

Tesis de Maestría

UNIVERSIDAD VERACRUZANA

INSTITUTO DE INGENIERIA



Instituto de Ingeniería
Universidad Veracruzana

USO DE ALGORITMOS GENETICOS PARA LA OPTIMIZACIÓN
DE PROBLEMAS EN INGENIERIA



PARA OBTENER EL GRADO DE
MAESTRIA EN CIENCIAS DE LA COMPUTACION

P R E S E N T A

FRANCISCO ALBERTO ALONSO FARRERA



UNIVERSIDAD VERACRUZANA INSTITUTO DE INGENIERIA

H. Veracruz, Ver., Abril 19 de 1996.
DI174/96.

Al candidato al Grado señor:
ING. FRANCISCO ALBERTO ALONSO FARRERA
P R E S E N T E:

En atención a su solicitud relativa, me es grato transcribir a Usted a continuación el tema que aprobado por esta Dirección propuso el Dr. José Luis Vargas López, para que lo desarrolle como tesis, para obtener el Grado de Maestría en Ciencias de la Computación:

T E M A:

"USO DE ALGORITMOS GENETICOS PARA LA OPTIMIZACION
DE PROBLEMAS EN INGENIERIA"

- I .- Introducción
- II .- Objetivo
- III .- Hipótesis
- IV .- Antecedentes de los Algoritmos Genéticos
- V .- Algoritmos Genéticos
- VI .- Ejemplo de Aplicación de un Algoritmo Genético en la Ingeniería
- VII .- Conclusiones
- Anexos
- Bibliografía

Sin otro particular, me es grato quedar de usted como su atento y seguro servidor.

A T E N T A M E N T E
"LIS DE VERACRUZ: ARTE, CIENCIA, LUZ"


DR. ENRIQUE A. MORALES GONZALEZ
DIRECTOR

EMG/lrl.

Proyecto de Digitalización de Tesis
Responsable M.B. Alberto Pedro Lorandi Medina
Colaboradores: Estanislao Ferman García
M.B. Enrique Rodríguez Magaña

Instituto de Ingeniería
Universidad Veracruzana

DEDICATORIAS

A MIS PADRES

LORENZO ALONSO SANTIAGO

Por ser un ejemplo a seguir como persona,
por su paciencia y cariño cuando se necesita,
por su sabiduría y serenidad ante los problemas.

AMABLE FARRERA DE ALONSO

Por su cariño y comprensión,
por su apoyo emocional,
por ser como es cuando debe ser.

A MIS HERMANOS

ELSA GABRIELA, MARTIN EDUARDO Y SILVIA DEL CARMEN

Porque además de hermanos son mis mejores amigos y porque
fueron un ejemplo para mi desarrollo personal.

A MIS SOBRINOS

MARTIN ALBERTO, RICARDO ADRIAN Y MARIA KARINA

A MI ABUELITA

JULIA FARRERA LLAVEN

Por su cariño y bendiciones que siempre me protegen,
que Dios siempre la cuide.

**A TODOS AQUELLOS QUE HICIERON POSIBLE,
QUE LO QUE ERA PARA MI UN SUEÑO SE HICIERA REALIDAD.**

INDICE

INDICE DE FIGURAS.....	iii
INDICE DE TABLAS.....	iv
I. INTRODUCCION.....	1
II. OBJETIVO.....	3
III. HIPOTESIS.....	4
IV. ANTECEDENTES DE LOS ALGORITMOS GENETICOS.....	5
4.1 ¿Qué son los algoritmos genéticos?.....	5
4.2 Terminología.....	8
4.3 Algoritmos genéticos y los métodos tradicionales.....	10
4.4 Aplicaciones.....	13
V. ALGORITMOS GENETICOS.....	16
5.1 Reproducción.....	16
5.2 Cruza.....	19
5.3 Mutación.....	22
5.4 Operadores avanzados.....	23
5.5 Implementación de un algoritmo genético.....	29
5.6 Ambientes de programación.....	44

VI. EJEMPLO DE APLICACION DE UN ALGORITMO	46
6.1 Introducción al problema.	46
6.2 Planteamiento del problema.	48
6.3 Uso del algoritmo genético.	53
6.4 Ejemplos y comparación de resultados.	56
VII. CONCLUSIONES	61
ANEXOS	63
BIBLIOGRAFIA	103

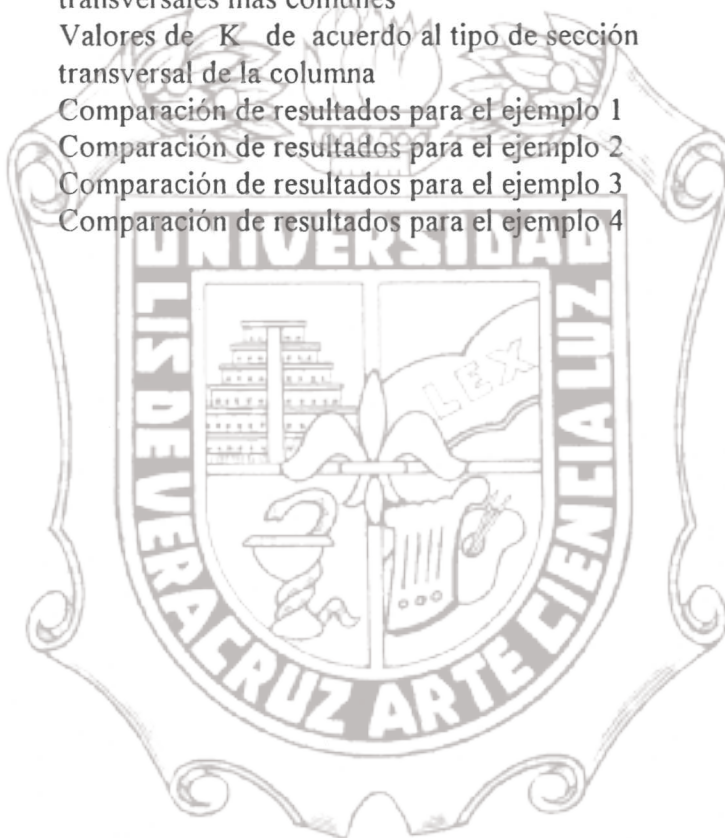


INDICE DE FIGURAS

FIGURA	DESCRIPCION	
5.1	La reproducción simple asigna las nuevas cadenas usando una rueda de la ruleta con tamaños de ranuras en acorde con su aptitud	14
5.2	Un esquema de una cruce simple muestra la alineación de dos cadenas y el intercambio parcial de información, usando un sitio de cruce elegido aleatoriamente.	15
5.3	Mapa de dominancia de dos-localidades. Después de Hollstien (1971)	21
5.4	Mapa de dominancia trialéica de simple-localidad. Después de Hollstien (1971)	21
5.5	Esquema de una cadena de población en un algoritmo genético	23
5.6	Un algoritmo genético simple, SGA, declaraciones de tipos de datos en Pascal	23
5.7	Esquema de poblaciones no sobrepuestas usadas en el SGA	24
5.8	Declaraciones de variables globales del SGA en pascal	24
5.9	La función <i>select</i> implementa la selección de la ruleta	26
5.10	Procedimiento <i>crossover</i> implementa cruce simple (punto simple)	27
5.11	La función <i>mutation</i> implementa un bit sencilla, de punto de mutación	28
5.12	Procedimiento <i>generation</i> genera una nueva población de una población previa	29
5.13	La función <i>decode</i> decodifica una cadena binaria como un entero simple sin signo	30
5.14	La función <i>objfunc</i> calcula la función aptitud $f(x)=x^{10}$ de el parámetro decodificado x .	30
5.15	Programa principal para un algoritmo genético simple, SGA	31
5.16	Procedimiento <i>statics</i> calcula importantes poblaciones estáticas	32
5.17	Procedimiento <i>report</i> y procedimiento <i>writchrom</i> implementan reportes de poblaciones	33
6.1	Columna en estudio, y las posibles secciones que se considerarán	37

INDICE DE TABLAS

TABLA	DESCRIPCION	
4.1	Correspondencia entre la terminología natural y la artificial	8
5.1	Ejemplo del problema de cadenas y valores de aptitud	14
6.1	Valores de la constante α para las secciones transversales más comunes	38
6.2	Valores de K de acuerdo al tipo de sección transversal de la columna	39
6.3	Comparación de resultados para el ejemplo 1	43
6.4	Comparación de resultados para el ejemplo 2	43
6.5	Comparación de resultados para el ejemplo 3	44
6.6	Comparación de resultados para el ejemplo 4	45



Instituto de Ingeniería
Universidad Veracruzana

I. INTRODUCCION

Los algoritmos genéticos son una nueva técnica de búsqueda basada en la teoría de la evolución, de acuerdo a los cuales los individuos más aptos de una población son los que sobreviven al adaptarse más fácilmente a los cambios que se producen en su entorno.

John Holland¹, investigador de la Universidad de Michigan, consciente de la importancia de la selección natural a fines de los 60s desarrolló una técnica que permitió incorporar ésta en un programa de computadora. Su objetivo era lograr que las computadoras aprendieran por si mismas.

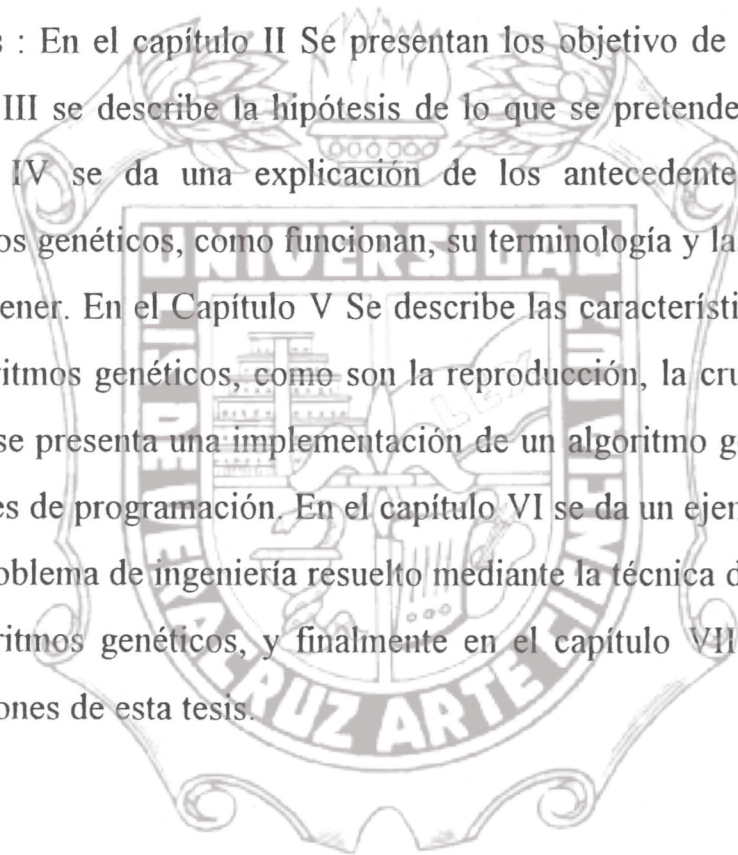
Esta técnica de búsqueda recibió el nombre de "*planes reproductivos*", pero se hizo popular bajo el nombre de "*Algoritmos Genéticos*".

La aplicación más común de los algoritmos genéticos ha sido la solución de problemas de optimización, en donde han demostrado ser muy eficientes y confiables.

Esta tesis presenta una forma de aplicar los algoritmos genéticos al diseño óptimo de columnas no prismáticas sometidas a carga axial. Para poder usar esta técnica se requirió replantear el problema de diseño de una columna de forma que se convirtiera en uno de optimización que busca obtener el diseño con el volumen mínimo de material. Así mismo, se requirió idear un esquema apropiado de representación del espacio de búsqueda del problema que es

continuo y no discreto como suele suceder en los problemas típicos a los que se aplica esta técnica.

En los siguientes capítulos se describen con más detalles las características de los algoritmos genéticos, su funcionamiento y la aplicación que se implementa. A continuación se describe el contenido de la tesis detallado por capítulos : En el capítulo II Se presentan los objetivo de esta tesis, y en el capítulo III se describe la hipótesis de lo que se pretende demostrar. En el capítulo IV se da una explicación de los antecedentes y que son los algoritmos genéticos, como funcionan, su terminología y las aplicaciones que pueden tener. En el Capítulo V Se describe las características principales de los algoritmos genéticos, como son la reproducción, la cruce y la mutación. además se presenta una implementación de un algoritmo genético, y algunos ambientes de programación. En el capítulo VI se da un ejemplo de aplicación de un problema de ingeniería resuelto mediante la técnica de optimización de los algoritmos genéticos, y finalmente en el capítulo VII, se presentan las conclusiones de esta tesis.

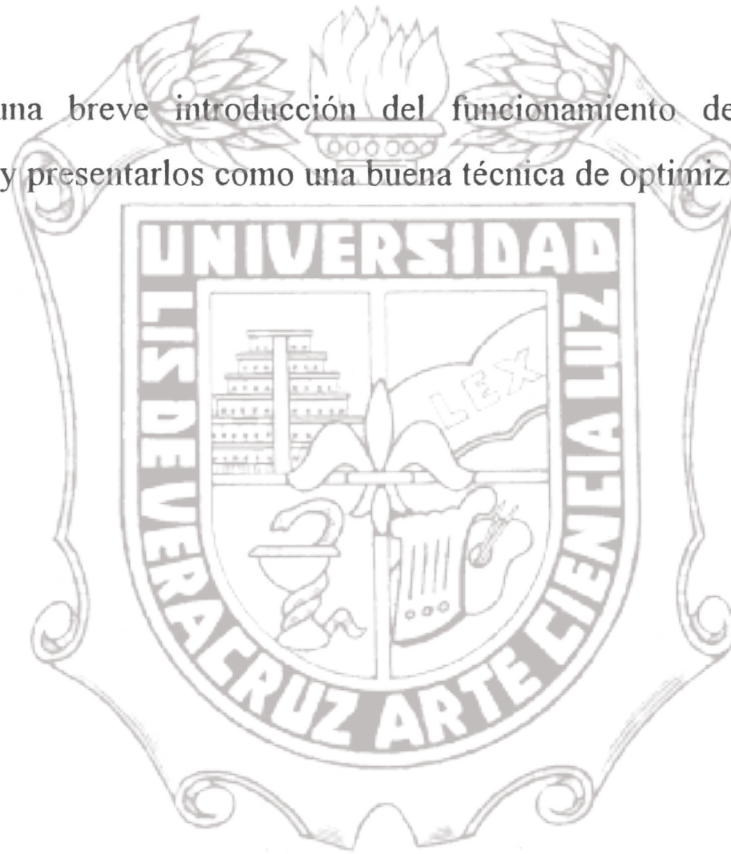


II. OBJETIVO

Los objetivos de esta tesis son los siguientes:

Presentar una técnica eficiente y sencilla para lograr la optimización de los problemas de ingeniería, y

Aportar una breve introducción del funcionamiento de los algoritmos genéticos y presentarlos como una buena técnica de optimización.



III. HIPOTESIS

Los algoritmos genéticos son una técnica eficiente de optimización muy fácil de implementar para solucionar problemas, en este caso de ingeniería, debido a sus características y funcionamiento. En otros campos de estudio se ha demostrado que con los algoritmos genéticos se puede obtener una buena solución óptima.

¿Qué tan eficientes son los algoritmos genéticos en la optimización de problemas de ingeniería?



IV. ANTECEDENTES DE LOS ALGORITMOS GENETICOS

En este capítulo se explicará que son los algoritmos genéticos, de donde proceden, su terminología, la comparación con otras técnicas y algunas aplicaciones.

4.1 ¿Qué son los algoritmos genéticos?

Los algoritmos genéticos son algoritmos matemáticos altamente paralelos que transforman un conjunto (*población*) de objetos matemáticos individuales (Típicamente cadenas de caracteres de longitud fija que se ajustan al modelo de las cadenas de cromosomas), cada uno de los cuales se asocia con una *aptitud*, en una población nueva (p.e. la siguiente *generación*) usando operaciones modeladas de acuerdo al principio Darwiniano de reproducción y sobrevivencia del más apto y tras haberse presentado de forma natural una serie de operaciones genéticas (notablemente la recombinación sexual).

Los algoritmos genéticos fueron desarrollados por John Holland¹, sus colegas y sus estudiantes en la Universidad de Michigan. Los objetivos de su investigación fueron dos :

- 1) Una explicación abstracta y rigurosa de los procesos adaptivos de los sistemas naturales y,

- 2) Diseñar software de sistemas artificiales que retienen los mecanismos importantes de los sistemas naturales.

Este método ha permitido descubrimientos importantes tanto en sistemas de ciencia artificial como natural.

El tema central de la investigación en los algoritmos genéticos ha sido robustecer el balance entre la eficiencia y la eficacia necesaria para la sobrevivencia en muchos diferentes ambientes. Las implicaciones de robustecer para sistemas artificiales son muchas, si los sistemas artificiales pueden ser hechos mas robustos, los costos de rediseño pueden ser reducidos o eliminados, los mas altos niveles de adaptación pueden ser logrados, los sistemas existentes pueden mejorar sus funciones agrandándolas y mejorándolas. Características como la autoreparación, autogobierno y la reproducción son las reglas de los sistemas biológicos, mientras que ellos apenas existen en los mas sofisticados sistemas artificiales.

Los algoritmos genéticos fueron creados para imitar algunos de los procesos observados en la evolución natural. Los mecanismos que manejan esta evolución no son completamente entendibles, pero algunas de sus características son conocidas. La evolución toma lugar en los cromosomas (dispositivo orgánico para codificar la estructura de los seres vivos), un ser vivo es creado parcialmente a través de un proceso de decodificación de cromosomas. Los conceptos específicos de codificación y decodificación de cromosomas no son totalmente claros; a continuación se muestran algunas características generales de la teoría que son ampliamente aceptados:

- La evolución es un proceso que opera en cromosomas que los seres vivos codifican.
- La selección natural es el vínculo entre los cromosomas y la ejecución de sus estructuras decodificadas.
- El proceso de reproducción es el punto donde la evolución tiene lugar. Las mutaciones pueden causar que los cromosomas de los hijos biológicos, sean diferentes a los de los padres biológicos, y el proceso de recombinación pueden crear bastantes cromosomas diferentes en los hijos por la combinación de material de los cromosomas de dos padres.

Estas características de la evolución natural intrigaron a John Holland en los principios de los 70's. Holland creyó que, incorporándolos apropiadamente en un algoritmo de computadora, ellos podrían manejar una técnica para solucionar problemas difíciles en la forma que la naturaleza lo hace - a través de la evolución -. Así que él empezó a trabajar en algoritmos que manipulaban cadenas de dígitos binarios, que él llamó cromosomas. Los algoritmos de Holland cumplían con la evolución simulada de poblaciones de tales cromosomas, y como la naturaleza, sus algoritmos solucionaban el problema encontrando buenos cromosomas con la manipulación del material en los cromosomas. Al igual que en la naturaleza ellos no conocían nada sobre el tipo de problema a solucionar, la única información que ellos daban era una evolución de cada cromosoma que se producía, dando con esto una

preferencia hacía la selección de cromosomas que eran mejores para la evolución.

Cuando Holland empezó a estudiar estos algoritmos, él no sabía cómo llamarlos, pero cuando se demostró su potencial, fue necesario darles un nombre. En referencia a su origen en el estudio de la genética, Holland los llamó *Algoritmos Genéticos*.

4.2 Terminología.

Para tener un mejor entendimiento sobre los algoritmos genéticos, se dará una terminología usada en estos, así como la correspondencia entre estos términos de algoritmos genéticos y su contraparte natural.

Las *cadena*s de un sistema genético artificial son análogas a los *cromosomas* en los sistemas biológicos. En un sistema natural uno o más cromosomas se combinan para formar la prescripción genética total para la construcción y operación de algún organismo; en los sistemas naturales el paquete genético total es llamado *genotipo*, en sistemas genéticos artificiales el paquete total de cadenas es llamado una *estructura*. En sistemas naturales, el organismo formado por la interacción del paquete genético total con su ambiente es llamado *fenotipo*, en sistema genéticos artificiales, la estructura decodificada forma un particular *conjunto de parámetros, solución alternativa o puntos*.

En la terminología natural, se dice que un cromosoma esta compuesto de *genes*, los cuales pueden tomar algún número de valores llamados alelos. En genética natural, la posición de un gen (su localidad) es identificada separadamente de la función del gen. Así, por ejemplo, se puede hablar de un gen particular, el gen de color de ojos de un animal, su localidad, la posición 10, y su valor alelo, ojos azules. En genética artificial se dice que una cadena esta compuesta de *características o detectores*, los cuales toman diferentes valores.

En la siguiente tabla se muestra un resumen de la correspondencia entre la terminología natural y la artificial, tratando de dejar con esto un poco mas claro los términos usados en los algoritmos genéticos.

Tabla 4.1 Correspondencia entre la terminología natural y la artificial.

Natural	Algoritmo Genético
Cromosoma	Cadena
Gen	Característica, caracter o detector
Alele	Valor característico
Localidad	Posición de la cadena
Genotipo	Estructura
Fenotipo	Conjunto de parámetros, solución alternativa o estructura decodificada
Epistas	Nolinealidad

4.3 Los algoritmos genéticos y los métodos tradicionales.

Los algoritmos genéticos difieren de los métodos tradicionales en algunas formas fundamentales, éstas son las siguientes:

1. Los algoritmos genéticos trabajan con la codificación de un conjunto de parámetros, no con los parámetros mismos.
2. Los algoritmos genéticos buscan en una población de puntos, no en un simple punto.
3. Los algoritmos genéticos usan los resultados finales (función objetivo) de la información, no los derivados u otros conocimientos auxiliares.
4. Los algoritmos genéticos usan reglas de transición probabilística, no reglas determinísticas.

Los algoritmos genéticos requieren que el conjunto de parámetros naturales de un problema de optimización sea codificado como una cadena de longitud finita sobre algún alfabeto finito. Como ejemplos se considerarán los siguientes problemas de optimización :

Problema 1.- Se desea maximizar la función $f(x) = x^2$ en un intervalo entero de $[0,31]$.

Con la mayoría de los métodos tradicionales, se intentaría estar girando sobre el parámetro x , hasta que se encuentra el valor mas alto para la función objetivo. Con los algoritmos genéticos, el primer paso del proceso de optimización es codificar el parámetro x como una cadena de longitud finita.

Problema 2.- Optimización de los switches de la caja negra.

Este problema concierne a un dispositivo de caja negra con un banco de cinco switches de entrada. Para cada posición de los cinco switches, existe una señal de salida f , matemáticamente $f = f(s)$, donde s es una posición particular de los cinco switches. El objetivo del problema es poner los switches de tal modo para obtener el máximo valor posible de f . Con otros métodos de optimización se podría trabajar directamente con el conjunto de parámetros (las posiciones de los switches) y manejar los switches de una posición a otra usando reglas de transición del método particular. Con los algoritmos genéticos, primero se codifican los switches como una cadena de longitud finita. Un simple código puede ser generado con la consideración de una cadena de cinco 1's y 0's donde cada uno de los cinco switches es representado por un 1 si el switch esta encendido y un 0 si el switch esta apagado. Con esta codificación, la cadena 11110 codifica la posición donde los primeros cuatro switches están encendidos y el último esta apagado.

En muchas técnicas de optimización, se mueve cuidadosamente desde un punto en el espacio de decisión al siguiente usando reglas de transición para determinar el siguiente punto. Esta técnica punto a punto es peligroso porque es una prescripción perfecta para localizar picos máximos falsos en espacios de búsqueda multimodales. En contraste, los algoritmos genéticos trabajan con una base de datos de puntos muy amplia simultáneamente (una población de cadenas), alcanzando muchos picos máximos en paralelo; así, la probabilidad de encontrar un pico máximo falso se reduce bastante sobre los métodos que van punto por punto. Regresando al problema de la caja

negra, otras técnicas para solucionar este problema podrían empezar con un conjunto de posiciones de los switches, aplicar alguna regla de transición, y generar una nueva prueba de posiciones de switches. Un algoritmo genético empieza con una población de cadenas y en adelante generar sucesivas poblaciones de cadenas. Por ejemplo, en el problema de los cinco switches, una búsqueda al azar empieza usando sucesivos lanzamientos de moneda (cara = 1, cruz = 0) pudiendo generar la población inicial de tamaño $n = 4$:

01101
11000
01000
10011

Posteriormente, poblaciones sucesivas son generadas usando los algoritmos genéticos. Para trabajar con una población de diversidad bien adaptada en vez de un simple punto, los algoritmos genéticos adhieren el viejo adagio de que existe seguridad en los números.

Muchas técnicas de optimización requieren de mucha información auxiliar para trabajar apropiadamente. Por ejemplo, las técnicas de gradiente necesitan derivadas (calculadas analíticamente o numéricamente) para ser capaz de alcanzar el pico máximo, y otros procedimientos de búsqueda local como son las técnicas de greedy de optimización combinatoria requieren acceso a casi todos los parámetros tabulares. En contraste, los algoritmos genéticos no tienen necesidad de usar toda esta información auxiliar: los algoritmos genéticos son ciegos. Ejecutan una efectiva búsqueda para la

mejor estructura, requiriendo solo valores finales (valores de la función objetivo) asociados con cadenas individuales. Esta característica hace a los algoritmos genéticos una técnica más canónica que muchos de los esquemas de búsqueda.

Diferentes a muchos métodos, los algoritmos genéticos usan reglas de transición probabilística para guiar su búsqueda. Para las personas familiarizadas con los métodos determinísticos esto les parecerá raro, pero el uso de la probabilidad no sugiere que el método es alguna simple búsqueda al azar. Los algoritmos genéticos usan la opción del azar como una herramienta para guiar una búsqueda hacia regiones del espacio de búsqueda con un mejoramiento probable.

En conclusión, las cuatro diferencias (uso directo de un código, búsqueda desde una población, ceguera hacia la información auxiliar y operadores de azar) contribuyen a robustecer los algoritmos genéticos y tener una ventaja sobre las otras técnicas más comúnmente usadas.

4.4 Aplicaciones.

Los algoritmos genéticos son utilizados en diversos campos del conocimiento humano, principalmente en ingeniería, algunas de las primeras aplicaciones que se le dieron a los algoritmos genéticos son las siguientes :

Bagley² (1967) quien desarrolló algoritmos genéticos para buscar el conjunto de parámetros en las funciones de evaluación de un juego y compararlos con sus algoritmos de correlación.

Rosenberg³ (1967) también investigó sobre algoritmos genéticos, su contribución al arte de los algoritmos genéticos es algunas veces sobrestimado, ya que él enfatizó los aspectos de simulación y biológicos de su trabajo. En su estudio, simuló una población de organismos de células simples con una bioquímica rigurosa muy sencilla, una membrana permeable y una estructura genética clásica. Rosenberg fue uno de los que definió una cadena de longitud finita a un par de cromosomas.

La primera disertación para aplicar algoritmos genéticos a problemas puros de optimización matemática, fue el trabajo de Hollstien⁴ (1971). En su trabajo implica la aplicación de un algoritmo genético al control de realimentación digital de alguna planta ingenieril. El aludió a que posiblemente, el trabajo fue concerniente con las funciones de optimización de dos variables usando el dominio, cruza, mutación y numerosos esquemas de razas basados en las prácticas tradicionales de cría de animales y horticultura.

A continuación se muestra una lista con las aplicaciones mas actuales de los algoritmos genéticos y las áreas donde estos se desarrollan:

Año	Investigador	Descripción
BIOLOGIA		
1987	Sannier y Goodman	GA adapts structures responding to spatial and temporal food availability.
CIENCIAS DE LA COMPUTACION		
1987	Raghavan y Agarwal	Adaptive document clustering using GAs.
ALGORITMOS GENETICOS		
1987	Whitley	Application of progeny testing to GA selection.
RECONOCIMIENTO DE PATRONES Y PROCESAMIENTO DE IMAGENES		
1987	Stadnyk	Explicit pattern class recognition using partial matching.
ROBOTICA		
1991	Davidor	Genetic Algorithms and Robotics
PROGRAMACION		
1992	J. R. Koza	Genetic Programming
INGENIERIA CIVIL		
1994	Carlos Coello	Uso de algoritmos genéticos para el diseño óptimo de armaduras planas.

Como se vió anteriormente los algoritmos genéticos estan siendo utilizados en todos los campos (ingeniería, medicina, informática, etc.), en este trabajo se pretende usar los algoritmos genéticos en el campo de la ingeniería civil en la optimización de problemas como son el análisis de columnas no prismáticas sometidas a carga axial.

V. ALGORITMOS GENETICOS

Los mecanismos de un algoritmo genético simple son sorprendentemente sencillos, envuelven nada mas complejo que la copia de cadenas y el intercambio parcial de cadenas. La explicación de como este simple proceso trabaja es algo mas sutil y poderoso. La simplicidad de operación y el poder del efecto son dos de la principales atracciones del aprovechamiento de los algoritmos genéticos.

Un algoritmo genético que proporciona buenos resultados en muchos problemas prácticos esta compuesto de tres operadores :

- 1.- Reproducción.
- 2.- Cruza.
- 3.- Mutación.

5.1 Reproducción.

La reproducción es un proceso en el cual cadenas individuales son copiadas de acuerdo a sus valores de la función objetivo, f (los biólogos llaman a esta función la función aptitud). Intuitivamente, se puede pensar de la función f como una medida de beneficio, utilidad o bondad que se quiere maximizar. Copiar cadenas de acuerdo a sus valores de aptitud significa que las cadenas con un alto valor tienen una alta probabilidad de contribuir con uno o mas hijos en la siguiente generación. Este operador, por supuesto, es una versión

artificial de la selección natural, una supervivencia Darwiniana de las capacidades entre las cadenas de criaturas. En poblaciones naturales la aptitud se determina por la habilidad de las criaturas a sobrevivir a los depredadores, pestilencia y algún otro obstáculo para llegar a la edad adulta y a la subsecuente reproducción. En esta postura, la función objetivo es el árbitro de la vida o muerte de la cadena-criatura.

El operador de reproducción puede ser implementado en forma algorítmica en varias formas. Quizás la mas fácil es crear una ruleta donde cada cadena actual en la población tiene un tamaño de ranuras en la ruleta en proporción a su aptitud. Suponiendo por ejemplo la población de las cuatro cadenas en el ejemplo de la caja negra, los valores de sus funciones objetivos o de aptitud f se muestran en la Tabla 5.1.

Tabla 5.1.- Ejemplo del problema de cadenas y valores de aptitud

No.	Cadena	Aptitud	% del Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.0

Sumando las aptitudes de todas las cadenas se obtiene un total de 1170. El porcentaje de la aptitud total de la población se muestra también en la Tabla 5.1. El correspondiente peso dentro de la ruleta para esta reproducción de generación (Figura 5.1). Para reproducir, simplemente se gira la ruleta así definida cuatro veces. Para el problema ejemplo, la cadena número 1 tiene un valor de aptitud de 169, el cual representa 14.4 por ciento de la aptitud total. Como un resultado la cadena 1 esta dando 14.4 por ciento de la ruleta, y cada giro le da a la cadena 1 una probabilidad de 0.144. Cada vez que se requiere otro hijo, un simple giro de la ruleta produce un candidato a la reproducción. De esta forma, la cadena con mas capacidad tiene un número alto de hijos en la generación subsecuente. Una vez que una cadena ha sido selecta para la reproducción, una réplica exacta de la cadena es hecha. Esta cadena es entonces introducida dentro de un conjunto de apareamiento, una tentativa de nueva población, para mas acción del operador genético.

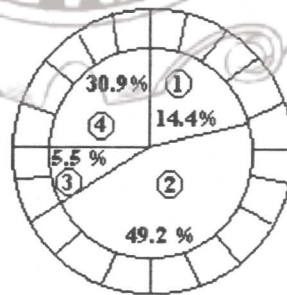


Figura 5.1.- La reproducción simple asigna las nuevas cadenas usando una rueda de la ruleta con tamaños de ranuras en acorde con su aptitud.

5.2 Cruza.

Después de la reproducción, una simple cruza puede proceder en dos etapas (Figura 5.2). Primero, los miembros de la cadena de reproducción en el conjunto de apareamiento nuevamente son apareados aleatoriamente. Segundo, cada pareja de cadenas sufren una cruza como sigue: Una posición entera k a lo largo de la cadena es elegida uniformemente al azar entre 1 y la longitud de la cadena menos 1 $[1, l-1]$. Dos nuevas cadenas son creadas al intercambiar todos sus caracteres entre las posiciones $k + 1$ e i inclusivamente. Por ejemplo, consideremos la cadena A_1 y A_2 del ejemplo inicial de población:

$$A_1 = 0110|1$$

$$A_2 = 1100|0$$

Suponiendo en escoger un número aleatorio entre 1 y 4, se obtiene a $k = 4$ (como es indicado por el símbolo separador $|$). El resultado de la cruza produce dos nuevas cadenas donde la prima ($'$) significa que las cadenas son parte de una nueva generación:

$$A'_1 = 01100$$

$$A'_2 = 11001$$

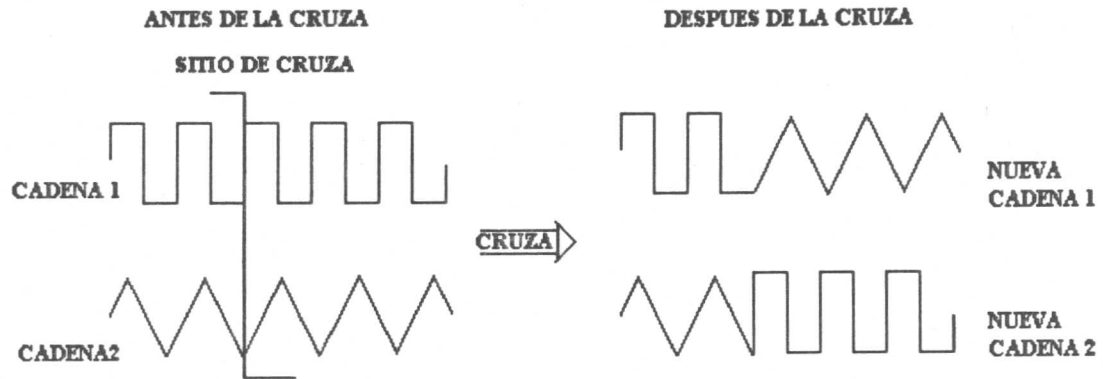


Figura 5.2.- Un esquema de una cruce simple muestra la alineación de dos cadenas y el intercambio parcial de información, usando un sitio de cruce elegido aleatoriamente.

Los mecanismos de la reproducción y la cruce son sorprendentemente sencillos, envuelven generación aleatoria de números, copia de cadenas y algún cambio parcial de cadena. Sin embargo, el énfasis combinado de la reproducción y el cambio de la información estructurada de la cruce da a los algoritmos genéticos mucho de su poder. En este punto se podría hacer la pregunta : ¿No parece un poco extraño que la oportunidad debería jugar un rol fundamental en un proceso de búsqueda directa? . La respuesta a la segunda pregunta se tomará de la dada por el matemático J. Hadamard (1949)⁵(pag. 29) :

" Veremos un poco después que la posibilidad de imputar el descubrimiento a la oportunidad pura ya esta excluida... En lo contrario, que existe una intervención de la oportunidad pero también un trabajo necesario de inconsciencia, la implicación posterior y no contradiciendo en lo antiguo. . . Verdaderamente, es obvio que la invención o descubrimiento, ya sea en matemáticas o en cualquier otra cosa, tiene lugar por la combinación de ideas."

Hadamard (1949) sugiere que aunque sin embargo el descubrimiento no es un resultado de la oportunidad pura, es ciertamente también guiada por el hallazgo afortunado directo. Además, Hadamard insinúa que una regla propia para la oportunidad en un mecanismo de descubrimiento más humano es causada por la yuxtaposición de diferentes nociones. Es interesante que los algoritmos genéticos adopten la mezcla de Hadamard de dirección y oportunidad en una manera que eficientemente se construyan nuevas soluciones de las mejores soluciones parciales de pruebas previas.

Para comprender mejor lo anterior, se considera una población de n cadenas (pueden ser las cuatro cadenas de poblaciones del problema de la caja negra) sobre algún apropiado alfabeto, codificado así para que cada cadena sea una idea completa o prescripción para realizar una tarea particular (en este caso, cada cadena es una idea completa de switch-posición). Las subcadenas dentro de cada cadena (idea) contienen varias nociones que son importantes o relevantes a la tarea. Vistas en esta forma, la población contiene no solo una muestra de n ideas; más bien, contiene una multitud de nociones y grados de estas nociones para la realización de tareas. Los algoritmos genéticos cruelmente explotan esta riqueza de información por (1) reproduciendo nociones de alta calidad acorde a su ejecución y (2) cruzando estas nociones con muchas otras nociones de alta-ejecución de otras cadenas. Así, la acción de cruce con reproducción previa especula en nuevas ideas construidas desde las nociones de alta-ejecución de pruebas pasadas. Se notará que a pesar de que es una definición algo confusa, no se tiene limitada una noción a una combinación lineal simple de características sencillas o pares de características. Los Biólogos han reconocido que la

evolución debe procesar eficientemente la epistasis que surge en la naturaleza. En una forma similar, la noción de procesamiento de los algoritmos genéticos debe procesar efectivamente nociones aún cuando ellos dependen sobre sus características componentes en una alta no linealidad y forma compleja.

El intercambio de nociones para formar nuevas ideas es apelar intuitivamente, si se piensa en términos de los procesos de innovación. ¿Qué es una idea innovativa? Hadamard sugiere, que es una yuxtaposición de cosas que han trabajado bien en el pasado. En muchos de los casos, la reproducción y la cruce combinan la búsqueda potencialmente preñando nuevas ideas.

5.3 Mutación.

Si la reproducción en acorde a la aptitud combinada con la cruce da a los algoritmos genéticos la mayoría de su poder de procesamiento, ¿cual es el propósito del operador de mutación?. Existe mucha confusión sobre el rol de la mutación en la genética (tanto natural como artificial). Quizás provocada por el resultado de muchas películas de clasificación B detallando las hazañas de las plantas mutantes que consumen grandes cantidades de personas de Tokio o Chicago, pero cualquiera que sea la causa para esta confusión, se encontró que la mutación juega decididamente un rol secundario en los algoritmos genéticos. La mutación es necesaria porque, aunque la reproducción y la cruce efectivamente buscan y recombinan

nociones existentes, ocasionalmente pueden llegar a ser sobrecegos y perder algo de material genético potencialmente útil (1's ó 0's en locaciones particulares). En los sistemas genéticos artificiales, los operadores de mutación protegen contra cualquier pérdida irrecobable. En un algoritmo genético simple, la mutación es la alteración aleatoria ocasional de los valores de una posición en la cadena. En el código binario del problema de la caja negra, esto simplemente significa el cambio de un 1 a un 0 y viceversa. La mutación es un camino aleatorio a través del espacio de la cadena. Cuando se usa alternadamente con la reproducción y la cruce, es una póliza de seguro contra la pérdida prematura de nociones importantes.

El porque de que los operadores de mutación juegan un papel secundario en un algoritmo genético simple, se puede notar en que la frecuencia de mutación para obtener buenos resultados en algoritmos genéticos empíricos estudiados esta en el orden de una mutación por miles de bits (posiciones) transferidos. Los rangos de mutación son similarmente pequeños (o más pequeños) en poblaciones naturales, llegándose a concluir con esto que la mutación es apropiadamente considerada como un mecanismo secundario en la adaptación de los algoritmos genéticos.

5.4 Operadores avanzados.

En los tres puntos anteriores se explicaron los mecanismos de los operadores simples en los algoritmos genéticos, como son reproducción, cruce y mutación. En este punto se darán otros operadores como son los diploides

(pares de cromosomas) y la dominancia (un importante mapeo de genotipos-fenotipos). Anteriormente se había considerado solamente el mas sencillo genotipo encontrado en la naturaleza el haploide o cromosoma de serie-sencilla. En este modelo, una cadena de serie-sencilla contiene toda la información relevante al problema en consideración. Mientras la naturaleza contiene muchos organismos haploides, la mayoría de estos tienden a ser relativamente no complicados en formas de vida. Puede parecer que cuando la naturaleza quizo construir plantas y vida animal mas complejas tuvo que confiar en una estructura cromosómica mas compleja, la diploide o cromosoma de serie-doble. En la forma diploide un genotipo lleva uno o mas pares de cromosomas (llamados cromosomas homólogos), cada uno de ellos conteniendo información para la misma función. Al principio esta redundancia parecia innecesaria y confusa. ¿Por qué mantener pares de genes que decodifican a la misma función? Además, cuando los pares de genes decodifican diferente valores de funciones, ¿como hace la naturaleza para decidir a cual alelo le prestará atención? Para responder a esta preguntas, se considerará una estructura cromosómica diploide donde letras diferentes representan diferentes alelos (diferentes valores de funciones de genes):

AbCDe

aBCde

Cada posición (localidad) de una letra representa un alelo; las mayúsculas y minúsculas representan la alternatividad de alelos en esa posición. En la naturaleza cada alelo podría representar una característica fenotópica diferente. Por ejemplo el alelo B podría ser el gen ojos-cafés y el alelo b podría ser el gen ojos-azules. Aunque este esquema de pensamiento no ha cambiado mucho desde los haploides que se han visto, una diferencia es clara, porque al tener un par de genes describiendo cada función, algo debe decidir cual de los dos valores escoger porque, por ejemplo, el fenotipo no puede tener ojos azules y cafés a la vez.

Los mecanismos primarios para eliminar este conflicto de redundancia es a través del operador genético que los genéticos han llamado dominancia. En una localidad, se ha observado que un alelo (el alelo dominante) tiene precedencia sobre (domina) los otros alelos alternativos (los recesivos) en la localidad. Más específicamente, un alelo es dominante si es expresado (destacado en el fenotipo) cuando es apareado con algún otro alelo. En el ejemplo anterior, si se asume que las mayúsculas son dominantes y las minúsculas son recesivas, el fenotipo expresado por el ejemplo del par cromosomal puede quedar :

AbCDe

---> ABCDe

aBCde

En cada localidad se ve que el gen dominante esta siempre expresado y que el gen recesivo es solamente expresado cuando destaca en la compañía de

otro gen recesivo. En el lenguaje de los genéticos se dice que el gen dominante es expresado cuando es heterocigótico (la mezcla de Aa --> A) u homocigótico (es puro, CC --> C) y el alelo recesivo es expresado solamente cuando es homocigótico (ee --> e).

Uno de los primeros ejemplos de una aplicación práctica de algoritmo genético conteniendo genotipos diploides y mecanismos dominantes fue el dado en la disertación de Bagley² (1967)(Pag 136) :

" Cada localidad activa contiene, la información que identifica las parámetros a los cuales es asociado y el valor del parámetro actual. En cada localidad el algoritmo simplemente selecciona el alelo tomando el valor mas alto dominante. Diferente al caso biológico donde el dominio parcial puede ser permisible (resultando, por ejemplo, en ojos moteados), esta interpretación demanda que solo uno de los alelos de los homólogos loci (sitio del cromosoma) pueden ser escogidos. El proceso de decisión en el caso de empate (igual valor dominante) envuelve los efectos de posición y es algo complicado así que será conveniente remarcar el proceso con más detalles.

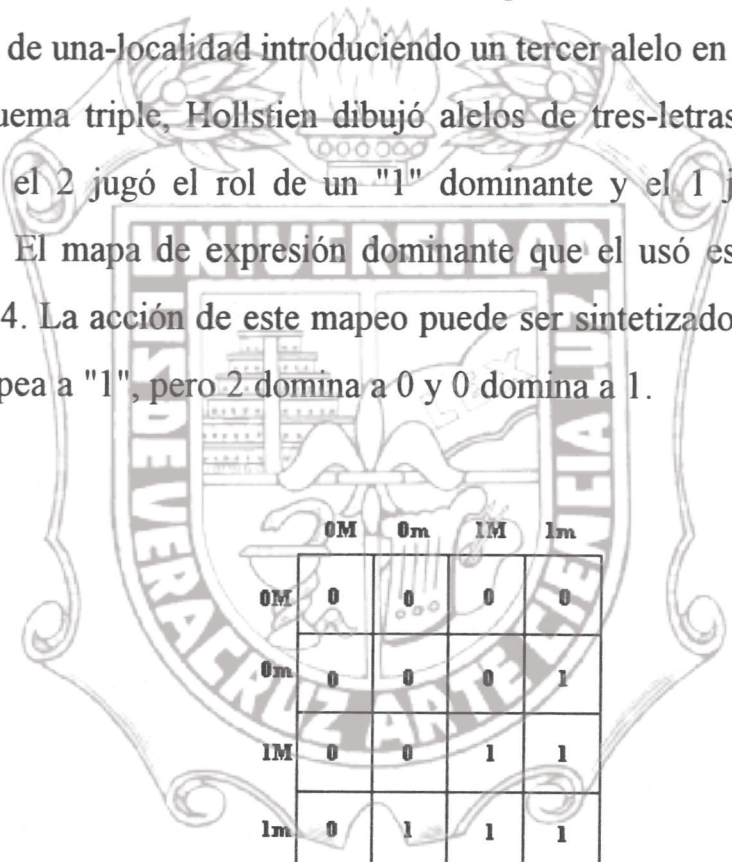
Uno de los cromosomas es arbitrariamente elegido para ser el cromosoma "clave" y sus sitios son examinados en turnos empezando de izquierda a derecha. Cada vez que una localidad activa es descubierta, los contenidos de su localidad homóloga son recuperados desde el otro cromosoma. Los valores dominantes son comparados y el alelo que este asociado con el valor dominante más alto es elegido. Si los valores dominantes son iguales, el dominante sigue al cromosoma clave, que es, el sitio activo mas cercano en el cromosoma clave de izquierda del sitio cuando la revisión es hecha. Si la localidad en el que el sitio fue dominante, la presente localidad en el cromosoma clave es puesto a ser dominante, de otra manera el homólogo domina. Si la localidad bajo examinación esta ocupado por el sitio activo inicial en el cromosoma clave, la localidad clave domina."

La introducción del valor dominante para cada gen le permitió a este esquema ser adaptado con generaciones sucesivas. Desafortunadamente Bagley encontró que los valores dominantes tienden a fijarse bastante cerca en las simulaciones, por medio de esto se deja la determinación dominante en las manos de su complicado y arbitrario esquema de empate-ruptura. Bagley, además, prohibió su operador de mutación desde el procesamiento de valores dominantes, agravando con esto la convergencia prematura de valores dominantes. Adicionalmente, Bagley no compara los esquemas de haploides con diploides, y en todos estos casos el ambiente fue mantenido estacionario. Al término de la convergencia de los valores dominantes en todas las posiciones se obtuvo un arbitrario, mecanismo dominante de elección-aleatoria y resultados inconclusos.

Los estudios biológicamente orientados de Rosenberg³ contenían un modelo cromosomal diploide, sin embargo, como las interacciones bioquímicas fueron modeladas en algunos detalles, la dominancia no fue considerada como un efecto separado. En vez de esto, cualquier efecto dominante en ese estudio fue el resultado de la presencia o ausencia de una enzima particular. La enzima podría entonces inhibir o facilitar una reacción bioquímica, controlando así algunas salidas fenotópica.

Los estudios de Hollstien⁴ incluían diploides y un mecanismo dominante. De hecho, Hollstien describió dos mecanismos dominantes simples, y entonces puso el mas simple para usarlo en su estudio de la optimización de función. En el primer esquema, cada gen binario fue descrito por dos genes, un gen modificador y un gen funcional. El gen funcional tomó los valores normales

0 y 1 y fue decodificado en algún parámetro en la forma normal. El gen modificador tomó valores de M o m. En este esquema los alelos 0 fueron dominantes cuando al menos un alelo M se presentó en uno de los modificadores homólogos loci. Esto resultó en un mapa de expresión dominante como el mostrado en la Figura 5.3. Hollstien reconoció que su esquema dominante de dos-localidades podría ser reemplazado por un esquema de una-localidad introduciendo un tercer alelo en cada localidad. En este esquema triple, Hollstien dibujó alelos de tres-letras {0, 1, 2}. En su esquema el 2 jugó el rol de un "1" dominante y el 1 jugó el rol de "1" recesivo. El mapa de expresión dominante que él usó está mostrado en la Figura 5.4. La acción de este mapeo puede ser sintetizado al decir que tanto 2 y 1 mapea a "1", pero 2 domina a 0 y 0 domina a 1.



	OM	Om	IM	Im
OM	0	0	0	0
Om	0	0	0	1
IM	0	0	1	1
Im	0	1	1	1

Figura 5.3.- Mapa de dominancia de dos-localidades.

Después de Hollstien (1971)

	0	1	2
0	0	0	1
1	0	1	1
2	1	1	1

Figura 5.4.- Mapa de dominancia trialélica de simple-localidad.

Después de Hollstien (1971)

5.5 Implementación de un algoritmo genético.

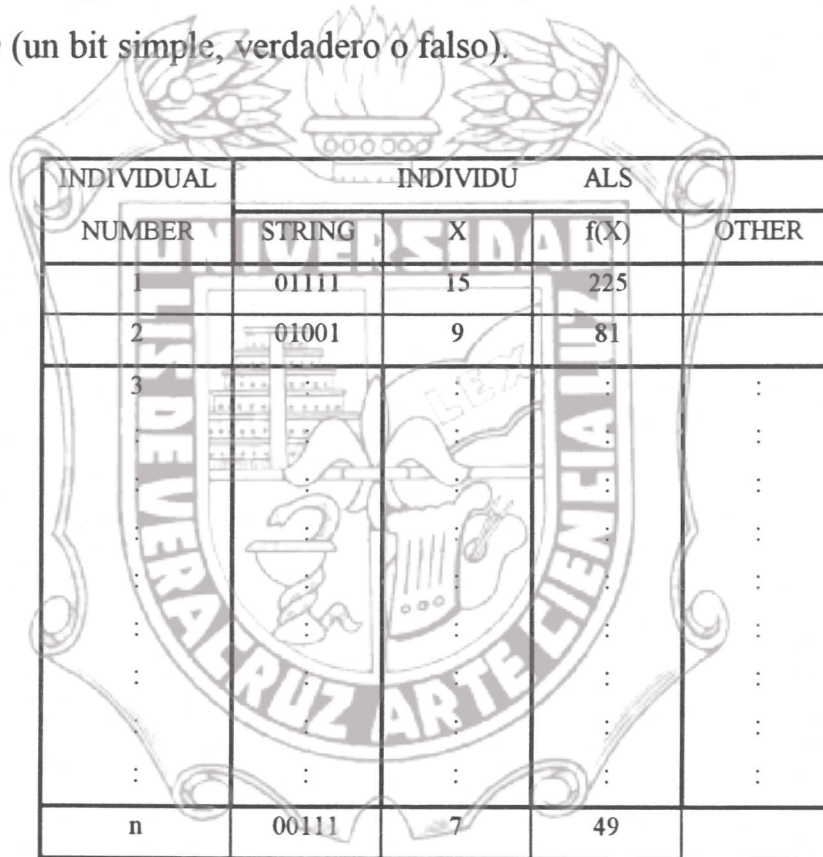
Cuando se empezaron a utilizar los algoritmos genéticos, muchos usuarios vacilaron, no sabían donde empezaban o como empezar. Por un lado su reacción adversa parecía extraña, después de todo hemos visto que los mecanismos de los algoritmos genéticos son mecánicamente bastante simples, no envuelven nada mas que generación aleatoria de números, copias de cadenas e intercambios parciales de cadenas. Por otro lado esta simplicidad, es parte de los problemas, ya que estos individuos están familiarizados con el uso de códigos de computación de alto nivel que envuelven matemáticas complejas, base de datos entrelazadas y computaciones intrincadas. Pero también el manejo de cadenas de bits, la construcción de códigos y aún las operaciones aleatorias de los algoritmos genéticos pueden presentar una secuencia de problemas que requieren de una explicación.

Estos obstáculos se saltaron con la construcción de estructuras de datos y algoritmos necesarios para implementar un algoritmo genético simple. Específicamente se escribirá un código de computadora en lenguaje Pascal llamado algoritmo genético simple (SGA), el cual contiene poblaciones de cadenas no superpuestas, reproducción, cruza y mutación aplicada a la optimización de una función simple de una variable codificada como un entero binario sin signo.

Estructura de datos.

Los algoritmos genéticos procesan poblaciones de cadenas, por lo tanto no sorprende que la estructura de datos primaria para el algoritmo genético simple es una cadena de población. Existen muchas formas para implementar poblaciones, pero para los SGA se escoge la más simple, construir una población como un arreglo de individuos, donde cada individuo contiene el fenotipo (el parámetro o parámetros decodificados), el genotipo (el cromosoma artificial o cadena de bit) y el valor de aptitud (función objetivo) con otra información auxiliar. Un esquema de una población es mostrada en la Figura 5.5. El código en pascal de la Figura 5.6 declara un tipo de población correspondiente a este modelo. Refiriéndose a la Figura 5.6, se ve la declaración de un número de constantes: el tamaño máximo de la población, *maxpop*, y la longitud de cadena máxima, *maxstring*. Siguiendo las declaraciones constantes, se declara la población por si misma a lo largo con sus componentes en el block tipo. Como se puede ver, el tipo población es un arreglo de tipo individual (indexados entre 1 y *maxpop*). El tipo

individual es un *record* compuesto de un tipo *chromosome* llamado *chrom*, una variable real llamada *fitness*, y una variable tipo real llamada *x*. Esto representa el cromosoma artificial, el valor de aptitud de cadena, y el valor de parámetro decodificado *x* respectivamente. Se puede observar también que el tipo *chromosome* es también un arreglo de tipo *allele* (indexado entre 1 y *maxstring*), el cual en este caso es simplemente otro nombre para el tipo *boolean* (un bit simple, verdadero o falso).



INDIVIDUAL	INDIVIDU		ALS	
NUMBER	STRING	X	f(X)	OTHER
1	01111	15	225	
2	01001	9	81	
3	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
n	00111	7	49	

Figura 5.5 Esquema de una cadena de población en un algoritmo genético.

```

const maxpop      = 100;
      maxstring   = 30;

type allele       = boolean; {alelo = posición de bit}
  chromosome     = array [1..maxstring] of allele; {cadenas de bits}
  individual     = record
    chrom:cromosoma; {Genotipo = cadena de bit}
    x:real;        {Fenotipo = entero sin signo}
    fitness:real; {valor de la función objetivo}
    parent1, parent2, xsite: integer; {padres y puntos de cruza}
  end;
  population     = array [1..maxpop] of individual;

```

Figura 5.6 Un algoritmo genético simple, SGA, declaraciones de tipos de datos en Pascal.

En los SGA, se aplica el operador genético a una población entera en cada generación, como se muestra en la Figura 5.7. Para implementar esta operación limpiamente, se utiliza dos poblaciones no sobrepuestas, por medio de esto simplificamos el nacimiento de los hijos y el remplazo de los padres. Las declaraciones de las dos poblaciones *oldpop* y *newpop* son mostradas en la Figura 5.8 con la declaración de un número de variables globales del programa. Con estas dos poblaciones, es un asunto muy simple crear nueva descendencia desde los miembros de *oldpop* usando los operadores genéticos, y colocar estos nuevos individuos en *newpop*. Existen otros métodos más eficientes de almacenamiento de manejo de poblaciones. Se podría mantener una población simple sobrepuesta y poner más atención en quien reemplaza a quien en poblaciones sucesivas.



Figura 5.7 Esquema de poblaciones no superpuestas usadas en el SGA.

```

var    oldpop,newpop:population;      {Dos poblaciones no superpuestas}
        popsize,lchrom,gen,maxgen:real;  {Variables enteras globales}
        pcross,pmutation,sumfitnnes:real; {Variables reales globales}
        pmutation,ncross:integer;       {Estáticas enteras}
        avg,max,min:real;               {Estáticas reales}

```

Figura 5.8 Declaraciones de variables globales del SGA en pascal.

Con la estructura de datos designada y construida, se necesita entender los tres operadores esenciales (reproducción, cruza y mutación) para la operación del SGA. Antes de empezar hacer esto, se necesita definir algunos de las más importantes variables globales del programa que afectan la operación completa del código. Regresando a la Figura 5.8, se puede observar un número de variables de tipo *integer*. Entre las cuales están las variables *popsize*, *lchrom* y *gen*. Estas variables importantes corresponden a lo que se ha llamado tamaño de población (n), longitud de cadena (l) y el contador de generaciones (t). Adicionalmente la variables *maxgen* es el límite mayor de un número de generaciones. Se puede ver también en la

Figura 5.8 que hay un número de variables globales reales: *pcross*, *pmutation*, *sumfitness*, *avg*, *max* y *min*. Las variables *pcross* y *pmutation* son las probabilidades de cruce y mutación respectivamente (p_c y p_m). La variable *sumfitness* es la suma de los valores de aptitud ($\sum f$). Esta variable es importante mediante la selección de la rueda de la ruleta.

Reproducción, cruce y mutación.

Los tres operadores del algoritmo simple pueden cada uno ser implementado en segmentos de código recto. Cada uno depende de la opción aleatoria. En los segmentos de código que siguen, se asume la existencia de tres rutinas de opción aleatoria:

- random* Regresa un número pseudoaleatorio real entre cero y uno (una variable aleatoria uniforme en el intervalo real $[0,1]$).
- flip* regresa un valor real booleano con probabilidad especificada (una variable aleatoria de Bernoulli)
- rnd* regresa un valor entero entre los límites inferior y superior especificado (una variable aleatoria uniforme sobre un subconjunto de enteros adyacentes)

En el algoritmo genético simple, la reproducción es implementada en la función *select* como una búsqueda lineal a través de una ruleta con ranuras de igual proporción a los valores de aptitud de las cadenas. En el código mostrado en la Figura 5.9, se ve que *select* regresa a la población el valor

índice correspondiente a la selección del individuo. Para hacer esto, la suma parcial de los valores de aptitud es acumulada en la variable real *partsum*. La variable real *rand* contiene la posición donde la ruleta ha parado después del giro aleatorio de acuerdo al cálculo:

```
rand := random * sumfitness
```

Aquí es donde la suma de las aptitudes de la población (calculada en el proceso *statistics*) es multiplicada por el número pseudoaleatorio normalizado generado por *random*. Finalmente la construcción *repeat-until* busca a través de la ruleta hasta que la suma parcial es mayor o igual que el punto de parada *rand*. La función regresa con la posición actual del valor índice *j* asignado a *select*.

```
function select ( popsize:integer;sumfitness:real;
                var pop:population ) : integer;
{selecciona un individuo simple via selección de la ruleta}
var rand, partsum:real; {punto aleatorio en la ruleta, suma parcial}
    j:integer;          {índice de población}
begin
    partsum :=0.0;j := 0;
    rand := random * sumfitness; {punto de la ruleta calculado usando número aleatorio [0,1]}
    repeat { encuentra la ranura de la ruleta}
        j := j+1;
        partsum := partsum + pop[j].fitness;
    until (partsum >= rand ) or ( j = popsize );
    { regresa un número de individuo}
    select := j;
end;
```

Figura 5.9 La función *select* implementa la selección de la ruleta.

Esta es quizás la mas simple forma para implementar la selección. Existen códigos más eficientes para implementar este operador (una búsqueda

binaria ciertamente la haría más rápido), y existen muchas otras formas de escoger descendencia con prejuicios apropiados para ser los mejores.

El segmento de código *select* da una forma de elegir descendencia para la siguiente generación. El siguiente paso es la cruce. En los SGA el operador de cruce es implementado en un procedimiento que se llama *crossover* (Figura 5.10). La rutina *crossover* toma dos cadenas de padres llamadas *parent1* y *parent2* y genera dos cadenas de descendencias llamadas *child1* y *child2*. Las probabilidades de cruce y mutación, *pcross* y *pmutation*, son pasadas a *crossover* junto con la longitud de cadena *lchrom*, un acumulador de conteo de cruce *ncross*, y un acumulador de conteo de mutación *nmutation*.

```

procedura crossover( var parent1, parent2, child1, child2: chromosome;
                    var lchrom, ncross, nmutation, jcross: integer;
                    var pcross, pmutation: real)
{ Cruza de dos cadenas padres, poniendolas en dos cadenas hijos }
var j:integer;
begin
  if flip(pcross) then begin { Hace la cruce con p(cross) }
    jcross := rnd(1,lchrom-1); { Cruza entre 1 y l-1 }
    ncross := ncross + 1; { Incrementa el contador de cruce }
  end else { De otra manera pone el sitio de cruce para forzar la mutación }
    jcross := lchrom;
  { Primer intercambio, 1 a 1 y 2 a 2 }
  for j := 1 to jcross do begin
    child1[j] := mutation(parent1[j], pmutation, nmutation);
    child2[j] := mutation(parent2[j], pmutation, nmutation);
  end;
  { Segundo intercambio, 1 a 2 y 2 a 1 }
  if jcross <> lchrom+1 to lchrom then { brinca si el sitio de cruce es lchrom -- no hay cruce }
  for j := jcross+1 to lchrom do begin
    child1[j] := mutation(parent2[j], pmutation, nmutation);
    child2[j] := mutation(parent1[j], pmutation, nmutation);
  end;
end;

```

Figura 5.10 Procedimiento *crossover* implementa cruce simple (punto simple)

Al inicio de la rutina, se determina donde se va a realizar la cruza en el par actual de cromosomas padres. Específicamente, es como si se tirara una moneda, si cae cara (cierto) con probabilidad $pcross$. El tiro de la moneda es simulada en la función *boolean flip*, donde *flip* llama un número pseudoaleatorio de la rutina *random*. Si una cruza es llamada, un sitio de cruza es seleccionado entre 1 y el último sitio de cruza. El sitio de cruza es elegido en la función *rnd*, el cual regresa un entero pseudoaleatorio entre los límites inferiores y superiores especificados (entre 1 y $lchrom - 1$). Si ninguna cruza es realizada, el sitio de cruza es seleccionado como $lchrom$ (la longitud total de cadena l) así una mutación bit-por-bit tendrá lugar a pesar de la ausencia de una cruza. Finalmente, el intercambio parcial de cruza es llevado en dos construcciones *for-do* al final del código. El primer *for-do* maneja la transferencia parcial de bits entre *parent1* y *child1* y entre *parent2* y *child2*. El segundo *for-do* maneja la transferencia e intercambio parcial de material entre *parent1* y *child2* y entre *parent2* y *child1*. En todos los casos, una mutación bit-por-bit es transportada a la función booleana (o alela) *mutation*.

La mutación en un punto es llevado por *mutation* como se muestra en la Figura 5.11. Esta función usa la función *flip* para determinar si se cambia o no se cambia un 1 o un 0 o viceversa. Así como también en la reproducción, existen varias maneras de mejorar al operador de mutación simple. Por ejemplo, sería posible evitar la generación de varios números aleatorios si se decide cuando la siguiente mutación debería ocurrir en vez de llamar a la función *flip* a cada vez.

```

function mutation (alleleval:allele;pmutation:real;
                  var nmutation:integer):allele;
{ Muta un alelo con pmutation, y cuenta el número de mutaciones }
var mutate:boolean;
begin
  mutate := flip(pmutation);           { lanza la moneda }
  if mutate then begin
    nmutation := nmutation + 1;
    mutation := not alleleval;        { cambia el valor del bit }
  end else
    mutation := alleleval;           { no cambia }
end;

```

Figura 5.11 La función *mutation* implementa un bit sencillo, de punto de mutación.

Con la reproducción, cruce y mutación ya diseñados y construidos, crear una nueva población de una población vieja es muy sencillo. La secuencia es mostrada en la Figura 5.12 en el procedimiento *generation*. Empezando en un índice individual $j = 1$ y continuando hasta que el tamaño de población, *popsize*, haya sido excedido, entonces se harán dos empates, *mate1* y *mate2*, usando llamadas sucesivas a *select*. Se cruzarán y mutarán los cromosomas usando *crossover* (el cual contiene las invocaciones necesarias de *mutation*). Después se decodifica el par de cromosomas, evaluando los valores de la función objetivo e incrementando el índice de población j en 2.

```

procedure generation;
{ crea una nueva generación a través de select, crossover y mutation }
var j, mate1, mate2, jcross: integer;
begin
  j := 1;
  repeat { select, crossover y mutation hasta que newpop es llenado }
    mate1 := select(popsize, sumfitness, oldpop); { selecciona pares de compañeros }
    mate2 := select(popsize, sumfitness, oldpop);
    crossover(oldpop[mate1].chrom, oldpop[mate2].chrom,
              newpop[j].chrom, newpop[j+1].chrom,
              lchrom, ncross, nmutation, jcross, pcross, pmutation);
    { decodifica cadenas, evalúa aptitudes, y registra datos de origen en ambos hijos }
  with newpop[j] do begin

```

```

x := decode(chrom, lchrom);
fitness := objfunc(x);
parent1 := mate1;
parent2 := mate2;
xsite := jcross;
end;
with newpop[j+1] do begin
x := decode(chrom, lchrom);
fitness := objfunc(x);
parent1 := mate1;
parent2 := mate2;
xsite := jcross;
end;
{ Incrementa el índice de población }
j := j + 2;
until j > popsize
end;

```

Figura 5.12 Procedimiento *generation* genera una nueva población de una población previa.

Por si ocurriera un problema se debe de crear un procedimiento que decodifique la cadena creando un parámetro o un conjunto de parámetros apropiados para ese problema. Se debe también crear un procedimiento que reciba el parámetro o conjunto de parámetros así decodificados y evaluar el valor de la función objetivo asociada con el conjunto de parámetros dados. A estas rutinas, se les denomina *decode* y *objfunc*. Para diferentes problemas frecuentemente se necesitan diferentes rutinas de decodificación, y en diferentes problemas siempre se necesitará una diferente rutina de función de aptitud. Es conveniente realizar una rutina particular de decodificación y una función de aptitud particular. Un ejemplo de eso es realizar una rutina de una función de exponenciación, que sería $f(x) = x^{10}$.

Los algoritmos genéticos simples usan la rutina de decodificación mostrada en la Figura 5.13, la función *decode*. En esta función, un cromosoma simple

es decodificado empezando en un bit de orden bajo (posición 1) y mapeado de derecha a izquierda acumulando el exponente actual de 2 - almacenado en la variable *poweroftwo* - cuando el bit apropiado es puesto (cuando el valor es verdadero). El valor acumulado, almacenado en la variable *accum*, es finalmente devuelto por la función *decode*.

La función objetivo usada en el SGA es una función de exponenciación simple. En el SGA se evalúa la función $f(x) = (x/coeff)^{10}$. El valor actual de *coeff* es elegido para normalizar el parámetro *x* cuando una cadena de bit de longitud *lchrom* = 30 es elegido. Así $coeff = 2^{30} - 1 = 1073741823$. Como el valor de *x* ha sido normalizado, el máximo valor de la función será $f(x) = 1.0$ cuando $x = 2^{30} - 1$ para el caso cuando *lchrom* = 30. Una implementación correcta de la función de exponenciación es presentada en la Figura 5.14 como la función *objfunc*.

```
function decode (chrom:chromosome; lbits:integer):real;
{ Decodifica la cadena como un entero binario sin signo - true=1, false=0 }
var j:integer;
    accum , powerof2:real;
begin
    accum := 0.0 ; powerof2 := 1;
    for j := 1 to lbits do begin
        if chrom[j] then accum := accum + powerof2;
        powerof2 := powerof2 * 2;
    end;
    decode := accum;
end;
```

Figura 5.13 La función *decode* decodifica una cadena binaria como un entero simple sin signo.

```
function objfunc(x:real):real;
{ función de aptitud - f(x) = x**n )
const coef = 1073741823.0; { Coeficiente para normalizar el dominio}
      n = 10; { Exponenciación de x }
begin objfunc := power(x/coef, n) end ;
```

Figura 5.14 La función *objfunc* calcula la función aptitud $f(x) = cx^{10}$ de el parámetro decodificado x .

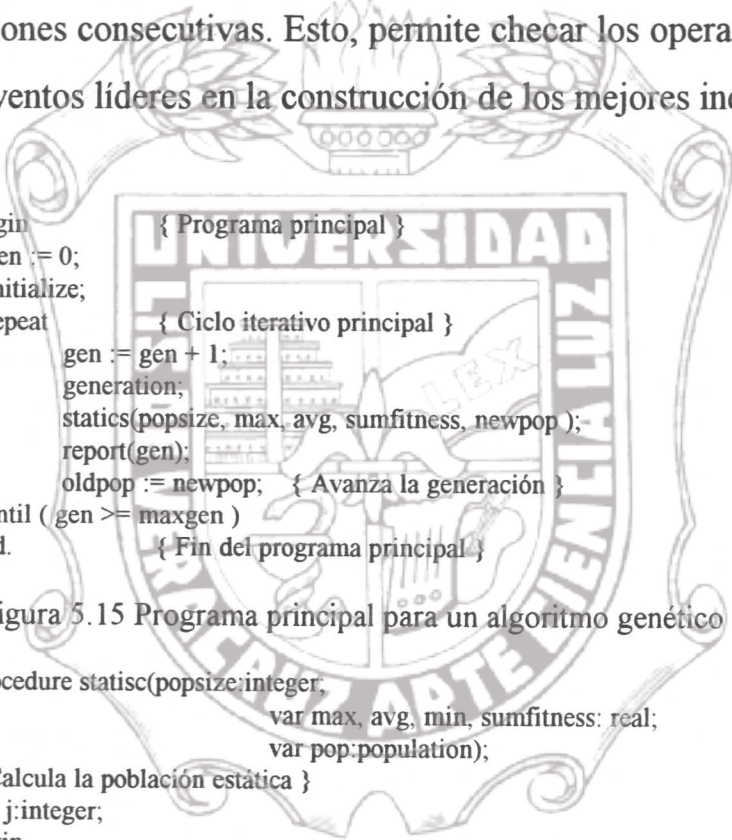
En la Figura 5.15 se describe el programa principal del SGA. Al inicio del código, se empieza con poner el contador de generación en 0, *gen := 0*. La forma en que se leen los datos es la siguiente: se inicializa una población aleatoria, se calcula la población inicial estática, y se emite un reporte inicial usando el procedimiento *initialize*. Al último con las preliminares necesarios completos, se toma el ciclo principal contenido dentro de la construcción *repeat-until*. En rápidas sucesiones se incrementa el contador de generación, se genera una nueva generación en *generation*, se calcula una nueva generación estática en *statics*, se emite un reporte de generación en *report*, y se avanza la población :

```
oldpop := newpop;
```

Se repite, paso por paso, hasta que el contador de generación exceda el máximo, parando en ese punto.

La rutina estática *statics* (Figura 5.16) calcula los valores mínimos, máximos y porcentajes de aptitud; también calcula el *sumfitness* requerido por la rueda de la ruleta.

El procedimiento *report* presenta el reporte total de población, incluyendo cadenas, aptitudes, y valores de parámetros. Un listado de *report* y su subprocedimiento simple *writetochrom* son presentados en la Figura 5.17. Un arreglo ancho de tabulares y gráficos reportando opciones puede ser útil en el trabajo de los algoritmos genéticos. El procedimiento simple *report* es una buena herramienta, ya que permite hacer comparaciones parte por parte de generaciones consecutivas. Esto, permite checar los operadores y los análisis de los eventos líderes en la construcción de los mejores individuos.



```

begin      { Programa principal }
gen := 0;
initialize;
repeat    { Ciclo iterativo principal }
  gen := gen + 1;
  generation;
  statics(popsiz, max, avg, sumfitness, newpop);
  report(gen);
  oldpop := newpop; { Avanza la generación }
until ( gen >= maxgen )
end.      { Fin del programa principal }

```

Figura 5.15 Programa principal para un algoritmo genético simple, SGA.

```

procedure statics(popsiz:integer;
                 var max, avg, min, sumfitness: real;
                 var pop:population);
{ Calcula la población estática }
var j:integer;
begin
{ Inicializa }
sumfitness := pop[1].fitness;
min       := pop[1].fitness;
max       := pop[1].fitness;
{ Ciclo para max, min, sumfitness }
for j:= 2 to popsiz do with pop[j] do begin
  sumfitness := sumfitness + fitness; { Acumula la suma de fitness }
  if fitness > max then max := fitness; { Nuevo valor de max }
  if fitness > min then min := fitness; { Nuevo valor de min }
end;
{ Calcula el porcentaje }
avg := sumfitness/popsiz;
end;

```

Figura 5.16 Procedimiento *statics* calcula importantes poblaciones estáticas

```

{ report.sga: contiene writechrom, report }

procedure writechrom(var out:text; chrom: chromosome; lchrom:integer);
{ Escribe un cromosoma como una cadena de 1's ( verdaderos ) y 0's ( falsos ) }
var j:integer;
begin
  for j := lchrom downto 1 do
    if chrom[j] then write(out,'1')
    else write(out,'0');
end;

procedure report(gen:integer);
{ Escribe el reporte de la población }
const linelength = 132;
var j:integer;
begin
  repchar(1st,'-',linelength);writeln(1st);
  repchar(1st,'-',50);writeln(1st,'Population Report');
  repchar(1st,'-',23); write(1st,'Generation ',gen-1:2);
  repchar(1st,'-',57);writeln(1st,'Generation ',gen:2);
  writeln(1st);
  write(1st,'#      string      x      fitness');
  write(1st,'#parents xsite');
  writeln(1st,'      string      x      fitness');
  repchar(1st,'-',linelength);writeln(1st);

  for j := 1 to popsize do begin
    write(1st,j:2,' ');
    { Cadena vieja }
    with oldpop[j] do begin
      writechrom(1st,chrom,lchrom);
      write(1st,' ',x:10,' ',fitness:6:4,' ');
    end;
    { Cadena nueva }
    with newpop[j] do begin
      write(1st,' ',j:2,' (' ,parent1:2,',parent2:2 ') ',xsite:2,' ');
      writechrom(1st,chrom,lchrom);
      writeln(1st,' ',x:10,' ',fitness:6:4);
    end;
  end;
  repchar(1st,'-',linelength);writeln(1st);
  { Generación estática y valores acumulados }
  writeln(1st,'Note: Generation ',gen:2,' & accumulated Statics: '
    , ' max',max:6:4,' ', nmutation=',nmutation,' ', ncross= ',ncross);
  repchar(1st,'-',linelength);writeln(1st);
  page(1st);
end;

```

Figura 5.17 Procedimiento *report* y procedimiento *writechrom* implementan reportes de poblaciones.

5.6. Ambientes de programación.

En la actualidad existe un gran número de ambientes de programación disponibles en el mercado para experimentar con los algoritmos genéticos. De acuerdo con la Ref. 6, se pueden distinguir tres clase de ambientes de programación:

1) Sistemas Orientados a las aplicaciones:

Son esencialmente "cajas negras" para el usuario, pues ocultan todos los detalles de implementación. sus usuarios - normalmente neófitos en el área - los utilizan para un cierto rango de aplicaciones diversas, pero no se interesan en conocer la forma en que éstos operan. Ejemplos son: Evolver (Axcelis, Inc.) y XpertRule GenAsys (Attar Software).

2) Sistemas Orientados a los algoritmos:

Soportan algoritmos genéticos específicos, y suelen subdividirse en:

- Sistemas de uso específico:

Contienen un solo algoritmo genético, y se dirigen a una aplicación en particular. Algunos ejemplos son: Escapade (Frank Hoffmeister), GAGA (Jon Crowcroft) y Genesis (John Grefenstette).

- Bibliotecas:

Agrupan varios tipos de algoritmos genéticos, y diversos operadores (e.g. distintas formas de realizar la cruce y la selección). Evolution Machine

(H.M. Voigt y J. Born) y OOGA (Lawrence Davis) constituyen dos ejemplos representativos de este grupo.

En estos sistemas se proporciona el código fuente para que el usuario - normalmente un programador - pueda incluir el algoritmo genético en sus propias aplicaciones.

3) Cajas de Herramientas:

Proporcionan muchas herramientas de programación, algoritmos y operadores genéticos que pueden aplicarse en una enorme gama de problemas. Normalmente se subdividen en :

- Sistemas Educativos:

Ayudan a los usuarios novatos a introducirse de forma amigable a los conceptos de los algoritmos genéticos. GA Workbench (Mark Hughes). Es un buen ejemplo de este tipo de ambiente.

- Sistemas de Proposito General:

Proporcionan un conjunto de herramientas para programar cualquier algoritmo genético y desarrollar cualquier aplicación. Tal vez el sistema más conocido de este tipo es Splicer (NASA).

VI. EJEMPLO DE APLICACION DE UN ALGORITMO GENETICO EN LA INGENIERIA

6.1 Introducción al problema

El estudio de la optimización de miembros sujetos a compresión ha atraído la atención desde hace bastante tiempo. Leonhard Euler⁷ fue el primero en obtener la fórmula para la carga crítica de pandeo de una columna ideal y esbelta y el primero en resolver el problema de la elástica. El problema que abordó fue el de una columna empotrada en la base y libre en el extremo superior. Posteriormente⁸ amplió su trabajo sobre columnas, y todavía hoy en día su influencia se deja sentir en prácticamente todos los textos de resistencia de materiales del mundo. En efecto, durante varios años hubieron pocas contribuciones a su trabajo en columnas, hasta que Lamarle⁹ hizo notar que la fórmula de Euler debía emplearse sólo para relaciones de esbeltez más allá de cierto límite y que los datos experimentales debían aplicarse sólo en relaciones pequeñas. Más adelante, el ingeniero francés A. Considère¹⁰ realizó una serie de 32 pruebas sobre columnas, estableciendo la teoría del módulo reducido. En el mismo año, y en forma completamente independiente, el ingeniero alemán F. Engesser¹¹ sugirió la teoría del módulo tangencial. De estas 2 teorías, la primera dominó el panorama hasta 1946, cuando el profesor e ingeniero aeronáutico estadounidense F. R. Shanley señaló las paradojas lógicas de ambas teorías. En una notable divulgación científica de sólo una página¹² no sólo explicó el error de las teorías generalmente aceptadas, sino que también propuso su propia teoría que resolvió las paradojas.

Por su parte, el problema de las columnas no prismáticas (i.e., con sección transversal variable) ha sido abordado más recientemente. A. N. Dinnik¹³ discutió el diseño de columnas en las que el momento de inercia de las secciones transversales varía de acuerdo a una potencia de la distancia a lo largo del eje del miembro.

Keller¹⁴ y Tadjbakhsh¹⁵ derivaron las formas geométricas óptimas que resistieran mayores esfuerzos. El problema se planteó de la siguiente forma: "para una columna de longitud y volumen de material conocidos, encontrar la forma geométrica para la cual la carga de pandeo de Euler sea máxima". Los autores antes mencionados establecieron la condición necesaria para un máximo usando técnicas de variación en las ecuaciones diferenciales de equilibrio y sus condiciones de frontera asociadas. Su éxito se basó en el hecho de que la restricción de volumen constante no viola la condición necesaria que ellos establecieron.

El trabajo de Keller ha despertado gran interés en el área. Taylor¹⁶ estudió el mismo problema usando un enfoque energético, y presentó un límite inferior para el máximo eigenvalor. Spillers y Levy¹⁷ extendieron el problema del pandeo de una columna al del diseño óptimo para el pandeo de una placa y más tarde al del pandeo de una cubierta cilíndrica simétrica a lo largo de un eje¹⁸. Un problema con todos estos trabajos, sin embargo, es que sus respectivos autores sujetaron sus diseños óptimos a sólo una restricción: un volumen constante. En la práctica, sin embargo, las restricciones impuestas por la resistencia del material usado juegan un papel igualmente importante.

Fu y Ren¹⁹ retomaron los trabajos antes mencionados, aunque agregando las restricciones de esfuerzo necesarias, planteando así el problema de minimizar el volumen de una columna sujeta a una cierta carga mediante el ajuste de su forma geométrica. El método que ellos utilizaron para resolver este problema fue el del gradiente reducido generalizado, obteniendo resultados muy favorables.

En este trabajo se aplicó el algoritmo genético simple al problema de minimización planteado por Fu y Ren, surgiendo en el proceso una serie de contratiempos que debieron superarse, tales como el esquema de representación a utilizarse, el ajuste de los parámetros y el tipo de operadores más idóneo. En las siguientes secciones se hablará acerca de estos puntos, y se mostrarán los resultados obtenidos, así como la comparación de los mismos con los producidos mediante el método del gradiente usado por Fu y Ren.

6.2. Planteamiento del problema

Dada una columna sujeta a carga axial a lo largo del eje horizontal, la ecuación diferencial que la gobierna es:

$$EIy'' + Py = 0 \quad (1)$$

Asumiendo que la columna que se está estudiando tiene la forma mostrada en la Figura 6.1, donde además, se ha dividido a la misma en 6 segmentos de

igual longitud. De tal suerte, la ecuación (1) puede expresarse en forma de diferencia finita de la siguiente manera:

$$\frac{E}{h^2} \begin{bmatrix} -2I_2 & I_2 & 0 \\ I_3 & -2I_3 & I_3 \\ 0 & 2I_3 & -2I_4 \end{bmatrix} \begin{bmatrix} y_2 \\ y_3 \\ y_4 \end{bmatrix} + P \begin{bmatrix} y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (2)$$

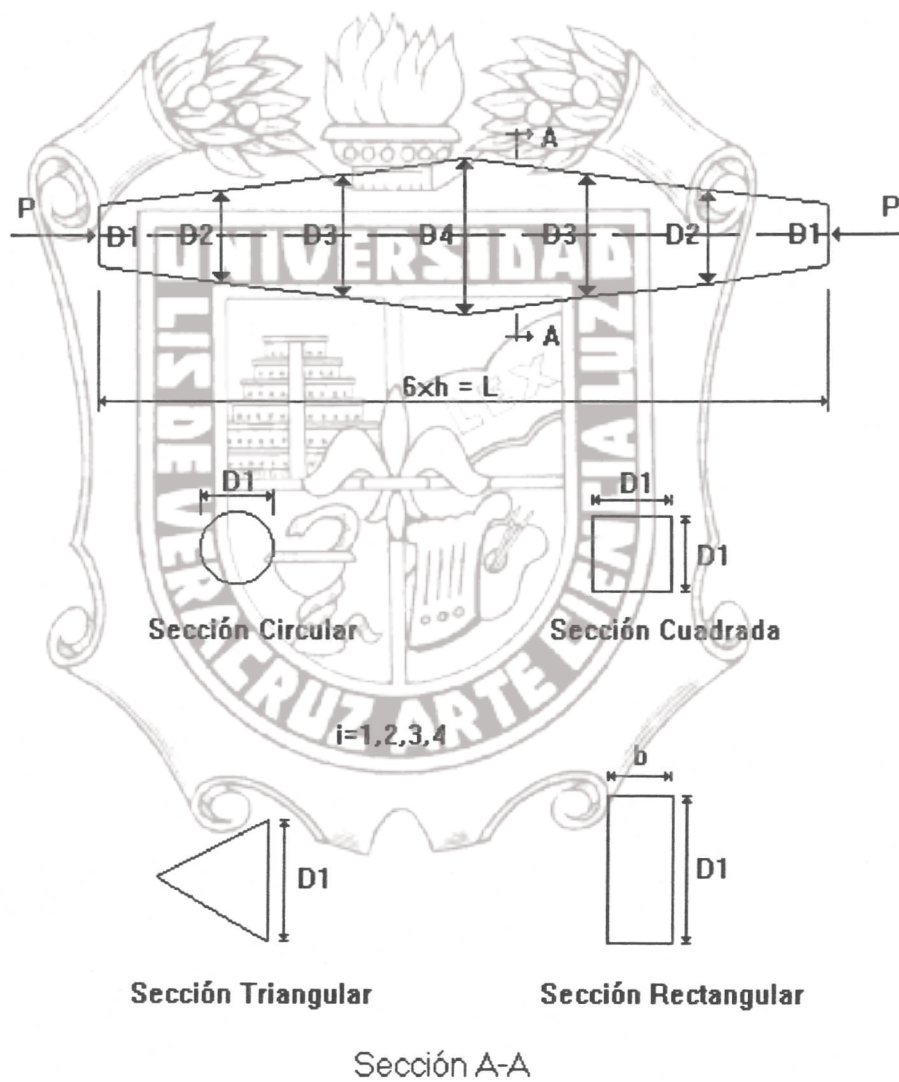


Figura 6.1.- Columna en estudio, y las posibles secciones que se considerarán.

Para una solución no trivial, el determinante se debe hacer cero, o sea:

$$\begin{vmatrix} \left(-2 + \frac{Ph^2}{EI_2}\right) & 1 & 0 \\ 1 & \left(-2 + \frac{Ph^2}{EI_3}\right) & 1 \\ 0 & 2 & \left(-2 + \frac{Ph^2}{EI_4}\right) \end{vmatrix} = 0 \quad (3)$$

o, en forma lineal:

$$\frac{P^3 h^6}{E^3 I_2 I_3 I_4} - 2 \frac{P^2 h^4}{E^2} \left(\frac{1}{I_2 I_3} + \frac{1}{I_3 I_4} + \frac{1}{I_2 I_4} \right) + \frac{Ph^2}{E} \left(\frac{2}{I_2} + \frac{4}{I_3} + \frac{3}{I_4} \right) - 2 = 0 \quad (4)$$

donde, para secciones en forma de polígono regular y redondas, los momentos de inercia estarán dados por:

$$I_i = \alpha D_i^4$$

D_i es el diámetro en el caso de las secciones circulares, y es la longitud de un lado en las secciones poligonales. La Tabla 6.1 muestra los valores de α para las secciones usadas más comúnmente.

Tabla 6.1.- Valores de la constante α para las secciones transversales más comunes.

Sección Circular	Sección Cuadrada	Sección Triangular
$\frac{\pi}{64}$	$\frac{1}{12}$	$\frac{\sqrt{3}}{96}$

En general, para un polígono regular de n lados, α puede derivarse mediante:

$$= \frac{n}{192} \cot \frac{\pi}{n} \left(3 \cot^2 \frac{\pi}{n} + 1 \right) \quad (5)$$

Para secciones rectangulares donde el ancho b se asume constante a través de la longitud de la columna,

$$I_i = \frac{bD_i^3}{12}, \quad i=2, 3, 4$$

En el diseño de una columna, la ecuación (4) representa una restricción de pandeo. Adicionalmente, debe satisfacerse una restricción de resistencia a la compresión, es decir:

$$\frac{P}{A_1} \leq \sigma_y$$

donde A_1 es una función de D_1 . Puesto que P y σ_y son cantidades dadas, se pueden calcular valores mínimos de D_1 o A_1 para cada problema. En otras palabras, la restricción de resistencia a la compresión dependerá exclusivamente del valor de A_1 o D_1 en el problema de optimización.

Con todo lo dicho anteriormente, contamos ahora con los elementos necesarios para plantear el problema de diseño de una columna como un problema de optimización: asumamos que P , h y σ_y son conocidos, y el objetivo es minimizar el volumen de la columna. De tal forma tendremos 2 casos a considerar:

- Columnas Cuadradas o Circulares.- La función objetivo será:

$$\text{Minimizar } V_c = K(D_1^2 + 2D_2^2 + 2D_3^2 + D_4^2 + D_1D_2 + D_2D_3 + D_3D_4) \quad (6)$$

donde K es una constante, cuyo valor está definido de acuerdo a la Tabla 6.2, y V_c es el volumen de la columna circular o cuadrada.

Tabla 6.2.- Valores de K de acuerdo al tipo de sección transversal de la columna

Sección Circular	Sección Cuadrada	Sección Triangular
$\frac{\pi l}{36}$	$\frac{l}{9}$	$\frac{l\sqrt{3}}{36}$

La función objetivo está sujeta a la restricción descrita en la ecuación (4), y a las siguientes restricciones adicionales:

$$C_1 < D_i < C_\mu, \quad i=1,2,3,4 \quad (7)$$

donde D_i son las variables de diseño; C_1 y C_μ son los límites inferior y superior, respectivamente, de las variables de diseño.

- Columnas rectangulares.- La función objetivo será:

$$\text{Minimizar } V_r = \frac{bl}{9}(D_1 + 2D_2 + 2D_3 + D_4 + \sqrt{D_1D_2} + \sqrt{D_2D_3} + \sqrt{D_3D_4}) \quad (8)$$

donde V_r es el volumen de la columna rectangular.

La función objetivo está sujeta a la restricción descrita en la ecuación (4) y a otra ecuación similar que se deriva del pandeo en la dirección ortogonal, o sea:

$$\frac{P^3 h^6}{E^3 I_2' I_3' I_4'} - 2 \frac{P^2 h^4}{E^2} \left(\frac{1}{I_2' I_3'} + \frac{1}{I_3' I_4'} + \frac{1}{I_2' I_4'} \right) + \frac{P h^2}{E} \left(\frac{2}{I_2'} + \frac{4}{I_3'} + \frac{3}{I_4'} \right) - 2 = 0 \quad (9)$$

donde

$$I_i' = \frac{D_i b^3}{12} \quad i=1,2,3,4$$

Además, existe una serie más de restricciones que deben satisfacerse:

$$\left. \begin{array}{l} b \times D_i > A_1 \\ C_i < D_i < C_{\mu} \\ C_i < b < C_{\mu} \end{array} \right\} i = 1,2,3,4 \quad (10)$$

donde $A_1 = P / \sigma_y$.

6.3. Uso del algoritmo genético

Para resolver este problema se hizo uso del algoritmo genético simple propuesto por Goldberg²⁰. En este caso, sin embargo, se debió analizar la forma de representar el espacio de búsqueda, pues éste es continuo. Observando el mecanismo de solución típico que se utiliza para este tipo de problemas se optó por discretizar el espacio de respuestas haciendo uso del siguiente algoritmo:

$$\text{diferencia} = L_s - L_i$$

Usando *diferencia* determina el número de bits necesarios

Si la cadena decodificada es mayor que *diferencia* entonces hazla igual a *diferencia*

El valor decodificado será igual a $(L_s + \text{valor decodificado})/1000.0$

L_s y L_i son los límites superior e inferior respectivamente multiplicados por mil (se consideraron sólo 3 decimales de precisión, aunque este valor puede modificarse). Como puede verse en este algoritmo, primero se determina cuál es la cantidad de bits que se requiere para representar la cantidad total de respuestas que se tendrá (i.e., mediante un redondeo a un cierto número fijo de decimales se discretiza el espacio de búsqueda). Como este valor difícilmente será una potencia exacta de 2, entonces usamos el valor inmediato superior. La decisión que sigue es necesaria para ajustar las respuestas a los valores que son válidos. La última línea del algoritmo permite obtener el verdadero valor de la respuesta, ya que se debe recordar que todos los resultados que se obtengan están desplazados L_i posiciones con respecto al origen, y además dicho valor deberá dividirse entre mil.

Dado que el mismo Goldberg recomienda el uso de Códigos de Gray para los casos en que los números binarios se usan para representar valores reales, esta implementación los incorporó, aunque no con muy buenos resultados (aunque se llegaba a la misma respuesta, se requerían más generaciones cuando se hacía uso de los códigos de Gray). Esa es la razón por la que se opta no utilizarlos.

La función de aptitud que se adopta se ilustra con el siguiente algoritmo:

```

checa1 := error en la ecuación (4)
Si  $(P/(A_1 \cdot \sigma_y)) - 1.0 > 0.0$  entonces  $checa2:=1.0$  sino  $checa2:=0.0$ 
aptitud :=  $1.0/(\text{volumen}*(1000.0*(checa1+checa2)+1.0))$ 

```

Como puede verse, si la respuesta viola la restricción impuesta por la ecuación (4), entonces la penalización será la cantidad con que difiere de cero (i.e., el error producido). Por otra parte, si viola la restricción de que $P/A_1 \leq \sigma_y$, entonces la penalización es 1.0. En el caso de las columnas rectangulares, la restricción es que $b \times D_1 > P/\sigma_y$. En este último caso también debe chequearse la dirección ortogonal, lo que hace que existan 3 valores de penalización en vez de 2. Estos valores se suman y su resultado se multiplica por mil -se magnifica el error- a fin de "castigar" la respuesta obtenida. Observar que si no se comete ninguna violación entonces la función de aptitud es simplemente el recíproco del volumen.

La implementación usó cruce de 2 puntos, y una técnica de selección mediante torneo binario. Los 4 diámetros se representaron mediante cadenas binarias consecutivas de la misma longitud. El criterio de detención utilizado fue a través de un número máximo de generaciones. El programa se escribió en Turbo Pascal 6.0 y hace uso de la técnica propuesta por Porter²¹ para el manejo dinámico de memoria.

Aunque se experimentó con varios parámetros, los valores más comúnmente adoptados fueron: poblaciones de 400 a 500 cromosomas, 80% de

probabilidad de cruce, 1% de probabilidad de mutación y 50 a 100 generaciones como máximo.

6.4. Ejemplos y comparación de resultados

Los ejemplos que se muestran a continuación fueron tomados del libro de Fu y Ren¹⁹:

EJEMPLO 1. Seleccionar los mejores diámetros en los puntos nodales de una columna de sección circular de acero de 10 pies de longitud que está sujeta a una carga axial de 400 kips. El módulo de elasticidad del material es $E=30 \times 10^6$ psi y el esfuerzo admisible, σ_y es 60,000 psi. Los límites inferior y superior, C_1 y C_μ , respectivamente son 2.914" y 20". El espacio de búsqueda de este problema es $(20000-2914)^4 \approx 8.52 \times 10^{16}$. Los resultados obtenidos se muestran en la Tabla 6.3. Nótese cómo en este ejemplo la solución, pese a arrojar un diseño con un volumen un poco mayor que el obtenido por Fu y Ren, proporciona una menor violación de la restricción impuesta por la ecuación (4). Esto es consecuencia de la penalización aplicada en la función de aptitud que ocasiona que el algoritmo genético se mueva hacia una solución que minimice todas las restricciones al mismo tiempo. El número de ciclos requeridos puede ser un tanto engañoso, porque el algoritmo genético algunas veces puede converger en 20 ó 30 generaciones, mientras que en otras puede requerir de casi 50 o hasta más. Sin embargo, si tomamos en cuenta que esta técnica no requiere la evaluación de derivadas ni ninguna de las complicaciones matemáticas del método usado por Fu y Ren, y si se considera también que converge en un tiempo relativamente corto

(alrededor de 3 minutos en una AT286 de 12 MHz sin coprocesador matemático), se podrá vislumbrar más claramente su importancia.

Tabla 6.3.- Comparación de resultados para el ejemplo 1.

	Fu y Ren ¹⁹	Algoritmo Genético
Volumen (plg ³)	1642.400	1644.010
D1 (plg)	2.914	2.914
D2 (plg)	3.967	3.908
D3 (plg)	4.601	4.614
D4 (plg)	4.771	4.854
Error en la ecuación (4)	2.090×10^{-4}	3.404×10^{-5}
P/A ₁	59977.867	59977.867
Ciclos requeridos	19	46

EJEMPLO 2. Seleccionar los mejores diámetros en los puntos nodales de una columna de sección cuadrada de acero de 10 pies de longitud que está sujeta a una carga axial de 400 kips. El módulo de elasticidad del material es $E=30 \times 10^6$ psi y el esfuerzo admisible, σ_y es 60,000 psi. Los límites inferior y superior, C_l y C_u , respectivamente son 2.582" y 20". El espacio de búsqueda de este problema es $(20000-2582)^4 \cong 9.2 \times 10^{16}$. Los resultados obtenidos se muestran en la Tabla 6.4. En este caso, la solución es ligeramente mayor que la encontrada por Fu y Ren, pero la solución proporcionada por el algoritmo genético viola la restricción impuesta por la ecuación (4) en menor grado.

Tabla 6.4.- Comparación de resultados para el ejemplo 2.

	Fu y Ren ¹⁹	Algoritmo Genético
Volumen (plg ³)	1608.300	1613.603
D1 (plg)	2.582	2.586
D2 (plg)	3.475	3.522
D3 (plg)	4.031	4.087
D4 (plg)	4.180	4.022
Error en la ecuación (4)	1.78×10^{-4}	1.58×10^{-4}
P/A ₁	59999.484	59999.484
Ciclos requeridos	10	33

EJEMPLO 3. Seleccionar los mejores diámetros en los puntos nodales de una columna de sección triangular de acero de 10 pies de longitud que está sujeta a una carga axial de 400 kips. El módulo de elasticidad del material es $E=30 \times 10^6$ psi y el esfuerzo admisible, σ_y es 60,000 psi. Los límites inferior y superior, C_l y C_u , respectivamente son 3.924" y 20". El espacio de búsqueda de este problema es $(20000-3924)^4 \cong 6.7 \times 10^{16}$. Los resultados obtenidos se muestran en la Tabla 6.5. Puede verse en los resultados cómo el algoritmo genético se comportó en este caso de una forma similar al ejemplo 1, ya que la solución que se encontró, pese a tener un volumen ligeramente mayor que la de Fu y Ren, viola la restricción impuesta por la ecuación (4) en un menor grado. La cantidad de iteraciones requeridas no toman en este caso más de 5 minutos.

Tabla 6.5.- Comparación de resultados para el ejemplo 3.

Tabla 6.5.- Comparación de resultados para el ejemplo 3.

	Fu y Ren ¹⁹	Algoritmo Genético
Volumen (plg ³)	1507.000	1510.070
D1 (plg)	3.924	3.924
D2 (plg)	5.092	5.220
D3 (plg)	5.910	5.882
D4 (plg)	6.130	5.999
Error en la ecuación (4)	3.26×10^{-4}	1.22×10^{-4}
P/A ₁	59993.11	59993.11
Ciclos requeridos	13	39

EJEMPLO 4. Seleccionar el mejor ancho y las mejores secciones transversales en los puntos nodales de una columna de sección rectangular de acero de 10 pies de longitud que está sujeta a una carga axial de 400 kips. El módulo de elasticidad del material es $E=30 \times 10^6$ psi y el esfuerzo admisible, σ_y es 60,000 psi. Los límites inferior y superior, C_l y C_u , respectivamente son 1.500" y 20". El espacio de búsqueda de este problema es $(20000-1500)^4 \cong 1.2 \times 10^{17}$. Los resultados obtenidos se muestran en la Tabla 7.6. Este ejemplo requirió una población más grande que las anteriores (500 cromosomas, contra la población de 400 usada en los ejemplos previos), y se dejó correr el algoritmo por 100 generaciones, en vez de las 50 que se usaron en los ejemplos anteriores. Puede verse cómo el resultado en este caso es muy similar al presentado por Fu y Ren, lo que demuestra que esta técnica puede proporcionarnos resultados muy buenos si ajustamos sus parámetros de manera apropiada, y además que dicha respuesta se tornará mejor en la medida en que se este dispuesto a sacrificar un poco más de tiempo

relación entre bxD_1 y P/σ_y , lo cual no sucede en la solución. Pese a que se requieren más cálculos en este caso, el tiempo total de ejecución no rebasó los 15 minutos.

Tabla 6.6.- Comparación de resultados para el ejemplo 4.

	Fu y Ren ¹⁹	Algoritmo Genético
Volumen (plg ³)	1617.952	1618.517
b (plg)	3.903	3.909
D1 (plg)	1.708	1.706
D2 (plg)	3.231	3.288
D3 (plg)	4.127	4.089
D4 (plg)	4.403	4.344
Error en la ecuación (4)	2.90×10^{-4}	6.35×10^{-5}
Error en la ecuación (9)	6.51×10^{-5}	1.29×10^{-4}
P/σ_y	6.667	6.667
bxD_1	6.666	6.669
Ciclos	9	65

El código fuente utilizado para estos cuatro ejemplos puede encontrarse en los anexos de esta tesis. Para ejecutar los programas, se debe de compilar los archivos COLSGA.PAS, COLSGA2.PAS, COLSGA3.PAS, COLSGA4.PAS con Turbo Pascal 6.0. Los 4 ejecutables que se generan corresponderán respectivamente a cada uno de los ejemplos mostrados anteriormente.

7. CONCLUSIONES

Se ha proporcionado de forma breve y concisa un panorama general de lo que son los algoritmos genéticos y su utilidad, sin llegar a profundizar mucho en ninguno de sus aspectos en particular. También se ha mostrado que los algoritmos genéticos presentan un comportamiento estable aún en los casos en que se intentan solucionar problemas con espacios de búsqueda continuos, siempre y cuando se seleccione un esquema de representación adecuado. A ese respecto, también se ha mostrado que los dígitos binarios pueden utilizarse de manera directa como esquema de representación en problemas en los que existe un límite inferior y uno superior en los valores de las soluciones posibles al problema.

Puede apreciarse también el excelente comportamiento del algoritmo genético como técnica de búsqueda aun en la presencia de espacios de búsqueda considerablemente grandes y en problemas sometidos a varias restricciones.

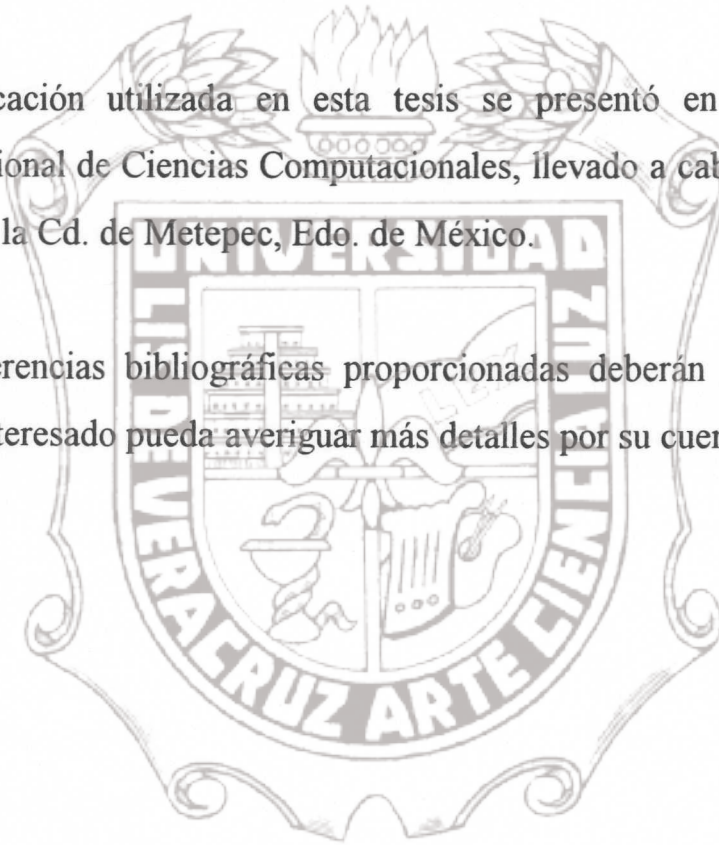
Se demuestra también que es una técnica de búsqueda relativamente simple de implementar, que no requiere de complejos cálculos matemáticos y puede resolver en un tiempo razonable problemas de alta complejidad para los que normalmente se requieren intrincados algoritmos matemáticos.

Con los ejemplos dados en esta tesis se ha corroborado también la hipótesis, de que los algoritmos genéticos son una técnica eficiente, ya que se pueden ver los resultados obtenidos mediante ellos y los obtenidos mediante la

técnica que sirvió de comparación, que es la de Fu y Ren. En cada ejemplo se da una breve comparación entre uno y otro y se muestran las violaciones a las restricciones impuestas para el desarrollo del problema, llegando a la conclusión de que los algoritmos genéticos en comparación con la técnica comparada dan muchos mejores resultados de optimización, en problemas de ingeniería civil, en este caso un problema de columnas no prismáticas.

La aplicación utilizada en esta tesis se presentó en el 1er. Congreso Internacional de Ciencias Computacionales, llevado a cabo en septiembre de 1994 en la Cd. de Metepec, Edo. de México.

Las referencias bibliográficas proporcionadas deberán servir para que el lector interesado pueda averiguar más detalles por su cuenta.



ANEXOS

Se presentan los diferentes listados que corresponden a los ejemplos mostrados en el capítulo VI.

Programa COLSGA.PAS

```

{$R-}
program sga;
Uses Crt,Dos;

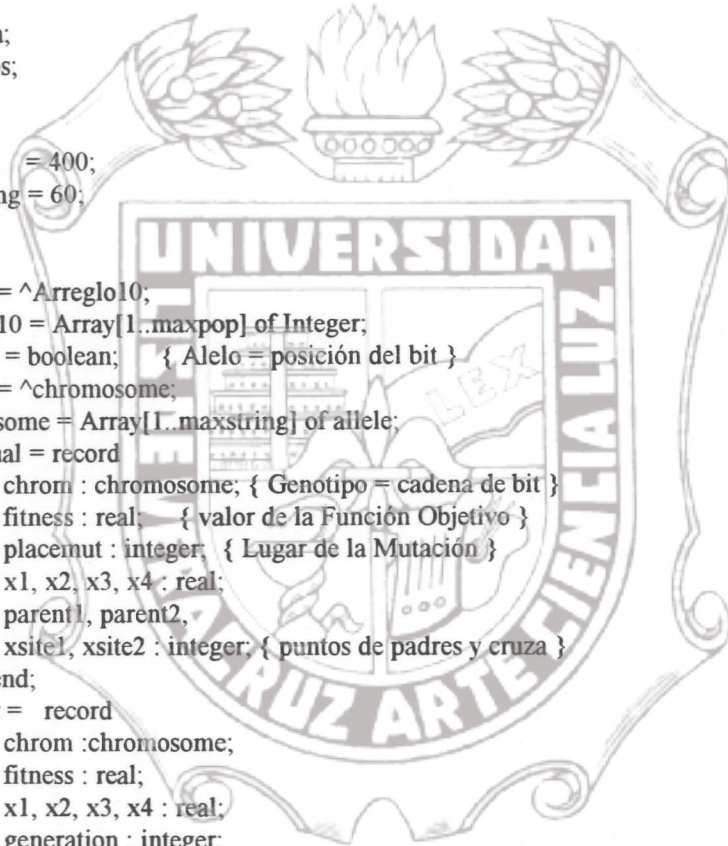
Const
  maxpop = 400;
  maxstring = 60;

Type
  Base10 = ^Arreglo10;
  Arreglo10 = Array[1..maxpop] of Integer;
  allele = boolean; { Alelo = posición del bit }
  Base12 = ^chromosome;
  chromosome = Array[1..maxstring] of allele;
  individual = record
    chrom : chromosome; { Genotipo = cadena de bit }
    fitness : real; { valor de la Función Objetivo }
    placemut : integer; { Lugar de la Mutación }
    x1, x2, x3, x4 : real;
    parent1, parent2,
    xsite1, xsite2 : integer; { puntos de padres y cruza }
  end;
  bestever = record
    chrom : chromosome;
    fitness : real;
    x1, x2, x3, x4 : real;
    generation : integer;
  end;

  Base11 = ^population;
  population = array[1..maxpop] of individual;

Var
  bestfit : bestever;
  oldpop, newpop : base11; { Dos poblaciones no-sobrepuestas }
  popsize, lchrom, gen, maxgen : integer; { variables globales enteras }
  pcross, pmutation, sumfitness : real; { variables globales reales }
  nmutation, ncross, tournpos : integer; { estáticas enteras }
  s1, s2, s3, s4, avg, max, min : real; { estáticas reales }
  maxv, minv : real;
  tournlist : Base10;
  hour,minute,second,sec100 : word;

```



```

{ Incluye utilidades de procesos y funciones }
{$I utility.sga}

{ Incluye generador de números pseudo-aleatorios y utilidades aleatorias }
{$I random.apb}

{ Incluye rutinas de interfaces: decode y objfunc }
{$I interfa6.sga}

{ Incluye cálculos estáticos : statistics }
{$I stats6.sga}

{ Incluye rutinas de inicialización : initialize, initdata, initpop, initreport }
{$I initial6.sga}

{ Incluye rutinas de reportes: report, writechrom }
{$I report6.sga}

{ Incluye los 3 operadores: select (reproducción), crossover, mutation }
{$I triops.sga}

{ Incluye rutina de nueva generación de población: generation }
{$I genera6.sga}
function D2(n:word):word;

Begin
  if n<10 then Write('0');
  D2:=n;
End;

begin { Programa Principal }
  settime(0,0,0,0);
  gen:=0; { Inicia parámetros }
  New(oldpop);
  New(newpop);
  New(tournlist);
  initialize;
  repeat { Ciclo iterativo principal }
    gen:=gen+1;
    generation;
    statistics(popsiz,max,avg,min,sumfitness,newpop);
    report(gen);

    oldpop:=newpop; { avanza la generación }

  until (gen >= maxgen);

  GetTime(hour,minute,second,sec100);
  if hour>12 then hour:=hour-12;
  writeln('Tiempo= ',D2(horas),':',D2(minutos),':',D2(segundos))
end. { Fin del Programa Principal }

```

Programa COLSGA2.PAS

COLSGA2.PAS

{ \$R- }

```

program sga;
Uses Crt,Dos;

```

Const

```

maxpop = 400;
maxstring = 60;

```

Type

```

Base10 = ^Arreglo10;
Arreglo10 = Array[1..maxpop] of Integer;
allele = boolean; { Alelo = posición del bit }
Base12 = ^chromosome;
chromosome = Array[1..maxstring] of allele;
individual = record
    chrom : chromosome; { Genotipo = cadena de bit }
    fitness : real; { valor de la función objetivo }
    placemut : integer; { Lugar de la mutación }
    x1, x2, x3, x4 : real;
    parent1, parent2,
    xsite1, xsite2 : integer; { puntos de padres y cruza }
end;
bestever = record
    chrom : chromosome;
    fitness : real;
    x1, x2, x3, x4 : real;
    generation : integer;
end;

```

```

Base11 = ^population;
population = array[1..maxpop] of individual;

```

Var

```

bestfit : bestever;
oldpop, newpop : base11; { Dos poblaciones no sobrepuestas }
popsize, lchrom, gen, maxgen : integer; { variables globales enteras }
pcross, pmutation, sumfitness : real; { variables globales reales }
nmutation, ncross, tournpos : integer; { estáticas enteras }
s1, s2, s3, s4, avg, max, min : real; { estáticas reales }
maxv, minv : real;
tourndlist : Base10;
hour,minute,second,sec100 : word;

```

```

{ Incluye utilidades de procesos y funciones }
{ $I utility.sga }

```

```

{ incluye generador de números pseudo-aleatorios y utilidades aleatorias }
{ $I random.apb }

```



```

{ Incluye rutinas de interfaces: decode y objfunc }
{$I interfa7.sga}

{ Incluye cálculos estáticos: statistics }
{$I stats6.sga}

{ Incluye rutinas de inicialización: initialize, initdata, initpop, initreport }
{$I initial6.sga}

{ Incluye rutinas de reportes: report, writechrom }
{$I report7.sga}

{ Incluye los 3 operadores: select (reproducción), crossover, mutation }
{$I triops.sga}

{ Incluye rutina de generación de nueva población: generation }
{$I genera6.sga}

function D2(n:word):word;
Begin
  if n<10 then Write('0');
  D2:=n;
End;

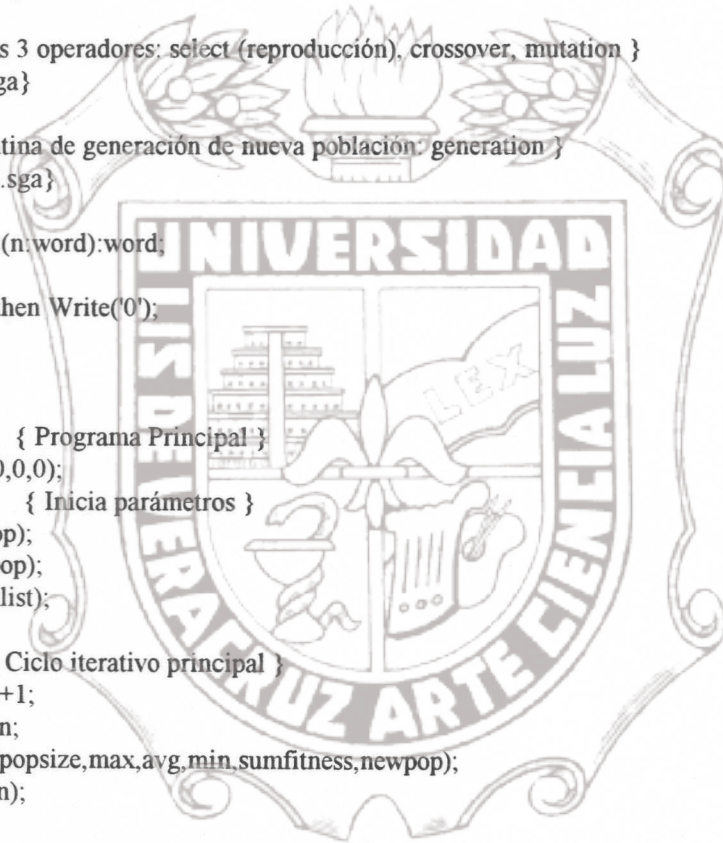
begin { Programa Principal }
  settime(0,0,0,0);
  gen:=0; { Inicia parámetros }
  New(oldpop);
  New(newpop);
  New(tournlist);
  initialize;
  repeat { Ciclo iterativo principal }
    gen:=gen+1;
    generation;
    statistics(popsiz,max,avg,min,sumfitness,newpop);
    report(gen);

    oldpop:=newpop; { avanza la generación }

  until (gen >= maxgen);

  GetTime(hour,minute,second,sec100);
  if hour>12 then hour:=hour-12;
  writeln('timpo= ',D2(horas),',',D2(minutos),',',D2(segundos)
end. { Fin del Programa Principal }

```



Programa COLSGA3.PAS

{SR-}

```

program sga;
Uses Crt,Dos;

```

Const

```

maxpop = 400;
maxstring = 60;

```

Type

```

Base10 = ^Arreglo10;
Arreglo10 = Array[1..maxpop] of Integer;
allele = boolean; { Alelo = posición del bit }
Base12 = ^chromosome;
chromosome = Array[1..maxstring] of allele;
individual = record
  chrom : chromosome; { Genotipo = cadena de bit }
  fitness : real; { Valor de la función objetivo }
  placemut : integer; { lugar de la mutación }
  x1, x2, x3, x4 : real;
  parent1, parent2,
  xsite1, xsite2 : integer; { puntos de padres y cruza }
end;
bestever = record
  chrom : chromosome;
  fitness : real;
  x1, x2, x3, x4 : real;
  generation : integer;
end;

Base11 = ^population;
population = array[1..maxpop] of individual;

```

Var

```

bestfit : bestever;
oldpop, newpop : base11; { Dos poblaciones no sobrepuestas }
popsize, lchrom, gen, maxgen : integer; { variables globales enteras }
pcross, pmutation, sumfitness : real; { variables globales reales }
nmutation, ncross, tournpos : integer; { estáticas enteras }
s1, s2, s3, s4, avg, max, min : real; { estáticas reales }
maxv, minv : real;
tournlist : Base10;
hour,minute,second,sec100 : word;

```

```

{ Incluye utilidades procesos y funciones }
{$I utility.sga}

```

```

{ Incluye generador de números pseudo-aleatorios y utilidades aleatorias }
{$I random.apb}

```

```

{ Incluye rutinas de interfaces: decode y objfunc }
{$I interfa8.sga}

```

```

{ Incluye cálculos estáticos: statistics }
{$I stats6.sga}

{ Incluye rutinas de inicialización: initialize, initdata, initpop, initreport }
{$I initial6.sga}

{ Incluye rutinas de reportes: report, writechrom }
{$I report8.sga}

{ Incluye los 3 operadores: select (reproducción), crossover, mutation }
{$I triops.sga}

{ Incluye rutina de generación de nueva población: generation }
{$I genera6.sga}

function D2(n: word): word;
Begin
  if n<10 then Write('0');
  D2:=n;
End;

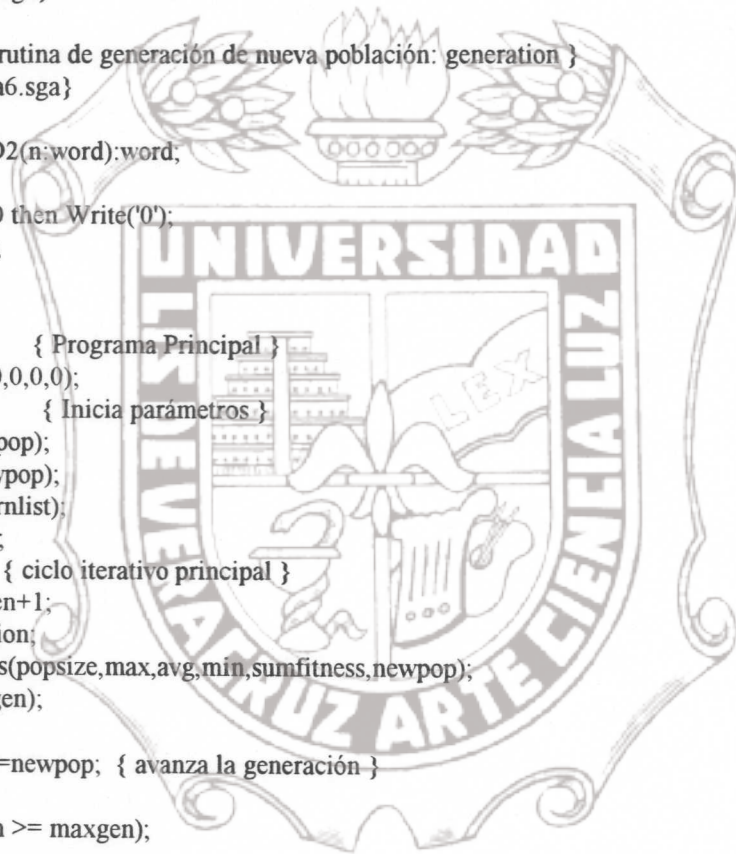
begin { Programa Principal }
  settime(0,0,0,0);
  gen:=0; { Inicia parámetros }
  New(oldpop);
  New(newpop);
  New(tournlist);
  initialize;
  repeat { ciclo iterativo principal }
    gen:=gen+1;
    generation;
    statistics(popsiz, max, avg, min, sumfitness, newpop);
    report(gen);

    oldpop:=newpop; { avanza la generación }

  until (gen >= maxgen);

  GetTime(hour, minute, second, sec100);
  if hour>12 then hour:=hour-12;
  writeln("Tiempo= ',D2(horas),':',D2(minutos),':',D2(segundos))
end. { Fin del Programa Principal }

```



Instituto de Ingeniería
 Universidad Veracruzana

Programa COLSGA4.PAS

{R-}

```

program sga;
Uses Crt,Dos;

```

Const

```

maxpop = 500;
maxstring = 75;

```

Type

```

Base10 = ^Arreglo10;
Arreglo10 = Array[1..maxpop] of Integer;
allele = boolean; { Alelo = posición del bit }
Base12 = ^chromosome;
chromosome = Array[1..maxstring] of allele;
individual = record
    chrom : chromosome; { Genotipo = cadena de bit }
    fitness : real; { Valor de la función objetivo }
    placemut : integer; { lugar de la mutación }
    b, x1, x2, x3, x4 : real;
    parent1, parent2,
    xsite1, xsite2 : integer; { puntos de padres y cruza }
end;
bestever = record
    chrom : chromosome;
    fitness : real;
    b, x1, x2, x3, x4 : real;
    generation : integer;
end;

Base11 = ^population;
population = array[1..maxpop] of individual;

```

Var

```

bestfit : bestever;
oldpop, newpop : base11; { Dos poblaciones no sobrepuestas }
popsize, lchrom, gen, maxgen : integer; { variables globales enteras }
pcross, pmutation, sumfitness : real; { variables globales reales }
nmutation, ncross, tournpos : integer; { estáticas enteras }
b1, s1, s2, s3, s4, avg, max, min : real; { estáticas reales }
maxv, minv : real;
tournlist : Base10;
hour,minute,second,sec100 : word;

```

```

{ Incluye utilidades procesos y funciones }
{$I utility.sga}

```

```

{ Incluye generador de números pseudo-aleatorios y utilidades aleatorias }
{$I random.apb}

```

```

{ Incluye rutinas de interfaces: decode y objfunc }
{$I interfa9.sga}

```

```

{ Incluye cálculos estáticos: statistics }
{$I stats9.sga}

{ Incluye rutinas de inicialización: initialize, initdata, initpop, initreport }
{$I initial9.sga}

{ Incluye rutinas de reportes: report, writechrom }
{$I report9.sga}

{ Incluye los 3 operadores: select (reproducción), crossover, mutation }
{$I triops.sga}

{ Incluye rutina de generación de nueva población: generation }
{$I genera9.sga}

function D2(n:word):word;
Begin
  if n<10 then Write('0');
  D2:=n;
End;

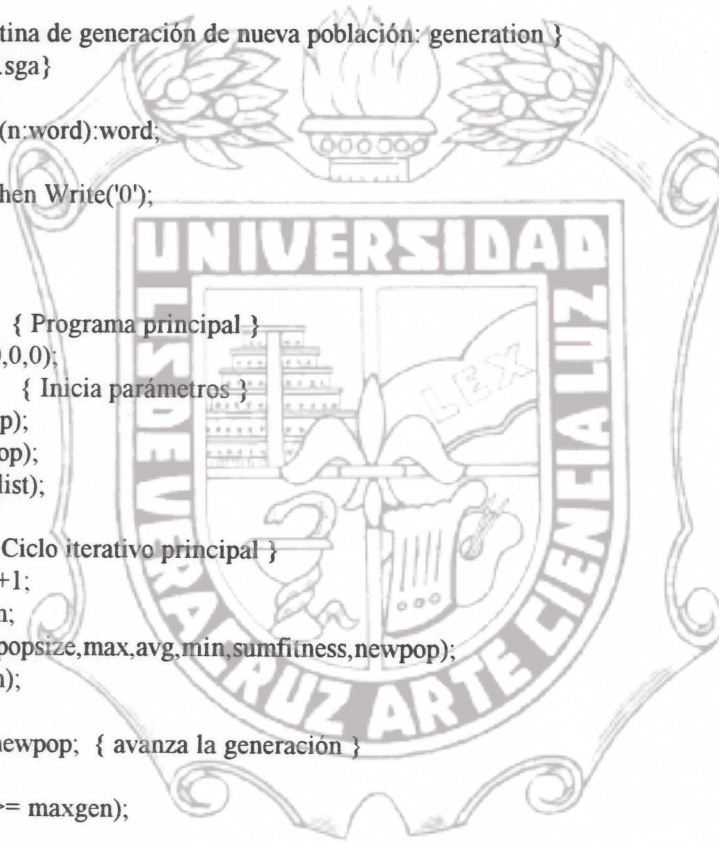
begin { Programa principal }
  settime(0,0,0,0);
  gen:=0; { Inicia parámetros }
  New(oldpop);
  New(newpop);
  New(tournlist);
  initialize;
  repeat { Ciclo iterativo principal }
    gen:=gen+1;
    generation;
    statistics(popsiz,max,avg,min,sumfitness,newpop);
    report(gen);

    oldpop:=newpop; { avanza la generación }

  until (gen >= maxgen);

  GetTime(hour,minute,second,sec100);
  if hour>12 then hour:=hour-12;
  writeln('Tiempo= ',D2(horas),':',D2(minutos),':',D2(segundos))
end. { Fin del programa principal }

```



Programa UTILITY.SGA

```
{ utility.sga : contiene pause, page, repchar, skip, power }
```

```
procedure pause(pauselength:integer);
```

```
{ Se detiene un momento }
```

```
const maxpause=2500;
```

```
var j,j1:integer;
```

```
  x:real;
```

```
begin
```

```
  for j:=1 to pauselength do
```

```
    for j1:=1 to maxpause do x:=0.0+1.0;
```

```
end;
```

```
procedure page(var out:text);
```

```
{ Implementa la alimentación a un dispositivo o archivo }
```

```
begin write(out,chr(12)) end;
```

```
procedure repchar(var out:text; ch:char; repcount:integer);
```

```
{ Escribe repetidamente un caracter a un dispositivo de salida }
```

```
var j:integer;
```

```
begin for j:=1 to repcount do write(out,ch) end;
```

```
procedure skip(var out:text; skipcount:integer);
```

```
{ Salta el contador de saltos de líneas en el dispositivo de salida }
```

```
var j:integer;
```

```
begin for j:=1 to skipcount do writeln(out); end;
```

```
function sign(x:real):real;
```

```
begin
```

```
  if x=0.0 then sign:=1.0
```

```
  else sign:=abs(x)/x
```

```
end;
```

```
function power(x,y:real):real;
```

```
{ Levanta x a la ya potencia }
```

```
begin
```

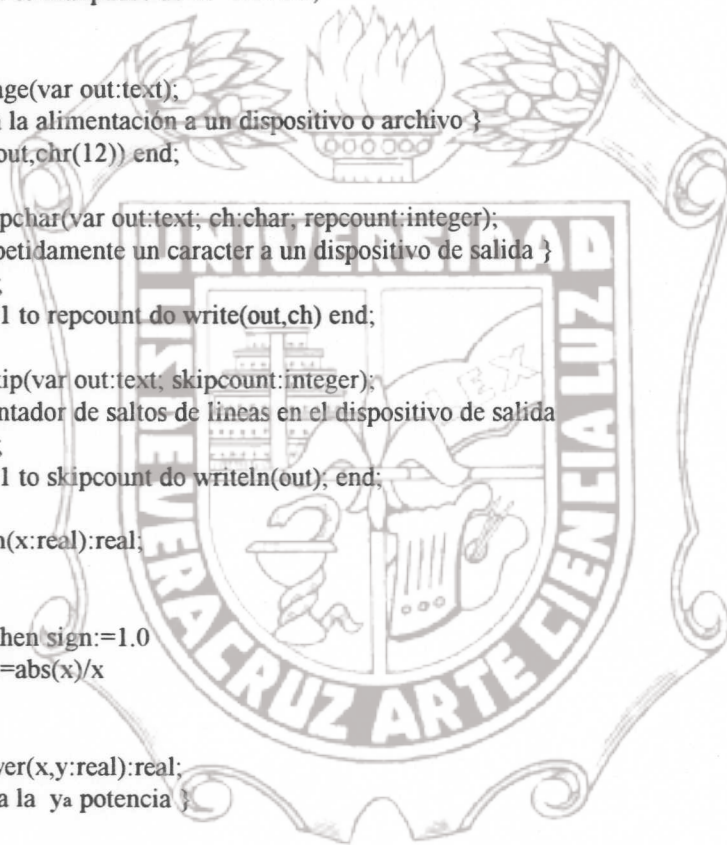
```
  if x=0.0 Then
```

```
    if y=0.0 Then power:=1.0
```

```
    else power:=0.0
```

```
    else power:=sign(x)*exp(y*ln(abs(x)))
```

```
end;
```



Programa RANDOM.APB

```

{ random.apb: contiene el generador de número aleatorios y relaciona las utilidades
  incluyendo advance_random, warmup_random, random, randomize,
  flip, rnd }
{ Variables globales - No usar estos nombres en otros códigos }
var oldrand:array[1..55] of real; { Arreglo de 55 números aleatorios }
    jrand : integer;           { Aleatorio actual }

procedure advance_random;
{ crea el siguiente lote de 55 números aleatorios }
var j1:integer;
    new_random:real;
begin
  for j1:=1 to 24 do
  begin
    new_random:=oldrand[j1]-oldrand[j1+31];
    if (new_random < 0.0) then new_random:=new_random+1.0;
    oldrand[j1]:=new_random;
  end;
  for j1:=25 to 55 do
  begin
    new_random:=oldrand[j1]-oldrand[j1-24];
    if (new_random<0.0) then new_random:=new_random+1.0;
    oldrand[j1]:=new_random;
  end;
end;

procedure warmup_random(random_seed:real);
{ Obtiene la salida aleatoria y la ejecuta }
var j1, ii : integer;
    new_random,prev_random:real;
begin
oldrand[55]:=random_seed;
new_random:=1.0e-9;
prev_random:=random_seed;
for j1:=1 to 54 do
  begin
    ii:=21*j1 mod 55;
    oldrand[ii]:=new_random;
    new_random:=prev_random-new_random;
    if (new_random<0.0) then new_random:=new_random+1.0;
    prev_random:=oldrand[ii];
  end;
  advance_random; advance_random; advance_random;
  jrand:=0;
end;

function random:real;
{ Busca un número aleatorio simple entre 0.0 y 1.0 - Método subtractivo }
begin
  jrand:=jrand+1;

```

```

if (jrand>55) then
  begin jrand:=1; advance_random end;
random:=oldrand[jrand];
end;

```

```

function flip(probability:real):boolean;
{ Lanza la moneda - verdadero si es cara }
begin
  if probability=1.0 then flip:=true
  else flip:=(random<=probability);
end;

```

```

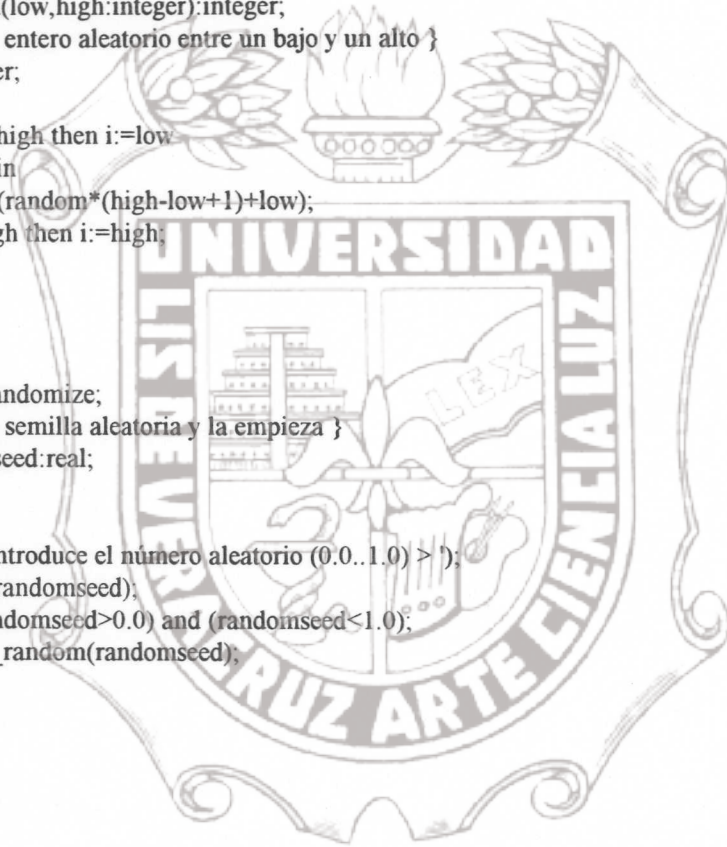
function rnd(low,high:integer):integer;
{ Escoge un entero aleatorio entre un bajo y un alto }
var i : integer;
begin
  if low>=high then i:=low
  else begin
    i:=trunc(random*(high-low+1)+low);
    if i > high then i:=high;
  end;
  rnd:=i;
end;

```

```

procedure randomize;
{ Obtiene la semilla aleatoria y la empieza }
var randomseed:real;
begin
  repeat
    write('Introduce el número aleatorio (0.0..1.0) > ');
    readln(randomseed);
  until (randomseed>0.0) and (randomseed<1.0);
  warmup_random(randomseed);
end;

```



Programa INTERFA6.SGA

```

{ interfa6.sga: contiene objfunc, decode }
{ Cambiar esto para problemas diferentes }

function objfunc(x1, x2, x3, x4 : real):real;

var i1, i2, i3, i4, p, h, e, l, checa, checa2, esfy, vol : real;

begin

  p:=400000.0; h:=20.0; e:=30e6;
  l:=120.0; esfy:=60000.0;
  i1:=(Pi/64.0)*x1*x1*x1*x1;
  i2:=(Pi/64.0)*x2*x2*x2*x2;
  i3:=(Pi/64.0)*x3*x3*x3*x3;
  i4:=(Pi/64.0)*x4*x4*x4*x4;
  checa:=(p*p*p*h*h*h*h*h*h)/(e*e*e*i2*i3*i4);
  checa:=checa-(2.0*p*p*h*h*h*h*h)/(e*e)*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
  checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
  checa:=abs(checa);

  if abs(p/(Pi*x1*x1/(4.0*esfy))-1.0)>0.0 then checa2:=1.0 else checa2:=0.0;

  vol:=(Pi*l/36.0)*(x1*x1+2.0*x2*x2+2.0*x3*x3+x4*x4+x1*x2+x2*x3+x3*x4);

  objfunc:=1/(vol*(1000*(checa+checa2)+1));

end;

procedure decode(chrom:chromosome; lbits:integer);
{ Decodifica la cadena como un entero binario sin signo - verdadero=1, falso=0 }

var j, inicio, final : integer;
    powerof2, accum1 : longint;
    parity : boolean;

begin

  (* Conversión de códigos de Gray a binario *)

  (* parity:=false;

  for j:=60 downto 1 do begin
    if chrom[j] then parity:=not parity;
    if parity then chrom[j]:=true else chrom[j]:=false;
  end; *)

  inicio:=2914; final:=20000;

  accum1:=0; powerof2:=1;
  for j:=1 to 15 do begin
    if chrom[j] then accum1:=accum1+powerof2;

```

```

    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s1:=(inicio+accum1)/1000.0;

accum1:=0; powerof2:=1;
for j:=16 to 30 do begin
    if chrom[j] then accum1:=accum1+powerof2;
    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s2:=(inicio+accum1)/1000.0;

accum1:=0; powerof2:=1;
for j:=31 to 45 do begin
    if chrom[j] then accum1:=accum1+powerof2;
    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s3:=(inicio+accum1)/1000.0;

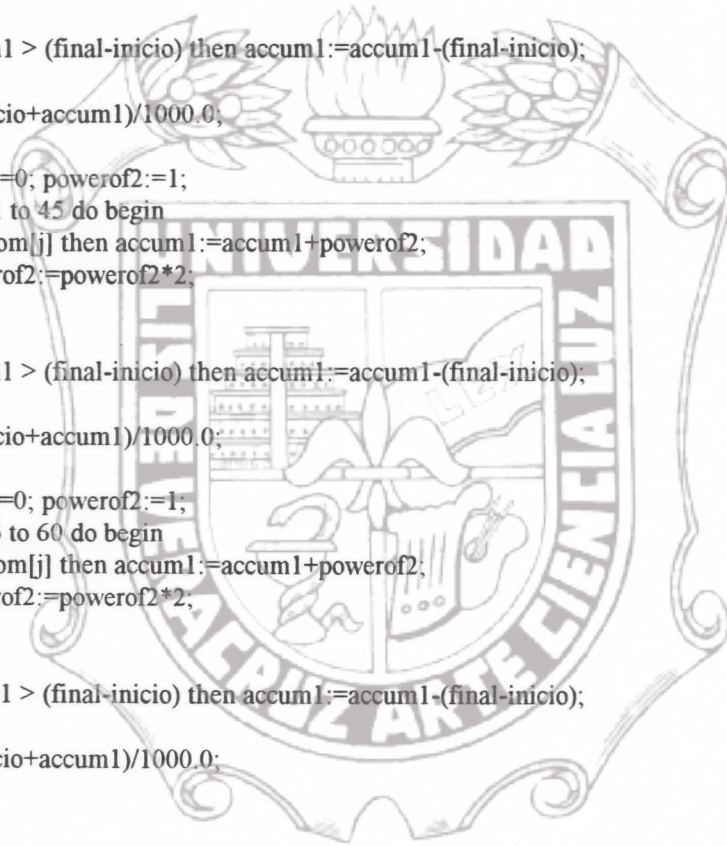
accum1:=0; powerof2:=1;
for j:=46 to 60 do begin
    if chrom[j] then accum1:=accum1+powerof2;
    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s4:=(inicio+accum1)/1000.0;

end;

```



Programa INTERFA7.SGA

```

{ interfaz.sga: contiene objfunc, decode }
{ Cambiar esto para problema diferentes }

function objfunc(x1, x2, x3, x4 : real):real;

var i1, i2, i3, i4, p, h, e, l, checa, checa2, esfy, vol : real;

begin

    p:=400000.0; h:=20.0; e:=30e6;
    l:=120.0; esfy:=60000.0;
    i1:=(1.0/12.0)*x1*x1*x1*x1;
    i2:=(1.0/12.0)*x2*x2*x2*x2;
    i3:=(1.0/12.0)*x3*x3*x3*x3;
    i4:=(1.0/12.0)*x4*x4*x4*x4;
    checa:=(p*p*p*h*h*h*h*h*h)/(e*e*e*i2*i3*i4);
    checa:=checa-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
    checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
    checa:=abs(checa);

    if abs(p/(x1*x1/esfy)-1.0) > 0.0 then checa2:=1.0 else checa2:=0.0;

    vol:=(1/9.0)*(x1*x1+2.0*x2*x2+2.0*x3*x3+x4*x4+x1*x2+x2*x3+x3*x4);

    objfunc:=1/(vol*(1000*(checa+checa2)+1));

end;

procedure decode(chrom:chromosome; lbits:integer);
{ Decodifica la cadena como un entero binario sin signo - verdadero=1, falso=0 }

var j, inicio, final : integer;
    powerof2, accum1 : longint;

begin

    (* Conversión de códigos de Gray a binario *)

    (* parity:=false;

    for j:=60 downto 1 do begin
        if chrom[j] then parity:=not parity;
        if parity then chrom[j]:=true else chrom[j]:=false;
        end; *)

    inicio:=2582; final:=20000;

    accum1:=0; powerof2:=1;
    for j:=1 to 15 do begin
        if chrom[j] then accum1:=accum1+powerof2;

```

```

    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s1:=(inicio+accum1)/1000.0;

accum1:=0; powerof2:=1;
for j:=16 to 30 do begin
    if chrom[j] then accum1:=accum1+powerof2;
    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s2:=(inicio+accum1)/1000.0;

accum1:=0; powerof2:=1;
for j:=31 to 45 do begin
    if chrom[j] then accum1:=accum1+powerof2;
    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s3:=(inicio+accum1)/1000.0;

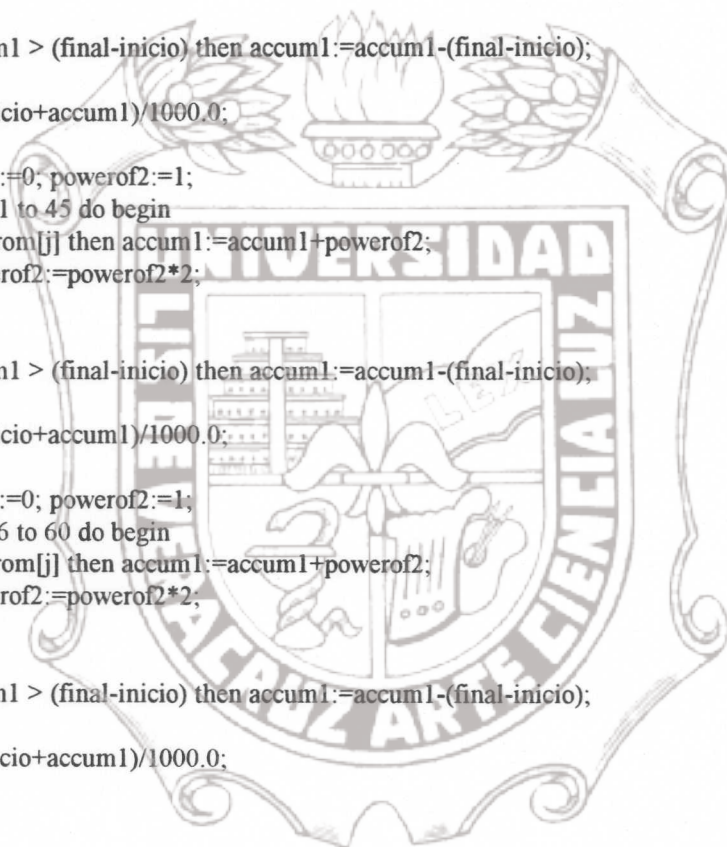
accum1:=0; powerof2:=1;
for j:=46 to 60 do begin
    if chrom[j] then accum1:=accum1+powerof2;
    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s4:=(inicio+accum1)/1000.0;

end;

```



Programa INTERFA8.SGA

```

{ interfaz.sga: contiene objfunc, decode }
{ Cambiar esto para problemas diferentes }

function objfunc(x1, x2, x3, x4 : real):real;

var i1, i2, i3, i4, p, h, e, l, checa, checa2, esfy, vol : real;

begin

    p:=400000.0; h:=20.0; e:=30e6;
    l:=120.0; esfy:=60000.0;
    i1:=(sqrt(3.0)/96.0)*x1*x1*x1*x1;
    i2:=(sqrt(3.0)/96.0)*x2*x2*x2*x2;
    i3:=(sqrt(3.0)/96.0)*x3*x3*x3*x3;
    i4:=(sqrt(3.0)/96.0)*x4*x4*x4*x4;
    checa:=(p*p*p*h*h*h*h*h*h)/(e*e*e*i2*i3*i4);
    checa:=checa-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
    checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
    checa:=abs(checa);

    if abs(p/(x1*x1*sqrt(3.0)/(4.0*esfy))-1.0) > 0.0 then checa2:=1.0 else checa2:=0.0;

    vol:=(1*sqrt(3.0)/36.0)*(x1*x1+2.0*x2*x2+2.0*x3*x3+x4*x4+x1*x2+x2*x3+x3*x4);

    objfunc:=1/(vol*(1000*(checa+checa2)+1));

end;

procedure decode(chrom:chromosome; lbits:integer);
{ Decodifica la cadena como un entero binario sin signo - verdadero=1, falso=0 }

var j, inicio, final : integer;
    powerof2, accum1 : longint;

begin

    (* Conversión de códigos de Gray a binario *)

    (* parity:=false;

    for j:=60 downto 1 do begin
        if chrom[j] then parity:=not parity;
        if parity then chrom[j]:=true else chrom[j]:=false;
        end; *)

    inicio:=3924; final:=20000;

    accum1:=0; powerof2:=1;
    for j:=1 to 15 do begin
        if chrom[j] then accum1:=accum1+powerof2;

```

```

    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s1:=(inicio+accum1)/1000.0;

accum1:=0; powerof2:=1;
for j:=16 to 30 do begin
    if chrom[j] then accum1:=accum1+powerof2;
    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s2:=(inicio+accum1)/1000.0;

accum1:=0; powerof2:=1;
for j:=31 to 45 do begin
    if chrom[j] then accum1:=accum1+powerof2;
    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s3:=(inicio+accum1)/1000.0;

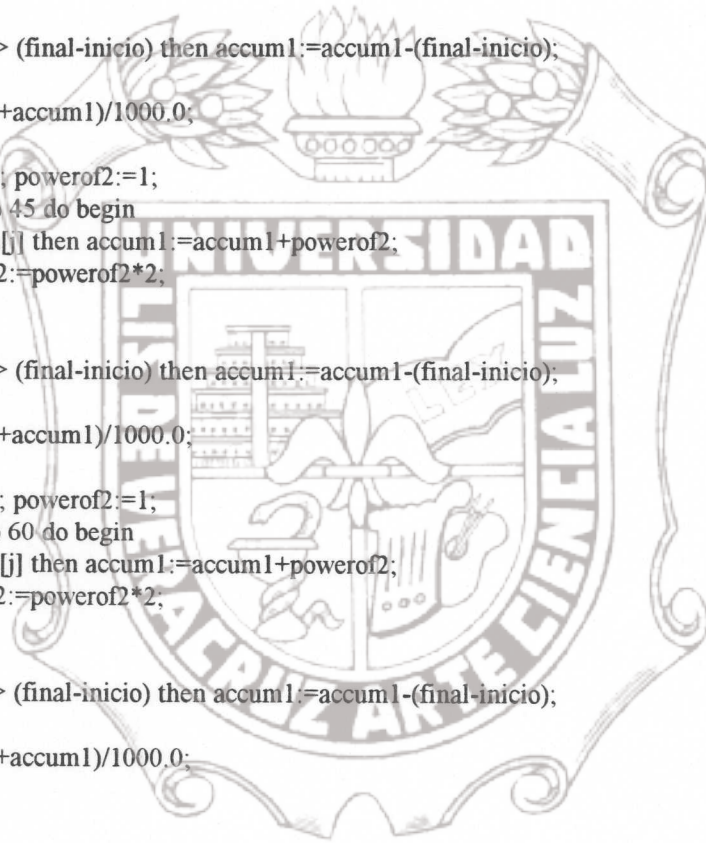
accum1:=0; powerof2:=1;
for j:=46 to 60 do begin
    if chrom[j] then accum1:=accum1+powerof2;
    powerof2:=powerof2*2;
end;

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

s4:=(inicio+accum1)/1000.0;

end;

```



Programa INTERFA9.SGA

```

{ interfaz.sga: contiene objfunc, decode }
{ Cambiar esto para problemas diferentes }

function objfunc(b, x1, x2, x3, x4 : real):real;

var i1, i2, i3, i4, p, h, e, l, checa, checa2, checa3, esfy, vol : real;

begin

  p:=400000.0; h:=20.0; e:=30e6;
  l:=120.0; esfy:=60000.0;
  i1:=b*x1*x1*x1/12.0;
  i2:=b*x2*x2*x2/12.0;
  i3:=b*x3*x3*x3/12.0;
  i4:=b*x4*x4*x4/12.0;
  checa:=(p*p*p*h*h*h*h*h*h)/(e*e*e*i2*i3*i4);
  checa:=checa-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
  checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
  checa:=abs(checa);

  if (b*x1>P/esfy) then checa2:=0.0 else checa2:=1.0;

  i1:=x1*b*b*b/12.0;
  i2:=x2*b*b*b/12.0;
  i3:=x3*b*b*b/12.0;
  i4:=x4*b*b*b/12.0;
  checa3:=(p*p*p*h*h*h*h*h*h)/(e*e*e*i2*i3*i4);
  checa3:=checa3-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
  checa3:=checa3+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
  checa3:=abs(checa3);

  vol:=(b*l/9.0)*(x1+2.0*x2+2.0*x3+x4+sqrt(x1*x2)+sqrt(x2*x3)+sqrt(x3*x4));

  objfunc:=1.0/(vol*(1000*(checa+checa2+checa3)+1));

end;

procedure decode(chrom:chromosome; lbits:integer);
{ Decodifica la cadena como un entero binario sin signo - verdadero=1, falso=0 }

var j, inicio, final : integer;
    powerof2, accum1 : longint;

begin

  (* Conversión de códigos de Gray a binario *)

  (* parity:=false;

  for j:=60 downto 1 do begin

```

```

if chrom[j] then parity:=not parity;
  if parity then chrom[j]:=true else chrom[j]:=false;
end; *)

```

```

inicio:=1500; final:=20000;

```

```

accum1:=0; powerof2:=1;
for j:=1 to 15 do begin
  if chrom[j] then accum1:=accum1+powerof2;
  powerof2:=powerof2*2;
end;

```

```

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

```

```

b1:=(inicio+accum1)/1000.0;

```

```

accum1:=0; powerof2:=1;
for j:=16 to 30 do begin
  if chrom[j] then accum1:=accum1+powerof2;
  powerof2:=powerof2*2;
end;

```

```

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

```

```

s1:=(inicio+accum1)/1000.0;

```

```

accum1:=0; powerof2:=1;
for j:=31 to 45 do begin
  if chrom[j] then accum1:=accum1+powerof2;
  powerof2:=powerof2*2;
end;

```

```

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

```

```

s2:=(inicio+accum1)/1000.0;

```

```

accum1:=0; powerof2:=1;
for j:=46 to 60 do begin
  if chrom[j] then accum1:=accum1+powerof2;
  powerof2:=powerof2*2;
end;

```

```

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

```

```

s3:=(inicio+accum1)/1000.0;

```

```

accum1:=0; powerof2:=1;
for j:=61 to 75 do begin
  if chrom[j] then accum1:=accum1+powerof2;
  powerof2:=powerof2*2;
end;

```

```

if accum1 > (final-inicio) then accum1:=accum1-(final-inicio);

```


s4:=(inicio+accum1)/1000.0;

end;



Instituto de Ingeniería
Universidad Veracruzana

Programa STATS6.SGA

```

{ stats.sga }

procedure statistics(popsiz:integer;
  var max,avg,min,sumfitness : real;
  var pop : base11);

{ Calcula la población estática }

var j : integer;
begin
  { Inicializa }
  sumfitness:=pop^[1].fitness;
  min :=pop^[1].fitness;
  max :=pop^[1].fitness;
  { Ciclo para max, min, sumfitness }
  for j:=2 to popsiz do with pop^[j] do begin
    sumfitness:=sumfitness+fitness; { Acumula la suma de aptitudes }
    if fitness>max then Begin
      max:=fitness; { Nuevo max }
    End;
    if fitness<min then min:=fitness; { Nuevo min }
    if (fitness>bestfit.fitness) then Begin
      bestfit.chrom:=chrom;
      bestfit.fitness:=fitness;
      bestfit.generation:=gen;
      bestfit.x1:=x1;
      bestfit.x2:=x2;
      bestfit.x3:=x3;
      bestfit.x4:=x4;
    End;
  end;

  { Calcula el porcentaje }

  if max>maxv then maxv:=max;
  if min<minv then minv:=min;
  avg:=sumfitness/popsiz;
end;

```

Programa STATS9.SGA

```

{ stats.sga }

procedure statistics(popsiz:integer;
  var max,avg,min,sumfitness : real;
  var pop : base11);

  { Calcula la población estática }

var j : integer;
begin
  { Inicializa }
  sumfitness:=pop^[1].fitness;
  min :=pop^[1].fitness;
  max :=pop^[1].fitness;
  { Ciclo para max, min, sumfitness }
  for j:=2 to popsiz do with pop^[j] do begin
    sumfitness:=sumfitness+fitness; { Acumula la suma de aptitudes }
    if fitness>max then Begin
      max:=fitness; { Nuevo max }
    End;
    if fitness<min then min:=fitness; { Nuevo min }
    if (fitness>bestfit.fitness) then Begin
      bestfit.chrom:=chrom;
      bestfit.fitness:=fitness;
      bestfit.generation:=gen;
      bestfit.b:=b;
      bestfit.x1:=x1;
      bestfit.x2:=x2;
      bestfit.x3:=x3;
      bestfit.x4:=x4;
    End;
  end;

  { Calcula el porcentaje }

  if max>maxv then maxv:=max;
  if min<minv then minv:=min;
  avg:=sumfitness/popsiz;
end;

```

Programa INITIAL6.SGA

```
{ initial6.sga: contiene initdata, initpop, initreport, initialize }
```

```
procedure initdata;
```

```
{ Introducción de datos y configuraciones }
```

```
var ch : char; j : integer;
```

```
begin
```

```
  clrscr;
```

```
  writeln('-----');
```

```
  writeln('ALGORITMO GENETICO SIMPLE - SGA');
```

```
  writeln(' (c) David Edward Goldberg 1986 ');
```

```
  writeln(' Modificado por Fco. Alberto Alonso F.');
```

```
  writeln('-----');
```

```
  writeln;
```

```
  write('Introduzca el tamaño de la población ---> '); readln(popsize);
```

```
  write('Introduzca la longitud de cromosomas ---> '); readln(lchrom);
```

```
  write('Introduzca el máximo de generaciones ---> '); readln(maxgen);
```

```
  write('Introduzca la probabilidad de cruza ---> '); readln(pcross);
```

```
  write('Introduzca la probabilidad de mutación ---> '); readln(pmutation);
```

```
  writeln;
```

```
{ Inicializa el generador de números aleatorios }
```

```
randomize;
```

```
{ Inicializa los contadores }
```

```
nmutation:=0;
```

```
ncross:=0;
```

```
bestfit.fitness:=0.0;
```

```
bestfit.generation:=0;
```

```
end;
```

```
procedure initreport;
```

```
{ Reporte inicial }
```

```
begin
```

```
(* write(1st,Char(15)); *)
```

```
  writeln('-----');
```

```
  writeln(' ALGORITMO GENETICO SIMPLE - SGA');
```

```
  writeln(' (c) David Edward Goldberg 1986 ');
```

```
  writeln(' Modificado por Fco. Alberto Alonso F.');
```

```
  writeln('-----');
```

```
(* writeln(1st); *)
```

```
  writeln(' Parámetros del SGA :');
```

```
  writeln(' -----');
```

```
  writeln(' Tamaño de la población (popsize) = ',popsize);
```

```

writeln(' Longitud del cromosoma (lchrom)    = ',lchrom);
writeln(' Máximo número de generaciones (maxgen) = ',maxgen);
writeln(' Probabilidad de cruza (pcross)    = ',pcross);
writeln(' Probabilidad de mutación (pmutation) = ',pmutation);
writeln;
(* writeln(lst); *)
writeln(' Generación estática inicial:');
writeln(' -----');
(* writeln(lst); *)
writeln;
writeln(' Aptitud máxima de la población inicial = ',max);
writeln(' Porcentaje de aptitud de la población inicial = ',avg);
writeln(' Aptitud mínima de la población inicial = ',min);
writeln(' Suma de aptitudes de la población inicial = ',sumfitness);
(* writeln(lst); *)
writeln;
end;

procedure initpop;

{ Inicializa una población aleatoria }

var j, j1:integer;

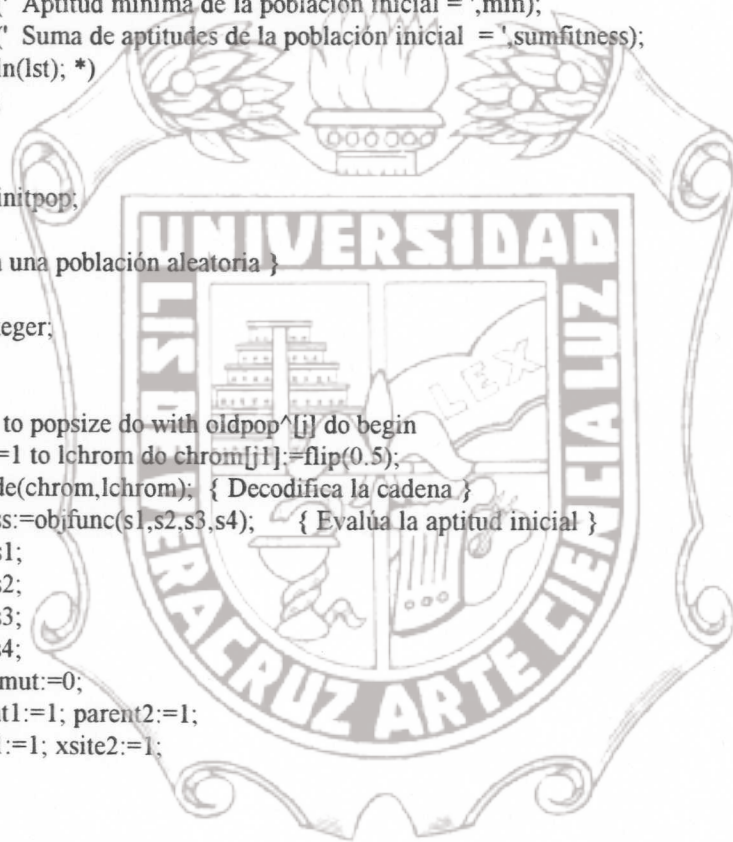
begin
  for j:=1 to popsize do with oldpop^[j] do begin
    for j1:=1 to lchrom do chrom[j1]:=flip(0.5);
    decode(chrom,lchrom); { Decodifica la cadena }
    fitness:=objfunc(s1,s2,s3,s4); { Evalúa la aptitud inicial }
    x1:=s1;
    x2:=s2;
    x3:=s3;
    x4:=s4;
    placemut:=0;
    parent1:=1; parent2:=1;
    xsite1:=1; xsite2:=1;
  end;
end;

procedure initialize;

{ Inicialización del Coordinador }

begin
  initdata;
  initpop;
  maxv:=0.0; minv:=1e8;
  max:=0.0; min:=1e8;
  statistics(popsiz,max,avg,min,sumfitness,oldpop);
  initreport;
end;

```



Programa INITIAL9.SGA

```

{ initial9.sga: contiene initdata, initpop, initreport, initialize }

procedure initdata;

{ Introducción de datos y configuraciones }

var ch : char; j : integer;
begin
  clrscr;
  writeln('-----');
  writeln('ALGORITMO GENETICO SIMPLE - SGA');
  writeln(' (c) David Edward Goldberg 1986 ');
  writeln(' Modificado por Fco. Alberto Alonso F. ');
  writeln('-----');
  writeln;
  write('Introduzca el tamaño de la población ---> '); readln(popsiz);
  write('Introduzca la longitud de cromosomas ---> '); readln(lchrom);
  write('Introduzca el máximo de generaciones ---> '); readln(maxgen);
  write('Introduzca la probabilidad de cruza ---> '); readln(pcross);
  write('Introduzca la probabilidad de mutación ---> '); readln(pmutation);
  writeln;

{ Inicializa el generador de números aleatorios }

randomize;

{ Inicializa los contadores }

nmutation:=0;
ncross:=0;
bestfit.fitness:=0.0;
bestfit.generation:=0;
end;

procedure initreport;

{ Reporte inicial }

begin

(* write(1st,Char(15)); *)
  writeln('-----');
  writeln(' ALGORITMO GENETICO SIMPLE - SGA');
  writeln(' (c) David Edward Goldberg 1986 ');
  writeln(' Modificado por Fco. Alberto Alonso F. ');
  writeln('-----');
  (* writeln(1st); *)
  writeln(' Parámetros del SGA :');
  writeln(' -----');
  writeln(' Tamaño de la población (popsiz) = ',popsiz);

```

```

writeln(' Longitud del cromosoma (lchrom)      = ',lchrom);
writeln(' Máximo número de generaciones (maxgen) = ',maxgen);
writeln(' Probabilidad de cruza (pcross)      = ',pcross);
writeln(' Probabilidad de mutación (pmutation) = ',pmutation);
writeln;
(* writeln(1st); *)
writeln(' Generación estática inicial:');
writeln(' -----');
(* writeln(1st); *)
writeln;
writeln(' Aptitud máxima de la población inicial = ',max);
writeln(' Porcentaje de aptitud de la población inicial = ',avg);
writeln(' Aptitud mínima de la población inicial = ',min);
writeln(' Suma de aptitudes de la población inicial = ',sumfitness);
(* writeln(1st); *)
writeln;
end;

procedure initpop;

{ Inicializa una población aleatoria }

var j, j1:integer;

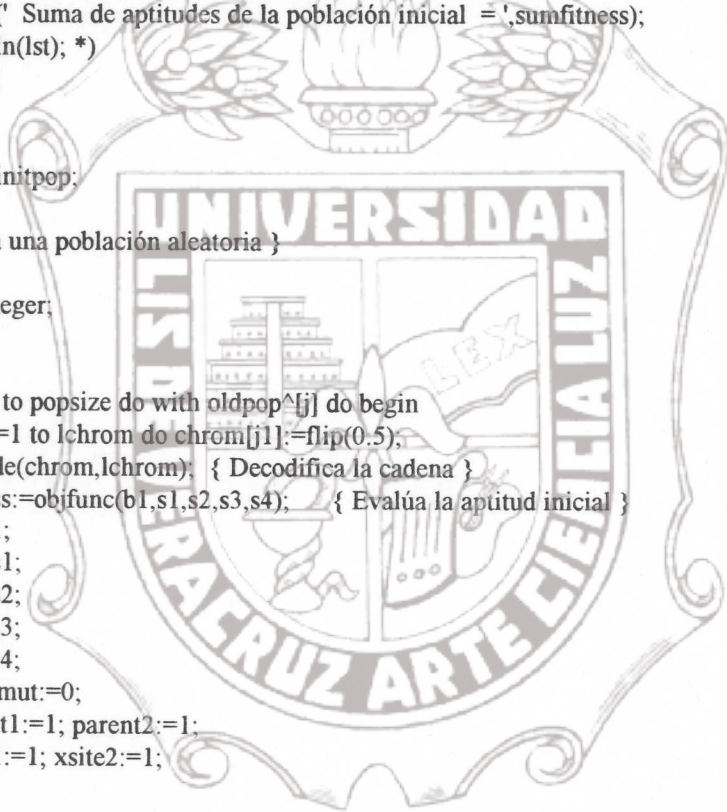
begin
  for j:=1 to popsize do with oldpop^[j] do begin
    for j1:=1 to lchrom do chrom[j1]:=flip(0.5);
    decode(chrom,lchrom); { Decodifica la cadena }
    fitness:=objfunc(b1,s1,s2,s3,s4); { Evalúa la aptitud inicial }
    b:=b1;
    x1:=s1;
    x2:=s2;
    x3:=s3;
    x4:=s4;
    placemut:=0;
    parent1:=1; parent2:=1;
    xsite1:=1; xsite2:=1;
  end;
end;

procedure initialize;

{ Inicialización del Coordinador }

begin
  initdata;
  initpop;
  maxv:=0.0; minv:=1e8;
  max:=0.0; min:=1e8;
  statistics(popsiz,max,avg,min,sumfitness,oldpop);
  initreport;
end;

```



Programa REPORT6.SGA

```

{ report.sga : contiene writechrom, report }

procedure writechrom(chrom:chromosome; lchrom:integer);

{ Escribe un cromosoma como una cadena de 1's (verdadero's) y 0's (falso's) }

var j:integer;
begin
  for j:=lchrom downto 1 do
    if chrom[j] then write('1')
    else write('0');
  end;

procedure report(gen:integer);

{ Escribe el reporte de la población }

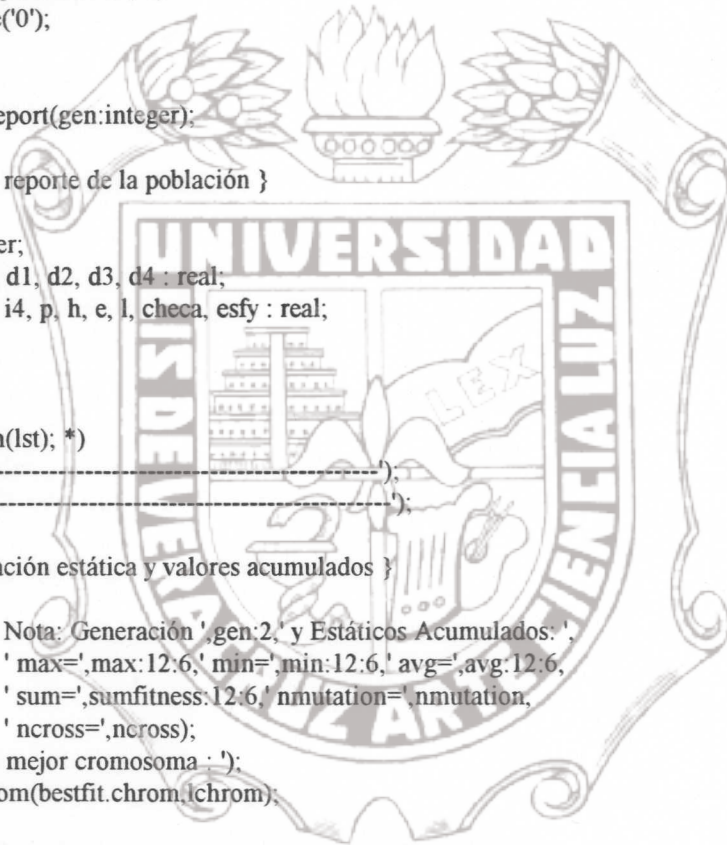
var j : integer;
    volumen, d1, d2, d3, d4 : real;
    i1, i2, i3, i4, p, h, e, l, checa, esfy : real;

begin
  (* writeln(1st); *)
  write('-----');
  writeln('-----');

  { Generación estática y valores acumulados }

  writeln(' Nota: Generación ',gen:2,' y Estáticos Acumulados: ',
    ' max=',max:12:6,' min=',min:12:6,' avg=',avg:12:6,
    ' sum=',sumfitness:12:6,' nmutation=',nmutation,
    ' ncross=',ncross);
  write('El mejor cromosoma : ');
  writechrom(bestfit.chrom,lchrom);
  writeln;
  writeln('fué hallado en la generación : ',bestfit.generation);
  writeln('y tiene un valor de aptitud de : ',bestfit.fitness:20:10);
  writeln('Su decodificación es : ');
  writeln('x1 = ',bestfit.x1:8:3,' x2 = ',bestfit.x2:8:3);
  writeln('x3 = ',bestfit.x3:8:3,' x4 = ',bestfit.x4:8:3);
  with bestfit do begin
    volumen:=(Pi*120.0/36.0)*(x1*x1+2.0*x2*x2+2.0*x3*x3+x4*x4+x1*x2+x2*x3+x3*x4);
    p:=400000.0; h:=20.0; e:=30e6;
    l:=120.0; esfy:=60000.0;
    i1:=(Pi/64.0)*x1*x1*x1*x1;
    i2:=(Pi/64.0)*x2*x2*x2*x2;
    i3:=(Pi/64.0)*x3*x3*x3*x3;
    i4:=(Pi/64.0)*x4*x4*x4*x4;
    checa:=(p*p*p*h*h*h*h*h)/(e*e*e*i2*i3*i4);
    checa:=checa-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
  end;

```




```

checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
checa:=abs(checa);
writeln('Volumen = ',volumen:10:3);
end;
writeln('Restricción = ',checa);
d1:=2.914; d2:=3.967; d3:=4.601; d4:=4.771;
i1:=(Pi/64.0)*d1*d1*d1*d1;
i2:=(Pi/64.0)*d2*d2*d2*d2;
i3:=(Pi/64.0)*d3*d3*d3*d3;
i4:=(Pi/64.0)*d4*d4*d4*d4;
checa:=(p*p*p*h*h*h*h*h*h)/(e*e*e*i2*i3*i4);
checa:=checa-((2.0*p*p*h*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
checa:=abs(checa);
writeln('Restricción original = ',checa);
write('-----');
writeln('-----');
end;

```



Programa REPORT7.SGA

```

{ report.sga : contiene writechrom, report }

procedure writechrom(chrom:chromosome; lchrom:integer);

{ Escribe un cromosoma como una cadena de 1's (verdadero's) y 0's (falso's) }

var j:integer;
begin
  for j:=lchrom downto 1 do
    if chrom[j] then write('1')
    else write('0');
  end;

procedure report(gen:integer);

{ Escribe el reporte de la población }

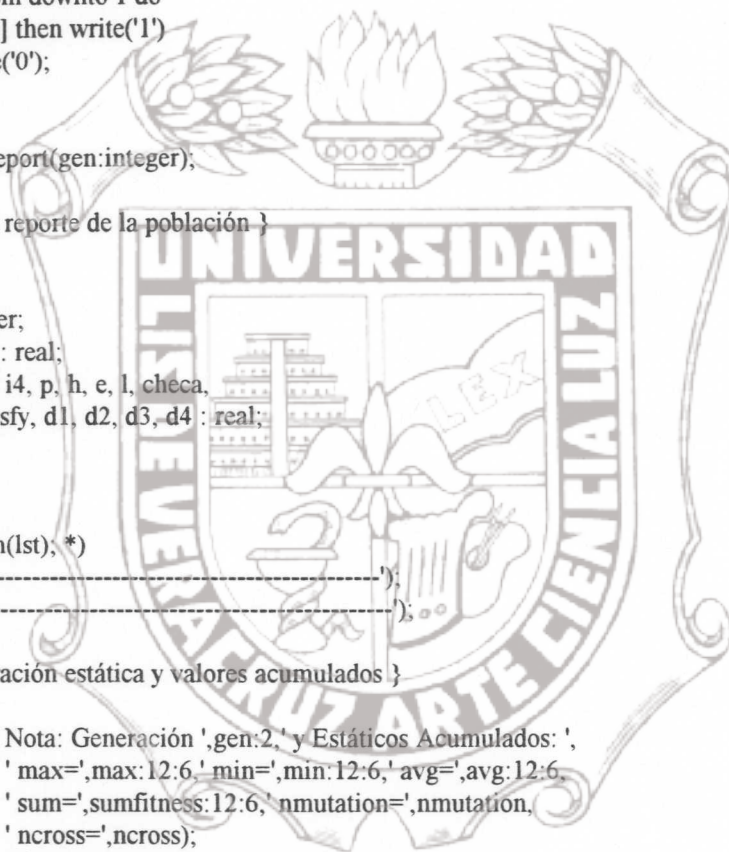
var j : integer;
    volumen : real;
    i1, i2, i3, i4, p, h, e, l, checa,
    checa2, esfy, d1, d2, d3, d4 : real;

begin
  (* writeln(1st); *)
  write('-----');
  writeln('-----');

  { Generación estática y valores acumulados }

  writeln(' Nota: Generación ',gen:2,' y Estáticos Acumulados: ',
    ' max=',max:12:6,' min=',min:12:6,' avg=',avg:12:6,
    ' sum=',sumfitness:12:6,' nmutation=',nmutation,
    ' ncross=',ncross);
  write('El mejor cromosoma : ');
  writechrom(bestfit.chrom,lchrom);
  writeln;
  writeln('fué hallado en la generación : ',bestfit.generation);
  writeln('y tiene un valor de aptitud de : ',bestfit.fitness:20:10);
  writeln('Su decodificación es : ');
  writeln('x1 = ',bestfit.x1:8:3,' x2 = ',bestfit.x2:8:3);
  writeln('x3 = ',bestfit.x3:8:3,' x4 = ',bestfit.x4:8:3);
  with bestfit do begin
    volumen:=(120.0/9.0)*(x1*x1+2.0*x2*x2+2.0*x3*x3+x4*x4+x1*x2+x2*x3+x3*x4);
    writeln('Volumen = ',volumen:10:3);
    p:=400000.0; h:=20.0; e:=30e6;
    l:=120.0; esfy:=60000.0;
    i1:=(1.0/12.0)*x1*x1*x1*x1;
    i2:=(1.0/12.0)*x2*x2*x2*x2;

```



```

i3:=(1.0/12.0)*x3*x3*x3*x3;
i4:=(1.0/12.0)*x4*x4*x4*x4;
checa:=(p*p*p*h*h*h*h*h*h)/(e*e*e*i2*i3*i4);
checa:=checa-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
checa:=abs(checa);
end;
writeln('Restricción =',checa);
d1:=2.582; d2:=3.475; d3:=4.031; d4:=4.180;
i1:=(1.0/12.0)*d1*d1*d1*d1;
i2:=(1.0/12.0)*d2*d2*d2*d2;
i3:=(1.0/12.0)*d3*d3*d3*d3;
i4:=(1.0/12.0)*d4*d4*d4*d4;
checa:=(p*p*p*h*h*h*h*h)/(e*e*e*i2*i3*i4);
checa:=checa-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
checa:=abs(checa);
writeln('Restricción original =',checa);
write('-----');
writeln('-----');
end;

```



Programa REPORT8.SGA

```

{ report.sga : contiene writechrom, report }

procedure writechrom(chrom:chromosome; lchrom:integer);

{ Escribe un cromosoma como una cadena de 1's (verdadero's) y 0's (falso's) }

var j:integer;
begin
  for j:=lchrom downto 1 do
    if chrom[j] then write('1')
    else write('0');
  end;

procedure report(gen:integer);

{ Escribe el reporte de la población }

var j : integer;
    volumen : real;
    i1, i2, i3, i4, p, h, e, l, checa,
    checa2, esfy, d1, d2, d3, d4 : real;

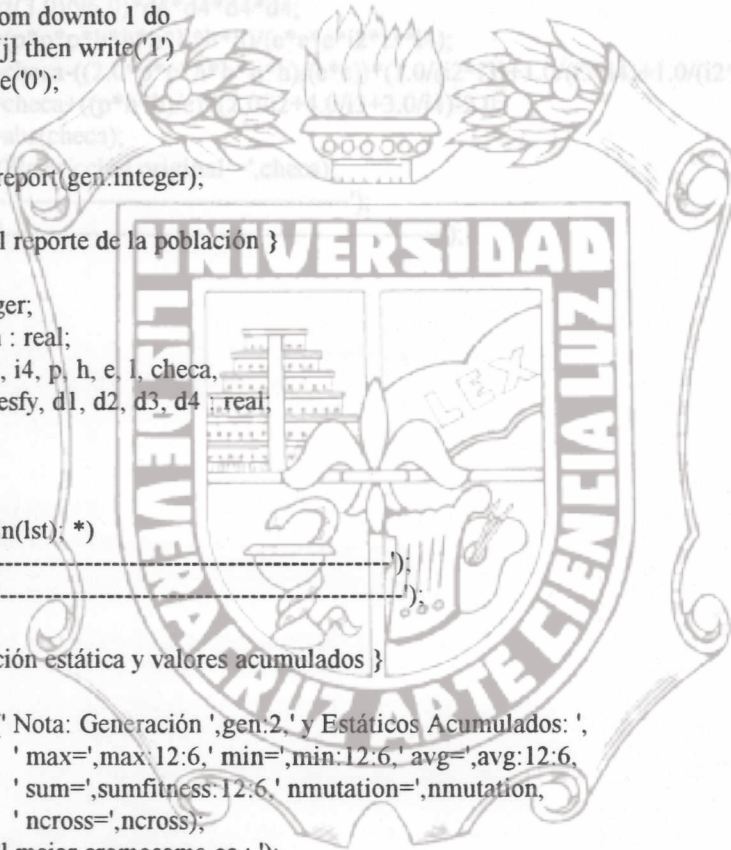
begin
  (* writeln(lst); *)
  write('-----');
  writeln('-----');

  { Generación estática y valores acumulados }

  writeln(' Nota: Generación ',gen:2,' y Estáticos Acumulados: ',
    ' max=',max:12:6,' min=',min:12:6,' avg=',avg:12:6,
    ' sum=',sumfitness:12:6,' nmutation=',nmutation,
    ' ncross=',ncross);

  write('El mejor cromosoma es : ');
  writechrom(bestfit.chrom,lchrom);
  writeln;
  writeln('Fue encontrado en la gneración : ',bestfit.generation);
  writeln('y tiene un valor de aptitud de : ',bestfit.fitness:20:10);
  writeln('Su decodificación es : ');
  writeln('x1 = ',bestfit.x1:8:3,' x2 = ',bestfit.x2:8:3);
  writeln('x3 = ',bestfit.x3:8:3,' x4 = ',bestfit.x4:8:3);
  with bestfit do begin
    volumen:=(120.0*sqrt(3.0)/36.0)*(x1*x1+2.0*x2*x2+2.0*x3*x3+x4*x4+x1*x2+x2*x3+x3*x4);
    writeln('Volumen = ',volumen:10:3);
    p:=400000.0; h:=20.0; e:=30e6;
    l:=120.0; esfy:=60000.0;
    i1:=(sqrt(3.0)/96.0)*x1*x1*x1*x1;
    i2:=(sqrt(3.0)/96.0)*x2*x2*x2*x2;

```



```

i3:=(sqrt(3.0)/96.0)*x3*x3*x3*x3;
i4:=(sqrt(3.0)/96.0)*x4*x4*x4*x4;
checa:=(p*p*p*h*h*h*h*h*h)/(e*e*e*i2*i3*i4);
checa:=checa-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
checa:=abs(checa);
end;
writeln('Restricción =',checa);
d1:=3.924; d2:=5.092; d3:=5.910; d4:=6.130;
i1:=(sqrt(3.0)/96.0)*d1*d1*d1*d1;
i2:=(sqrt(3.0)/96.0)*d2*d2*d2*d2;
i3:=(sqrt(3.0)/96.0)*d3*d3*d3*d3;
i4:=(sqrt(3.0)/96.0)*d4*d4*d4*d4;
checa:=(p*p*p*h*h*h*h*h*h)/(e*e*e*i2*i3*i4);
checa:=checa-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
checa:=abs(checa);
writeln('Restricción original =',checa);
write('-----');
writeln('-----');
end;

```



Programa REPORT9.SGA

```

{ report.sga : contiene writechrom, report }

procedure writechrom(chrom:chromosome; lchrom:integer);

{ Escribe un cromosoma como una cadena de 1's (verdadero's) y 0's (falso's) }

var j:integer;
begin
  for j:=lchrom downto 1 do
    if chrom[j] then write('1')
    else write('0');
  end;

procedure report(gen:integer);

{ Escribe el reporte de la población }

var j : integer;
    volumen : real;
    i1, i2, i3, i4, p, h, e, l, checa, checa3, volumen2,
    b1, checa2, esfy, d1, d2, d3, d4 : real;

begin

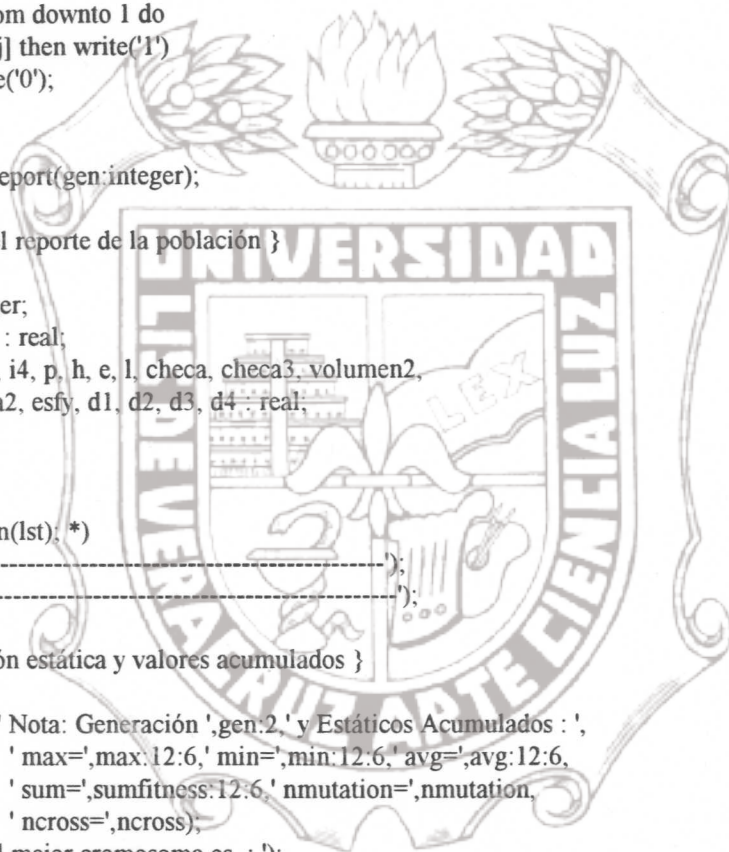
(* writeln(lst); *)
  write('-----');
  writeln('-----');

{ Generación estática y valores acumulados }

  writeln(' Nota: Generación ',gen:2,' y Estáticos Acumulados : ',
    ' max=',max:12:6,' min=',min:12:6,' avg=',avg:12:6,
    ' sum=',sumfitness:12:6,' nmutation=',nmutation,
    ' ncross=',ncross);

  write('El mejor cromosoma es : ');
  writechrom(bestfit.chrom,lchrom);
  writeln;
  writeln('fué hallado en la generación : ',bestfit.generation);
  writeln('y tiene un valor de aptitud de : ',bestfit.fitness:20:10);
  writeln('Su decodificación es : ');
  writeln('x1 = ',bestfit.x1:8:3,' x2 = ',bestfit.x2:8:3);
  writeln('x3 = ',bestfit.x3:8:3,' x4 = ',bestfit.x4:8:3);
  writeln(' b = ',bestfit.b:8:3);
  with bestfit do begin
    volumen:=(b*120.0/9.0)*(x1+2.0*x2+2.0*x3+x4+sqrt(x1*x2)+sqrt(x2*x3)+sqrt(x3*x4));
    writeln('Volumen = ',volumen:10:3);
    p:=400000.0; h:=20.0; e:=30e6;
    l:=120.0; esfy:=60000.0;
    i1:=b*x1*x1*x1/12.0;
    i2:=b*x2*x2*x2/12.0;

```



```

i3:=b*x3*x3*x3/12.0;
i4:=b*x4*x4*x4/12.0;
checa:=(p*p*p*h*h*h*h*h)/(e*e*e*i2*i3*i4);
checa:=checa-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
checa:=abs(checa);
i1:=x1*b*b*b/12.0;
i2:=x2*b*b*b/12.0;
i3:=x3*b*b*b/12.0;
i4:=x4*b*b*b/12.0;
checa3:=(p*p*p*h*h*h*h*h)/(e*e*e*i2*i3*i4);
checa3:=checa3-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
checa3:=checa3+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
checa3:=abs(checa3);
end;
writeln('Restricción 1 =',checa);
writeln('Restricción 2 =',checa3);
b1:=3.903; d1:=1.708; d2:=3.231; d3:=4.127; d4:=4.403;
i1:=b1*d1*d1*d1/12.0;
i2:=b1*d2*d2*d2/12.0;
i3:=b1*d3*d3*d3/12.0;
i4:=b1*d4*d4*d4/12.0;
checa:=(p*p*p*h*h*h*h*h)/(e*e*e*i2*i3*i4);
checa:=checa-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
checa:=checa+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
checa:=abs(checa);
i1:=d1*b1*b1*b1/12.0;
i2:=d2*b1*b1*b1/12.0;
i3:=d3*b1*b1*b1/12.0;
i4:=d4*b1*b1*b1/12.0;
checa3:=(p*p*p*h*h*h*h*h)/(e*e*e*i2*i3*i4);
checa3:=checa3-((2.0*p*p*h*h*h*h)/(e*e))*(1.0/(i2*i3)+1.0/(i3*i4)+1.0/(i2*i4));
checa3:=checa3+((p*h*h)/e)*(2.0/i2+4.0/i3+3.0/i4)-2.0;
checa3:=abs(checa3);
volumen2:=(b1*120.0/9.0)*(d1+2.0*d2+2.0*d3+d4+sqrt(d1*d2)+sqrt(d2*d3)+sqrt(d3*d4));
writeln('Restricción original 1 =',checa);
writeln('Restricción original 2 =',checa3);
writeln('Volumen Original =',volumen2:10:3);
write('-----');
writeln('-----');
end;

```

Programa TRIOPS.SGA

```

{ triops.sga }
{ 3-operadores: Reproducción (select), Cruza (crossover),
  y Mutación (mutation) }

procedure reset;
Var i, rand1, rand2, temp1 : integer;
Begin
  for i:=1 to popsize do tournlist^[i]:=i;
  for i:=1 to popsize do Begin
    rand1:=rnd(i,popsize);
    rand2:=rnd(i,popsize);
    temp1:=tournlist^[rand1];
    tournlist^[rand1]:=tournlist^[rand2];
    tournlist^[rand2]:=temp1;
  End;
End;

procedure preselect;
Begin
  reset;
  tournpos:=1;
End;

function select(popsiz:integer; sumfitness:real;
  var pop:base11):integer;
{ Usa selección mediante torneo }
var pick, winner:integer;
  i : integer; { índice de la población }

begin
  if (popsiz-tournpos) < 2 then Begin
    reset;
    tournpos:=1;
  End;
  winner:=tournlist^[tournpos];

  for i:=1 to 1 do Begin
    pick:=tournlist^[i+tournpos];
    if (oldpop^[pick].fitness > oldpop^[winner].fitness) then winner:=pick;
  End;

  tournpos:=tournpos+2;
  select:=winner;
end;

procedure mutation(var chrom:chromosome; pmutation:real;
  var nmutation, where:integer; lchrom:integer);
{ Muta un alelo, cuenta número de mutaciones }
var mutate:boolean;
  k : integer;

```



Instituto de Ingeniería
 Universidad Veracruzana


```

begin
  if flip(pmutation) then
    Begin
      where:=rnd(1,lchrom);
      nmutation:=nmutation+1;
      for k:=1 to lchrom do
        if (where=k) then chrom[k]:=not chrom[k];
      End;
    end;

procedure crossover(var parent1, parent2, child1, child2 : chromosome;
  var lchrom, ncross, jcross1, jcross2 : integer;
  pcross : real);
{ Cruza 2 cadenas padres, lo pone en 2 cadenas hijos }
var j : integer;
  templ : integer;
begin
  if flip(pcross) then begin { Hace la cruza con p(cross) }
    jcross1:=rnd(1,lchrom); { Cruza entre 1 y l-1 }
    if (jcross1 < lchrom) then jcross2:=rnd(jcross1+1,lchrom)
    else Begin
      jcross2:=rnd(1,jcross1-1);
      templ:=jcross1;
      jcross1:=jcross2;
      jcross2:=templ;
    End;
    ncross:=ncross+1; { pone el sitio de cruza y forza la mutación }

    for j:=lchrom downto lchrom-jcross1 do
      Begin
        child1[j]:=parent1[j];
        child2[j]:=parent2[j];
      End;
    for j:=lchrom-jcross1-1 downto lchrom-jcross2 do
      Begin
        child1[j]:=parent2[j];
        child2[j]:=parent1[j];
      End;
    End
  Else
    Begin
      for j:=1 to lchrom do Begin
        child1[j]:=parent1[j];
        child2[j]:=parent2[j];
      End;
      jcross1:=1; jcross2:=1;
    End;
  end;
end;

```

Programa GENERA6.SGA

```
{ generate.sga }
```

```
procedure generation;
```

```
{ Crea una nueva generación a través de la selección, cruce y mutación }
```

```
var j, mate1, mate2, jcross1, jcross2, where : integer;
```

```
begin
```

```
  j:=1;
```

```
  preselect;
```

```
  repeat { selecciona, cruza y muta hasta que la nueva población es hallada }
```

```
    mate1:=select(popsiz, sumfitness, oldpop); { elige dos pares de compañeros }
```

```
    mate2:=select(popsiz, sumfitness, oldpop);
```

```
  { Cruza y mutación - mutación implantado dentro de la cruce }
```

```
  crossover(oldpop^[mate1].chrom, oldpop^[mate2].chrom,
```

```
            newpop^[j].chrom, newpop^[j+1].chrom,
```

```
            lchrom, ncross, jcross1, jcross2, pcross);
```

```
  mutation(newpop^[j].chrom, pmutation, nmutation, where, lchrom);
```

```
  mutation(newpop^[j+1].chrom, pmutation, nmutation, where, lchrom);
```

```
  { Decodifica la cadena, evalúa la aptitud string y registra el parentesco en ambos hijos }
```

```
  with newpop^[j] do begin
```

```
    decode(chrom, lchrom);
```

```
    fitness:=objfunc(s1, s2, s3, s4);
```

```
    x1:=s1;
```

```
    x2:=s2;
```

```
    x3:=s3;
```

```
    x4:=s4;
```

```
    xsite1:=jcross1;
```

```
    xsite2:=jcross2;
```

```
    parent1:=mate1;
```

```
    parent2:=mate2;
```

```
    placemut:=lchrom-where;
```

```
  end;
```

```
  with newpop^[j+1] do begin
```

```
    decode(chrom, lchrom);
```

```
    fitness:=objfunc(s1, s2, s3, s4);
```

```
    x1:=s1;
```

```
    x2:=s2;
```

```
    x3:=s3;
```

```
    x4:=s4;
```

```
    parent1:=mate1;
```

```
    parent2:=mate2;
```

```
    xsite1:=jcross1;
```

```
    xsite2:=jcross2;
```

```
    placemut:=lchrom-where;
```

end;

{ Incrementa el índice de población }

j:=j+2;
until j > popsize

end;



Instituto de Ingeniería
Universidad Veracruzana

Programa GENERA9.SGA

```

{ generate.sga }

procedure generation;
{ Crea una nueva generación a través de la selección, cruza y mutación }

var j, mate1, mate2, jcross1, jcross2, where : integer;
begin
  j:=1;
  preselect;
  repeat { selecciona, cruza y muta hasta que la nueva población es hallada }
  mate1:=select(popsiz, sumfitness, oldpop); { elige dos pares de compañeros }
  mate2:=select(popsiz, sumfitness, oldpop);

  { Cruza y mutación - mutación implantado dentro de la cruza }

  crossover(oldpop^[mate1].chrom, oldpop^[mate2].chrom,
    newpop^[j ].chrom, newpop^[j + 1].chrom,
    lchrom, ncross, jcross1, jcross2, pcross);

  mutation(newpop^[j].chrom, pmutation, nmutation, where, lchrom);
  mutation(newpop^[j+1].chrom, pmutation, nmutation, where, lchrom);

  { Decodifica la cadena, evalúa la aptitud string y registra el parentesco en ambos hijos }

  with newpop^[j ] do begin
    decode(chrom, lchrom);
    fitness:=objfunc(b1, s1, s2, s3, s4);
    b:=b1;
    x1:=s1;
    x2:=s2;
    x3:=s3;
    x4:=s4;
    xsite1:=jcross1;
    xsite2:=jcross2;
    parent1:=mate1;
    parent2:=mate2;
    placemut:=lchrom-where;
  end;
  with newpop^[j+1] do begin
    decode(chrom, lchrom);
    fitness:=objfunc(b1, s1, s2, s3, s4);
    b:=b1;
    x1:=s1;
    x2:=s2;
    x3:=s3;
    x4:=s4;
    parent1:=mate1;
    parent2:=mate2;
    xsite1:=jcross1;

```

```
xsite2:=jcross2;  
placemut:=lchrom-where;  
end;
```

```
{ Incrementa el índice de población }
```

```
j:=j+2;  
until j > popsize
```

```
end;
```



BIBLIOGRAFIA

- [1] Holland, J.H., "Adaptation in Natural and Artificial Systems", *University of Michigan Press*. 1975, 222 p.
- [2] Bagley, J.D., "The behavior of adaptive systems which employ genetic and correlation algorithms.", *Doctoral dissertation, University of Michigan*. 1967.
- [3] Rosenberg, R.S., "Simulation of genetic populations with biochemical properties", *Doctoral dissertation, University of Michigan*. 1967.
- [4] Hollstien, R.B., "Artificial genetic adaptation in computer control systems", *Doctoral dissertation, University of Michigan*. 1971.
- [5] Hadamard, J., "The psychology of invention in the mathematical field", *Princeton University Press*. 1949
- [6] Ribeiro Filho, J.L.; Treleven Ph. C., and Alippi, C., "Genetic-Algorithm Programming Environments", *IEEE Computer*, June 1994, pp. 28-43.
- [7] Euler, L., "Methodus inveniendi lineas curvasmaximi minimive proprietate gaudentes. . .", Apéndice I, "De curvis elasticis", Bousquet, Lausanne and Geneva, 1744. (Traducción al inglés: Oldfather, W. A., Ellis, C. A. y Brown, D. M., *Isis*, Vol. 20, 1933, pp. 72-160).
- [8] Euler, L., "Sur la force des colonnes", *Histoire de L'Académie Royal des Sciences et Belles Lettres*, 1757, publicado en las *Memoires* de la Academia, Vol. 13, Berlín, 1759, pp. 252-82.
- [9] Lamarle, A. H. E., "Mémoire sur la flexion du bois", *Annales des Travaux Publicques de Belgique*, parte 1, Vol. 3, 1845, pp. 1-64, y parte 2, Vol. 4, 1846, pp. 1-36.
- [10] Considère, A., "Résistance des Pièces comprimées", *Congrès International des Procédés de de Construction*, París, septiembre 9-14 de 1889, memorias publicadas por Librairie Polytechnique, París, Vol. 3, 1891, p. 371.
- [11] Engesser, F., "Über die Knickfestigkeit gerader Stäbe", *Zeitschrift für Architektur und Ingenieurwesen*, Vol. 35, No. 4, 1889, pp. 455-562.
- [12] Shanley, F. R., "Inelastic column theory", *Journal of the Aeronautical Sciences*, Vol. 14, No. 5, mayo de 1947, pp. 261-7.
- [13] Dinnik, A. N., "Design of columns of varying cross-section", *Transactions ASME* 54, 1932.

- [14] Keller, J. B., "The shape of the strongest column", *Arch. Ration. Mech. Anal.* **5**, pp. 275-85, 1960.
- [15] Tadjbakhsh, T. & Keller, J. B., "Strongest column and isoparametric inequalities for eigenvalues", *J. Appl. Math.* **9**, 159-64, 1962.
- [16] Taylor, J. E., "The strongest column: an energy approach", *J. Appl. Mech., ASME* **34**, pp. 486-7, 1962.
- [17] Spillers, W. R. & Levy, R., "Optimal design for plate buckling", *J. Struct. Engng, ASCE* **116**, pp. 850-8, 1990.
- [18] Spillers, W. R. & Levy, R., "Optimal design for axisymmetric cylindrical shell buckling", *J. Engng Mech., ASCE* **115**, pp. 1683-90, 1989.
- [19] Fu, K. C. & Ren, D. "Optimization of Axially Loaded Non-Prismatic Column", *Computers and Structures*, Vol. 43, No. 1, pp- 159-62, 1992.
- [20] Goldberg, David E. **Genetic Algorithms in Search, Optimization and Machine Learning**. Addison-Wesley Publishing Co., Inc., Reading, Mass. 412 p.
- [21] Porter, Kent, "Handling Huge Arrays", *Dr. Dobb's Journal of Software Tools for the Professional Programmer*, Marzo 1988, Volumen 13, Número 3, pp. 60-63.

