**Reporte Técnico:**

# Motor Genético Genérico
# *Generic Genetic Engine – GGEngine 1.0*
# (versión Serial 1.0)


Vincent Ho vincenth@fing.edu.uy
Sergio Nesmachnow sergion@fing.edu.uy
Gerardo Ares gares@fing.edu.uy
Pablo Ezzatti pezzatti@fing.edu.uy
Nelson Calero ncalero@fing.edu.uy

Área de Trabajo e Investigación
en Procesamiento Paralelo
y sus Aplicaciones

Centro de Cálculo

Instituto de Computación
Facultad de Ingeniería

Resumen:

El presente reporte resume el desarrollo de la primera etapa del motor genético genérico realizado a fines del año 2002, como resultado del mismo, se obtuvo una versión funcional serial del motor. La continuación de su desarrollo forma parte de los planes actuales del grupo de trabajo, y se busca llegar a uno o varios motores genéticos genérico paralelo para las diferentes plataformas a nuestra disposición (multiprocesador con memoria compartida, cluster de PC).

Con motor genético se refiere a un *solver* que implementa los algoritmos genéticos en forma abstracta, sin instanciarse para ningún problema en particular, hay precedente de motores como lo es *Genesis* [Gen90] pero muchos de ellos solo admiten limitadas representaciones de los cromosomas. En nuestro caso, apuntamos a un motor genérico que manipula los cromosomas y operadores genéticos en forma abstracta, con la meta de ampliar el campo de aplicación. Hay pocos motores genéticos genéricos existentes, el objetivo del grupo de trabajo es desarrollar uno propio, sobre el cual aplicar técnicas de paralelismo y concurrencia.

Palabras claves:

Algoritmo Genético, Metaheurística, Optimización, Combinatoria, Computación Evolutiva, Motor, Motor Genético.

## Índice:


1. Introducción ..... 4
   1.1 Motivación ..... 4
   1.2 Algoritmos Genéticos (Síntesis de la teoría) ..... 4
   1.3 Requerimientos ..... 6
   1.4 Organización del reporte ..... 6
2. Arquitectura del motor de Alg. Gen. ..... 7
   2.1 Introducción ..... 7
   2.2 Módulo “Population” ..... 7
   2.3 Módulo “Engine” ..... 8
   2.4 Módulo “Utilities” ..... 9
3. Implementación del Motor ..... 10
   3.1 Introducción ..... 10
   3.2 Módulo “Population” ..... 10
   3.3 Módulo “Engine” ..... 10
      3.3.1 Operators ..... 10
      3.3.2 Encoding ..... 12
      3.3.3 Engine ..... 12
   3.4 Módulo “Utilities” ..... 13
      3.4.1 Configuration ..... 13
      3.4.2 Random ..... 13
      3.4.3 DB-fitness ..... 13
4. Resultados y sus Análisis ..... 14
5. Trabajos futuros ..... 16
Apéndice ..... 17
   A. API ..... 17
   B. Archivos de configuración ..... 33
   C. Manual de Usuario ..... 34
Referencias ..... 39

# 1. Introducción

## 1.1 Motivación

Algoritmo Genético es una metaheurística que ataca problemas de optimización, con origen sobre los años 1960s. Basándose en la simulación del proceso de la evolución donde solo los individuos más aptos de una población sobrevivan a la selección natural, y así cada generación evoluciona adaptándose a su entorno.

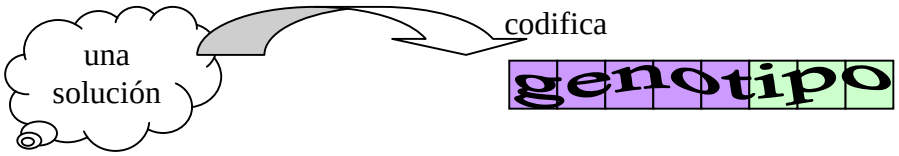
La meta principal del proyecto es la creación de una herramienta (un motor) que facilite la aplicación de ésta técnica a los diversos problemas.

## 1.2 Algoritmos Genéticos (Síntesis de la teoría)

Para aplicar algoritmo genético en la resolución de un problema de optimización, se debe modelar el problema adecuadamente, seleccionando un componente adecuado para cada una de las siguientes partes:

codificación particular a cada problema:

- Codificar las soluciones (***phenotypes***) al problema de interés en forma de individuos (***genotypes***), tomando en cuenta que cada individuo estará compuesto por componentes atómicos llamados "***genes***".

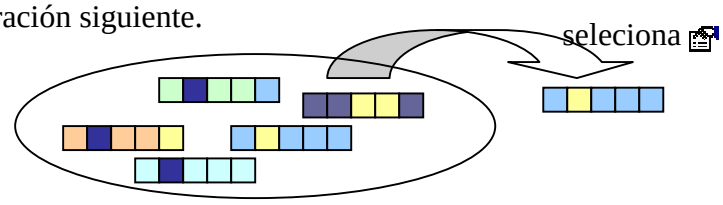



- La función de "***fitness***", es una función que para cada solución (*phenotype*) le asigna un número real que representa lo apto que es la solución respecto al problema.

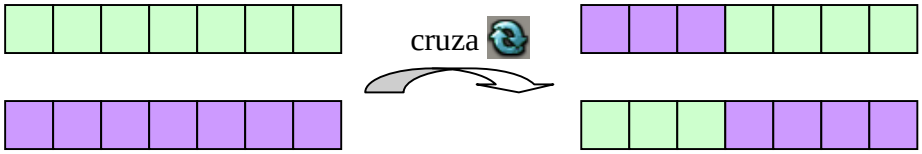
***fitness***: "una solución" $\rightarrow$ $\Re$

operadores genéticos:

- El operador de selección (***selection***): selecciona en cada generación los más aptos tienen mayores chances de ser seleccionados y así propagar sus genes a la generación siguiente.

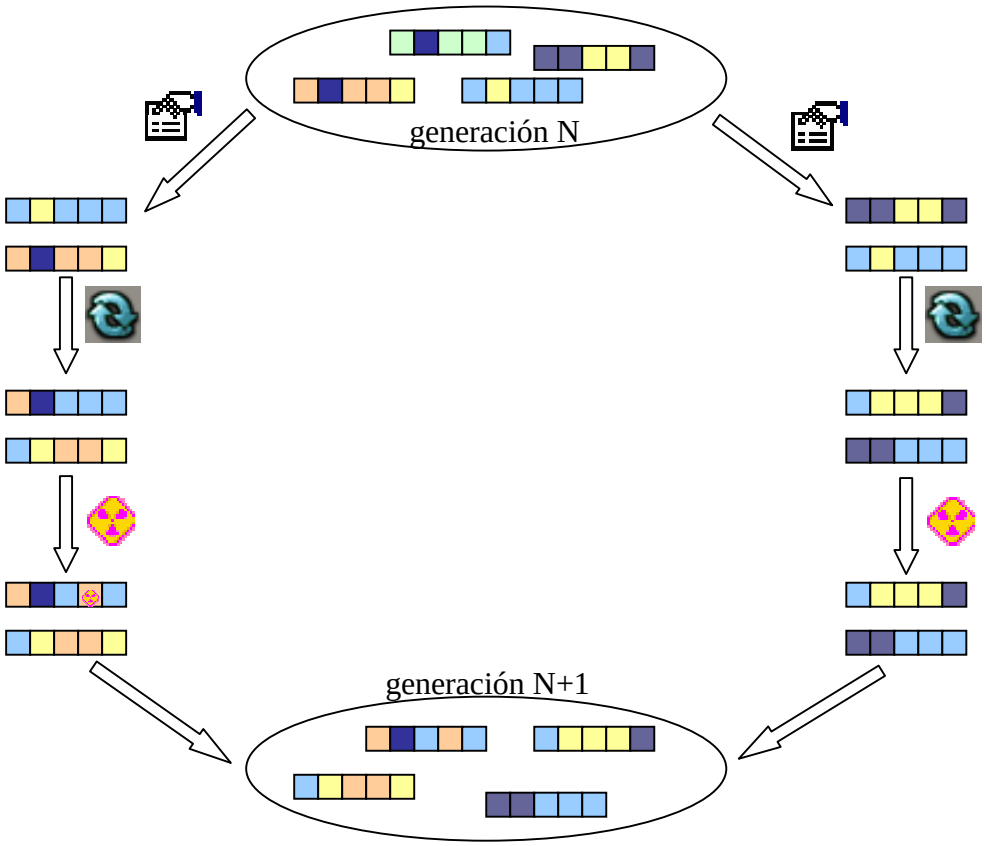


- El operador de cruzamiento (***crossover***): toma dos o múltiples individuos y generan nuevos individuos combinando sus genes.




- El operador de mutación (***mutation***): genera alteraciones en los individuos, sirve para introducir genes a la población y le ayuda a escapar de los óptimos locales.


Para simular la evolución, el algoritmo se inicia con una generación cero, dicha generación puede ser generada en forma aleatoria o generada a partir de alguna heurística. Para avanzar, se selecciona individuos utilizando el operador de selección, luego se les aplicará el operador de cruzamiento y mutación, generando así la siguiente generación. Según el teorema de esquemas [Gol89], las generaciones convergen (evolucionan) hacia buenas soluciones del problema.



## 1.3 Requerimientos

**facilidad de uso**
Para facilitar la aplicación de los Algoritmos Genéticos, el motor a implementar debe ofrecer los operadores genéticos y tipos de genes más estándares, además debe contemplar el uso de operadores o genes definidos por el usuario. Modificaciones en los parámetros de entrada, tipo de genes, u operadores genético a usar deberán poder ser efectuados sin la necesidad de recompilación del motor. Más detalladamente, se ofrece las representaciones de lista de bits, de enteros y reales; como operador de cruzamiento, el cruzamiento en un punto, en dos, en n puntos, cruzamiento uniforme, cruzamiento ponderado; como operador de mutación, la mutación uniforme; como operador de selección, la selección con ruleta, torneo y ranking.

**performance**
Se implementará mecanismos como caché de *fitness*, ahorro en llamadas al generador de números aleatorios, etc... para estudiar posible contribución en la performance de los algoritmos genéticos.

**puertas abierta para el paralelismo**
Muchos de los problemas de interés manejan un gran volumen de datos o requieren de procesamiento intensivo, y es nuestro interés atacar estos problemas con técnicas de paralelismo y distribución. En la primera versión, solo se desarrollará un motor genético genérico serial, pero se debe contemplar el desarrollo de futuras versiones cuyas metas serán distribuir el procesamiento con técnicas de programación paralela y distribuida. El estado de arte de las librerías para la programación distribuida (como PVM, MPI, etc.) tienen mayor desarrollo en C que C++, por dicho motivo, el motor se implementará en C.

## 1.4 Organización del reporte

Las próximas dos secciones describen el diseño del motor. Primero en la sección 2 se describe su arquitectura y cada uno de los módulos, luego en la sección 3 se describe su implementación.

Para continuar en la sección 4 se presenta un análisis de los resultados obtenidos al aplicar el motor a una serie de problemas sencillos. Y se lista las tareas pendientes y los pasos a seguir en la sección 5.

Por último se adjunta el API del motor, el formato del archivo de configuración y un manual de usuario respectivamente en los apéndices A, B y C.

# 2. Arquitectura del motor de Alg. Gen.

## 2.1 Introducción

El objetivo principal de nuestro motor apunta a la resolución de problemas generales, dado que los diversos problemas son modelables en formas muy variadas, el motor debe manejar una abstracción que lo permite operar independiente de las representaciones. Para ello, su diseño apunta a una arquitectura modular que admite el reemplazo de los diferentes componentes, como los operadores genéticos a aplicar, la representación de cada una de las entidades (*gene*, *genotype*, *phenotype*, *population*, etc...).

Se divide en tres módulos: Population, Engine, y Utilities.

## 2.2 Módulo "Population"

Es el módulo que contiene las estructuras básicas con que manipula el motor, como indica su nombre, en él se encapsula la manipulación de la entidad población. Una población es una colección de individuos (genotipos), y a su vez cada individuo (también llamado genotipo o cromosoma), es una agrupación de genes. Los diferentes componentes de este módulo modelan las entidades: ***Population***, ***Genotype***, ***Gene***.

***Population***: representa una población de genotipos, ofrece operaciones de inserción y consulta sobre el conjunto de genotipos.

***Genotype***: representa un genotipo o cromosoma[1], las codificaciones más populares son listas o arreglos de bits, de enteros o decimales; para algunos problemas sobre grafos también se utilizan matrices. En nuestro caso un genotipo es una colección abstracta de genes abstractos.

***Gene***: representa un gene (una característica genética), puede ser implementado de muchas formas, los más estándares son los genes de bit, de enteros o decimales, también admitimos la definición de genes por el usuario.

[1] en la biología, un genotipo está formado por un grupo de cromosomas. Los seres humanos tienen 46 cromosomas, y cada cromosoma es una cadena de ADN. En Alg. Gen. muchas veces se utilizan las terminologías de cromosoma y genotipo como agrupación de genes.

## 2.3 Módulo “Engine”

Éste es el módulo central de nuestro motor genético, y está compuesto por los siguientes componentes: ***Operators***, ***Encoding***, ***Engine***.

***Operators***: hay tres tipos de operadores genéticos, éstos son *Selection*, *Crossover* y *Mutation*. La implementación de cada uno de ellos se puede realizar de formas diversas, dado que es nuestro interés permitir la integración con las diversas variantes, es imprescindible establecer una interfaz para cada uno de los tres tipos de operadores, volviéndolos reemplazables en forma transparente para el resto del sistema. Y además, la implementación de estos operadores debe utilizar genes o genotipos abstractos, para poder independizarse del tipo de genes o genotipos seleccionado.

*Selection*: los operadores de selección más comunes son Ruleta (“*Standard Proportional Selection*”), “*Tournament*” y “*Ranking*”. Se puede ver como una caja negra que toma una población y la evaluación de sus integrantes para luego obtener una población selecta según los parámetros de configuración dada. En general los operadores de selección no seleccionan la población en forma determinística, sino a través de un algoritmo aleatorio.

Población + Evaluación + Generador aleatorio + Configuración -> Población selecta

*Crossover*: los operadores de cruzamiento más comunes son los cruzamientos en uno, en dos, o en **n** puntos [Gol89]; cruzamiento uniforme [Sys89], cruzamiento ponderado, etc. Se puede ver definiciones sobre pares de genotipo, pero para simplificar la estructura del motor, utilizaremos una definición equivalente sobre población. En este caso tomando una población padre como entrada, genera una población hija según los parámetros de configuración, y potencialmente opera con algoritmo no determinístico.

Población padre + Generador aleatorio + Configuración -> Población hija

*Mutation*: mutación uniforme sobre cada gene de los individuos es la más usada, la desventaja de la mutación uniforme es que requiere un gran número de sorteos y la cantidad de llamadas al generador de números aleatorio consume tiempo de procesamiento. Nuevamente utilizaremos la definición sobre población en lugar de individuo para simplificar la estructura del motor. Tomando una población, con la ayuda de un generador de números aleatorio genera una población mutada, respectando siempre los parámetros de configuración.

Población original + Generador aleatorio + Configuración -> Población mutada

***Encoding***: éste componente tiene dos partes: *Encoding* y *Fitness*, se encapsulan los cambios necesarios para modelar los diferentes problemas de optimización. La primera representa la codificación y decodificación entre fenotipos y genotipos, en otras palabras la traducción entre soluciones al problema de optimización a una codificación con lista de genes. La segunda parte "*Fitness"* implementa la función de evaluación (función de *fitness*) para valora cada solución del problema. La función de *fitness* juega un rol muy importante ya que las generaciones van evolucionando para adaptarse (lograr valores altos) a esta función.

***Engine***: es el cuerpo principal del algoritmo genético, aplica los operadores genéticos sobre la población y simula el proceso de la selección natural.

---

*Esqueleto*:

{genera la generación cero al azar}
<- generación $G_0$
mientras {condición}
<- generación $G_N$
{selecciona los más aptos con "op. selection"}
{aplica "op. crossover" sobre la población selecta}
{aplica "op. mutation" sobre la población hija}
<- generación $G_{N+1}$
fin mientras

---

## 2.4 Módulo "Utilities"

Los otros componentes de utilidad se agrupan en el módulo de utilitarios, en él se encuentra los componentes: ***Configuration***, ***Random***, ***DB-fitness***.

***Configuration***: el motor tiene una cantidad de parámetros configurables, éstas se almacenan en un diccionario y puede ser leído desde un archivo de configuración. Con el componente actual se encapsula las operaciones sobre dicho diccionario.

***Random***: en los algoritmos genéticos, hay múltiples puntos donde se requiere la generación de número aleatorio (generación de la población inicial, los diferentes operadores, etc...). Éste componente se encarga de la generación de las diferentes distribuciones a utilizar en el motor.

***DB-fitness***: parte del objetivo del proyecto es desarrollar y estudiar mecanismos que mejoran la performance de los algoritmos genéticos, uno de estos mecanismos es el caché de la evaluación sobre los genotipos (sus *fitness*). Con la interfaz definida de éste componente, facilitamos el reemplazo del mecanismo de caché, ya que puede haber diferentes implementaciones de caché.

# 3. Implementación del Motor

## 3.1 Introducción

En esta sección se discutirá la implementación serial del motor descrito en las secciones anteriores. Con la implementación se mostrará la aplicabilidad de los diversos implementaciones de genes y operadores genéticos a la hora de resolver los diferentes problemas, gracias a un diseño modular y la técnica de puntero a funciones.

## 3.2 Módulo "Population"

Se admite el uso de cuatro tipos diferentes de *Gene*, los booleanos, enteros y reales son implementados, mientras el cuarto tipo es abierto para ser implementado por el usuario. En todos los puntos donde se requiere diferenciar los diferentes tipos de genes se utiliza puntero a función, y según un parámetro de configuración se accede a la implementación correcta de la función. Con ésta técnica, ofrecemos el cambio de representación sin necesitad de recompilación.

En cuando a *Genotype*, se implementó lista de genes como única representación. Durante la redacción del presente documento, otras líneas de trabajo[2] se encontraron con la necesitad de otros tipos de implementación como matrices de genes, esto último se incluye fácilmente reemplazando el componente de genotipo, pero forma parte de la siguiente versión del motor incluir el uso de puntero a función en genotipo y admitir el uso de definición personalizada.

Para *Population* se implementó una única representación, éste es el tipo abstracto de datos Lista.

## 3.3 Módulo "Engine"

### 3.3.1 Operators

El nivel de abstracción en el módulo "Population" permite que los operadores manipulen las diferentes entidades en forma abstracta, en otras palabras, una única implementación sirve para los diferentes tipos de genes o genotipos que hay.

Ofrecemos múltiples implementaciones para los diferentes tipos de operadores, y se deja abierto la posibilidad de incluir nuevas implementaciones. Para aplicar cualquiera de ellas, alcanza con activar los parámetros de configuración correspondientes en el motor.

[2] la versión actual del motor está siendo utilizada entre otros por un Daniel Calegari en su proyecto de grado "Algoritmos Genéticos aplicados al diseño de una Red de Comunicación Confiable", año 2002-2003.

Para el caso de selección, hay tres implementaciones, la selección por ruleta ("*Standard Proportional Selection*"), "*Tournament*" y "*Ranking*". El operador de selección es un punto donde se evalúa los genotipos según sus *fitness*, y acá es donde interviene el caché de *fitness* que proponemos. Uno de los parámetros de entrada es el "evaluador de *fitness*" y tratará los accesos repetidos según su implementación (se discutirá en 3.4 Módulo "Utilities").

Para el caso de cruzamiento, actualmente solo se implementó los siguientes cruzamientos bi-genotipos: un punto, dos puntos, uniforme en un punto, uniforme en dos puntos, ponderado.
Durante la redacción del presente documento otra línea de trabajo[2] se encontró con la necesitad de un mecanismo de corrección en el cruzamiento, esto es debido a que tanto el cruzamiento como la mutación, según la codificación del usuario, pueden generar genotipos corresponden a soluciones no factibles. Para incluir la corrección, se puede solicitar al usuario la implementación de una función de corrección y invocarla luego del cruzamiento, los cambios son mínimos ya que solo requiere incluir el pasaje del un nuevo parámetro.

Ofrecemos dos implementaciones del operador de mutación de la mutación uniforme, debido que es la más popular entre las literaturas. La primera de ellas es la inversión y solo es aplicable en genes de bits, mientras que la segunda es para genes genéricos, para mutar invoca la inicialización particular según el tipo de gene y éste se asegura de generar un valor (puede ser igual al original) válido para el gene mutado.

Como próximas metas, aparte de incluir el uso de mecanismo de corrección, se cambiará la forma de usar el generador de números aleatorios. Actualmente se sortea en cada gene para determinar si se realiza mutación, éste encare cuesta **N** sorteos por genotipo (siendo N la cantidad de genes). La idea es cambiar **N** sorteos de distribución uniforme con probabilidad **p** de éxito, por unos **N*p** sorteos de distribución uniforme sobre las posiciones para elegir las posiciones a mutar, y así bajar la cantidad de llamadas al generador de números aleatorios. En la práctica es más conveniente si el número **N** es grande, para ello es más adecuado aplicar ésta simplificación sobre una población en lugar de sobre cada genotipo.

### 3.3.2 Encoding

Para modelar el problema se define un conjunto de componentes a implementar por los usuarios, tales como la codificación en cromosoma (fenotipo a genotipo y viceversa), la evaluación de su *fitness*.

*user_encoding*:

- definición de fenotipo
- codificación: fenotipo -> genotipo
- decodificación: genotipo -> fenotipo
- evaluación: fenotipo -> $\Re$

Solo se requiere compilar éste archivo y linkeditarlo para adaptar el motor a un nuevo problema. Vale destacar que para realizar ésta tarea, se dispone de un conjunto de utilidades que facilita su desarrollo (vea C. Manual de Usuario, por más detalle sobre estas facilidades).

### 3.3.3 Engine

La implementación serial del motor es simplemente un loop con el esqueleto presentado en la sección anterior, éste que se encarga de mantener una población, aplicando los diversos operadores genéticos sobre él.

Actualmente la condición de parada es por cantidad de generaciones, como próxima tarea se incluirá nuevos mecanismos en cuando a condición de parada. También está en los planes incluir la funcionalidad de salvar y recuperar, así periódicamente o mediante una señal se salvará los estados necesarios para poder retomar la ejecución posteriormente. Los estados necesarios para poder retomar la ejecución son los parámetros de configuración, el estado del generador de números aleatorio, la generación actual, y si corresponde también el caché de *fitness*.

## 3.4 Módulo “Utilities”

### 3.4.1 Configuration

El diccionario en que se guarda los parámetros de configuración, incluye la funcionalidad de lectura desde archivo y se implementará el salvado en la próxima versión junto con el salvado y retoma de ejecución.

El archivo de configuración contiene registros de la forma “*Nombre = Valor*”, entre el archivo de configuración y el motor, hay dos niveles de mapeo, de modo que se puede renombrar los nombres de los parámetros en el archivo de configuración (facilitando el cambio de idioma de interfaz).

### 3.4.2 Random

La implementación del generador de números pseudo-aleatorio se basa en la siguiente propiedad:

$$X_{n+1} = (a * X_n + c) \bmod m \quad \textit{tiene período } m$$
$$\Leftrightarrow \begin{cases} c \textit{ es relativamente primo de } m \\ a-1 \textit{ es múltiplo de } p, \forall p \textit{ factor de } m \\ a-1 \textit{ es múltiplo de } 4, \textit{ si } m \textit{ es múltiplo de } 4 \end{cases}$$

Seleccionando los parámetros correctos se tiene una secuencia de número pseudo-aleatorio de período tan largo como **m**. Vea [Knu81].

### 3.4.3 DB-fitness

Durante el desarrollo se realizó dos implementaciones, la primera no cachea realmente sino es un simple pasa mano que invoca en cada instancia a la función de evaluación. La segunda extiende la definición del genotipo con un valor real que en caso de un genotipo ya evaluado, aprovecha lo procesado. Si en la población hay múltiples genotipos idénticos, al ser de diferentes instancias, se evaluarán una vez para cada una de ellas, pero como ventaja utiliza muy poco espacio extra y no requiere de comparación.

En la versión estable solo se incluye la primera implementación, en la próxima versión se incluirá el “caché en genotipo” junto con otras ideas como caché con envejecimiento.

# 4. Resultados y sus Análisis

Se realizó una serie de pruebas para revisar de forma preliminar, el funcionamiento del motor. Los problemas que los integran son problemas triviales que apuntan a la verificación de características particulares.

Parámetros de configuración serie A:

- cantidad de generaciones: 150
- tamaño de la población: 200
- largo de genotipo: 10
- tasa de cruzamiento: 0.7
- tasa de mutación: 0.01
- operador de selección: ruleta
- operador de cruzamiento: 1 punto
- operador de mutación: uniforme

Parámetros de configuración serie B:

- cantidad de generaciones: 50
- tamaño de la población: 100
- largo de genotipo: 10
- tasa de cruzamiento: 0.7
- tasa de mutación: 0.01
- operador de selección: ruleta
- operador de cruzamiento: 1 punto
- operador de mutación: uniforme

---

Problema 1:

- codificación binaria de números naturales
- problema de la búsqueda de máximo
- función lineal f(x)=x

Dio con el resultado correcto con los parámetros en la serie A y buena aproximación en la serie B (959 en lugar de 1023). Nota que la serie se caracteriza por tener menor cantidad de generaciones, en este caso no suficiente para llegar a la solución, pero se obtiene una aproximación.

Problema 2:

- codificación binaria de números naturales
- problema de la búsqueda de máximo
- función escalón $f(x) = \begin{cases} 40 & si\ x \leq 8 \\ 3 & si\ no \end{cases}$

Dio resultados correctos con los parámetros en las series A y B, lo cual indica que se encontró con la solución con poco procesamiento (serie B 100 individuos hasta 50 generaciones) y se mantuvo cuando se incrementó la diversidad genética (serie A 200 individuos hasta 150 generaciones). Si aparecen en una generación alguno de los pocos individuos aptos, éstos son seleccionados con alta probabilidad en la selección ruleta y prevalecen.

Problema 3:

- codificación binaria de números naturales
- problema de la búsqueda de máximo
- función escalón $f(x) = \begin{cases} 3.1 & si\ x \leq 8 \\ 3 & si\ no \end{cases}$

No dio resultado correcto con las series A ni B, sin embargo cambiando la tasa de mutación a 0.5, sí resultó en ambos casos.

Vale destacar la dificultad de este problema, $2^{10} = 1024$ posibilidades en el espacio de solución, solo 8 de ellos son óptimos en una función tipo escalón, además la mejora en su *fitness* no los vuelven suficientemente atractivos especialmente para la selección ruleta.

En realidad no debería atribuir crédito, al hecho de haberse dado con la solución luego de aumentar la tasa de mutación, ya que 0.5 es una tasa muy alta, muta 5 de cada 10 genes, generando una situación sumamente al azar, similar a elegir mejor de N al azar.

nota: con el problema 2 y 3, se muestra la importancia de la función de *fitness*, muchas veces depende de ella el éxito de la resolución.

Problema 4:

- codificación binaria de números naturales
- problema de la búsqueda de máximo
- función parabólica $f(x) = \begin{cases} 100 - (x - 128)\text{^}2 & si\ 118 \leq x \leq 138 \\ 50 & si\ no \end{cases}$
- el óptimo está en 128, con *fitness* 100

La serie A llegó a buena aproximación 127 con *fitness* 99.

Como segunda ronda, se probó cambiar los diferentes operadores genéticos para verificarlos, y han mostrado resultados similares. No entra en el alcance del reporte actual la comparación entre los diferentes operadores genéticos, la segunda ronda de prueba se realizó con la simple meta de poner en práctica los diferentes operadores en la resolución de problemas.

# 5. Trabajos futuros

Como próxima meta del grupo de trabajo, está el desarrollo de una segunda versión estable del motor serial, en el cual se incluirá las características no implementadas en la versión actual y tomando la retroalimentación de los usuarios, se agregará otras características nuevas. Una vez finalizada la segunda versión, se tomará como base para las versiones sobre plataformas de memoria compartida, cluster de PC, etc.

Un listado de características a incluir en la próxima versión:

- corrección de soluciones no factibles: admite la corrección definida por el usuario en los operadores de cruzamiento y mutación.
- genotipo predefinido + personalizado: permite la coexistencia del genotipo predefinido y definición del usuario, similar al esquema actual de genes predefinido vs. personalizado.
- mutación más liviana: reducir llamadas al generador de números aleatorio en mutación uniforme.
- cruzamiento uniforme más liviana: ídem anterior.
- *save* & *load*: funcionalidad de suspender y retomar ejecución.
- más condición de parada: implementar más variedad de condición de parada en el motor.
- caché de *fitness*: implementación y comparación entre diferentes formas de caché.
- testing más amplios: existen funciones de pruebas clásicas como las funciones de testeo que utilizó De Jong en su tesis PhD [DeJ75], éstas pueden formar parte de las pruebas de futuros motores.

# Apéndice

A. API

A.1.1 Introducción

A continuación se mostrará en detalle la estructura del motor, ofreciendo también la interfaz definida en cada uno de los módulos. En A.1.2 se mostrará un resumen sobre los archivos en cada módulo, luego en A.1.3 se detalla para cada archivo .h la definición de tipos que encapsula, constantes y la interfaz de sus funciones.

A.1.2 Archivos en cada módulo

Population
- user_gene_def.h
- user_gene.h .c
- gene.h .c
- genotype.h .c
- population.h .c

Engine:
- selection.h .c
- crossover.h .c
- mutation.h .c
- user_encoding.h .c
- engine.c <- main

Utilities:
- cte.h
- configuration.h .c
- nrand.h .c
- fitnessdb.h .c

A.1.3 Cabezal de cada archivo

## user_gene_def.h

**typedef:**

*TYPE_GENE_USER_DEF*
Una definición particular de gene, personalizado por el usuario.

*TYPE_LIMIT_USER_DEF*
Una definición particular de dominio para valores del gene definido en "*TYPE_GENE_USER_DEF*", personalizado por el usuario.

## user_gene.h

**procedimientos:**

*code createGene_user(TYPE_GENE*, const Configuration*)*
Crea un puntero vacío de gene genérico ("*TYPE_GENE*") asignándole memoria en caso de contener estructura dinámica. Para liberarlo luego, debe invocar a *freeGene_user*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code initiateGene_user(TYPE_GENE*, const Limit*, Nrand*, const Configuration*)*
Inicializa un puntero a un gene genérico ("*TYPE_GENE*"), tomando un rango genérico ("*Limit*") y los parámetros de configuración. En casos estándares se realiza asignación de memoria y luego lo inicializa con la ayuda del generador de números aleatorios ("*Nrand*"). Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code freeGene_user(TYPE_GENE*)*
Libera un puntero a un gene genérico ("*TYPE_GENE*"), desasignando la memoria asignado con *initiateGene_user*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*int displayGene_user(TYPE_GENE*, char*)*
Obtiene un *String* que representa el gene genérico ("*TYPE_GENE*") terminado en '*\0*' y lo escribe sobre el segundo parámetro. Retorna el largo del *String*.

*code cloneGene_user(TYPE_GENE*, const TYPE_GENE*)*
Clona (copia) un gene sobre otro, ambos deben haber sido creado previamente con *createGene_user*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code readRange_user(Limit*, const Configuration*)*
Carga el rango genérico instanciandolo con *TYPE_LIMIT_USER_DEF*, puede realizar una lectura desde un archivo de configuración cuyo nombre se obtiene con los parámetros de configuración ("*Configuration*") o cargar el rango directamente con datos de configuración. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

---

## gene.h

**typedef:**

*TYPE_GENE*
Definición de gene genérico, es una unión de definiciones de genes.

*Limit*
Definición de rango genérico, es una unión de definiciones de rangos.

*code FUN_createGene(TYPE_GENE*, const Configuration*)*
Definición abstracta de creación de genes. Una función tipo crea un punteo puntero vacío de gene genérico (“*TYPE_GENE*”) asignándole memoria en caso de contener estructura dinámica. Para liberarlo luego, debe invocar a *FUN_freeGene*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*FUNPOINTER_createGene*
Puntero a *FUN_createGene*.

*code FUN_initiateGene(TYPE_GENE*, const Limit*, Nrand*, const Configuration*)*
Definición abstracta de inicialización de genes. Una función tipo inicializa un puntero a un gene genérico (“*TYPE_GENE*”), tomando un rango genérico (“*Limit*”) y los parámetros de configuración. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*FUNPOINTER_initiateGene*
Puntero a *FUN_initiateGene*.

*code FUN_freeGene (TYPE_GENE*)*
Definición abstracta de liberación de genes. Una función tipo libera un puntero a un gene genérico (“*TYPE_GENE*”), desasignando la memoria asignado con *FUN_initiateGene*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*FUNPOINTER_freeGene*
Puntero a *FUN_freeGene*.

*int FUN_displayGene (TYPE_GENE*, char*)*
Definición abstracta de conversión a *String*. Una función tipo obtiene un *String* que representa el gene genérico (“*TYPE_GENE*”) terminado en ‘*\0*’ y lo escribe sobre el segundo parámetro. Retorna el largo del *String*.

*FUNPOINTER_displayGene*
Puntero a *FUN_displayGene*.

*code FUN_cloneGene(TYPE_GENE*, const TYPE_GENE*)*
Definición abstracta de la clonación (duplicado) de genes genéricos (“*TYPE_GENE*”). Como condición previo, ambos genes deben tener memoria asignados con *FUN_createGene*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*FUNPOINTER_cloneGene*
Puntero a *FUN_cloneGene*.

**procedimiento:**

*code cloneLimit(Limit*, const Limit*)*
Duplica un rango genérico ("*Limit*"). Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

---

## genotype.h

**typedef:**

*Genotype*
Definición del genotipo como lista de genes.

*GeneDomain*
Definición a la par con *Genotype*, agrupando los rangos para cada gene, posición a posición.

*code FUN_readRange(Limit*, const Configuration*)*
Definición abstracta de carga inicial de rangos. Una función tipo recorre la lista de rangos e inicializa cada posición, puede realizarse con una lectura a algún archivo de configuración.

*FUNPOINTER_readRange*
Puntero a *FUN_readRange*.

**procedimientos:**

*code initiateGeneDomain(GeneDomain*, FUNPOINTER_readRange[],*
*Configuration*)*
Inicializa un puntero a un *GeneDomain*, asignando memoria y cargando los datos con la ayuda de una instancia particular de *FUN_readRange*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code freeGeneDomain(GeneDomain*)*
Libera un puntero a *GeneDomain*, desasignando la memoria asignada por *initiateGeneDomain*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code getRangeAt(const GeneDomain*, Limit*, const int)*
Obtiene un rango ("*Limit*"), dado un *GeneDomain* y el índice, el rango es copiado en el segundo parámetro. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code getRangeReferenceAt(const GeneDomain*, Limit**, const int)*
Versión optimizada para obtiene un rango ("*Limit*"), dado un *GeneDomain* y el índice. No copia el rango, sino devuelve una referencia a la estructura almacenada en *GeneDomain*. Se recomienda el cuidado en su uso, porque cualquier modificación afectará al original. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code setRangeAt(GeneDomain*, const Limit*, const int)*
Asigna un rango ("*Limit*") en la posición indicada del *GeneDomain*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code createGenotype(Genotype*, const FUNPOINTER_createGene[],*
*const Configuration*)*
Crea un puntero vacío de un genotipo ("*Genotype*") asignándole memoria. Para liberarlo luego, debe invocar a *freeGenotype*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code initiateGenotype(Genotype*, const FUNPOINTER_initiateGene[],*
*const GeneDomain*, Nrand*, const Configuration*)*
Inicializa un puntero a un genotipo ("*Genotype*"), solo se puede invocar luego de *createGenotype*! Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code freeGenotype(Genotype* , const FUNPOINTER_freeGene[],*
*const Configuration*)*
Libera un puntero a un genotipo ("*Genotype*"), desasigna la memoria asignado por *createGenotype*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*int sizeOfGenotype(const Genotype*)*
Retorna el tamaño de un genotipo ("*Genotype*").

*code getGeneAt(const Genotype*, TYPE_GENE*, const int,*
*const FUNPOINTER_cloneGene[], const Configuration*)*
Permite consultar el gene genérico (*TYPE_GENE*) en la posición dada del genotipo ("*Genotype*") copiándolo en el segundo parámetro. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code getGeneReferenceAt(const Genotype*, TYPE_GENE**, const int)*
Versión optimizada para consultar el gene genérico ("*TYPE_GENE*") en la posición dada del genotipo ("*Genotype*"). No realiza copia, sino devuelve una referencia sobre el gene original, se recomienda cuidado con su uso ya que cualquier modificación afectará al gene original. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code setGeneAt(Genotype*, const TYPE_GENE*, const int,*
*const FUNPOINTER_cloneGene[], const Configuration*)*
Asigna un gene genérico ("*TYPE_GENE*") en la posición dada del genotipo ("*Genotype*"). Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code cloneGenotype(Genotype*, const Genotype*,*
*const FUNPOINTER_cloneGene[], const Configuration*)*
Duplica un genotipo ("*Genotype*"). Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*int displayGenotype(const Genotype*, char*, const FUNPOINTER_displayGene [],*
*const Configuration*)*
Obtiene un *String* que representa al genotipo ("*Genotype*"), con la gracia de *FUN_displayGene* particulares. Retorna el largo del *String* como resultado.

---

## population.h

**typedef:**

*Population*
Definición de población, como una lista de *Genotype*.

**procedimientos:**

*code createPopulation(Population*, const int, const FUNPOINTER_createGene[],*
*const Configuration*)*
Crea un puntero vacío de población ("*Population*") asignándole memoria. Para liberarlo luego, debe invocar a *freePopulation*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code initiatePopulation(Population*, const int, const FUNPOINTER_initiateGene[],*
*const GeneDomain*, Nrand*, const Configuration*)*
Inicializa un puntero a una población ("*Population*"), solo se puede invocar luego de *createPopulation*! Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code freePopulation(Population* , const FUNPOINTER_freeGene[],*
*const Configuration*)*
Libera un puntero a una población ("*Population*"), desasigna la memoria asignado por *createPopulation*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*int sizeOfPopulation(const Population*)*
Retorna el tamaño actual de una población ("*Population*"), siempre es menor que la capacidad de la población.

*int capacityOfPopulation(const Population*)*
Retorna la capacidad máxima de la población ("*Population*").

*code getGenotypeAt(const Population*, Genotype*, const int,*
*const FUNPOINTER_cloneGene[], const Configuration*)*
Permite consultar el genotipo ("*Genotype*") en la posición dada de la población (*Population)* copiándolo en el segundo parámetro. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code getGenotypeReferenceAt(const Population*, Genotype**, const int)*
Versión optimizada para consultar el genotipo ("*Genotype*") en la posición dada de la población ("*Population*"). No realiza copia, sino devuelve una referencia sobre el genotipo original, se recomienda cuidado con su uso ya que cualquier modificación afectará al genotipo original. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code setGenotypeAt(Population *, const Genotype *, const int,*
*const FUNPOINTER_cloneGene[], const Configuration*)*
Asigna un genotipo ("*Genotype*") en la posición dada de la población ("*Population*"). Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code pushGenotype(Population*, const Genotype*,*
*const FUNPOINTER_cloneGene[], const Configuration*)*
Agrega un genotipo ("*Genotype*") a la población ("*Population*"), le asigna el úlitmo índice disponible. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code popGenotype(Population*, Genotype*,*
*const FUNPOINTER_cloneGene[], const Configuration*)*
Remueve el último genotipo de la población ("*Population*"), el genotipo es copiado en el segundo parámetro. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

int displayPopulation(const Population*, char*, const FUNPOINTER_displayGene[],
const Configuration *)
Obtiene un *String* que representa a la población ("*Population*"), con la gracia de *FUN_displayGenotype* particulares. Retorna el largo del *String* como resultado.

---

## selection.h

**typedef:**

*code FUN_selection(Population*, const Population*, FitnessDB*,*
*const FUNPOINTER_cloneGene[], Nrand*, const Configuration*)*
Definición abstracta del operador genético selección. Una función tipo realiza la selección sobre una población ("*Population*") y devuelve la población selecta en el primer parámetro. Tiene a su disposición los parámetros de configuración ("*Configuration*") y el caché de *fitness* ("*FitnessDB*"). Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*FUNPOINTER_selection*
Puntero a *FUN_selection*.

**constantes:**

*SELECTION_ROULETTE*
Índice para el operador de selección ruleta (*Standard Proportional Selection*)

*SELECTION_TOURNAMENT*
Índice para el operador de selección torneo.

*SELECTION_RANKING*
Índice para el operador de selección ranking.

---

## crossover.h

**typedef:**

*code FUN_crossover(Population*, const Population*,*
*const FUNPOINTER_createGene[], const FUNPOINTER_freeGene[],*
*const FUNPOINTER_cloneGene[], Nrand*, const Configuration*)*
Definición abstracta del operador genético cruzamiento. Una función tipo realiza el cruzamiento sobre una población ("*Population*") y devuelve la población hija en el primer parámetro. Tiene a su disposición los parámetros de configuración ("*Configuration*"). Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*FUNPOINTER_crossover*
Puntero a *FUN_crossover*.

**constantes:**

*CROSSOVER_1P*
Índice para el operador de cruzamiento bigenotipo en un punto.

*CROSSOVER_2P*
Índice para el operador de cruzamiento bigenotipo en dos punto.

*CROSSOVER_UNIF1*
Índice para el operador de cruzamiento bigenotipo uniforme, devuelve un único genotipo resultante.

*CROSSOVER_UNIF2*
Índice para el operador de cruzamiento bigenotipo uniforme, devuelve dos genotipos resultantes.

*CROSSOVER_WEIGHTED*
Índice para el operador de cruzamiento bigenotipo ponderado, devuelve un único genotipo resultante.

---

## mutation.h

**typedef:**

*code FUN_mutation(Population*, const Population*, const GeneDomain*,*
*const FUNPOINTER_initiateGene*, const FUNPOINTER_cloneGene[],*
*Nrand*, const Configuration*)*
Definición abstracta del operador genético mutación. Una función tipo realiza la mutación sobre una población ("*Population*") y devuelve la población mutada en el primer parámetro. Tiene a su disposición los parámetros de configuración ("*Configuration*"), colección de rangos para el genotipo ("*GeneDomain*"), puntos a las funciones de inicialización de los genes particulares ("*FUNPOINTER_initiateGene*") y el generador de números aleatorios ("*Nrand*").
Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*FUNPOINTER_selection*
Puntero a *FUN_selection*.

**constantes:**

*MUTATION_SIMPLE*
Índice para el operador de mutación simple.

*MUTATION_INVERT*
Índice para el operador de mutación inversión (solo aplicable para genes booleanos).

## user_encoding.h

**typedef:**

*Phenotype*
Definición de fenotipo.

**procedimientos:**

*code createPhenotype(Phenotype*, const Configuration*)*
Crea un puntero vacío de fenotipo ("*Phenotype*") asignándole memoria. Para liberarlo luego, debe invocar a *freePhenotype*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code freePhenotype(Phenotype*)*
Libera un puntero a un fenotipo ("*Phenotype*"), desasigna la memoria asignado por *createPhenotype*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code decode(Phenotype*, const Genotype*, const Configuration*)*
Traduce un genotipo ("*Genotype*") a fenotipo ("*Phenotype*"). Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code encode(Genotype*, const Phenotype*, const Configuration*)*
Traduce un fenotipo ("*Phenotype*") a un genotipo ("*Genotype*"). Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*float user_fitness(const Phenotype*, const Configuration*)*
Función de *fitness*, retorna el *fitness* correspondiente para el fenotipo ("*Phenotype*").

---

## configuration.h

**typedef:**

*Configuration*
Definición del diccionario que contiene el mapeo de parámetro -> valor.

**procedimientos:**

*code createConf(Configuration*)*
Crea un puntero vacío de configuración ("*Configuration*") asignándole memoria. Para liberarlo luego, debe invocar a *destroyConf*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code destroyConf(Configuration*)*
Libera un puntero a una configuración ("*Configuration*"), desasigna la memoria asignado por *createConf*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code readConf(Configuration*, char*)*
Carga la configuración a partir de un archivo de texto cuyo nombre es el segundo parámetro. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*int sizeOfConf(const Configuration*)*
Devuelve la cantidad de parámetros registrados en la configuración ("*Configuration*").

*code getConfInt(const Configuration*, const char* param, int* p_int)*
Obtiene el valor entero *p_int* correspondiente al parámetro *param*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code getConfFloat(const Configuration*, const char* param, float* p_real)*
Obtiene el valor real *p_real* correspondiente al parámetro *param*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code getConfStr(const Configuration*, const char* param, char* str)*
Obtiene el string *str* correspondiente al parámetro *param*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*void displayConf(const Configuration*)*
Despliega el mapeo de los pares parámetro-configuración en pantalla.

---

## cte.h

**typedef:**

*code*
Definición de código de respuesta: {*OK*, *ERROR*}

*boolean*
Definición de booleano: {*true*, *false*, *TRUE*, *FALSE*}

sobre genes

*INDEX_BOOL*
Índice para el gene booleano.

*INDEX_INT*
Índice para el gene entero.

*INDEX_FLOAT*
Índice para el gene real.

MAX_GENE_STRLEN
Cota superior para el largo del string representante de un gene.

sobre archivos del sistema

*RANDOM_FILE*
Nombre del archivo para la inicialización del generador de números aleatorio.

*CONFIGURATION_FILE*
Nombre del archivo de configuración.

sobre nombres de parámetros de configuración

*PARAM_TYPE_GENE*
Nombre del parámetro que indica el tipo de gene.

*PRARM_GENOTYPE_LEN*
Nombre del parámetro que indica el largo del genotipo.

*PARAM_POPULATION_SIZE*
Nombre del parámetro que indica la dimensión de la población.

*PARAM_CROSSOVER_RATE*
Nombre del parámetro que indica la tasa de cruzamiento.

*PARAM_MUTATION_RATE*
Nombre del parámetro que indica la tasa de mutación.

*PARAM_GENERATIONS*
Nombre del parámetro que indica la cantidad de generaciones a evolucionar.

*PARAM_GENEDOMAIN_BOOL_FILE*
Nombre del parámetro que indica el archivo de inicialización para los genes booleanos.

*PARAM_GENEDOMAIN_INT_FILE*
Nombre del parámetro que indica el archivo de inicialización para los genes enteros.

*PARAM_GENEDOMAIN_FLOAT_FILE*
Nombre del parámetro que indica el archivo de inicialización para los genes reales.

*PARAM_SELECTION_OPERATOR*
Nombre del parámetro que indica el operador genético de selección a utilizar.

*PARAM_CROSSOVER_OPERATOR*
Nombre del parámetro que indica el operador genético de cruzamiento a utilizar.

*PARAM_MUTATION_OPERATOR*
Nombre del parámetro que indica el operador genético de mutación a utilizar.

*PARAM_CROSSOVER_WEIGHT*
Nombre del parámetro que indica el peso a favor del genotipo más apto, en un cruzamiento con ponderación.

*PARAM_SELECTION_SIZE*
Nombre del parámetro que indica el tamaño de la población selecta.

*PARAM_TOURNAMENT_SIZE*
Nombre del parámetro que indica el tamaño del torneo en una selección con torneo ("Tournament").

valores por defectos del sistema

*DEFAULT_TYPE_GENE*
Valor por defecto para el tipo de gene, *INDEX_BOOL*.

*DEFAULT_GENOTYPE_LEN*
Valor por defecto en el largo de los genotipos, 10.

*DEFAULT_POPULATION_SIZE*
Valor por defecto para el tamaño de la población, 100.

*DEFAULT_CROSSOVER_RATE*
Valor por defecto para la tasa de cruzamiento, 0.7

*DEFAULT_MUTATION_RATE*
Valor por defecto para la tasa de mutación, 0.01

*DEFAULT_GENERATIONS*
Valor por defecto para la cantidad de generaciones a evolucionar, 150.

*DEFAULT_GENEDOMAIN_BOOL_FILE*
Valor por defecto para el nombre del archivo de inicialización de genes booleanos, "range_bool.dat".

*DEFAULT_GENEDOMAIN_INT_FILE*
Valor por defecto para el nombre del archivo de inicialización de genes enteros, "range_int.dat".

*DEFAULT_GENEDOMAIN_FLOAT_FILE*
Valor por defecto para el nombre del archivo de inicialización de genes reales, "range_float.dat".

*DEFAULT_GENEDOMAIN_USER_FILE*
Valor por defecto para el nombre del archivo de inicialización de genes personalizados, “range_user.dat”.

*DEFAULT_GENE_TRUE_RATE*
Valor por defecto de la tasa que se utiliza para la inicialización de genes booleanos, 0.5 .

*DEFAULT_GENE_INT_MIN*
Valor por defecto para la cota inferior de genes enteros, 0.

*DEFAULT_GENE_INT_MAX*
Valor por defecto para la cota superior de genes enteros, 10.

*DEFAULT_GENE_FLOAT_MIN*
Valor por defecto para la cota inferior de genes reales, 0.0 .

*DEFAULT_GENE_FLOAT_MAX*
Valor por defecto para la cota superior de genes reales, 1.0 .

*DEFAULT_SELECTION_OPERATOR*
Valor por defecto de operador genético de selección a utilizar, 0 que corresponde a la selección por ruleta.

*DEFAULT_CROSSOVER_OPERATOR*
Valor por defecto de operador genético de cruzamiento a utilizar, 0 que corresponde al cruzamiento en un punto.

*DEFAULT_MUTATION_OPERATOR*
Valor por defecto de operador genético de cruzamiento a utilizar, 0 que corresponde a la mutación uniforme.

*DEFAULT_CROSSOVER_WEIGHT*
Valor por defecto del peso en caso de un cruzamiento ponderado, 0.6 .

*DEFAULT_SELECTION_SIZE*
Valor por defecto del tamaño de la población selecta, *SELECTION_SIZE*/2.

---

## nrand.h

**typedef:**

*Nrand*
Estructura del generador de números aleatorios.

**procedimientos:**

*code initiateNrandFromFile(Nrand*, char* filename)*
Inicializa el generador de números aleatorios ("*Nrand*") a partir de un archivo de texto cuyo nombre es *filename*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*unsigned long NrandNext(Nrand*)*
Retorna un natural al azar.

*unsigned long NrandNextInt(unsigned long max_val, Nrand*)*
Retorna un natural al azar en un rango entre cero y *max_val*.

*double NrandUni(Nrand*)*
Retorna un valor entre [0,1] sorteado con distribución uniforme.

*int NrandIntRank(int min, int max, Nrand*)*
Retorna un valor entero entre *int_min* e *int_max*, sorteado con distribución uniforme.

*float NrandFloatRank(double float_min, double float_max, Nrand*)*
Retorna un valor real entre *float_min* y *float_max*, sorteado con distribución uniforme.

*boolean NrandFlip(double p, Nrand*)*
Retorna un valor booleano sorteado con probabilidad *p*.

---

## fitnessdb.h

**typedef:**

*FitnessDB*
Estructura del caché de fitness.

**procedimientos:**

*code createFitnessDB(FitnessDB*, const Configuration*)*
Crea un puntero vacío al caché de fitness (“*FitnessDB*”) asignándole memoria. Para liberarlo luego, debe invocar a *freeFitnessDB*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*code freeFitnessDB(FitnessDB*)*
Libera un puntero a caché de fitness (“*FitnessDB*”), desasigna la memoria asignado por *createFitnessDB*. Retorna el código *OK* si se completó con éxito, y *ERROR* en caso contrario.

*float getFitness(const Phenotype*, FitnessDB*, const Configuration*)*
Retorna dado un fenotipo (“*Phenotype*”), su fitness correspondiente.

---

B. Archivos de configuración

**config.dat**

nota:
- se puede insertar comentarios con la ayuda del símbolo #
- la sintaxis es de la forma Parámetro = Valor

listado de los parámetros:

| Parámetro | Propósito | Valores |
|---|---|---|
| TYPE_GENE | tipo de gene | 0 booleano<br>1 entero<br>2 real<br>3 personalizado |
| GENOTYPE_LEN | largo de genotipo | número natural |
| BOOLEAN_GENE_RANGE_FILE | archivo de rango para genes booleanos | ruta y nombre |
| INTEGER_GENE_RANGE_FILE | archivo de rango para genes enteros | ruta y nombre |
| FLOAT_GENE_RANGE_FILE | archivo de rango para genes reales | ruta y nombre |
| USER_GENE_RANGE_FILE | archivo de rango para genes personalizados | ruta y nombre |
| GENERATIONS | cantidad de generaciones a evolucionar | número natural |
| POPULATION_SIZE | tamaño de la población | número natural |
| CROSSOVER_RATE | tasa de cruzamiento | real entre [0,1] |
| MUTATION_RATE | tasa de mutación | real entre [0,1] |
| SELECTION_OPERATOR | operador de selección | 0 ruleta<br>1 torneo<br>2 ranking |
| CROSSOVER_OPERATOR | operador de cruzamiento | 0 un punto<br>1 dos puntos<br>2 uniforme 1 hijo<br>3 uniforme 2 hijos<br>4 ponderado 1 hijo |
| MUTATION_OPERATOR | operador de mutación | 0 uniforme<br>1 uniforme inversión |
| CROSSOVER_WEIGHT | peso en caso de cruzamiento ponderado | real entre [0,1] |
| SELECTION_SIZE | tamaño de la población selecta (*mating pool*) | número natural |
| TOURNAMENT_SIZE | tamaño del torneo en caso de selección con torneo | número natural |

C. Manual de Usuario

Para utilizar el motor en la resolución de un problema, primero se debe modelar las soluciones al problema en forma de genotipos, y definir una función de *fitness* adecuado, éstos dos elementos son claves en la resolución de los problemas.
A continuación mostramos los pasos a seguir en el momento de resolver un problema con nuestro motor.

**Primer paso:** se debe elegir un tipo de genes en el archivo de configuración "*config.dat*" asigna el parámetro *TYPE_GENE* en:

0 genes booleanos
1 genes enteros
2 genes reales (float)
3 definición de gene del usuario

Si la elección es **3**, se utiliza la definición del usuario, y se debe realizar adicionalmente lo siguiente:

- en el archivo "*user_gene_def.h*" definir la estructura del gene *TYPE_GENE_USER_DEF* y la de su dominio *TYPE_LIMIT_USER_DEF*.

ej. 1.1.a:

```
typedef char TYPE_GENE_USER_DEF;     (define un gene como un char)
```

ej. 1.1.b:

```
typedef struct NODO_DE_LISTA {
        char                 a;
        char                 b;
        NODO_DE_LISTA * next;
        };

typedef struct{
        NODO_DE_LISTA * dat;
        } TYPE_GENE_USER_DEF;        (o estructuras más avanzadas)
```

ej. 1.2.a:

```
typedef enum {e1='A', e2='B', e3='C'} AUX;

typedef struct{
        AUX   dominio;               (define el dominio)
        } TYPE_LIMIT_USER_DEF;
```

ej. 1.2.b:

```
typedef  struct{
        int     cant_nodo_minimo;
        int     cant_nodo_maximo;
        AUX   dominio;               (dominio tan raro
        }TYPE_LIMIT_USER_DEF;        como se necesita)
```

- en el archivo "*user_gene.c*" se debe implementar cada procedimiento establecida en la interfaz (vea los ejemplos).
    - ***createGene_user*** (asigna memoria, si gene contiene estructura)
    - ***initiateGene_user*** (inicializa un gene con el dominio dado)
    - ***freeGene_user*** (libera memoria, si gene contiene estructura)
    - ***cloneGene_user*** (copia un gene)
    - ***displayGene_user*** (representación en un String)
    - ***readRange_user*** (carga el dominio desde un archivo de dominio)

  Los parámetros que se disponen son de tipo "*TYPE_GENE*":

  *typedef union{*

  *...*

  *TYPE_GENE_USER_DEF TYPE_GENE_UDEF;*

  *}* ***TYPE_GENE****;*

  De tipo "*Limit*"

  *typedef union{*

  *...*

  *TYPE_LIMIT_USER_DEF TYPE_LIMIT_UDEF;*

  *}* ***Limit****;*

  Algunos de los procedimientos reciben también parámetro de tipo "*Configuration* conf*" y/o "*Nrand* NR*", el primero de ellos permite el acceso a parámetros de configuración interesantes como:

  - *PARAM_GENEDOMAIN_USER_FILE*
  - *DEFAULT_GENEDOMAIN_USER_FILE*
  - o cualquier otro que aparezca en ***cte.h*** con el prefijo *PARAM_* o *DEFAULT_*

  se puede acceder a sus valores con *getConfStr* (caso string) y *getConfInt* (entero):

  *getConfStr(conf, PARAM_GENEDOMAIN_USER_FILE, &file_name);*
  *getConfStr(conf, DEFAULT_GENEDOMAIN_USER_FILE, &file_name);*

  Por otro lado con *Nrand* se accede al generador de números aleatorios (vea la API de ***nrand.h***), en resumen:

  - *double NrandUni(Nrand *)*
  - *int NrandIntRank(int min, int max, Nrand*)*
  - *float NrandFloatRank(double min, double max, Nrand*)*
  - *boolean NrandFlip(double, Nrand *)*

algo sumamente útil al implementar *initiateGene_user*.

ej. 1.3.a: (caso sencillo, siguiente el hilo de la serie 1.1.a, 1.2.a)

```
code createGene_user(TYPE_GENE* gen, const Configuration* conf){
    //no requiere memoria dinámica
    return OK;
    }

code initiateGene_user(TYPE_GENE* gen, const Limit* lim, Nrand* NR,
const Configuration* conf){
    int n;

    n = NrandIntRank(0, 2,NR);
    // u otro sorteo que dependa de información en lim
    switch(n){
        case 0:
            gen-> TYPE_GENE_UDEF = e1;
            break;
        case 1:
            gen-> TYPE_GENE_UDEF = e2;
            break;
        case 2:
            gen-> TYPE_GENE_UDEF = e3;
            break;
        }
    return OK;          // en caso de error debería ser "return ERROR"
    }


code freeGene_user(TYPE_GENE* gen){
    // no nada que liberar
    return OK;
    }

int displayGene_user(TYPE_GENE* gen, char* str){
    str[0]=gen-> TYPE_GENE_UDEF;// recuerda que es un char
    str[1]='\0';
    return 1;           // largo del string
    }

code cloneGene_user(TYPE_GENE* dest, const TYPE_GENE* src){
    dest-> TYPE_GENE_UDEF = src-> TYPE_GENE_UDEF;
    return OK;
    }

code readRange_user(Limit*, const Configuration*){
    // podría no hacer nada, pero en general parsea un archivo
    // y carga datos relevantes para formar el dominio
    return OK;
    }
```

**Segundo paso:** se define el fenotipo y su codificación en genotipo:

- en el archivo "*user_encoding.h*" definir la estructura del fenotipo, en otras palabras, la estructura de las soluciones del problema.
- luego se debe implementar la evaluación del *fitness*, la codificación de fenotipo a genotipo y viceversa.
  - ***user_fitness***: es la evaluación de fenotipo, indica que tan apta es una solución en el problema.
  - ***encode***: codifica fenotipo en genotipo, traduce cada solución del problema en un genotipo según el tipo de genes seleccionado.
  - ***decode***: decodifica un genotipo en fenotipo, traduce genotipo en solución.

ej. 2.a:

Supongamos que el problema original es minimizar una función trivial **f** siendo **f(x) = x**, con **x** una variable natural. Y que la codificación es en base tres (3 símbolos en el alfabeto), siendo un genotipo de largo **l**, se puede representar ***x*** en $[0, 3^l]$. Entonce el modelo se implementa como sigue:

```
typedef int Phenotype;


code createPhenotype(Phenotype* phen, const Configuration* conf){
        // no hay memoria dinámica que alojar en Phenotype
        // o sea no requiere las sentencias de malloc
        return OK;
        }

code freePhenotype(Phenotype* phen){
        // tampoco se requiere liberar
        return OK;
        }

code decode(Phenotype* phen, const Genotype* geno,
const Configuration* conf){
        // nota que en este punto, el usuario conoce que tipo de gene se usa
        // usuarios avanzados que se quieren tener una decodificación o
        // codificación aplicable a múltiples tipos de genes, se puede
        // averiguar el tipo de genes en uso con
        // getConfInt(conf, PARAM_TYPE_GENE, &n)
        // y los constantes: INDEX_BOOL, INDEX_INT, INDEX_FLOAT,
        //                   INDEX_USER_TYPE
        int i, l;
        TYPE_GENE *gen;
        getConfInt(conf, PRARM_GENOTYPE_LEN, &l);
```

```
        (*phen)=0;
        for(i=0;i<l;i++){
                getGeneReferenceAt(geno,&gen,i);
                switch(gen-> TYPE_GENE_UDEF){
                        case e1: break;
                        case e2: (*phen)+= pow(3, i); break;
                        case e3: (*phen)+= 2*pow(3, i); break;
                        }
                }
        return OK;
        }



code encode(Genotype* geno,const Phenotype* phen,
const Configuration* conf){
        int i, l, value;
        TYPE_GENE *gen;
        getConfInt(conf,PRARM_GENOTYPE_LEN,&l);

        value=(*phen);
        for(i=l-1;i>=0;i--){
                getGeneReferenceAt(geno,&gen,i);  // modificación en gen
                remain=pow(3, i);                 // se refleja en geno
                switch( (value/remain){
                        case 0:
                                gen-> TYPE_GENE_UDEF = e1;
                                break;
                        case 1:
                                gen-> TYPE_GENE_UDEF = e2;
                                break;
                        case 2:
                                gen-> TYPE_GENE_UDEF = e3;
                                break;
                        }
                        value-=remain;
                }
        return OK;
        }


float user_fitness(const Phenotype* phen, const Configuration* conf){
        return (float)(*phen);          // f(x)=x
        }
```

**Tercer paso**: por último se selecciona los operadores genéticos a aplicar junto con la condición de parada, se setea simplemente en el archivo de configuración. Vea el apéndice B. Archivos de configuración.

Luego de la ejecución del motor, simplemente se desplegará en la pantalla el mejor individuo junto con su *fitness*. En la próxima versión se incluirá otros mecanismos de representación.

# Referencias


[Gol89] David E. Goldberg, "Genetic Algorithm in Search, Optimization, and Machine Learning", Addison Wesley Longman, Inc., ISBN 0-201-15767-5.

[Knu81] D. Knuth, The Art of Computing, Vol. II: Seminumerical Algorithms, sección 3.2, Addison Wesley Longman, Inc., 1981.

[DeJ75] K. De Jong. An analysis of the behaviour of a class of genetic adaptive systems. PhD thesis, University of Michigan.

[Gen90] Genesis 5.0, por John Grefenstette. Software no comercial obtenible en: http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/genetic/ga/systems/genesis/0.html

[Sys89] Uniform Crossover in Genetic Algorithms, G. Syswerda. Proceedings of the Third International Conference on Genetic Algorithms, pp 2-9, 1989.