statement>
      | <Arithmetic Statement>
      | <Identifier>
      | <String>
      | <Char>)
  | <Factory Statement>

<Factory Statement> ::=
  Factory"("<Identifier>,"{"
    { <Logical Statement>
      | <Arithmetic Statement>
      | <Identifier>
      | <String>
      | <Char>};}"
  ")"

<Logical Statement> ::=
  <Logical Object> ("&&" | "||" | "^") <Logical Object>)?

<Logical Object> ::=
  (Not)? (True | False | <Comparison Statement> | <Identifier> |
  <Logical Statement>)

<Comparison Statement> ::=
  <Comparable Object> ("<" | ">" | "=" | "<=" | ">=" | "<>")
  <Comparable Object>)?

<Comparable Object> ::=
  <String> | <Char> | <Arithmetic Statement> | <Identifier>

<Arithmetic Statement> ::=
  <Simple Arithmetic> | <Complex Arithmetic>

<Simple Arithmetic> ::=
  <Arithmetic Object> ((+ | - | * | / | % | **) <Arithmetic Object>)?

<Arithmetic Object> ::=
  (-)? (<Number> | <Identifier> | <Arithmetic Statement>)

<Number> ::= (0..9)+ | PI | E

<Complex Arithmetic> ::=
  <Operator>("("<Arithmetic Object> (<Arithmetic Object>*)"")?

<Operator> ::=
  Abs | Max | Min | Root | Log | Sin | Cos | Tan | Asin | Acos | Atan |
  Hypot | toRadians | toDegrees | Round | Ceil | Floor | RandI | RandD
  | RandB

<Identifier> ::=
  $(<Name> | "{"<Arithmetic Statement>"}")

```

```

<Name> ::=
    (A..Z|a..z|_) <Chars>*

<String> ::=
    "(<Chars> | <WhiteSpace>)*"

<Chars> ::=
    (A..Z|a..z|_|0..9)

<Char> ::=
    '(<Chars> | <WhiteSpace>)' | RandC"("<Chars>,<Chars>")"

```

GAL_Parser permite utilizar ciclos, condicionales, asignaciones, operaciones lógicas, operaciones aritméticas y uso de identificadores (variables y constantes). Este lenguaje se caracteriza en que todas las instrucciones retornan un valor, de manera que la última instrucción que se ejecuta representa el retorno de la función. En el caso de la función de aptitud el código debe retornar un double, mientras que para la función de terminación debe retornar un boolean.

Se decidió que Jparsec arrojara un árbol de sintaxis abstracta en lugar del retorno. Esto debido a que ambas funciones a ser implementadas con *GAL_Parser* se ejecutan con alta frecuencia al momento de ejecutar el AG e interpretar el código múltiples veces es costoso en tiempo. Sin embargo, interpretar un árbol es más rápido, de manera que se crearon estructuras especiales que pudieran ser interpretadas rápidamente al momento de la ejecución.

Por ejemplo, para un código en *GAL_Parser*, como el mostrado en la Figura 2.30.

```

$i:=0;
While $i<10 Do
    $i:=$i+1;
End;

```

Figura 2.30. Código escrito en GAL_Parser

Se forma un árbol de sintaxis abstracto, como el que se aprecia en la Figura 2.31.

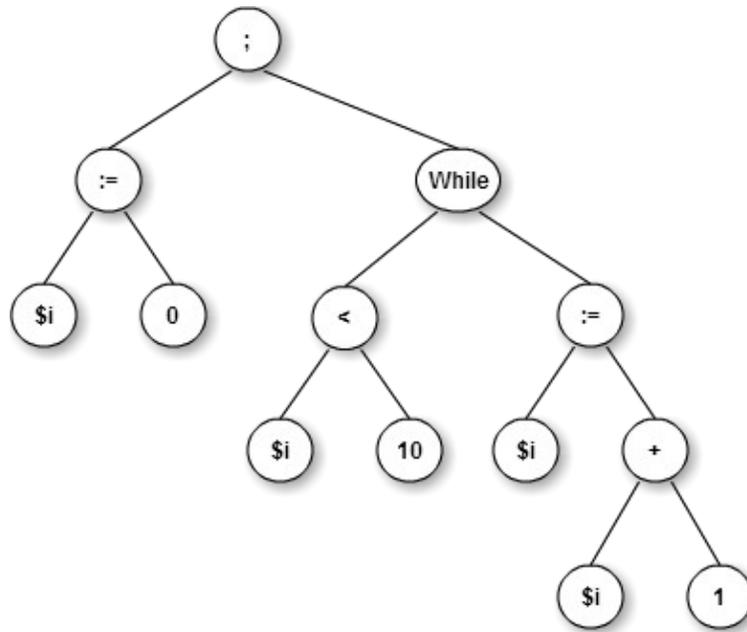


Figura 2.31. Árbol de un código en GAL_Parser

d) Configurar Algoritmo

Configurar Algoritmo se muestra en el área de trabajo como se aprecia en la Figura 2.32 y es representado por el color naranja (255,50,0). Permite definir el selector natural, los operadores genéticos y otros parámetros.

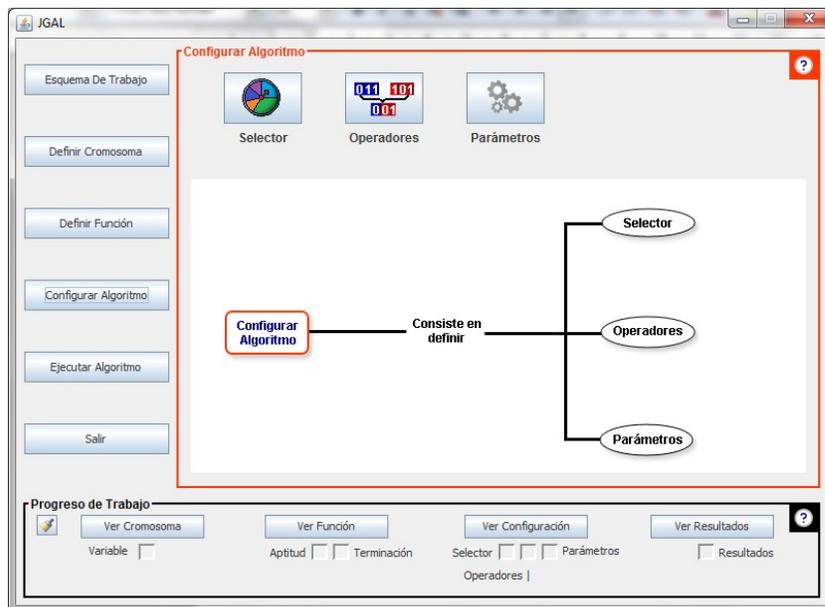


Figura 2.32. Interfaz para Configurar Algoritmo

Al presionar sobre el botón Selector, se abrirá una ventana como la que se observa en la Figura 2.33.

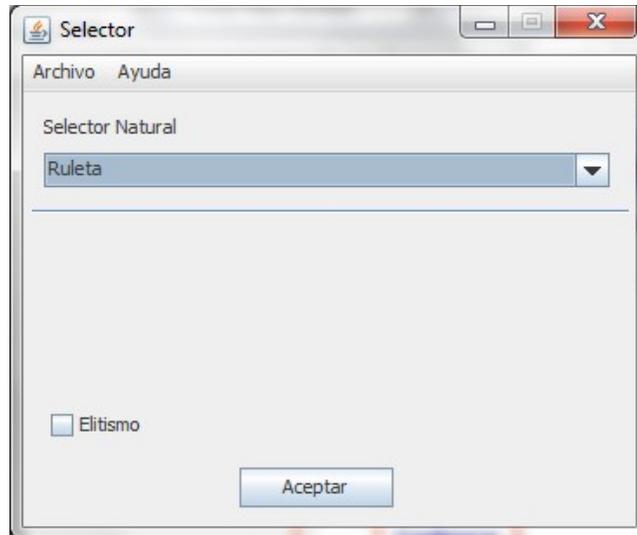


Figura 2.33. Interfaz para el Selector

En esta pantalla se podrá elegir el selector natural a utilizar y de manera opcional agregar elitismo.

Al presionar sobre el botón Operadores, se abrirá una ventana similar a la de Definir Genes, como se aprecia en la Figura 2.34.

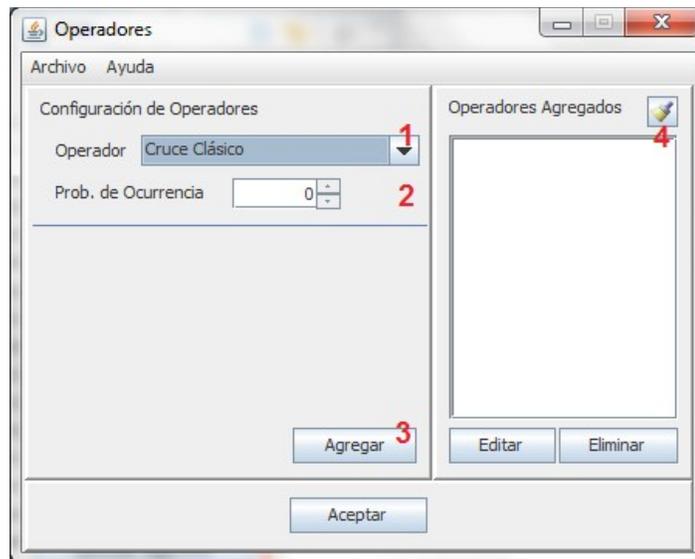


Figura 2.34. Interfaz para los Operadores

Cada número implica un paso en la definición de los operadores:

1. Definir Tipo de Operador
2. Definir Probabilidad de Ocurrencia
3. Agregar Operador
4. Operadores Agregados

Finalmente, al presionar sobre el botón Parámetros, se abrirá una ventana como la que se muestra en la Figura 2.35, donde se asignan los últimos parámetros como el tipo de manejador a utilizar, el tamaño de la población y el número máximo de generaciones..



Figura 2.35. Interfaz para los Parámetros

e) Ejecutar Algoritmo

Ejecutar Algoritmo se representa con el color azul (0,0,256) y como su nombre lo indica permite ejecutar el AG una vez se haya realizado todas las configuraciones. Además permite Guardar y Abrir configuraciones completas.

Para acceder a éste basta con presionar sobre Ejecutar Algoritmo, en cuyo caso el área de trabajo se verá como se refleja en la Figura 2.36.

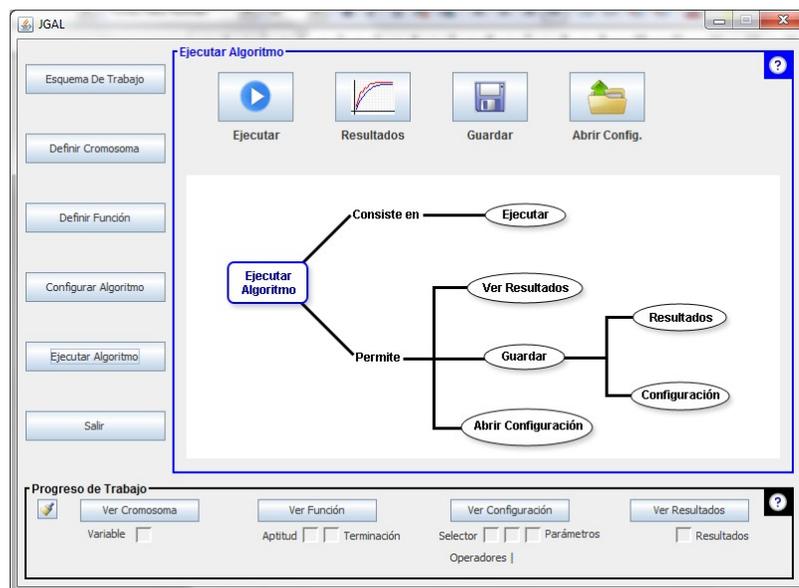


Figura 2.36. Interfaz para Ejecutar Algoritmo

Al presionar sobre el botón ejecutar, si no falta nada para poder realizar una ejecución exitosa, se mostrará una ventana de opción donde se pregunta que tipo de optimización se desea

realizar (Maximización o Minimización). Luego de elegir una opción, el AG se ejecutará y si no ocurre ningún error de ejecución, entonces se abrirá una ventana como la que se muestra en la Figura 2.37. Esta ventana puede ser accedida luego a través del botón Resultados o el botón Ver Resultados en el Progreso de Trabajo.

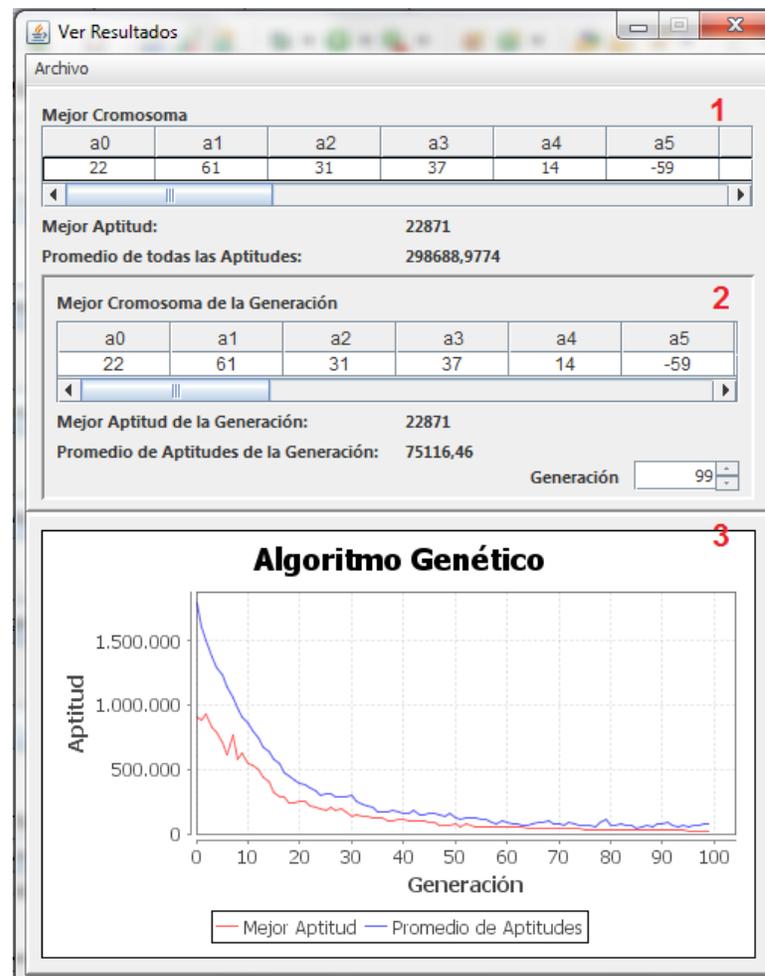


Figura 2.37. Interfaz para Resultados

Cada número representa el área para:

1. Mejor Cromosoma
2. Mejor Cromosoma de una generación
3. Gráfica de Aptitud/Generación

Para la gráfica se utilizó una librería Open Source de Java llamada jfreechart⁴, la cual permite crear gráficas con facilidad en el lenguaje de programación Java.

f) Progreso de Trabajo

El progreso de trabajo, como se mencionó anteriormente, permite visualizar el progreso al

⁴ jfreechart: <http://www.jfree.org/index.html>

momento de desarrollar un AG. Al iniciar la aplicación el progreso debería estar vacío como se muestra en la Figura 2.38.



Figura 2.38. Interfaz para Progreso de Trabajo 1

Pero si por ejemplo se define el cromosoma de manera exitosa, entonces cambiará de color la casilla relacionada a dicho progreso, como se observa en la Figura 2.39.



Figura 2.39. Interfaz para Progreso de Trabajo 2

El color de la casilla será el mismo que el color que representa dicho caso de uso. Es decir, como Definir Cromosoma es representado por el color verde, entonces la casilla para Variable se coloreará con ese mismo color.

En la Figura 2.40 se observa como se vería el progreso de trabajo hasta justo antes de ejecutar el AG, recordando que la Terminación no es necesario que se encuentre activada.



Figura 2.40. Interfaz para Progreso de Trabajo 3

El progreso de trabajo permite visualizar los datos del AG que se está desarrollando, a través de los botones que éste presenta. Al presionar sobre Ver Cromosoma se muestra una pantalla como la que se visualiza en la Figuras 2.41.

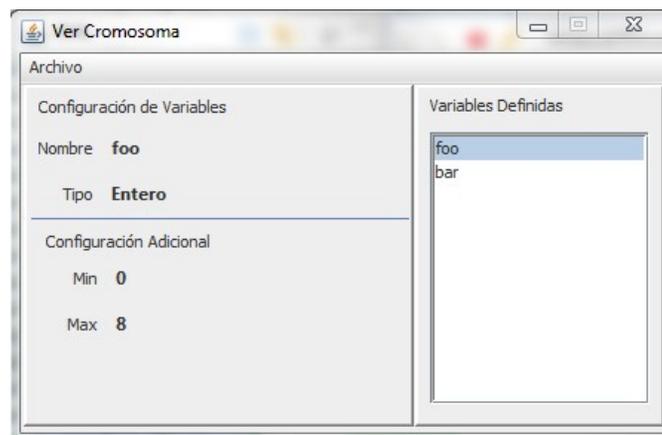


Figura 2.41. Interfaz para Ver Cromosoma

Al presionar sobre el botón Ver Función, se mostrará una pantalla como la que se visualiza en la Figura 2.42.

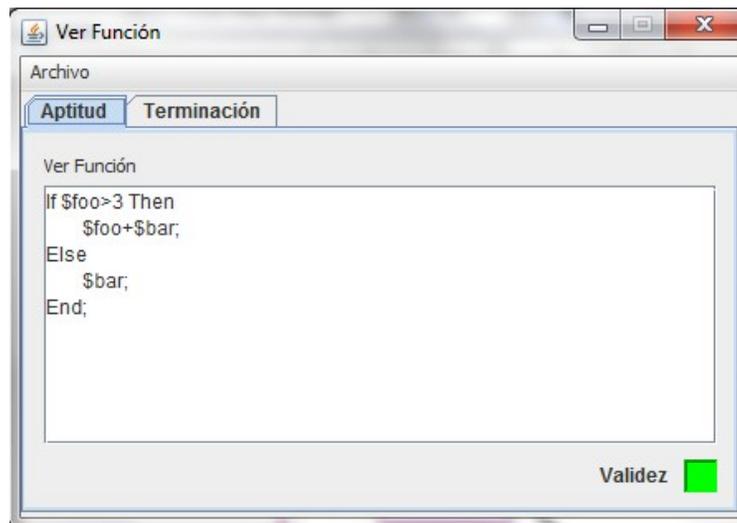


Figura 2.42. Interfaz para Ver Función

Y al presionar sobre el botón Ver Configuración se muestra una ventana como la que se aprecia en la Figura 2.43.

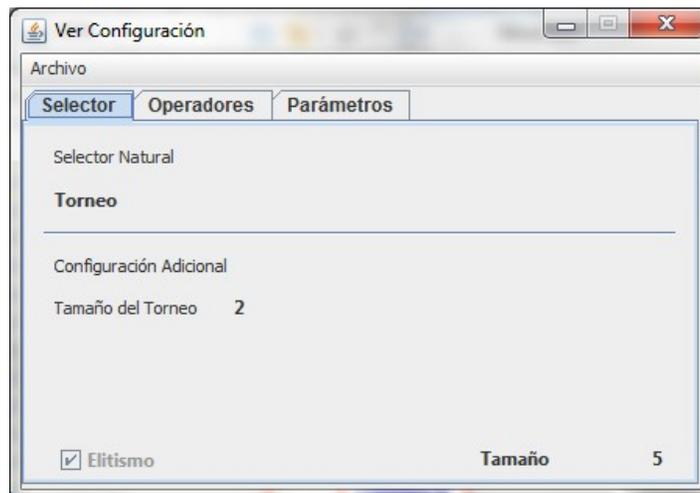


Figura 2.43. Interfaz para Ver Configuración

Finalmente, el progreso de trabajo incluye un pequeño botón con el símbolo de una escoba, con el cual se puede borrar todo el progreso realizado hasta el momento en que fue presionado.

g) Sistema de Ayuda de la Aplicación

La aplicación consta con un sistema de ayuda el cual fue desarrollado usando JavaHelp⁵, una librería OpenSource para el diseño de ayudas en Java.

⁵ JavaHelp: <http://javahelp.java.net/>

La ayuda puede ser accedida de múltiples maneras:

- Presionando el signo de interrogación en el área de trabajo.
- Presionado el signo de interrogación en el progreso de trabajo.
- En cualquier subventana de configuración del programa en el menú Ayuda/Abrir Ayuda
- En la función de aptitud y terminación en el menú Ayuda/Sintaxis

Donde cada una abre la ayuda en un tema relacionado a la ubicación donde fue abierta, como se observa en la Figura 2.44.

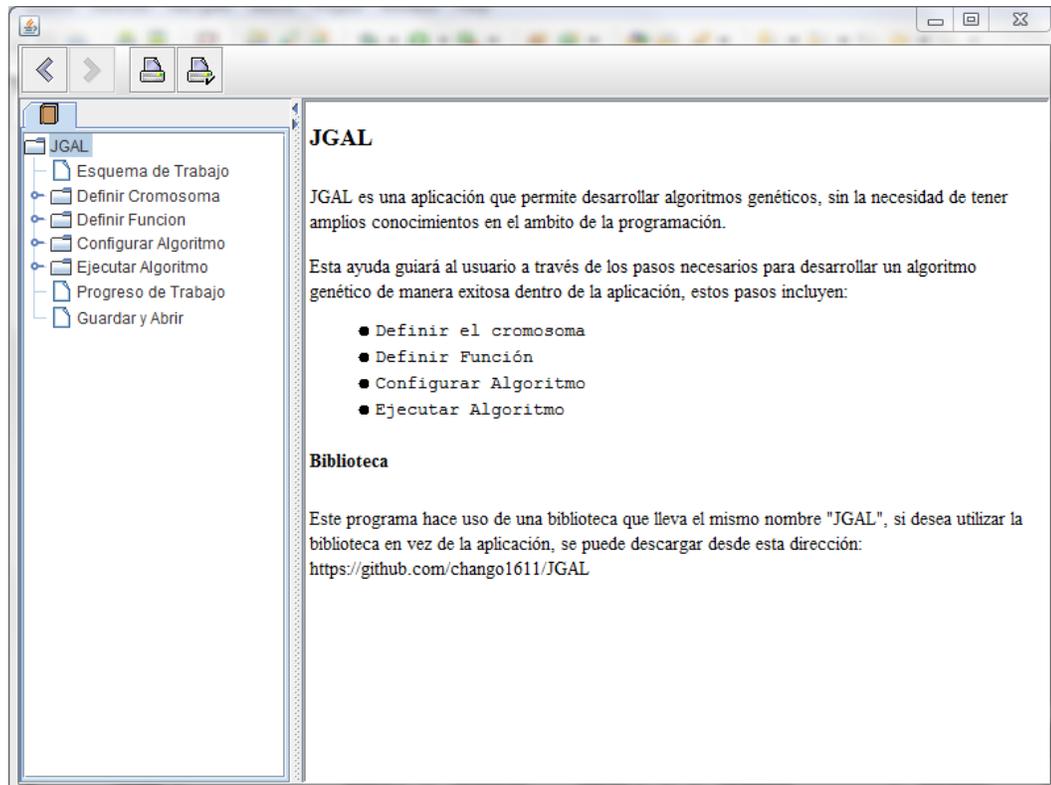


Figura 2.44. Interfaz para el Sistema de Ayuda

h) Guardar y Abrir

La aplicación permite Guardar y Abrir partes de la configuración del AG en cualquier subventana de configuración, mediante las opciones del menú Archivo/Guardar y Archivo/Abrir.

Los archivos que se crean con esta aplicación utilizan la sintaxis que se muestra a continuación:

```
<Archivo> ::=  
  <Archivo_Gen>  
  | <Archivo_Aptitud>  
  | <Archivo_Terminación>  
  | <Archivo_Selector>  
  | <Archivo_Operadores>
```

```

| <Archivo_Parámetros>
| <Archivo_Completo>

<Archivo_Gen> ::=
    <Nro_Genes> {<Genes>}

<Genes> ::=
    <Tipo> <Nombre> <Restricciones>

<Archivo_Aptitud> ::=
    <Nro_Líneas> {<Lineas>}

<Archivo_Terminación> ::=
    <Nro_Líneas> {<Lineas>} <Tamaño_Ventana>
<Archivo_Selector> ::=
    <Tipo> <Configuración_Selector>

<Archivo_Operadores> ::=
    <Nro_Operadores> {<Operadores>}

<Operadores> ::=
    <Tipo> <Configuración_Operador>

<Archivo_Parámetros> ::=
    <Tipo> <Tamaño_Población> <Máx_Generación> <Parámetro_Modificado>

<Archivo_Completo> ::=
    <Archivo_Gen> <Archivo_Aptitud> <Archivo_Terminación>
<Archivo_Selector> <Archivo_Operadores> <Archivo_Parámetros>

```

Donde el tipo es un número. Por ejemplo, para los genes: 0 es entero, 1 es double, 2 es binario, 3 es carácter y 4 es nominal.

i) Idioma

La aplicación permite su visualización en múltiples idiomas (por defecto Ingles y Español), con la posibilidad de expandir a nuevos idiomas. Para incluir un nuevo idioma se debe agregar una línea en el archivo language.ini que siga el formato:

```
Idioma id help_id image_id
```

Por ejemplo, si se está agregando Italiano la línea podría quedar como la siguiente:

```
Italiano it help_es image_es
```

Una vez hecho esto, se debe crear un archivo *id.language*, para el ejemplo anterior sería *it.language*, el cual debe seguir el formato de los otros archivos language (*es.language* y *en.language* se encuentran por defecto en la aplicación).

Si se desea se puede crear una ayuda para el idioma, para esto se crea una carpeta llamada *help_id* que contenga la ayuda, sin embargo se puede utilizar los que se encuentran ya por defecto *help_es* o *help_en*.

Finalmente, las imágenes no pueden ser cambiadas, pero incluyen la opción de ponerlas en español (*image_es*) o en inglés (*image_en*).

j) Metadata

La biblioteca permite implementar nuevas clases, con la finalidad de poder desarrollar AG con diferentes configuraciones a las que se encuentran por defecto. La herramienta permite agregar dichas clases a través del archivo `metadata.file`.

Las nuevas clases que se vayan a agregar debe estar ya compiladas (`.class`) y deben estar ubicadas en una carpeta que se encuentre dentro del `classpath`. Además, el código de éstas clases deben seguir 2 restricciones para funcionar correctamente:

- Los parámetros del constructor pueden ser: Integer, Short, Byte, Long, Double, Float, Character o String.
- Todos los parámetros del constructor deben tener un método consultor (`get`).

2.3.3.2. Conexión entre la biblioteca y la interfaz

Como se mencionó en el diagrama de clases de la Biblioteca en la Figura 2.2, se creó una clase llamada *GAL_Interface* que se encarga de la comunicación entre la biblioteca y la interfaz. Para esto, se creó una instancia de *GAL_Interface* desde el constructor como un objeto estático para que pueda ser accedido desde cualquier punto de la aplicación.

Se busco utilizar una arquitectura MVC donde la instancia de *GAL_Interface* se encarga del control y modelo de la aplicación, de manera que la interfaz manejara únicamente la vista.

De forma general los métodos en *GAL_Interface* se subdividen en:

- Metodos consultores (*get*) y modificadores (*set*) para la configuración..
- Lectura y escritura de archivos (Utilizados para guardar y abrir).
- Llamador de ejecución.
- Consultores para resultados de la ejecución.

2.3.3.3. Pruebas realizadas sobre la interfaz

Se utilizaron los mismos casos de prueba que para la Biblioteca en la Sección 2.3.2.2, por lo que no se explicaran nuevamente los problemas, sino que se mostrará directamente su resolución.

Caso de Prueba 1

Para este caso de prueba, se mostrará paso a paso su resolución usando la aplicación.

Primero se define el cromosoma en definir genes, como todos comparten una misma

configuración se puede utilizar la opción Agregar Varios. Para esto se crea la configuración del gen como se muestra en la Figura 2.45.

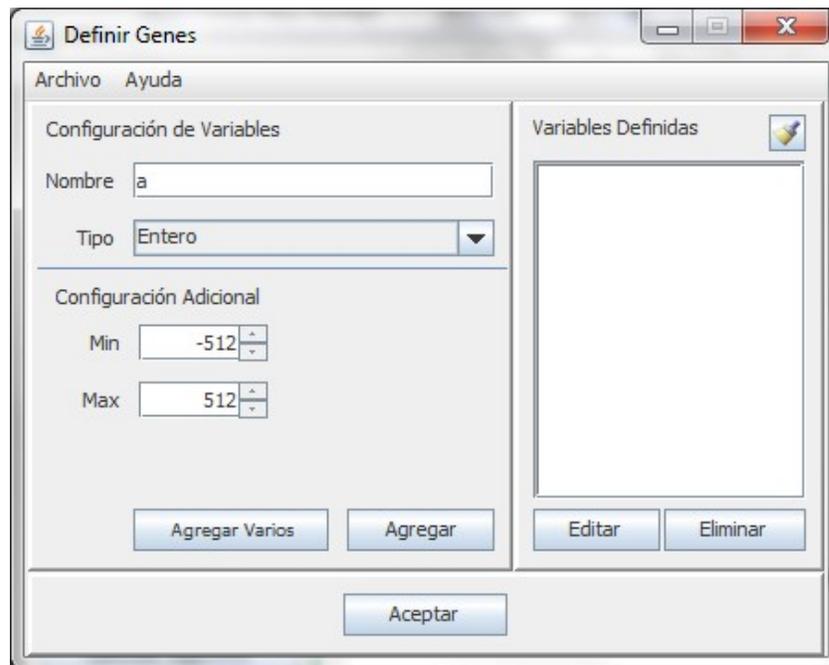


Figura 2.45. Caso de Prueba 1 – Definir Genes 1

Luego se presiona Agregar Varios, abriéndose una ventana emergente donde se pide cuántos genes agregar; como se necesitan 20 genes, entonces se coloca 20. Al pisar aceptar, la ventana debería verse como en la Figura 2.46.

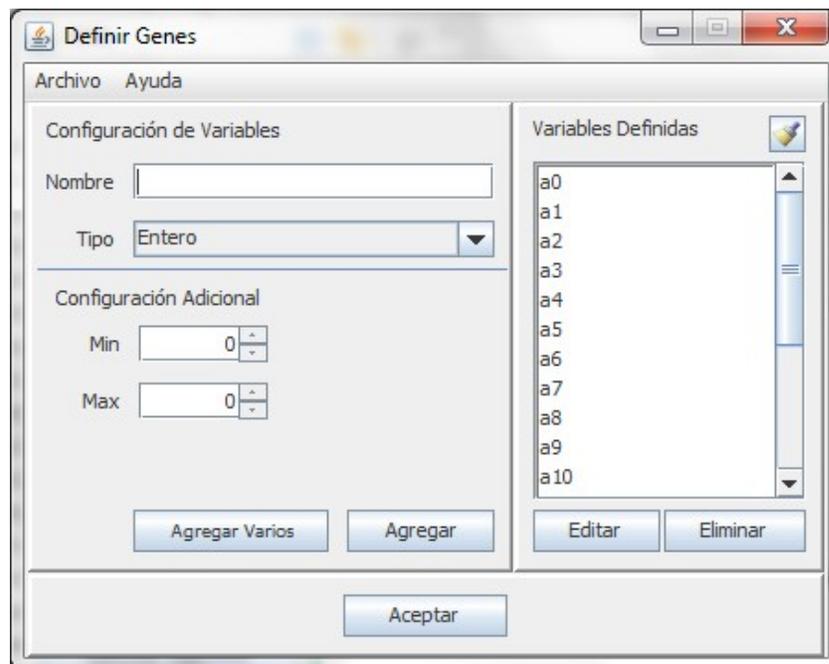


Figura 2.46. Caso de Prueba 1 – Definir Genes 2

Posterior a esto, se cierra la ventana de definir genes y se prosigue a crear la Función de Aptitud, el cual debería observarse parecido al que se muestra en la Figura 2.47.

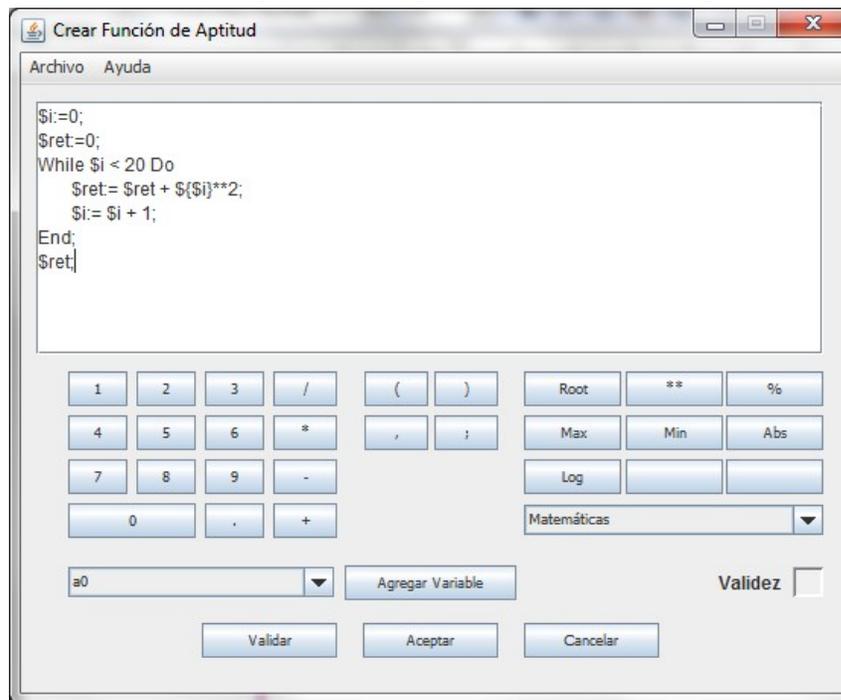


Figura 2.47. Caso de Prueba 1 – Crear Función de Aptitud

Se presiona aceptar y se prosigue a elegir el Selector, ya que para este caso no se planea utilizar condición de terminación. Se decidió utilizar el selector que obtuvo mejores resultados en la prueba con la biblioteca como se observa en la Figura 2.48.

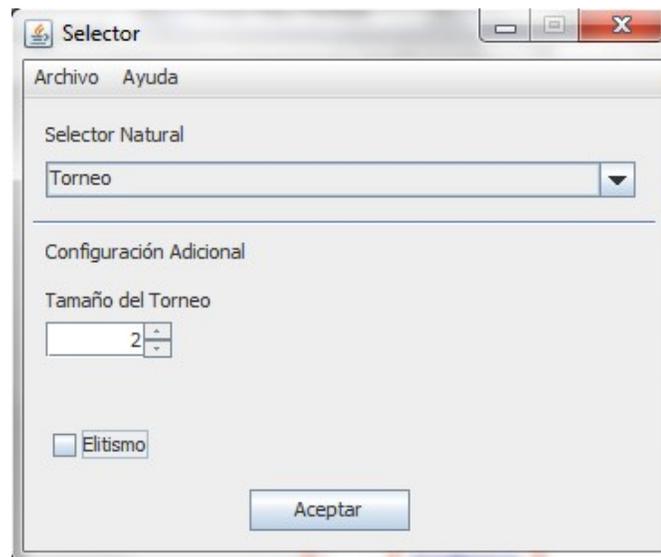


Figura 2.48. Caso de Prueba 1 – Selector

Se presiona aceptar y se prosigue a seleccionar los operadores. Se decidió utilizar los

operadores que obtuvieron mejores resultados en la prueba con la biblioteca como se observa en la Figura 2.49.

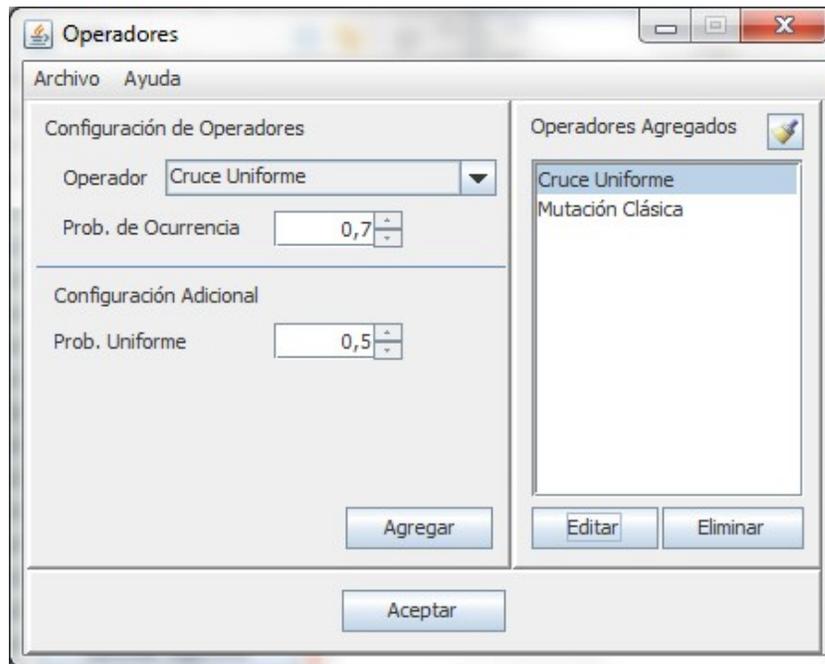


Figura 2.49. Caso de Prueba 1 – Operadores

Se prosigue a colocar los parámetros que faltan como se visualiza en la Figura 2.50.

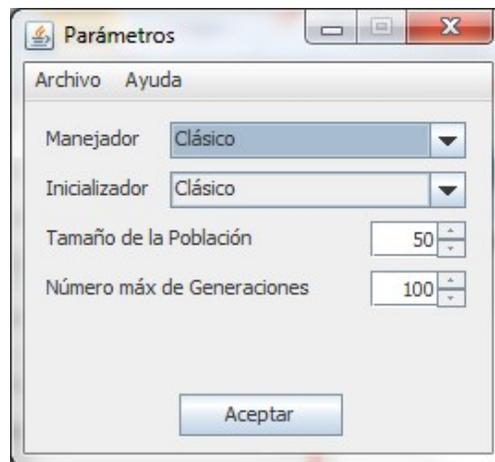


Figura 2.50. Caso de Prueba 1 – Parámetros

Finalmente, se ejecuta el algoritmo seleccionando minimización como tipo de optimización a utilizar. Luego de ejecutarse correctamente el AG, se mostrará una ventana de resultados como la de la Figura 2.51.

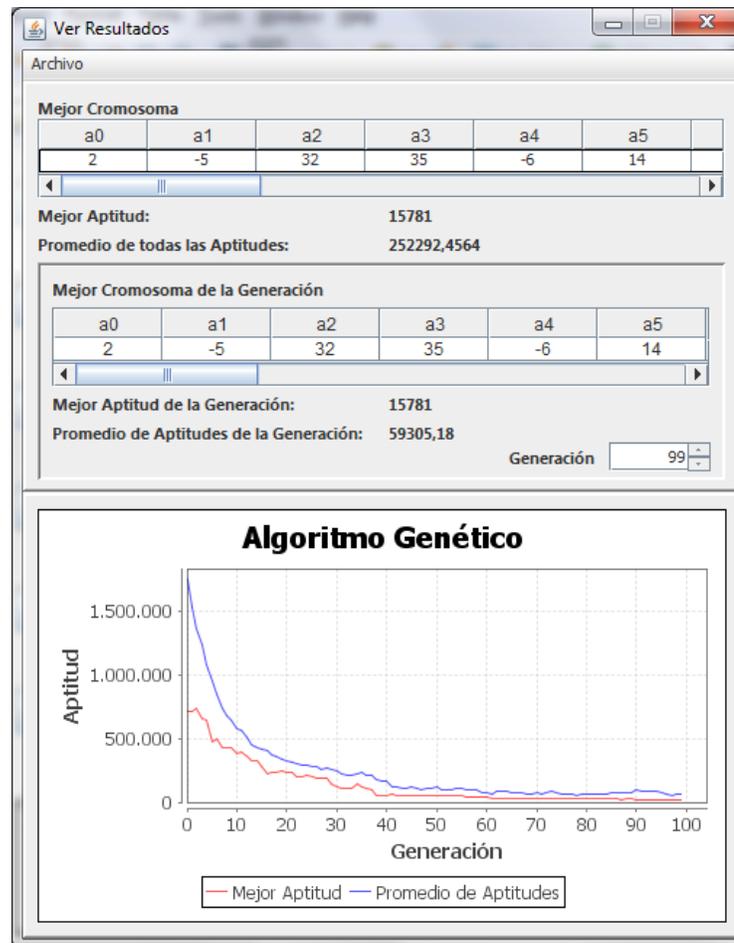


Figura 2.51. Caso de Prueba 1 – Resultados

Se obtuvo como mejor aptitud de la población 15.781, el cual coincide con el mejor cromosoma de la última generación. El promedio de aptitudes de la última generación fue de 59.305, mientras que el promedio total fue de 252.292. Finalmente, se observa en la gráfica que el AG actuó como se esperaba (a mayor generación mejor aptitud).

Caso de Prueba 2

El segundo caso de prueba se manejará a partir de la opción abrir configuración. Para ello se presiona sobre Abrir Config. y se selecciona el archivo de configuración que para este problema debería contener un texto como el que se aprecia en la Figura 2.52.

```

10
2 elem0
2 elem1
2 elem2
2 elem3
2 elem4
2 elem5
2 elem6
2 elem7
2 elem8
2 elem9
16
Array($v,{45;10;71;49;41;26;32;18;23;15;});
Array($w,{12;30;54;23;56;42;23;60;54;10;});
$val:=0; $peso:=0;
$i:=0;
While $i<10 Do
  If ${$i}=1 Then
    $valor:= $valor+${$i+10};
    $peso:= $peso+${$i+20};
  End;
  $i:= $i+1;
End;
If $peso>250 Then
  0;
Else
  $valor;
End;
1
$bestFitness_0>288;
1
0 0 1
2
0 0.700000000000000001
5 0.0175
0 0 50 20 1

```

Figura 2.52. Caso de Prueba 2 - Archivo

El archivo se interpreta de la siguiente manera:

1. Se leen 10 genes, estos genes son de tipo 2 (binario), por lo que solo necesitan el nombre.
2. Se lee la función de aptitud, ésta posee 16 líneas. Se observa que se utilizó la función Array, el cual permite realizar múltiples asignaciones de una manera más rápida y elegante.
3. Se lee la función de terminación, ésta contiene una sola línea. La línea que sigue corresponde al tamaño de la ventana, el cual para este problema es 1
4. Se lee el selector, el tipo 0 representa el selector de ruleta, el resto de parámetros en esa línea corresponden a la configuración adicional, en este caso elitismo desactivado.
5. Se leen 2 operaciones. La primera de tipo 0 (Cruce Clásico) y la segunda de tipo 5 (Mutación Clásica).
6. Finalmente se leen los parámetros restantes en el siguiente orden: Tipo de manejador (0-Clásico, 1-Modificado), Tipo de Inicializador(0-Clásico, 1-Permutaciones), Tamaño de la

Población, Número máximo de Generaciones y Parámetro del Manejador Modificado.

Si al abrir la configuración no ocurre ningún error, se prosigue a ejecutar con la opción de maximización, en cuyo caso se abrirá la ventana de resultados que se muestra en la Figura 2.53.

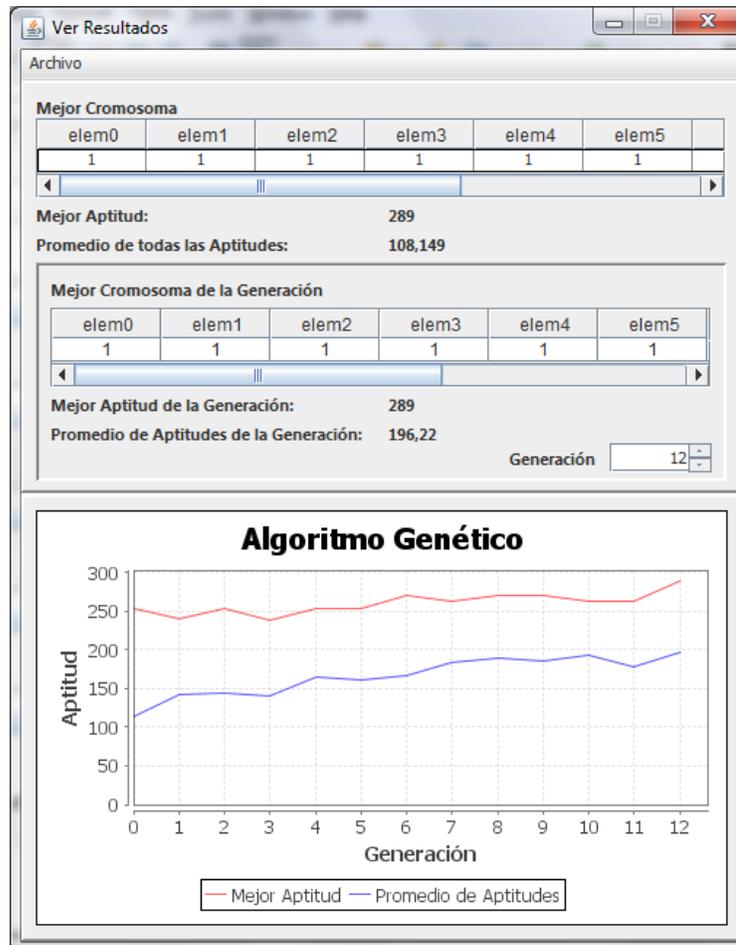


Figura 2.53. Caso de Prueba 2 - Resultados

Se consiguió el resultado esperado en la doceava generación. Se observa que la mejor aptitud incremento levemente a través de las generaciones, siendo en el promedio de aptitudes donde se observa con más facilidad la propiedad de mejores aptitudes a mayor generación.

Caso de Prueba 3

Se realizó el mismo procedimiento que con el primer caso de pruebas. Al guardar la configuración se generó un archivo como el que se observa en la Figura 2.54.

```

6
0 x_0 1 7
0 x_1 1 7
0 x_2 1 7
0 x_3 1 7
0 x_4 1 7
0 x_5 1 7
14
Array($c_0, {0;500;200;185;205;104;232;});
Array($c_1, {500;0;305;360;340;225;154;});
Array($c_2, {200;305;0;320;165;300;250;});
Array($c_3, {185;360;320;0;302;205;213;});
Array($c_4, {205;340;165;302;0;100;198;});
Array($c_5, {104;225;300;205;100;0;345;});
Array($c_6, {232;154;250;213;198;345;0;});
$fitness:= ${6 + $x_0};
$i:=1;
While ($i<6) Do
    $fitness:= $fitness + ${6 + ${$i-1}*7 + ${$i}};
    $i:= $i+1;
End;
$fitness + ${6 + ${$i-1}*7};
0
1
4 0.5 0 1
2
5 0.70000000000000000001
9 0.3
0 1 20 20 10

```

Figura 2.54. Caso de Prueba 3 - Archivo

Se obtuvo el resultado que se muestra en la Figura 2.55, el cual coincide con el mejor resultado que se encontró con la biblioteca en la Tabla 2.3.

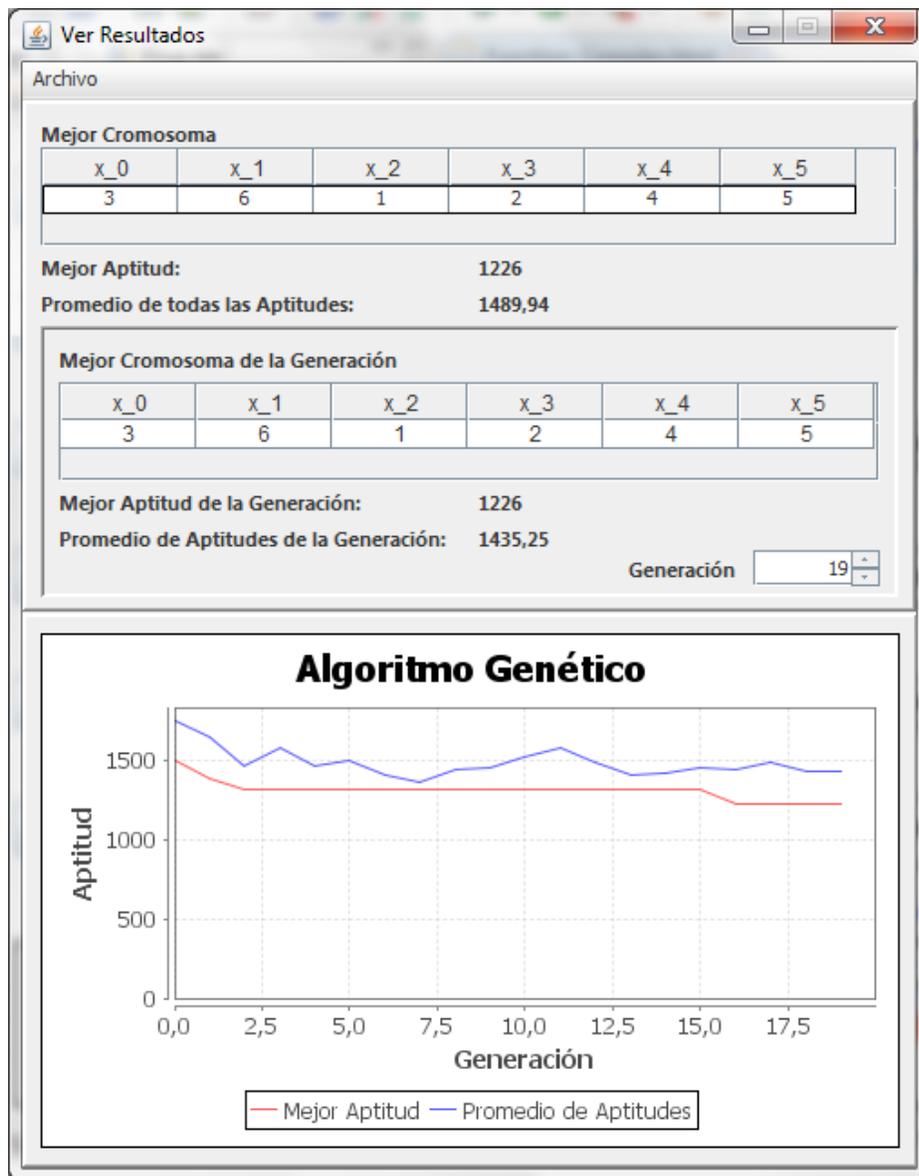


Figura 2.55. Caso de Prueba 2 – Resultados

Como se observa en la gráfica de Generación contra Aptitud en la Figura 2.55, a partir de la 16va generación se consiguió la mejor solución, con el cromosoma 3;6;1;2;4;5 con una aptitud de 1226.

Conclusiones y Recomendaciones

Se desarrolló una aplicación que permite a usuarios resolver problemas de Algoritmos Genéticos sin importar el nivel de experiencia que tenga éste en programación. Se respetaron los lineamientos que fueron utilizados en [18] y se agregaron nuevos basados en las necesidades de la aplicación que se diseñó. La aplicación fue liberada como un proyecto de Software Libre con el nombre JGAL_GUI(Java Genetic ALgorithm Graphical User Interface)⁶. Aunque se utilizó una GUI como base para la interfaz, se le agregaron mejoras, como la inclusión de un sistema de ayuda para el usuario y la posibilidad de guardar la configuración del AG por partes.

A pesar de que se podría haber usado una biblioteca para el desarrollo de Algoritmos Genéticos como las que se muestran en la Sección 1.4, se decidió desarrollar una nueva, con la finalidad de que sea usado para futuros proyectos, material de apoyo o su distribución. Esta biblioteca fue liberada como un proyecto de Software Libre con el nombre JGAL⁷.

La aplicación requería de una forma de representar la función de aptitud y la condición de terminación, por lo que se decidió diseñar un lenguaje de programación simple, el cual está basado en el lenguaje de programación Pascal. Este lenguaje es un híbrido entre imperativo y funcional, dado que sigue la estructura de un lenguaje imperativo, pero todas las instrucciones actúan como una función, es decir tienen retorno, siendo la última instrucción el retorno total de la función. A pesar de que el lenguaje fue diseñado específicamente para esta aplicación, se puede utilizar en futuros proyectos que requieran de un lenguaje simple de usar.

Para lograr la comunicación entre la biblioteca y la interfaz se diseñó un módulo de comunicación, el cual permitió separar la vista del modelo y el controlador. Además, se utilizó una metadata, la cual junto con el API de Java Reflection permitió reflejar las extensiones de la librería en la aplicación, permitiendo que la interfaz sea independiente a la biblioteca y viceversa; de manera que de ser necesario modificar la biblioteca, serían pocas las modificaciones requeridas para la interfaz.

Aunque se logró desarrollar una aplicación que cumple con los objetivos de diseño, es posible implementar algunas funcionalidades que no fueron agregadas a la solución:

- Incorporar otras representaciones para el cromosomas distintas al arreglo de genes como árboles, matrices y listas.
- Permitir al usuario guardar los resultados en formato PDF, de manera que se pueda incluir la gráfica de Aptitud/Generación para una mejor visualización de éstos.
- Diseñar un módulo que permita al usuario crear nuevos Operadores Genéticos, Selectores Naturales, Inicializadores o Genes directamente desde la interfaz de la aplicación.

6 JGAL_GUI: https://github.com/chango1611/JGAL_GUI

7 JGAL: <https://github.com/chango1611/JGAL>

Referencias

1. Michalewicz, Z., *Genetics Algorithm + Data Structures = Evolution Programs*, Third Edition, Springer, 1996
2. Dang, X.T., *Genetic Algorithms and Application in Examination Scheduling*, Scholarly Research Paper, GRIN, 2009
3. Holland, J., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975
4. De Jong, K.A., *Genetic Algorithms: A 10 Years Perspective*, 1985
5. De Jong, K.A., *An Analysis of the Behaviour of a Class of Genetic Adaptative System*, Doctoral Dissertation, University of Michigan, 1975
6. Mitchell, M., *An Introduction to Genetic Algorithm*, Fifth Print, 1999
7. Bethke, A.D., *Genetic Algorithms as Function Optimizers*, Doctoral Dissertation, University of Michigan, 1980.
8. Baker, J.E., *Adaptive Selection Methods for Genetic Algorithms*, in *Proceedings of the First International Conference of Genetic Algorithm*, Hillsdale, NJ, 1985, pp. 101–111.
9. Bäck, T., Hoffmeister, F., *Extended Selection Mechanisms in Genetic Algorithms*, in *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991, pp. 92–99.
10. Eiben, A.E., Smith, J., *Introduction to Evolutionary Computing*, Second Edition, Springer, 2007.
11. Eshelman, L.J., Caruana, R.A., Schaffer, J.D., *Biases in the Crossover Landscape*, in *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, CA, 1989, pp. 10–19.
12. Syswerda, G., *Uniform Crossover in Genetic Algorithms*, in *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, CA, 1989, pp. 2 – 9.
13. Spears, W.M., De Jong, K.A., *On the Virtues of Parametrized Uniform Crossover*, in *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991, pp. 367–384.
14. Wall, M., *GAlib: A C++ Library of Genetic Algorithm Components*, Revision B, 1996
15. Chipperfield, A.J., Fleming, P.J., *The MATLAB Genetic Algorithm Toolbox*.

16. Goldberg, D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley Publishing Company, 1989.

17. *JGAP: Java Genetic Algorithm Package*, fecha de consulta: 3 de Agosto del 2012, disponible en: <http://jgap.sourceforge.net/>.

18. Torres, G., A., *Integración y mejoramiento continuo a la herramienta de Redes Neuronales del Laboratorio de Inteligencia Artificial*, Trabajo Especial de Grado, Universidad Central de Venezuela, 2006.