



**Escola Politècnica Superior
de Castelldefels**

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE CARRERA

TÍTULO DEL TFC: Optimización global con algoritmos genéticos

**TITULACIÓN: Ingeniería Técnica en Telecomunicaciones, especialidad en
Telemática**

AUTOR: Félix Carretero López

DIRECTOR: Luis Delgado Muñoz

FECHA: 11 de Noviembre de 2010

Título: Optimización global con algoritmos genéticos

Autor: Félix Carretero López

Director: Luis Delgado Muñoz

Fecha: 11 de Noviembre de 2010

Resumen

El objetivo principal de este TFC es dar a conocer al lector el mundo de los métodos de optimización global mediante los algoritmos genéticos.

Para alcanzar este objetivo empezaremos por exponer las bases teóricas en las que se fundamentan y a continuación las intentaremos corroborar empíricamente mediante un conjunto de pruebas simples. A partir de los resultados y con todo lo aprendido intentaremos resolver un problema real más complejo.

En concreto intentaremos equilibrar la carga de las bodegas de un avión. Al tratarse de un problema más complicado sufriremos los contratiempos, típicos en estos casos, de partir de una idea y tener que ir la modelando y corrigiendo hasta conseguir un sistema capaz de resolver el problema.

Title: Global Optimization with Genetic Algorithms

Author: Félix Carretero López

Director: Luis Delgado Muñoz

Date: November 11, 2010

Overview

The main goal of this TFC is to introduce the reader into the world of global optimization methods with Genetic Algorithms.

In order to achieve this goal, we will begin by explaining the theoretical basis they are based upon, next we will try to corroborate them empirically by means of some simple tests. Taking into account the obtained results and all we have learnt we will try to solve a more complex and real problem.

Specifically, we will try to equilibrate the load of the decks of an airplane. Due to the complexity of our problem we will find some trouble, common in those cases, like beginning with an idea and having to change it and model it until we achieve a system able to solve the problem.

ÍNDICE

INTRODUCCIÓN	1
CAPÍTULO 1. INTRODUCCIÓN A LOS MÉTODOS DE OPTIMIZACIÓN	2
1.1. Definiciones básicas	2
1.2. Métodos tradicionales	5
1.3. Métodos Modernos.....	6
CAPÍTULO 2. INTRODUCCIÓN A LOS ALGORITMOS GENÉTICOS	9
2.1. Computación evolutiva	9
2.1.1. Programación Evolutiva	10
2.1.2. Estrategias Evolutivas.....	10
2.1.3. Programación Genética.....	10
2.2. Algoritmos Genéticos.....	11
2.2.1. Algoritmo Genético Simple o Canónico	12
2.2.2. Población inicial [Anexo A, apartados A.1.1 y A.1.2]	13
2.2.3. Función de Adaptación o <i>Fitness</i> [Anexo A, apartado A.1.3].....	14
2.2.4. Función de Selección [Anexo A, apartado A.1.4].....	14
2.2.5. Función de cruce [Anexo A, apartado A.1.5].....	16
2.2.6. Función de mutación [Anexo A, Apartado A.1.6].....	17
2.2.7. Función de inserción [Anexo A, apartado A.1.7]	18
2.2.8. Criterio de parada [Anexo A, apartado A.1.8].....	18
CAPÍTULO 3. IMPLEMENTACIÓN Y PRUEBAS BÁSICAS.....	19
3.1. Implementación de un Algoritmo Genético.....	19
3.2. Funciones seleccionadas.....	20
3.3. Test y resultados	21
3.3.1. Consideraciones previas	21
3.3.2. Test de población.....	23
3.3.3. Test de número de generaciones	26
3.3.4. Test de probabilidad de cruce	27
3.3.5. Test de probabilidad de mutación.....	28
3.3.6. Test de mutación y cruce simultáneamente	29
3.3.7. Test de <i>Sharing</i>	30
3.4. Mejoras.....	32
CAPÍTULO 4: APLICACIÓN PRÁCTICA	35
4.1. Descripción del problema.	35
4.1.1. Efectos negativos de la distribución de la carga.....	35
4.1.2. Espacio de búsqueda del problema.....	37

4.2. Por qué utilizar Algoritmos Genéticos.....	39
4.3. Especificaciones del código	40
4.3.1. Codificación utilizada y métodos de selección, cruce y mutación	42
4.3.2. Sistema de <i>Fitness</i> y penalización	44
4.4. <i>Primeros resultados</i>	45
4.5. <i>Resultados finales</i>	50
CAPÍTULO 5: GESTIÓN DEL PROYECTO.....	53
5.1. Diagrama de Gantt	53
5.2. Costes del proyecto	54
5.3. Ambientalización del proyecto	55
CONCLUSIONES	56
BIBLIOGRAFÍA	57
ANEXO A: ANÁLISIS EN PROFUNDIDAD DE LOS DIFERENTES ASPECTOS QUE DEFINEN UN ALGORITMO GENÉTICO.....	61
A.1. Aspectos básicos de un algoritmo genético	61
A.1.1. Codificación	61
A.1.2. Población inicial	64
A.1.3. Función de adaptación o <i>Fitness function</i>	64
A.1.3.1. Fitness puro $[r(i,t)]$	65
A.1.3.2. Fitness estandarizado $[s(i,t)]$	65
A.1.3.3. Fitness ajustado $[a(i,t)]$	65
A.1.3.4. Fitness normalizado $[n(i,t)]$	65
A.1.4. Función de selección.....	67
A.1.4.1. Selección puramente elitista.	67
A.1.4.2. Selección por ruleta.	67
A.1.4.3. Selección por rango.	68
A.1.4.4. Selección por torneo.	69
A.1.4.5. Selección del valor esperado.	70
A.1.5. Función de cruce	71
A.1.5.1. Operador de cruce basado en un punto (SPX).	71
A.1.5.2. Operador de cruce basado en n puntos	72
A.1.5.3. Operador de cruce uniforme (UPX)	72
A.1.5.4. Operador de cruce basado en la función de <i>fitness</i>	73
A.1.5.5. Operador de cruce inspirado en el <i>Simulated Annealing</i>	73
A.1.5.6. Operador de cruce baricéntrico	74
A.1.6. Función de mutación	75
A.1.7. Función de inserción	76

A.1.8. Condición de parada	78
A.2 Evaluación de los Algoritmos Genéticos	78
A.2.1. Evaluación On-Line	78
A.2.2. Evaluación Off-Line:	79
A.3. Problemas específicos	79
A.3.1. Sharing	81
A.3.2. Scaling	83
A.3.2.1. <i>Scaling</i> lineal	83
A.3.2.2. Scaling exponencial	84
A.3.3. Otros.....	84
A.3.3.1. <i>Crowding</i>	84
A.3.3.2. <i>Restarting</i>	85
A.3.3.3. <i>Restricted mating</i>	85
A.3.3.4. <i>Isolation by distance</i> :	85
A.4. Bases matemáticas de los algoritmos genéticos	85
A.5. Mejoras	87
A.5.1. Algoritmos en paralelo (Modelos de islas)	87
A.5.1.1. Arquitectura en estrella	88
A.5.1.2. Arquitectura en red	88
A.5.1.3. Arquitectura en anillo	89
A.5.2. Algoritmos celulares	89
A.5.3. Procesos en paralelo (Balanceado de carga de procesado)	90
A.5.4. Algoritmos Genéticos Híbridos.....	91
A.6. Cromosomas de longitud variable.....	92
A.6.2. Población inicial con cromosomas de longitud variable	92
A.6.3. Función de mutación con cromosomas de longitud variable	92
A.6.3. Función de cruce con cromosomas de longitud variable	93
ANEXO B: FUNCIONES ESCOGIDAS	94
B.1. Función A.....	94
B.2. Función B.....	95
B.3. Función C.....	95
B.4. Función D.....	96
B.5. Función E	97
ANEXO C: PROBLEMA FINAL	98
C.1. Tipos de contenedores.....	98
C.2. Cálculo del centro de gravedad de un avión.....	100

C.3. Cuantificación de los parámetros utilizados en las hojas de carga y equilibrado del avión103

C.4. Número de combinaciones posibles al introducir los paquetes en contenedores. .107

C.5. Cálculo de las diferentes combinaciones posibles al introducir los contenedores en las bodegas.108

ANEXO D: CODIFICACIONES..... 111

D.1. Análisis de la primera codificación utilizada proveniente del *Kanpur Genetic Algorithm's Laboratory*..... 111

D.1.1. Input_parameters()112

D.1.2. Select_memory().....113

D.1.3. init_report()113

D.1.4. initialize().....113

D.1.5. statistics()113

D.1.6. report()114

D.1.7. generate_new_pop()114

D.1.7.1. Proceso de selección.....114

D.1.7.2. Proceso de cruce115

D.1.7.3. Proceso de mutación:115

D.1.8. free_all()116

D.2. Codificación final para la aplicación práctica 117

D.3. Script's Matlab para el análisis de resultados 143

D.3.1. Script para la función A..... 143

D.3.2. Script para la función B..... 145

D.3.3. Script para la función C..... 147

D.3.4. Script para la función D..... 150

D.3.5. Script para la función E..... 152

ÍNDICE DE FIGURAS

Fig.1.1 Frente de Pareto de una función con dos variables	4
Fig.1.2 Fragmento de una función con múltiples máximos y mínimos [2]	4
Fig.1.3 Listado de algunos de los métodos de optimización más representativos. [1]...	5
Fig.2.1 Clasificación de los Algoritmos Evolutivos [7]	9
Fig.2.2 Pseudocódigo de un Algoritmo Genético Simple [12]	12
Fig.2.3 Algoritmo Genético expresado de forma abstracta como un conjunto de métodos	13
Fig.2.4 Operador de cruce basado en un punto. [6].....	17
Fig.3.1 Representación gráfica de las funciones elegidas para las pruebas.	21
Fig.3.2 Tiempo de proceso para diferentes longitudes de cromosomas binarios.	22
Fig.3.3 Precisión en los resultados para diferentes longitudes de cromosoma.	22
Fig.3.4 Resultados del test de población en la función A (codificación real)	23
Fig.3.5 Resultados del test de población en la función A (codificación binaria).....	24
Fig.3.6 Resultados del test de población en la función B (codificación real)	24
Fig.3.7 Resultados del test de población en la función B (codificación binaria).....	24
Fig.3.8 Resultados del test de población en la función C (codificación real)	25
Fig.3.9 Resultados del test de población en la función C (codificación binaria)	25
Fig.3.10 Resultados del test de generaciones en las funciones A, B y C	26
Fig.3.11 Efectividad del Algoritmo Genético en función de la probabilidad de cruce utilizada.....	28
Fig.3.12 Efectividad del AG en función de la probabilidad de mutación utilizada	29
Fig.3.13 Efectividad del AG con diferentes combinaciones de probabilidad de cruce y probabilidad de mutación	30
Fig.3.14 Efectividad y tiempo de proceso del AG en el test de <i>Sharing</i>	31
Fig.3.15 Resultados para las funciones A, B y C con y sin mejoras (real)	32
Fig.3.16 Resultados para las funciones A, B y C con y sin mejoras (binario).....	33
Fig.3.17 Resultados normalizados para la función E	34

Fig.4.1 Diagrama de fuerzas básicas que actúan sobre el centro de gravedad (izquierda) y representación de los ejes del avión (derecha)	36
Fig.4.2 Ejemplo de codificación	38
Fig.4.3 Detalle del contenedor AKH [15].....	39
Fig.4.4 Esquema de un Airbus A320 [15]	41
Fig.4.5 Detalle de las bodegas de un Airbus A320 y de los contenedores AKH.....	41
Fig.4.6 Ejemplo de la codificación A y de la codificación B	42
Fig.4.7 Ejemplo de cruce utilizando la codificación B con repeticiones	43
Fig.4.8 Ejemplo de cruce utilizando la codificación B una vez se ha resuelto el problema de los duplicados	43
Fig.4.9 Tiempo de proceso en función del número de bultos	52
Fig.5.1 Planificación inicial del proyecto	53
Fig.5.2 Reorganización del proyecto	54
Fig.A.1 Representación decimal, binaria y en codificación Gray de los 8 primeros dígitos.	62
Fig.A.2 Representación en forma real y binaria de los cromosomas de un problema de optimización [18]	62
Fig.A.3 Ejemplo de la relación directa entre genotipo y fenotipo [2].....	63
Fig.A.4 Ejemplos de los diferentes individuos que podrían aparecer [18]	63
Fig.A.5 Ruleta con las probabilidades de los cuatro elementos que componen la población.....	68
Fig.A.6 Representación de la distribución de probabilidad que ofrece la selección por ruleta (izquierda) y la selección por rango (derecha).	69
Fig.A.7 Ejemplo del funcionamiento del operador de cruce en un punto [6].....	71
Fig.A.8 Ejemplo del funcionamiento del operador de cruce en dos puntos [6]	72
Fig.A.9 Funcionamiento del operador de cruce uniforme [6]	73
Fig.A.10 Efectos del coeficiente α en el operador de cruce baricéntrico	74
Fig.A.11 Permutación de genes dentro de un cromosoma.	76
Fig.A.12 Ejemplo de espacios de búsqueda particulares donde los AG's no funcionan correctamente. [2]	79
Fig.A.13 Función demasiado irregular (<i>Ruggedness</i>) [2].....	80

Fig.A.14 Espacio de búsqueda con una gran sección neutral. [2].....	80
Fig.A.15 Efectos de utilizar <i>sharing</i> : a la izquierda distribución de la población en una función normal, a la derecha distribución de la población utilizando técnicas de <i>sharing</i> [22]	81
Fig.A.16 Ejemplo de esquema.....	86
Fig.A.17 Ejemplo de espacio de búsqueda con un óptimo aislado donde difícilmente daremos con el utilizando un <i>Algoritmo Genético</i> [2]	87
Fig.A.18 AG's en paralelo unidos por una arquitectura en red [31]	89
Fig.A.19 AG's conectados en paralelo mediante arquitectura en anillo [31].....	89
Fig.A.20 Ejemplo de algoritmo genético Celular [31]	90
Fig.A.21 Distribución de la carga de procesamiento del <i>algoritmo genético</i> [22]	91
Fig.A.22 Ejemplos de cruces con cromosomas de longitud variable [2].....	93
Fig.B.1 Gráfica de la función A.....	94
Fig.B.2 Gráfica de la función B	95
Fig.B.3 Gráfica de la función C	96
Fig.B.4 Gráfica de la función D.....	96
Fig.B.5 Gráfica de la función E	97
Fig.C.1 Tabla con diferentes tipos de contenedores (1/2) [15]	98
Fig.C.2 Tabla con diferentes tipos de contenedores (2/2) [15]	99
Fig.C.3 Hoja tabulada para calcular los efectos de las masas en el avión	100
Fig.C.4 Ejemplo del cálculo del centro de gravedad del Airbus A320 mediante las hojas tabuladas de equilibrado [38]	101
Fig.C.5 Limitaciones físicas aportadas por el fabricante del Airbus A320 [15]	102
Fig.C.6 Parte final de la hoja de equilibrado donde aparece el valor del ángulo TRIM asociado al índice obtenido [38]	102
Fig.C.7 Datos técnicos del avión y el vuelo (recuadro verde), así como la fórmula para determinar su índice DOW (recuadro rojo) [38]	103
Fig.C.8 Correcciones del índice DOW por el peso del personal y catering [38]	103
Fig.C.9 Contribuciones de la carga en las bodegas, zonas de pasajeros y combustible [38].....	104
Fig.C.10 Relaciones entre los valores del índice DOW y el ángulo TRIM [38], [15] ..	106

Fig.C.11 Distribución de los contenedores AKH dentro de las bodegas	109
Fig.D.1 Diagrama de bloques con el esquema de funcionamiento del AG inicial	111

ÍNDICE DE ECUACIONES

(ec.1.1) Definición de máximo local	3
(ec.1.2) Definición de mínimo local	3
(ec.2.1) Método de selección por ruleta	15
(ec.3.1) Función A	20
(ec.3.2) Función B	20
(ec.3.3) Función C	20
(ec.3.4) Función D	20
(ec.3.5) Función E	21
(ec.3.2) Criterio de parada	27
(ec.4.1) Espacio de búsqueda del problema final	38
(ec.4.2) Espacio de búsqueda con posibilidad de dejar mercancía en tierra	38
(ec.4.3) Cálculo del espacio de búsqueda para un ejemplo simple	39
(ec.4.4) Resultado final del espacio de búsqueda una vez se han eliminado las combinaciones repetidas	39
(ec.4.5) Penalización cuando existe <i>Overload</i>	44
(ec.4.6) Penalización cuando no existe <i>Overload</i>	44
(ec.4.7) Ecuación de <i>fitness</i> inicial	47
(ec.4.8) Ecuación de <i>fitness</i> definitiva cuando existe <i>Overload</i>	47
(ec.4.9) Ecuación de <i>fitness</i> definitiva cuando no existe <i>Overload</i>	47
(ec.A.1) Fitness estandarizado $s(i,t)$	65
(ec.A.2) Fitness ajustado $a(i,t)$	65
(ec.A.3) Fitness normalizado	66
(ec.A.4) Cálculo de $P(x)$ para el método de selección por ruleta	68
(ec.A.5) Cálculo de $P(i,t)$ para el método de selección por rango	69
(ec.A.6) Cálculo del <i>contador</i> en el método de selección por valor esperado	70
(ec.A.7) Cálculo del parámetro p para la distribución de Bernoulli del método de cruce basado en la función de fitness	73

(ec.A.8) Probabilidad p perteneciente al método de cruce inspirado en el <i>simulated annealing</i>	74
(ec.A.9) Operador de cruce baricéntrico	74
(ec.A.10) Probabilidad de mutación calculada empíricamente [23]	76
(ec.A.11) Método de evaluación <i>on-line</i>	79
(ec.A.12) Método de evaluación <i>off-line</i>	79
(ec.A.13) <i>Sharing_{triσ}</i> (d)	82
(ec.A.14) <i>Sharing_{cvexσ,p}</i> (d)	82
(ec.A.15) <i>Sharing_{ccavσ,p}</i> (d)	82
(ec.A.16) <i>Sharing_{expσ,p}</i> (d)	82
(ec.A.17) <i>Niche count</i> (cuenta de nicho)	83
(ec.A.18) <i>Scaling</i> lineal	83
(ec.A.19) <i>Scaling</i> exponencial	84
(ec.A.20) Parámetro k del <i>Scaling</i> exponencial	84
(ec.B.1) Ecuación de la función A	94
(ec.B.2) Ecuación de la función B	95
(ec.B.3) Ecuación de la función C	95
(ec.B.4) Ecuación de la función D	97
(ec.B.5) Ecuación de la función E	97
(ec.C.1) Cálculo del índice DOW mediante las diferentes contribuciones y factores predefinidos	106
(ec.C.2) Cálculo del TRIM mediante el índice DOW y los factores predefinidos	106
(ec.C.3) Cálculo del número de soluciones sin dejar mercancía en tierra	107
(ec.C.4) Cálculo del número de soluciones dejando mercancía en tierra	107
(ec.C.5) Formula genérica para determinar el número de soluciones posibles no repetidas tanto cuando se deja mercancía en tierra como cuando se carga toda	107
(ec.C.6) Número de soluciones en el caso práctico (sin dejar carga en tierra)	108
(ec.C.7) Número de soluciones en el caso práctico (eliminando las combinaciones representan la misma solución)	108

(ec.C.8) Número de permutaciones posibles de los 7 contenedores AKH dentro de las bodegas del Airbus A320	109
(ec.C.9) Número de combinaciones posibles de los 7 contenedores AKH eliminando las “repetidas”	109

ÍNDICE DE TABLAS

Tabla 3.1 Valores óptimos de la probabilidad de cruce.....	28
Tabla 3.2 Valores óptimos de la probabilidad de mutación	29
Tabla 4.1 Resultados sin reparar las soluciones (problema final)	46
Tabla 4.2 Resultados reparando las soluciones (problema final)	46
Tabla 4.3 Resultados del nuevo sistema sin penalización y con nuevo <i>fitness</i>	48
Tabla 4.4. Resultados con el sistema de permutación de contenedores.....	49
Tabla 4.5. Mejores individuos por codificación con el sistema completo	50
Tabla 4.6 Detalles de los diferentes escenarios de pruebas	50
Tabla 4.7 Valores medios de las diferentes pruebas al sistema completo	51
Tabla 4.8 Individuos con mejor fitness obtenidos con el sistema completo.....	51
Tabla 4.9 Individuos con mejor ocupación obtenidos con el sistema completo.....	51
Tabla C.1 Contribuciones cuantificadas del peso de los tripulantes, catering, etc.....	104
Tabla C.2 Contribuciones por kilo de peso de las bodegas y zonas de pasajeros	105
Tabla C.3 Parámetros CERO_TRIM y FACTOR_TRIM	106
Tabla C.4 Tabla con las 210 combinaciones diferentes que se pueden formar con los 7 contenedores AKH dentro de las bodegas del Airbus A320	110

INTRODUCCIÓN

Aunque su nombre suene algo extraño, los algoritmos genéticos son unos potentes métodos de optimización pertenecientes a la rama de los llamados algoritmos *heurísticos*.

A lo largo de este TFC iremos profundizando en su origen y funcionamiento e iremos comprobando poco a poco, no sólo que funcionan correctamente, sino que además resultan ser de las pocas técnicas efectivas en problemas que antes se resolvían gracias a la intuición y experiencia de los investigadores.

La finalidad de este TFC no es hacer un ensayo exhaustivo sobre estas técnicas, ya que se trata de unos sistemas abiertos y muy flexibles que aún están creciendo y que admiten infinitas posibilidades. Intentar plasmar todo este contenido en un sólo trabajo sería del todo imposible. La idea es dar unas pinceladas sobre sus aspectos más básicos para que el lector pueda obtener una visión global de cómo son, cómo funcionan y cuáles son sus debilidades.

Para alcanzar nuestro objetivo comenzaremos con una breve introducción general sobre los métodos de optimización global. Veremos que existen diferentes ramas de clasificación para estos sistemas en función del tipo de problema y de su metodología a la hora de intentar resolverlo.

En el capítulo 2 nos centraremos de lleno en los algoritmos genéticos. Nuestra intención sería mostrar con un mínimo de detalle las bases en las que se sustenta el funcionamiento de estos métodos. Debido a la gran cantidad de documentación existente subdividiremos este capítulo en dos partes: por un lado relegaremos a los anexos el grueso de información teórica y por otro lado dejaremos un breve resumen básico en el cuerpo del TFC. De esta forma intentaremos que el lector pueda seguir todo el trabajo sin problemas, pero que al mismo tiempo tenga la posibilidad de profundizar más en el tema si así lo desea. Así pues, ningún apartado de los anexos es necesario para seguir el hilo del TFC, más bien son complementos opcionales.

Una vez conocidas las bases y el mecanismo interno de los algoritmos genéticos, utilizaremos el capítulo 3 para poner a prueba la teoría con una serie de problemas simples. En cuanto a la codificación utilizada, nos serviremos de una de las múltiples implementaciones libres que existen y la modificaremos para que responda a nuestras necesidades. Realizaremos diferentes pruebas y extraeremos datos estadísticos para verificar empíricamente la teoría al mismo tiempo que profundizamos más en el entendimiento de estos métodos.

Por último, el capítulo final lo dedicaremos a intentar resolver un problema real más complejo. En concreto intentaremos equilibrar la carga de las bodegas de un Airbus A320 mediante estos métodos. Al tratarse de un problema totalmente diferente, no podremos utilizar la implementación del código inicial y deberemos partir de cero, así pues veremos la evolución ha de seguir desde la idea inicial hasta alcanzar el método definitivo que pueda resolver el problema.

CAPÍTULO 1. INTRODUCCIÓN A LOS MÉTODOS DE OPTIMIZACIÓN

1.1. Definiciones básicas

De forma genérica, puede definirse la optimización como aquella ciencia encargada de determinar las mejores soluciones a problemas matemáticos que a menudo modelan una realidad física [1]. Bajo esta definición tan simple se presenta todo un mundo de métodos y algoritmos diferentes que resultarían del todo imposibles de tratar de forma correcta en un solo TFC.

Los problemas de optimización no sólo están presentes en ingenierías de cualquier tipo. Forman parte de casi todas las ramas de la ciencia haciendo posible la resolución de una variedad infinita de problemas distintos y de cuyas características depende la elección de uno u otro método de optimización. Para poder comprender un poco mejor el amplio abanico de posibles soluciones que se nos ofrecen es necesario comenzar con una serie de definiciones básicas extraídas del trabajo de Thomas Weise, *Global optimization algorithms – theory and application* [2].

- **Solución candidata:** Consiste en todas y cada una de las soluciones posibles al problema. En el caso de estar trabajado en un problema de optimización con restricciones veremos que no todas las soluciones posibles son aceptables. Esto implica que previamente se ha modelado dicho problema de manera que se pueda trabajar como una función matemática, cosa que resulta muy complicada en algunos casos. De hecho, en muchas ocasiones el problema a optimizar no tiene una representación matemática directa, sino que se trabaja con algún tipo de función de coste o incluso con simulaciones de los resultados.
- **Espacio del problema:** Se define el *espacio del problema* X como la unión de todas las soluciones candidatas al problema que queremos optimizar.
- **Espacio de soluciones:** El *espacio de soluciones* S está definido como el conjunto de todas las soluciones aceptables al problema¹.
- **Espacio de búsqueda:** El *espacio de búsqueda* G se define como el conjunto de todos los elementos que pueden ser operados por los métodos de optimización. Esta diferenciación cobra especial sentido en los *algoritmos genéticos* y en la mayoría de los algoritmos heurísticos puesto que, como se verá más adelante, no trabajan con soluciones directamente, sino con individuos codificados que representan las posibles soluciones.

¹ Como se ha dicho antes, en caso de tratarse de optimización sin restricciones, el *espacio del problema* y el *espacio de soluciones* son iguales.

- **Vecindad:** La vecindad se define como un entorno acotado del espacio de búsqueda alrededor del punto que se está evaluando donde todos los puntos son *adyacentes*.
- **Adyacentes:** Se dice que dos puntos p y p' , pertenecientes a un espacio de búsqueda de una función, son adyacentes si se puede saltar de uno a otro con una única operación de búsqueda.
- **Operación de búsqueda:** Cualquiera de las operaciones de las que se sirva el algoritmo de optimización para llevar a cabo su cometido que es, cómo no, encontrar el valor óptimo (ya sea local o global).
- **Máximo local:** Se define el *máximo local* $x' \in X$ de una función $f: X \rightarrow R$ es un elemento tal que, $f(x') \geq f(x)$ para cualquier otro elemento x que pertenezca al vecindario. O dicho de otra forma, se puede definir un intervalo ε tal que para todo elemento $x \in X$ se cumpla lo siguiente (ec.1.1).

$$\forall x' \exists \varepsilon > 0 : f(x') \geq f(x) \forall x \in \mathbb{X}, |x - x'| < \varepsilon \quad (\text{ec.1.1})$$

- **Máximo global:** Se define el *máximo global* $x' \in X$ de una función $f: X \rightarrow R$ como aquel elemento que para cualquier $x \in X$ se cumple que $f(x') \geq f(x)$.
- **Mínimo local:** Se define que el *mínimo local* $x' \in X$ de una función $f: X \rightarrow R$ es un elemento tal que, $f(x') \leq f(x)$ para cualquier otro elemento x que pertenezca al vecindario. O dicho de otra forma, se puede definir un intervalo ε tal que para todo elemento $x \in X$ se cumpla lo siguiente (ec.1.2).

$$\forall x' \exists \varepsilon > 0 : f(x') \leq f(x) \forall x \in \mathbb{X}, |x - x'| < \varepsilon \quad (\text{ec.1.2})$$

- **Mínimo global:** Se define el *mínimo global* $x' \in X$ de una función $f: X \rightarrow R$ como aquel elemento que para cualquier $x \in X$ se cumple que $f(x') \leq f(x)$.
- **Óptimo local:** Se define el *óptimo local* $x^* \in X$ de una función $f: X \rightarrow R$ como aquel elemento que es máximo o mínimo local de la función.
- **Óptimo global:** Se define el *óptimo global* $x^* \in X$ de una función $f: X \rightarrow R$ como aquel elemento que es máximo o mínimo global de la función.
- **Óptimo de Pareto:** Este término aparece cuando nos enfrentamos a problemas de optimización de más de una variable y con restricciones.

En estos casos no todas las soluciones son viables. Y más aún, los *óptimos de Pareto*² hacen referencia a soluciones en las que no se puede mejorar más los valores de una variable, sin empeorar alguna de las otras condiciones. Es decir que más que alcanzar una solución única, lo que conseguimos es definir una frontera de soluciones equilibradas (*frente de Pareto*) (**Fig.1.1**) donde no es posible encontrar una solución que mejore en ningún sentido sin empeorar en otro.

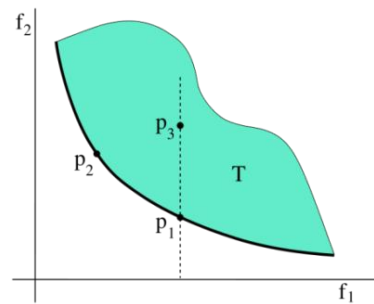


Fig.1.1 Frente de Pareto de una función con dos variables²

Una vez definidos todos estos conceptos, en la figura siguiente (**Fig. 1.2**) podemos ver un ejemplo de lo que sería un fragmento de una función cualquiera. En esta figura podemos ver múltiples *máximos* y *mínimos locales*. Y marcados en rojo podemos observar lo que serían los valores *máximos* y *mínimos globales*. Dado que no se trata del dominio completo de la función, sino de un fragmento acotado, es interesante remarcar como el *mínimo global* coincide con el límite del dominio de la función.

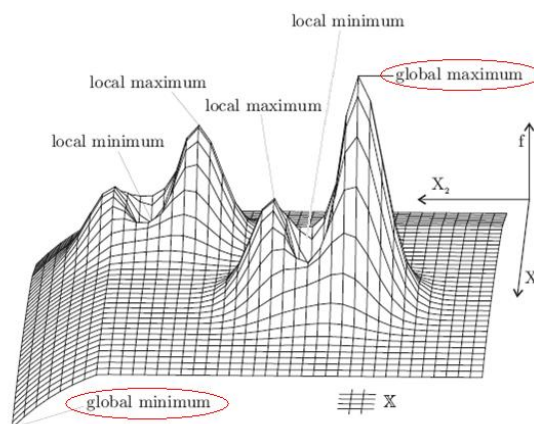


Fig.1.2 Fragmento de una función con múltiples máximos y mínimos [2].

En lo que a clasificación se refiere, la gran variedad de métodos de optimización existentes da lugar a multitud de esquemas diferentes según si se clasifican en función del sistema que utilizan, el tipo de espacio de búsqueda

² http://es.wikipedia.org/wiki/Eficiencia_de_Pareto

sobre el que trabajan, etc. A continuación podemos ver un intento incompleto de clasificación (**Fig.1.3**) de los métodos de optimización más representativos en función del sistema de búsqueda que utilizan.

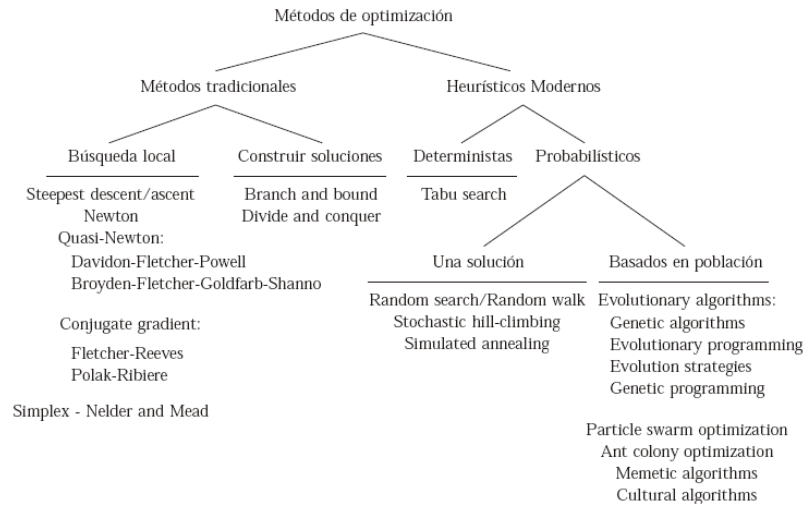


Fig.1.3 Listado de algunos de los métodos de optimización más representativos. [1]

1.2. Métodos tradicionales

Siguiendo el esquema de la **figura 1.2**, dentro de los métodos tradicionales encontraríamos una primera subdivisión en función de si se trata de algoritmos de optimización local o global.

Por un lado, los **algoritmos de búsqueda local** son métodos basados principalmente en la búsqueda de valores extremos de la función, y que se sirven básicamente de la utilización de ecuaciones diferenciales y gradientes para la resolución de problemas. El más destacado es el método **Newton**, (también conocido como método de **Newton-Raphson** [3]) que es un método iterativo numérico simple, que se utiliza para encontrar las raíces o ceros de las funciones basándose en el uso de ecuaciones diferenciales.

En esta primera clasificación también podríamos destacar el algoritmo **Simplex** [4], utilizado en la optimización de problemas de programación lineal. Este método, perteneciente a los algoritmos de búsqueda directa, define el espacio de búsqueda como los vértices de una figura geométrica de N-dimensiones. En cada iteración el algoritmo evalúa la función en los N vértices y desplaza el espacio de búsqueda hacia el punto más óptimo.

El otro gran grupo dentro de los algoritmos clásicos (**Fig.1.2**) lo formarían los **algoritmos** llamados **constructivos**, más orientados hacia la **optimización global**. Éstos se caracterizan por descomponer el problema inicial en sub-

espacios de búsqueda que se van examinando de forma aislada para finalmente obtener como solución el valor más óptimo de los encontrados. Quizás el ejemplo más claro de estos algoritmos sea **Divide and Conquer** [4] (Divide y vencerás), basado en la resolución recursiva de un problema dividiéndolo en dos o más sub-problemas de igual tipo o similar.

Otro ejemplo de este tipo de métodos sería el llamado **Branch and Bound** [4]. Este algoritmo de búsqueda general inicia el proceso considerando el espacio completo del problema original, dando lugar a lo que se denomina como *problema raíz*. A continuación se le aplican los procedimientos de búsqueda del límite inferior y límite superior entre los que se encuentran las soluciones locales al problema.

En caso de coincidir, el proceso termina puesto que se ha alcanzado la solución más óptima. De no ser así se divide el espacio de búsqueda en dos, de forma que obtenemos dos sub-problemas hijos del nodo raíz. Así sucesivamente se va generando un árbol de sub-problemas. Es importante tener en cuenta que el hecho de encontrar una solución óptima a alguno de estos sub-problemas, no garantiza que se trate de la solución óptima para el problema raíz. El proceso continúa hasta que todos los sub-problemas se hayan resuelto o se hayan “*podado*” (descartado) por ofrecer unos límites que se alejan de las soluciones óptimas obtenidas en otras partes del árbol de sub-problemas. Al final del proceso, lo que obtenemos es una solución que puede ser la óptima del problema raíz, o al menos un umbral entre la mejor solución y los límites de los nodos no resueltos.

1.3. Métodos Modernos

Siguiendo con la clasificación (**Fig.1.2**) encontraríamos otro gran grupo de métodos de optimización más modernos, comúnmente llamados **métodos heurísticos**. Estos algoritmos se pueden aplicar a funciones *simples* donde se podrían utilizar los métodos clásicos antes descritos, pero donde realmente demuestran su utilidad es en funciones con un espacio de soluciones demasiado grande o con muchas dimensiones. En estos casos no resulta eficiente recorrer todas las posibles soluciones en busca de la más óptima. Es necesario utilizar técnicas más *creativas* que, a partir de unas pequeñas muestras y de la evolución de las mismas, puedan *decidir* en qué regiones es más probable encontrar la solución al problema.

Dentro de este grupo podríamos hacer una primera gran distinción a partir del patrón de conducta utilizado. Así pues, se generarían dos grandes grupos: por un lado los **métodos deterministas** en los que el espacio de búsqueda se explora siguiendo un patrón concreto, y de los que la búsqueda **Tabú** [2] sería un claro ejemplo. Este método parte de un punto del espacio de búsqueda y analiza las soluciones vecinas en busca de la más óptima, almacenando todas ellas en un registro de soluciones. A partir de los resultados, el patrón de conducta es siempre el mismo, y consiste en desplazarse siempre hacia la solución más óptima, sin aceptar una solución de peor categoría excepto para

evitar volver a espacios de búsqueda que ya se hayan explorado. Este método permite escapar de soluciones locales, así como definir una nueva posible clasificación de los algoritmos de optimización en función de si trabajan con memoria (como en este caso, en el que todas las soluciones obtenidas se almacenan en un registro) o si no lo hacen, de manera que corren el riesgo de explorar una y otra vez el mismo espacio de búsqueda, y recaer una y otra vez en las mismas soluciones.

A parte de los algoritmos deterministas, el otro gran grupo en el que se dividen los sistemas de búsqueda heurísticos son los **métodos estocásticos** que, en contra de los deterministas, intentan aportar un mayor grado de aleatoriedad a la búsqueda, de manera que esta se aproxime más a un sistema de búsqueda probabilístico que no a uno de búsqueda secuencial. Este último grupo es el que más auge ha tenido en las últimas décadas gracias sobre todo a la aparición de procesadores cada vez más potentes que permiten realizar cálculos complejos con un coste de tiempo cada vez más pequeño. Dentro de esta categoría se produce una nueva diferenciación en función de si el algoritmo inicia la búsqueda a partir de un único punto de partida, o si por el contrario se dedica a observar una población de posibles soluciones.

El **Stochastic Hill-climbing** [4] es el algoritmo de búsqueda más utilizado dentro de los métodos que utilizan un único punto de partida. Se divide en tres pasos: primero empieza en un estado generado al azar. El siguiente paso es desplazarse repetidas veces hacia el vecino que ofrezca un valor mejor. Una vez alcanzado un mínimo local el proceso se reinicia desde una nueva posición generada de forma aleatoria. Para evitar quedar atrapado en los valores locales se define un parámetro denominado *Max_Flips* que limita el número de desplazamientos que pueden realizarse entre cada reinicio. El proceso se repite tantas veces como sea necesario hasta conseguir una solución.

Otro método característico dentro de este tipo de algoritmos sería el **Simulated Annealing** [2] que es un método consistente en imitar el proceso de cristalización de los sólidos. En la realidad, este sistema consiste en calentar un material y luego enfriarlo (como por ejemplo hierro o cristal) normalmente para endurecerlo y hacer que el material sea menos frágil. *Simulated Annealing* intenta imitar este proceso para tratar de obtener una solución más óptima al problema. La técnica consiste en imaginar la función matemática como si se tratase de una representación de la energía cinética de las partículas que la componen. Así pues, si calentamos suficientemente el sólido la posición de sus moléculas (o sea nuestra solución) podrá superar las barreras de potencial y saltar a un nuevo estado (es decir superar un mínimo local y desplazarse a una nueva zona del espacio de búsqueda). Basándose en esto, la técnica prueba variaciones al azar de la solución actual. La probabilidad de aceptar una solución peor como nueva solución disminuye con el tiempo (enfriamiento). Cuanto más lentamente se produzca el enfriamiento, más probable será que el algoritmo encuentre una solución óptima o casi óptima.

Por otro lado, dentro de los **algoritmos basados en poblaciones de soluciones** encontramos una gran variedad de métodos divididos en dos grupos: los que trabajan con memoria y los que no. En el lado de los que

trabajan sin memoria encontramos los **Evolutionary Algorithms**, que son métodos que intentan imitar el proceso de selección natural de la población. En este grupo es donde se encontrarían definidos los **Algoritmos Genéticos** (que serán tratados en profundidad en los capítulos siguientes). A grandes rasgos, estos métodos parten de una población de soluciones y las hacen *evolucionar* a base de combinarlas entre sí. Así mismo añaden aleatoriedad al sistema mediante un proceso de *mutación*. Pero su sistema de selección se basa meramente en el *elitismo* y sin ningún tipo de memoria, de forma que prevalecen las soluciones más óptimas frente a las otras que son desechadas.

Por otro lado, dentro de los *algoritmos con memoria*, encontramos métodos como el **Particle Swarm Optimization** (enjambre de partículas) o el **Ant Colony Optimization** [5] (colonia de hormigas) que consisten en poblaciones de individuos (en este caso soluciones) organizados, que tienen en cuenta los logros obtenidos por sus predecesores y que se organizan para cooperar entre sí. Por ejemplo, en el caso del de la Colonia de hormigas que define un paradigma que intenta imitar el comportamiento de las hormigas cuando están buscando un camino en la colonia que les lleve a una fuente de comida. A medida que las hormigas se van desplazando hacia posiciones más óptimas van marcando el camino mediante feromonas. Esta información sirve de guía para las siguientes hormigas que identificarán las zonas más óptimas ya que serán las que más cantidad de feromonas tendrán, mientras que las secciones menos fructíferas apenas ofrecerán rastros. Así pues, imitando este sistema natural, obtendremos un método distribuido en el que un conjunto de agentes individuales cooperan para llevar a cabo un objetivo común: encontrar una solución óptima.

Dentro de este grupo también se encuentran métodos similares como los **Cultural Algorithms**, que intentan imitar la evolución cultural. De manera que las sucesivas generaciones de individuos se enriquecen con el conocimiento del espacio de búsqueda que se obtiene durante el proceso de exploración. Esta información se transmite de generación en generación para así encontrar al final la solución óptima. Así como los **Memetic Algorithms**, que combinan técnicas de los *Evolutionary Algorithms* y *Conjugate Gradients*. Es decir de la búsqueda de poblaciones y la mejora local, y por ello también son considerados como algoritmos genéticos híbridos.

Queda patente pues la gran variedad de métodos de optimización existentes. En definitiva el tipo de algoritmo utilizado depende en gran medida del tipo de espacio de búsqueda con el que nos enfrentamos ya que métodos existentes hay muchos y cada uno goza de ciertas virtudes y padece ciertas debilidades.

CAPÍTULO 2. INTRODUCCIÓN A LOS ALGORITMOS GENÉTICOS

2.1. Computación evolutiva

Retomando el hilo de la clasificación utilizada en el capítulo anterior (**Fig.1.3**) nos encontramos con un grupo de algoritmos de optimización global, estocásticos y basados en población denominados **Algoritmos Evolutivos**.

Estos métodos de optimización obtienen su nombre de grupo de su rasgo en común que consiste en intentar imitar los procesos naturales de evolución genética planteados por Darwin. Haciendo referencia a lo dicho por M. Gestal [6] podríamos explicar la existencia de esta rama como la respuesta a una pregunta muy simple: “*Si algo funciona bien, ¿por qué no imitarlo?*”.

Durante millones de años las diferentes especies han estado evolucionando y compitiendo entre sí para tratar de convertirse en individuos cada vez más óptimos y mejor adaptados. Partiendo de esa base, lo que pretenden los Algoritmos Evolutivos es arrancar desde una población de potenciales soluciones a un problema y mediante sucesivos cruces, mutaciones y procesos de selección de los individuos más óptimos, converger hacia lo que sería el óptimo global del problema inicial (**Fig. 2.1**).

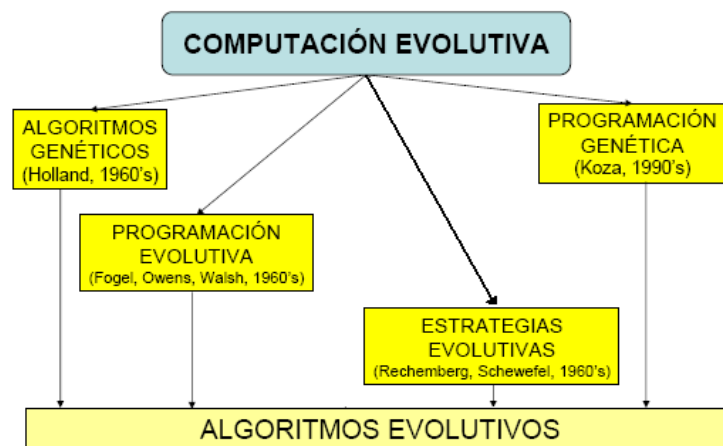


Fig.2.1 Clasificación de los Algoritmos Evolutivos [7]

A grandes rasgos, las diferencias internas dentro del grupo de los Algoritmos Evolutivos hacen referencia al tipo de individuos con los que trabajan o a la forma en la que se rigen para hacer evolucionar a sus poblaciones. Dejando a un lado los algoritmos genéticos, puesto que son el tema central del capítulo, nos quedan tres grandes grupos a definir (ver [2], [8], [9] y [10]).

2.1.1. Programación Evolutiva

La **Programación Evolutiva** surgió de las manos de David Fogel [8] en la década de los sesenta en E.E.U.U. como alternativa a la creación de redes neuronales y a la programación heurística. Ambos métodos se centraban en emular a los humanos como los organismos más inteligentes y avanzados creados por la evolución. Fogel pretendía utilizar la evolución simulada como proceso de aprendizaje destinado a generar inteligencia artificial.

Durante estos años se ha aplicado al aprendizaje de los autómatas de estados finitos y, más recientemente, a la optimización numérica. Este método se caracteriza por tener un gran marco de trabajo. Admite cualquier tipo de representación o mutación, aunque a grandes rasgos utiliza vectores de números reales y la perturbación gaussiana como tipo de mutación. Selecciona a los padres de manera determinística y a sus supervivientes de forma probabilística. De esta técnica cabe destacar que no admite la recombinación y que posee una gran auto-adaptación de la mutación a los parámetros estándar. Es un híbrido de los métodos que veremos a continuación, es decir, muchas de sus características serán comunes a los otros métodos.

2.1.2. Estrategias Evolutivas

Las **Estrategias Evolutivas** fueron introducidas por Rechenberg [10] en la Alemania de los años sesenta y setenta. Comparten con la Programación Evolutiva el tipo de representación codificada con números reales, la mutación gaussiana y la fácil auto-adaptación. Y se diferencian de ella por su rapidez, así como porque admiten recombinación discreta o intermedia. También difieren en el modo uniforme y aleatorio de seleccionar a los padres, y por los métodos de selección de supervivientes.

Estos métodos imitan la evolución biológica, más concretamente el proceso de selección natural y el principio de “supervivencia del más fuerte”. Son métodos que priorizan la selección y la mutación de las soluciones como sistema para hacer evolucionar a la población, dejando en segundo plano los métodos de cruce. Su aplicación más notoria es la optimización numérica.

2.1.3. Programación Genética

La **Programación Genética** apareció por primera vez de la mano de Koza [9] a principios de los noventa. Es un método relativamente nuevo que se inspira en la evolución biológica para encontrar programas destinados a realizar una tarea concreta (resolver un problema). Es un método automatizado que recibe la información de un nivel superior que le indica que hay un problema y automáticamente crea un programa para solucionar dicho problema. Tradicionalmente se ha aplicado a tareas de aprendizaje de autómatas tales como predecir o clasificar.

La programación genética igual que los algoritmos genéticos se caracteriza por seleccionar a los padres mediante *fitness*, así como porque en la selección de supervivientes se produce una sustitución generacional. Compite con las redes neuronales y sus derivados y necesita poblaciones enormes (miles de individuos) para llevarse a cabo. Se representa mediante estructuras en forma de árboles, y el proceso de recombinación se produce mediante el intercambio de subárboles. Por otro lado, la mutación, que es posible pero no necesaria, se realiza mediante cambios aleatorios en los árboles.

2.2. Algoritmos Genéticos

Los **Algoritmos Genéticos** (AG's de ahora en adelante) son métodos adaptativos que pueden utilizarse para resolver problemas reales de búsqueda y de optimización basándose en imitar los procesos genéticos de los organismos vivos.

Los principios básicos de los AG's se atribuyen a Holland en 1975 [11]. En gran parte, su auge y proliferación en los últimos tiempos se debe a su carácter robusto, que les permite abordar con éxito gran variedad de problemas de áreas muy diferentes, incluyendo aquellos en los que otros métodos encuentran dificultades [12]. Además de esto, los AG's ofrecen otras características interesantes como el hecho de que, aunque no se pueda garantizar que el AG encuentre la solución óptima al problema, existe la certeza empírica de que ofrecerá una solución de un nivel aceptable, en un tiempo competitivo. Es cierto que no son la respuesta perfecta para abordar todos los problemas. Existen situaciones que cuentan con métodos concretos más rápidos y efectivos que los AG's. Aun así, otra de las ventajas que ofrecen los AG's es que permiten mejorar estas técnicas específicas hibridándolas con la metodología de los AG's.

Haciendo referencia de nuevo al trabajo publicado por el Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad del País Vasco [12] a continuación describiremos a grandes rasgos las características de los AG's para, posteriormente, centrarnos más en cada una de sus partes.

Así pues, los AG's se basan en imitar el proceso genético de los seres vivos como se ha dicho anteriormente. Parten de una población inicial de individuos generados al azar de forma que cada uno de ellos representa una posible solución al problema dado. A cada uno de estos individuos se les debe asignar un valor que represente su grado de adaptación, o dicho de otra forma, cómo de buena es esa solución para el problema.

Así pues, del mismo modo que en la naturaleza los individuos mejor dotados (es decir, los que ofrezcan las mejores soluciones) serán los que más probabilidades tendrán de ser seleccionados para reproducirse. Mientras que los individuos peor adaptados (los que ofrezcan peores soluciones) tendrán más difícil el propagar su material genético a las nuevas generaciones. Sucesivamente cada nueva generación contará con una mayor proporción de

buenas características de forma que, si el AG ha sido diseñado correctamente, la población convergerá hacia una solución óptima del problema.

2.2.1. Algoritmo Genético Simple o Canónico

Bajo la explicación tan simple antes expuesta subyace un proceso probabilístico que puede llegar a ser muy complejo. El primer paso para poder explicar con más detalle la sistemática que utilizan los AG's es plantear la versión más sencilla posible de éstos denominada **Algoritmo Genético Canónico**. A continuación en la **figura 2.2** podemos ver el pseudocódigo de lo que sería el Algoritmo Genético Simple o Canónico.

```

BEGIN /* Algoritmo Genetico Simple */
  Generar una poblacion inicial.
  Computar la funcion de evaluacion de cada individuo.
  WHILE NOT Terminado DO
    BEGIN /* Producir nueva generacion */
      FOR Tamaño poblacion/2 DO
        BEGIN /*Ciclo Reproductivo */
          Seleccionar dos individuos de la anterior generacion,
          para el cruce (probabilidad de seleccion proporcional
          a la funcion de evaluacion del individuo).
          Cruzar con cierta probabilidad los dos
          individuos obteniendo dos descendientes.
          Mutar los dos descendientes con cierta probabilidad.
          Computar la funcion de evaluacion de los dos
          descendientes mutados.
          Insertar los dos descendientes mutados en la nueva generacion.
        END
      END
      IF la poblacion ha convergido THEN
        Terminado := TRUE
      END
    END
  END
END

```

Fig.2.2 Pseudocódigo de un Algoritmo Genético Simple [12]

Al observar la **figura 2.2** comprobamos que representa claramente el proceso descrito con anterioridad. Partimos de una *población inicial* de individuos a los que se les asigna un valor mediante una *función de adaptación (función de fitness)* que representa cómo de buena es la solución que proponen.

El cuerpo del algoritmo es un bucle que realiza las mismas tareas en tanto que no se cumpla la condición de convergencia previamente impuesta: en primer lugar selecciona los individuos progenitores (*función de selección*). Seguidamente se procede a cruzar estos individuos de forma que intercambien parte de sus *genes (función de cruce)* para obtener la nueva generación de descendientes.

A continuación introduce un determinado grado de mutación sobre estos nuevos individuos (*función de mutación*) y finalmente inserta y reajusta el tamaño o número de individuos que conforman la población para que esta no crezca de forma desmesurada (*función de inserción*).

Este proceso se repetirá en tanto que no se cumpla la *condición de parada* impuesta previamente. De nuevo esta es una explicación muy simple del funcionamiento de los AG's, pero que nos permite expresarlos como un conjunto de procesos más o menos sencillos (**Fig.2.3**). A continuación daremos unas breves pinceladas sobre cada uno (para profundizar más en ellos será necesario recurrir a los anexos (Anexo A) debido a la limitada extensión de que dispone este TFC).

```
BEGIN {  
    Población inicial  
    Función de Adaptación.  
    DO {  
        Función de Selección  
        Función de Cruce  
        Función de Mutación  
        Función de Adaptación.  
        Función de inserción  
    } WHILE (Condición de Parada == false);  
} END;
```

Fig.2.3 Algoritmo Genético expresado de forma abstracta como un conjunto de métodos

Para no tener que remitir constantemente al lector a los anexos y poder continuar con los objetivos propuestos daremos unas breves pinceladas sobre los aspectos más importantes y característicos de los AG's. Con esto será suficiente para poder proseguir con el hilo conductor del proyecto y poder comprender las pruebas y resultados del siguiente capítulo (Capítulo 3 Implementación y pruebas básicas). Aun así, quedarán referenciados los capítulos del Anexo donde se amplía la información que se está tratando por si el lector quiere conocer más en profundidad alguno de estos aspectos.

2.2.2. Población inicial [Anexo A, apartados A.1.1 y A.1.2]

En primer lugar deberíamos plantearnos la codificación a utilizar. Como se ha dicho anteriormente, cada individuo representa una posible solución, pero no es la solución misma. Cada individuo estará definido como un cromosoma compuesto por uno o varios genes que representan las diferentes variables que componen el problema. Así pues, cada individuo estaría representado no por el valor de la solución que aporta, sino por el valor que han de adoptar cada una de las variables y que dan como resultado una solución al problema.

Partiendo de esta base, uno de los primeros problemas que se nos plantea es cómo codificar dichos genes o variables. Realmente cualquier codificación puede ser válida: caracteres, números reales, binario, etc. La codificación binaria es una de las más utilizadas porque ofrece una gran flexibilidad a la

hora de codificar y trabajar con múltiples variables, aunque también tiene aspectos negativos como el hecho de que una simple mutación en un único bit puede dar lugar a un individuo totalmente diferente. Esto en sí mismo no es un efecto definitivamente negativo (puesto que añade diversidad y permite explorar mejor el espacio de búsqueda) pero puede alejarte radicalmente de la zona correcta de búsqueda de forma inesperada.

Una vez decidida la codificación más adecuada (y la longitud de la misma en el caso de tratarse de codificación en binario), la siguiente pregunta que surge es cuántos individuos deben formar esta población inicial (ver Anexo A, apartado A.1.2). Algunos autores se decantan por un número prudente como $n=50$ para codificaciones reales [13]. Otros estudios sugieren que si la población está codificada en binario con longitud L sería aconsejable utilizar entre L y $2L$ individuos [14]. Por último solo queda decidir cómo se escogen estos individuos primigenios: la opción más común suele ser hacerlo al azar mediante una distribución uniforme.

2.2.3. Función de Adaptación o *Fitness* [Anexo A, apartado A.1.3]

En la naturaleza, los individuos más adaptados son los que tienen más oportunidades de reproducirse y de que su material genético se propague de generación en generación. Los AG's intentan imitar este proceso y es la función de Adaptación o *Fitness* la encargada de evaluar cómo de "buenos" son los individuos.

Existen multitud de formas de evaluar a los individuos de una población: desde las más simples como la de asignar un valor real que defina cómo de bueno es el individuo comparado con el mejor encontrado hasta la fecha (*Fitness Puro*), hasta normalizar el valor entre 0 y 1 en función de los valores obtenidos por todos y cada uno de los individuos (*Fitness Normalizado*). Además de tener en cuenta posibles penalizaciones puesto que no todos los individuos pueden ser soluciones viables. Pueden incumplir alguna de las restricciones propias del problema y no por ello han de ser descartados sin más (aunque sería una opción posible). Pueden ser conservados aplicándoles algún tipo de hándicap que los penalice frente a los otros individuos viables, pero que los permita evolucionar para facilitar la exploración de los límites del problema.

2.2.4. Función de Selección [Anexo A, apartado A.1.4]

Como se puede imaginar por su nombre, la función de selección es la encargada de elegir a los individuos que harán de progenitores para la nueva generación. Como en el caso anterior existen diferentes formas de tomar esta decisión. La más evidente sería la de escoger a los individuos más adaptados pero esto degeneraría en problemas de *diversidad genética*, donde rápidamente aparecerían *súper individuos* que provocarían uno de los grandes problemas que presentan los algoritmos genéticos: *la convergencia prematura*. Estos tres términos son básicos y aparecerán en multitud de ocasiones de

ahora en adelante, así que es necesario hacer un alto en el camino y ofrecer una pequeña definición:

- **Diversidad genética:** es quizás el más intuitivo de los tres términos. Hace referencia a la distribución de las soluciones en el espacio de búsqueda. Uno de los problemas que presentan los AG's es el de cómo mantener una diversidad genética aceptable evitando que las soluciones se agolpen alrededor de óptimos locales impidiendo que se explore todo el espacio de búsqueda.
- **Súper individuos:** son óptimos locales. En esencia son buenas soluciones al problema (en ocasiones las mejores soluciones) pero que, si no se tratan correctamente, "absorben" a la población hacia ellos de forma que el resto del espacio de búsqueda queda sin explorar.
- **Convergencia prematura:** se produce cuando los súper individuos colapsan la población; es decir, los súper individuos (como mejores soluciones que son) se eligen una y otra vez como progenitores de forma que asfixian a los individuos menos dotados haciéndolos desaparecer. Al final el resultado es una población formada en su mayoría por individuos muy "similares" al súper individuo y por tanto muy próximos a él. De esta forma el AG no tiene ninguna posibilidad de explorar nuevos territorios de manera que la población *converge* hacia el *súper individuo* (óptimo local).

Como hemos dicho antes una solución tan trivial como escoger siempre al mejor no es efectiva a largo plazo, pues reduce la diversidad genética dejando al AG a merced de la suerte. Existen otros métodos más complejos que intentan hacer esta selección de un modo más "distribuido", dando siempre prioridad a los mejores individuos pero dejando cierto margen para los menos adaptados (ver Anexo A, apartado A.1.4). De entre estos diferentes métodos los más utilizados son la *selección por ruleta* y la *selección por torneo*.

- **Selección por ruleta:** consiste en asignar a cada individuo una probabilidad de ser elegido. Esta probabilidad estará basada en el valor de su función de adaptación y vendrá normalizada (entre 0 y 1) por la suma de la función de *fitness* de todos los individuos (**ec.2.1**). Así pues, con este método todos los individuos pueden ser seleccionados aunque claro está, los mejores individuos tendrán mayores posibilidades.

$$P(x) = \frac{\text{Adaptación}(x)}{\sum_{j=1}^N \text{Adaptación}(x_j)} \quad (\text{ec. 2.1})$$

- **Selección por torneo:** es un método considerablemente más simple. Consiste en escoger *n* individuos al azar y hacerlos competir entre sí. La manera de escoger al vencedor puede ser muy simple (escogiendo al que tenga un mejor valor de *fitness*) o más aleatoria (asignando una probabilidad de selección a cada individuo en función de su valor de

fitness y dejando que sea el azar quien decida. Evidentemente la probabilidad de ser elegido será mayor cuanto mejor sea el individuo, pero aun así todos tendrían posibilidades de ser vencedores).

Queda patente pues que la lucha por mantener la diversidad genética y evitar resultados no deseados comienza con la elección de una correcta función de selección, pero además existen otras técnicas dedicadas específicamente a ello. Una vez más, el grueso de las explicaciones quedarán relegadas al anexo (concretamente al Anexo A, apartado A.3 de *Problemas específicos*) pero es necesario definir como mínimo uno de estos métodos puesto que es sujeto específico de una de las pruebas del capítulo siguiente. En concreto se trata de las técnicas de *Sharing*.

- **Técnicas de *Sharing*:** simplificándolo mucho estas técnicas intentan evitar que los individuos más dotados se reproduzcan sin control (haciendo desaparecer a los menos favorecidos) a base de imponer ciertas penalizaciones.

Existen diferentes formas de aplicar esta penalización pero todas parten de un nexo común: la definición de un *nicho* o distancia entre soluciones. A los individuos que estén dentro de un mismo nicho se les penaliza con respecto a los de otros nichos en función de su densidad poblacional. Es decir, un individuo de un nicho muy poblado sufrirá una penalización en su *fitness* al competir contra otro individuo de un nicho más “desértico”.

Esto es así puesto que esa elevada densidad poblacional es debida a la existencia de un óptimo local, es decir, que ese nicho está superpoblado de individuos muy dotados contra los que poco podrán hacer los individuos residuales de otras zonas del espacio de búsqueda. De esta forma, aplicando las técnicas de *Sharing* se consigue una mayor diversidad genética.

2.2.5. Función de cruce [Anexo A, apartado A.1.5]

Es, junto con la función de mutación, la encargada de hacer evolucionar a la población de soluciones. Al igual que en el mundo real, una vez elegidos a los progenitores llega el momento de la reproducción que puede darse de dos formas diferentes: sexual o asexualmente.

Por un lado, el método asexual (que traducido al mundo de los AG's significa que los descendientes son exactamente iguales a los padres) es una idea que por sí sola no es nada recomendable (no aporta diversidad alguna a la población más allá de las posibles mutaciones que puedan surgir después) aunque combinada con la reproducción sexual puede resultar interesante.

Por otra parte, el método sexual, consistente en combinar de alguna forma el material genético de ambos progenitores para dar lugar a nuevos individuos,

(soluciones en este caso) diferentes a sus padres, pero con ciertas similitudes. De este modo la población evoluciona en todas direcciones, de manera que se obtienen tanto individuos más dotados como de peor calidad. Una vez más, existen multitud de formas de llevar a cabo este proceso y no es posible abordarlas todas en este capítulo (ver Anexo A, apartado A.1.5.)

Destacaremos una, la *función de cruce en un punto*, (**Fig. 2.4**) que además de ser la más simple y de las más utilizadas, sienta las bases para multitud de variantes. La idea en la que se sustenta es muy simple: se escoge un punto al azar del cromosoma, dando lugar a dos secciones, que seguidamente se intercambian entre los progenitores.

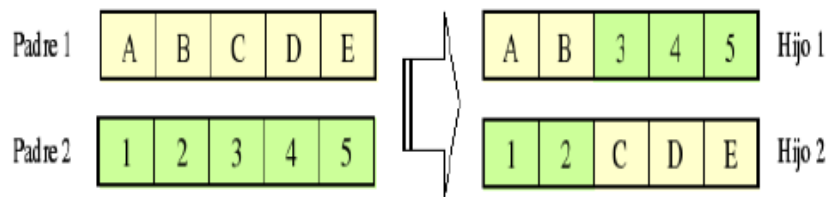


Fig.2.4 Operador de cruce basado en un punto. [6]

2.2.6. Función de mutación [Anexo A, Apartado A.1.6]

La misión de la función de mutación es complementar a la de cruce insertando cambios inesperados en los individuos. La idea es que la función de cruce recorra el espacio de búsqueda en todas direcciones y que la de mutación permita la aparición de individuos diferentes garantizando que ninguna zona del espacio búsqueda tenga una probabilidad cero de ser explorada. Para ello se define como una cierta probabilidad de que un gen, o parte del mismo, pueda ser mutado.

El caso más simple de mutación sería el de cambiar un bit del cromosoma del estado 0 al 1 o viceversa, pero como de costumbre existen multitud de métodos diferentes: por ejemplo, existe una variante llamada *permutación* que consiste en elegir al azar dos elementos de un mismo cromosoma e intercambiar sus posiciones [3, pp.148].

A parte de la elección del método, el problema principal radica en seleccionar un valor apropiado para esta probabilidad. Un valor demasiado alto haría crecer enormemente la diversidad genética (lo que significaría que el espacio de búsqueda sería explorado a fondo). Pero convertiría la optimización en una mera búsqueda aleatoria dificultando la convergencia final del problema. Por el contrario, un valor demasiado bajo reduciría la diversidad dejando al AG a merced del primer óptimo local en el que cayera.

Así pues, no existe un valor correcto, sino más bien un compromiso entre ambos factores. Lo único que parece estar claro es que se recomienda un valor muy pequeño: por ejemplo De Jong sugiere que la probabilidad de mutación sea la inversa de la longitud del cromosoma [13].

2.2.7. Función de inserción [Anexo A, apartado A.1.7]

En lo que se refiere a la función de inserción, su labor consiste en adecuar el tamaño de la población después de cada nueva generación. Esto es debido a que en un principio el AG se define con una población n , pero después de las funciones de selección y cruce a la población n se le han de sumar los m descendientes. Existen implementaciones en las que el tamaño de la población adopta un valor variable, pero lo normal es que sea de tamaño fijo y que tenga que ser “recortado” siguiendo algún criterio.

El criterio más sencillo sería el de no conservar individuos entre generaciones (*Reducción simple* [Anexo A, apartado A.1.7]). El polo opuesto sería utilizar un criterio más elitista de manera que se seleccionen los n individuos mejores y se descarten el resto. Esto aumentaría la velocidad de convergencia a costa de reducir la diversidad con el consiguiente riesgo que eso comporta.

Un camino intermedio parece la solución más completa, aunque como de costumbre cada método tiene aspectos positivos y negativos que se han de sopesar antes de decidirse por cualquiera de ellos.

2.2.8. Criterio de parada [Anexo A, apartado A.1.8]

En pocas palabras, el criterio de parada consistiría en un baremo para decidir cuando el AG ha alcanzado la solución óptima y podemos dar por finalizada la búsqueda. El método más sencillo sería definir un número máximo de generaciones. No es la solución perfecta puesto que en ocasiones la solución óptima aparece antes y el resto del tiempo hasta que se detiene el AG se convierte en ciclos de proceso innecesario. O todo lo contrario, de manera que el AG no dispone de ciclos suficientes y finaliza antes de alcanzar su objetivo.

Otro método consiste en tener en cuenta la convergencia genética de la población y definir un criterio de parada basado en su saturación genética. Según De Jong se dice que un gen ha convergido cuando el 95% de la población comparte el mismo valor para dicho gen [13]. De esta forma se podría establecer un criterio de parada tal, que se pueda concluir la labor del AG cuando un número determinado de genes del cromosoma hayan convergido.

Igualmente, el método anterior tampoco es un método recomendable al cien por cien. Hay que recordar que al AG sigue un método pseudoaleatorio de forma que en ocasiones la convergencia genética puede necesitar demasiadas generaciones. Quizás la solución más adecuada sería combinar ambos métodos.

CAPÍTULO 3. IMPLEMENTACIÓN Y PRUEBAS BÁSICAS

3.1. Implementación de un Algoritmo Genético

Una vez concluida la exposición teórica sobre los objetivos y técnicas utilizadas en los procesos de optimización (Capítulo 1), así como las bases en las que se sustentan los algoritmos genéticos (Capítulo 2 y Anexo A), llega el momento de aplicar todo lo explicado y comprobar que la teoría y la práctica van de la mano.

En este caso, en lugar de programarlo dedicando el TFC únicamente a ello, utilizaremos una de las múltiples implementaciones libres disponibles dedicando los esfuerzos a corroborar la teoría anteriormente expuesta y a la posterior aplicación hacia un problema de mayor envergadura.

En cuanto a la implementación escogida, recurriremos a uno de los grandes nombres en el campo de la investigación sobre los algoritmos genéticos: el Dr Kalyanmoy Deb³, director del *Kanpur Genetic Algorithms Laboratory* y editor adjunto del *Evolutionary Computation Journal* del MIT entre otros. De las múltiples implementaciones disponibles en la página oficial de sus laboratorios (*Kanpur Genetic Algorithms Laboratory*), hemos seleccionado una de las más simples puesto que implementa a la perfección los aspectos básicos que debe tener un AG y, gracias a su relativa simplicidad, permite interactuar con el código, modificándolo y ampliándolo, de manera que se puedan poner a prueba todos los parámetros principales utilizados en la optimización mediante AG's.

Una vez más, debido al espacio limitado de que consta este TFC los detalles sobre el funcionamiento del código se encuentran en el anexo D, apartado D.1 para que sean consultados si el lector lo desea. No obstante, a continuación repasaremos los aspectos más importantes de la implementación elegida antes de someterla a los test del apartado siguiente.

En esencia el AG implementado se ajusta al modelo del *Algoritmo Genético Simple o Canónico* (ver Apartado 2.2). Permite utilizar tanto variables codificadas en forma binaria como real y se sirve de un parámetro llamado *RIGID* para permitir, o no, que se generen soluciones que estén fuera del espacio de búsqueda. Así mismo y para complementar esta última parte, además de que a cada individuo se le asocie un valor de *fitness* mediante la función objetivo, también se le puede asociar una penalización.

En lo que a funcionamiento se refiere, utiliza una función aleatoria para generar los individuos de la población inicial a partir una “semilla” (número real) introducida por teclado al inicio. El método elegido para escoger a los progenitores es el *Torneo* entre dos individuos elegidos al azar.

³ <http://www.iitk.ac.in/kangal/deb.shtml>

Como función de cruce, utiliza el método *Single Point Crossover* (Cruce en un punto) y se sirve de una probabilidad de cruce P_{xover} para decidir qué individuos se reproducirán sexualmente y cuáles lo harán de forma asexual. Para las mutaciones utiliza, en el caso binario, una probabilidad de mutación para decidir bit a bit si éste debe mutar o no, y para los números reales, un sistema de *mutación polinómica*.

El método de inserción por su parte es el *Simplex*, donde la nueva generación sustituye por completo a la vieja. Por último, el criterio de parada utilizado consiste en definir un número máximo de generaciones a realizar. Además de esto, implementa técnicas de *Sharing* opcionales que en caso de utilización permiten definir el tamaño del nicho mediante el parámetro σ_{share} .

Como podemos ver, es un algoritmo bastante simple, pero que implementa a la perfección los aspectos básicos para entender el funcionamiento de estas técnicas. En el próximo apartado, intentaremos corroborar la teoría antes expuesta mediante unos sencillos test que pondrán a prueba los parámetros más interesantes del AG: el tamaño óptimo de la población, el número de generaciones, la probabilidad de cruce, la probabilidad de mutación, la efectividad de las técnicas de *Sharing* y el tamaño óptimo del nicho. Y para esto hemos elegido cinco funciones, muy simples todas ellas, pero que permitirán observar el funcionamiento del AG en diferentes escenarios.

3.2. Funciones seleccionadas

Para poder poner a prueba los diferentes parámetros del AG es necesario definir algunos escenarios de búsqueda. En este caso no se trata de funciones excesivamente complicadas, más bien lo contrario. Se trata de un conjunto de cinco funciones simples, con valores óptimos conocidos de forma que se puedan evaluar fácilmente los resultados. A continuación podemos ver las ecuaciones a las que responden estas funciones **(ec.3.1)**, **(ec.3.2)**, **(ec.3.3)**, **(ec.3.4)** y **(ec.3.5)**, así como su representación gráfica **(Fig.3.1)**.

$$\text{Función A} \quad y = \frac{x^2}{25} - 10,12345678901234 \quad \forall x \in [-50,50] \quad (\text{ec. 3.1})$$

$$\text{Función B} \quad y = \cos x^2 \times \cos^2 x \quad \forall x \in [-5,5] \quad (\text{ec. 3.2})$$

$$\text{Función C} \quad z = (\cos x * \cos y) * (x + y) \quad \forall x, y \in [-5,5] \quad (\text{ec. 3.3})$$

$$\text{Función D} \quad y = x * \sin x \quad \forall x \in [0,100] \quad (\text{ec. 3.4})$$

$$\text{Función E} \quad y = \begin{cases} \cos 200x & \text{si } x > -\frac{\pi}{400} \\ \cos 200x * 20 & \text{si } x \in [-\frac{\pi}{400}, \frac{\pi}{400}] \\ \cos 200x & \text{si } x < \frac{\pi}{400} \end{cases} \quad (\text{ec. 3.5})$$

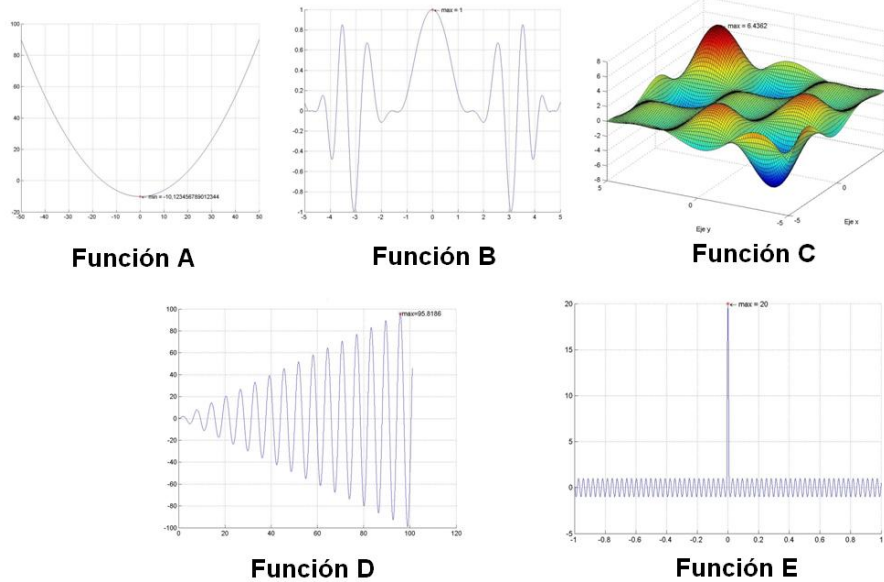


Fig.3.1 Representación gráfica de las funciones elegidas para las pruebas.

Una vez más, por limitaciones de espacio, los motivos e intenciones que nos llevaron a escoger estas funciones quedan relegados al anexo B, aunque no son imprescindibles para comprender los resultados. Baste decir que las **funciones A, B y C** serán las utilizadas para las consideraciones previas (3.3.1) y los test de número de generaciones (3.3.2), población (3.3.3), probabilidad de cruce (3.3.4) y mutación (3.3.5). Por otro lado, la función D ha sido escogida específicamente para ser utilizada en el test de *Sharing* (3.3.6) y por último la función E se ha diseñado específicamente para comprobar la efectividad del AG en uno de sus escenarios más desfavorables (3.3.7).

3.3. Test y resultados

3.3.1. Consideraciones previas

Antes de comenzar con los test propiamente dichos y con la intención de poder evaluar los resultados de la forma más eficiente posible hay que tener en cuenta un par de consideraciones: por un lado, es necesario implementar un método para evaluar el tiempo de proceso que ocupa el AG. Este método consistirá en un sencillo contador que se iniciara al comienzo de la evolución del AG (una vez que ya se hayan introducido todos los parámetros por teclado) y que se detendrá al final, evitando los procesos que no sea propiamente de la

evolución (impresión de ficheros de salida, etc.) de manera que se evalúe exclusivamente el tiempo de proceso evolutivo.

Por otro lado, y en referencia también al tiempo de proceso, hemos dicho que el AG puede trabajar con variables reales o binarias. En concreto las variables reales utilizan *double's* para guardar los resultados. Esto traducido en bits significaría que utilizan un espacio de 64 bits por variable. Para ser justos, y sobre todo para poder comparar correctamente los datos deberemos escoger una codificación binaria de similar magnitud. La idea más simple sería codificar cada variable binaria con 64 bits dejándola en igualdad de condiciones, pero como veremos más adelante, eso puede no resultar lo más eficaz.

En primer lugar, es evidente que la precisión de los resultados aumentará conforme aumente el número de bits (el número podrá tener más decimales). Pero debido a la implementación y al hecho de que el lenguaje escogido (en este caso c) presenta mejores tiempos de proceso trabajando con variables reales que con tiras de bits, veremos que los tiempos de proceso de variables en binario son muy superiores a los de variables reales (del orden de 3 a 10 veces dependiendo del número de bits usados). Así pues, se trataría de encontrar un punto intermedio entre precisión y tiempo de proceso para el AG codificado en binario. En los gráficos siguientes (**Fig.3.2** y **Fig.3.3**) encontraremos mediciones sobre los tiempos de proceso y precisiones obtenidas para las funciones A, B y C en función de la longitud del cromosoma.

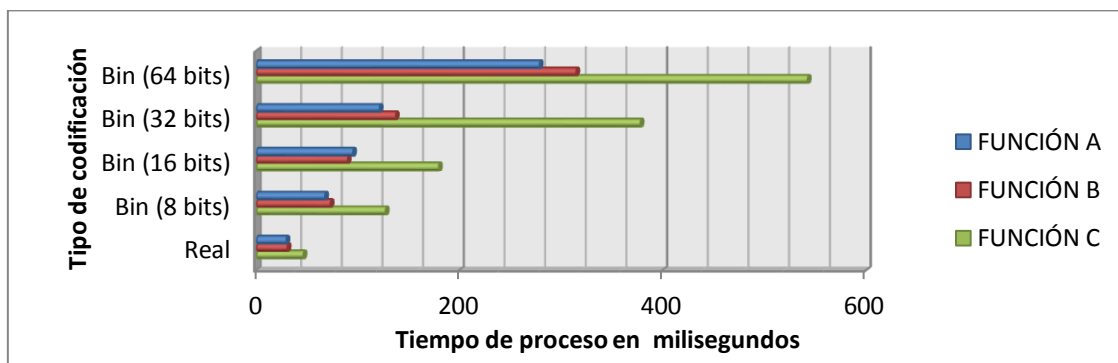


Fig.3.2 Tiempo de proceso para diferentes longitudes de cromosomas binarios.

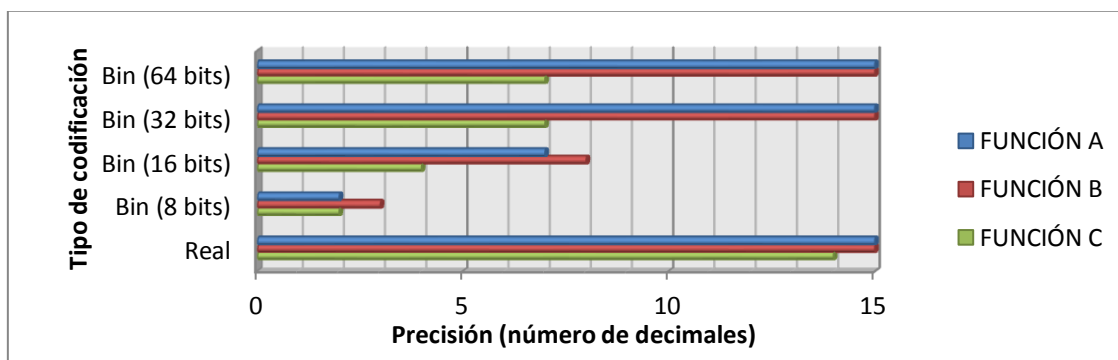


Fig.3.3 Precisión en los resultados para diferentes longitudes de cromosoma.

Como podemos observar, los tiempos de proceso aumentan de manera notable al aumentar el número de bits. En cambio la precisión de los resultados parece estancarse entre los 32 y los 64 bits. Así pues, a igualdad de resultados y para perjudicar lo mínimo posible a la codificación binaria (en lo que a tiempos se refiere), de ahora en adelante utilizaremos 32 bits en lugar de 64 para las variables binarias.

Por último, la idea básica de estos test es la de corroborar la teoría expuesta sobre los diferentes parámetros que definen un AG evaluándolos uno a uno. Ahora bien, en cada caso el valor que utilizaremos para el resto de los parámetros será el teóricamente correcto.

Así pues, definiremos los parámetros “teóricamente correctos” como siguen: población de 50 individuos, 32 bits por variable binaria, probabilidad de cruce alta (0,9) y probabilidad de mutación muy baja, de 0,05 (cercana a la inversa de la longitud del cromosoma (apartado 2.2 o Anexo A, apartado A.1.6) [13].

3.3.2. Test de población

Este test pretende observar los diferentes resultados que se obtienen sobre una misma función haciendo variar el tamaño de la población entre 4 y 100 individuos. El proceso se repite 30 veces para cada tamaño de la población y para cada una de las tres funciones A, B y C para conseguir un valor medio.

El objetivo es contrastar de forma práctica lo expuesto en la teoría y al mismo tiempo encontrar un valor óptimo para el parámetro de la población. Para cada población se evalúa el número de generaciones máximo, mínimo y medio en el que se alcanza la solución, así como el tiempo de proceso requerido. Seguidamente veremos un conjunto de gráficas en las que aparecen reflejados todos estos parámetros (**Fig.3.4**, **Fig. 3.5**, **Fig. 3.6**, **Fig. 3.7**, **Fig. 3.8** y **Fig. 3.9**).

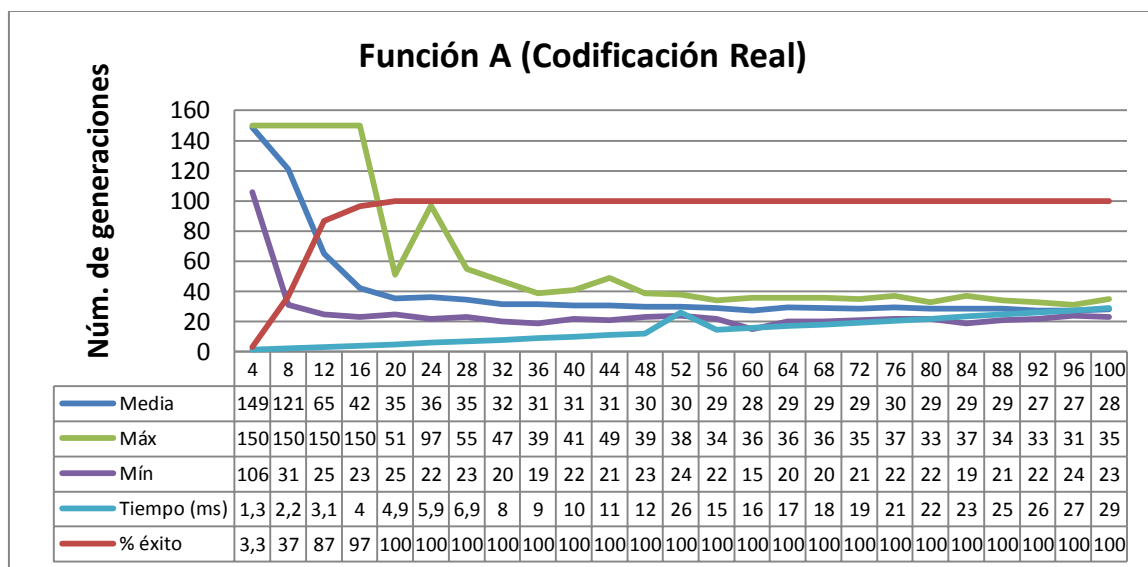


Fig.3.4 Resultados del test de población en la función A (codificación real)

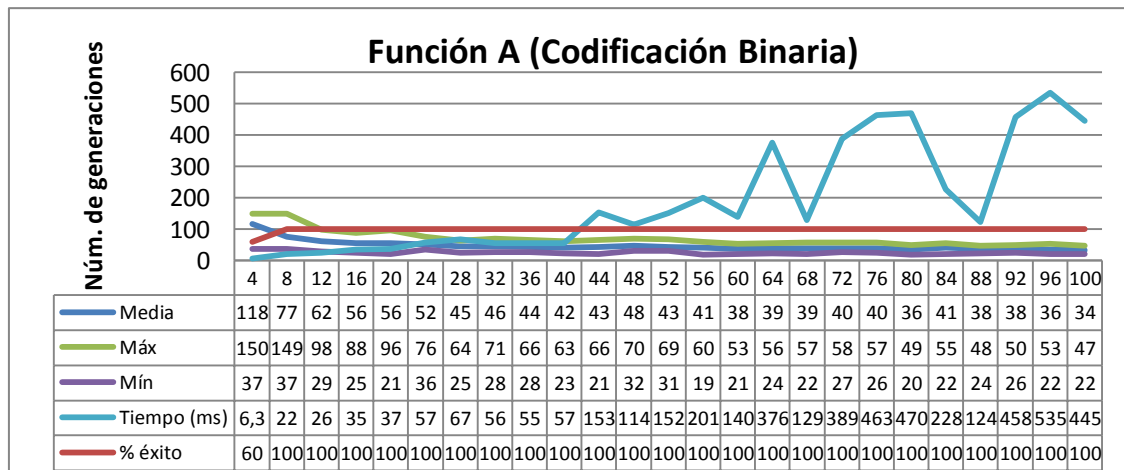


Fig.3.5 Resultados del test de población en la función A (codificación binaria)

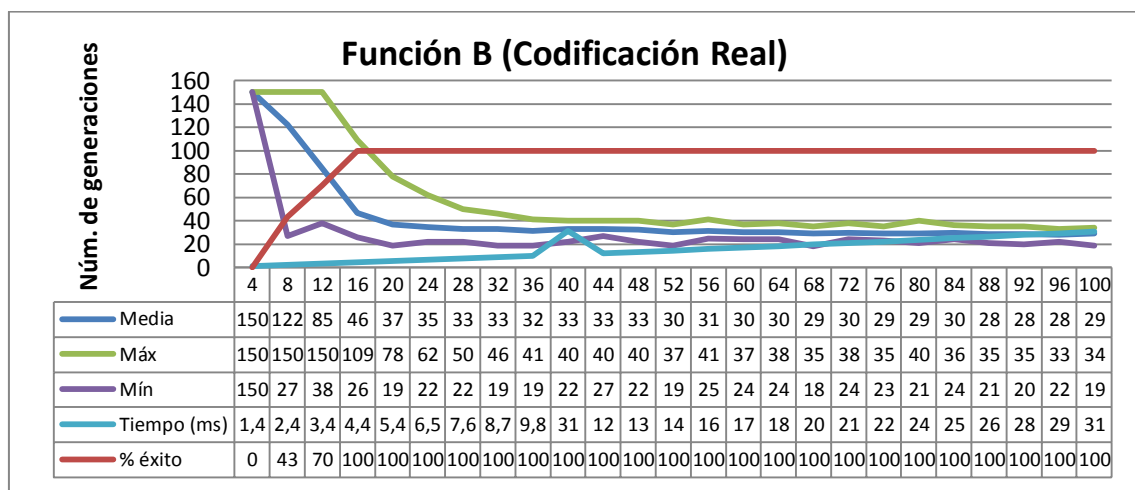


Fig.3.6 Resultados del test de población en la función B (codificación real)

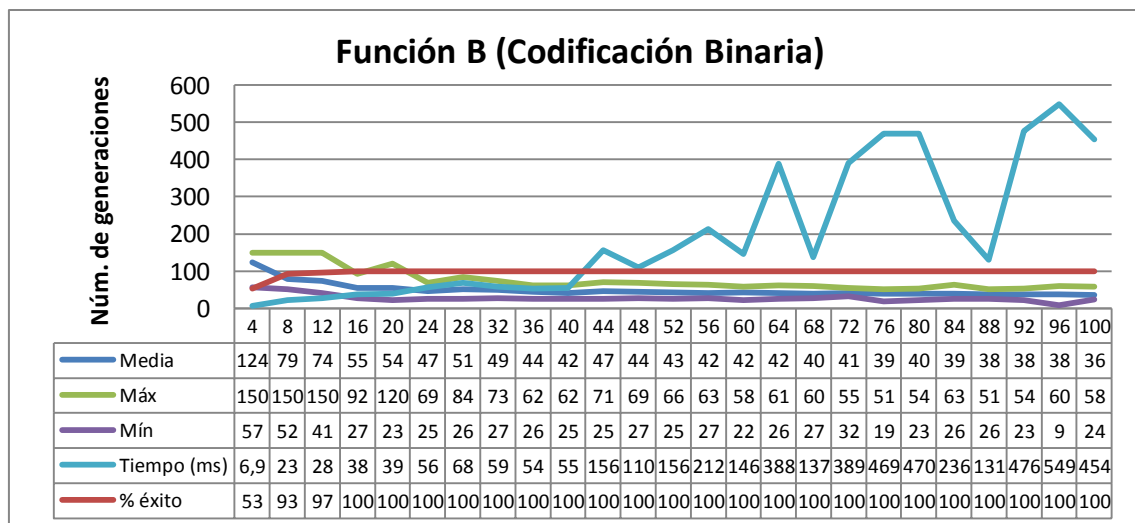


Fig.3.7 Resultados del test de población en la función B (codificación binaria)

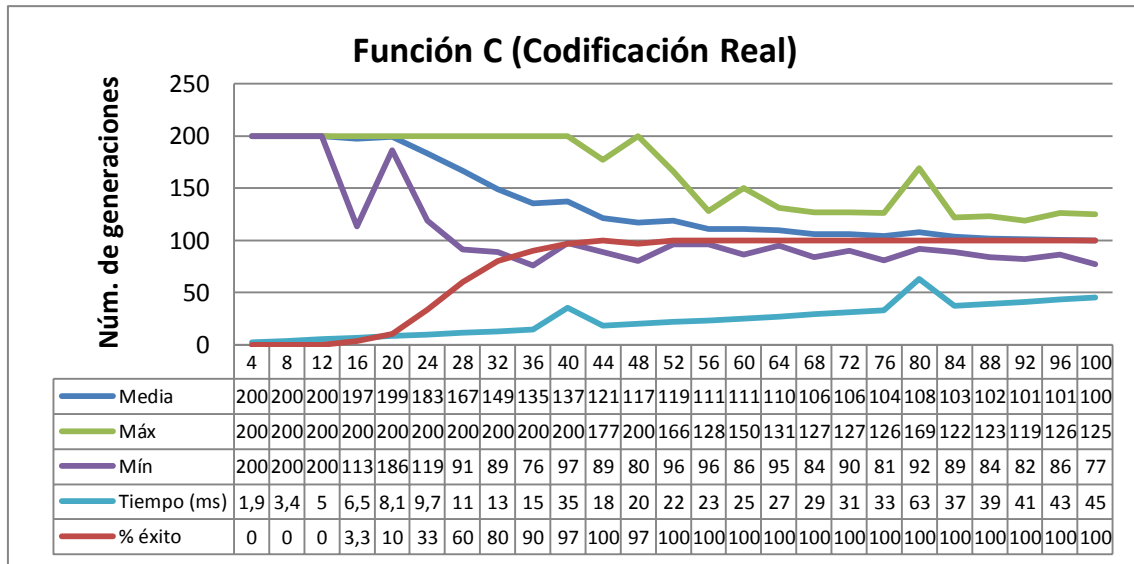


Fig.3.8 Resultados del test de población en la función C (codificación real)

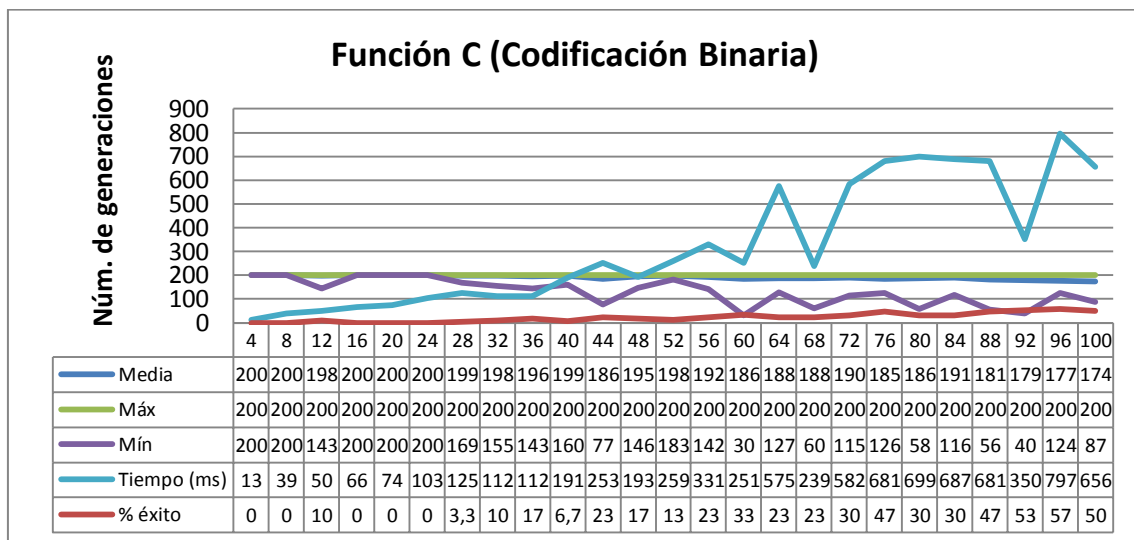


Fig.3.9 Resultados del test de población en la función C (codificación binaria)

Observando las gráficas podemos concluir que el intervalo de valores óptimos para estos casos particulares se encuentra entre los 40 y 48 individuos. Más allá de estos valores los tiempos de proceso suben demasiado sin aportar grandes mejoras.

Por el contrario, por debajo de estos valores los tiempos de proceso no son sustancialmente mejores como para compensar la falta de acierto que sufre el AG en algunos casos.

Así pues, por división simple concluiremos que el valor óptimo debe ser de 44 individuos, que comparándolo con los 50 que recomendaba la teoría ofrece un resultado más que aceptable.

3.3.3. Test de número de generaciones

En principio, este AG utiliza un criterio de parada basado únicamente en el número de generaciones máximas definidas por el usuario. Así pues, el interés de este test se centra en encontrar un número óptimo de iteraciones de forma que no se desperdicie tiempo de proceso en generaciones innecesarias, ni tampoco se finalice precozmente el AG sin darle tiempo a alcanzar la solución óptima. Si bien es cierto que ningún número garantiza el 100% de efectividad siempre, mediante estas pruebas encontraremos un valor aceptable que garantice el éxito en la mayor parte de los casos.

Para esto el test evaluará una población de tamaño definido (el valor óptimo antes encontrado) haciendo variar el número de generaciones desde 10 hasta 150 en 100 ocasiones para cada una de las tres funciones (una vez codificadas en binario y otra vez en real). A continuación podemos ver un gráfico (**Fig. 3.10**) donde se ve representado el número medio de generaciones necesarias para que el AG alcance el objetivo para cada función y codificación.

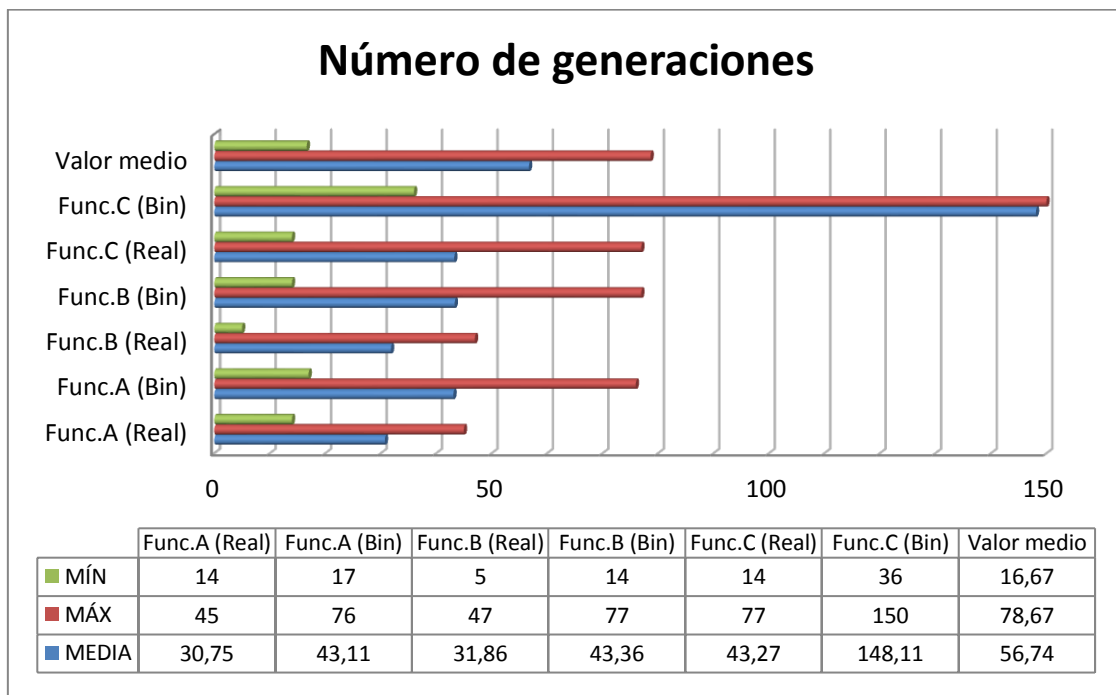


Fig. 3.10 Resultados del test de generaciones en las funciones A, B y C

Analicemos ahora los resultados obtenidos: la información que aporta el valor mínimo es prácticamente anecdótica (puesto que es puro azar que el óptimo se revele tan pronto) así que es mejor ignorarla. Observando los valores medios vemos que se mantienen constantes alrededor de las 40 generaciones salvo para el caso de codificación binaria con múltiples variables (función C). Este dato junto con el obtenido en las consideraciones previas sobre el tiempo de proceso, empieza a señalar que la codificación binaria no es muy recomendable para problemas de varias variables.

Dejando eso a un lado de momento, si tomamos los resultados reales y binarios por separado observamos que para problemas de una sola variable (funciones A y B) mantiene unas cifras más o menos constantes. Teniendo en cuenta el bajo tiempo de proceso que tiene el AG codificado en real podemos definir el valor óptimo de ambos como el más alto de las dos, es decir unas 80 generaciones.

Si tomamos ahora los datos obtenidos para el problema de dos variables (función C) observamos que las 80 generaciones se quedan cortas. Los valores prácticamente se han duplicado tanto para la codificación real como para la binaria. Siguiendo la misma lógica aplicada en el caso anterior, el valor óptimo en este caso sería de 150 generaciones.

Parece pues existir una relación directa entre el número de variables y el factor que multiplica al número de generaciones. A partir de todo esto, podemos concluir que el número de generaciones óptimas (que será utilizado como criterio de parada) para funciones de una o dos variables, parece responder a la **ecuación 3.2**.

$$\text{Criterio de parada} = (\text{Núm. Variables} \times 80) [\text{Generaciones}] \quad (\text{ec. 3.2})$$

Para dar por válida esta hipótesis a nivel general sería necesario hacer pruebas con funciones de 3 o más variables, pero esto queda fuera de los propósitos de este TFC. La función de este test era determinar cuál era el valor óptimo de generaciones necesarias para definir correctamente un criterio de parada, y en vista de que la teoría solo nos habla de escoger el valor más adecuado, podemos concluir que por el momento el valor que hemos elegido es tan válido como cualquier otro. Veremos después en futuras pruebas si hemos hecho la elección correcta o no.

3.3.4. Test de probabilidad de cruce

Una vez elegidos a los progenitores, es la probabilidad de cruce la que decidirá si estos intercambiarán su material genético, permitiéndoles evolucionar, o si se limitarán a pasar a la nueva generación sin sufrir ningún cambio. Teóricamente ambos casos son necesarios, eso sí, en su justa medida.

Partiendo de esta base el test siguiente intentará reflejar cómo afecta la probabilidad de cruce al desarrollo del AG. Para esto, haremos variar dicha probabilidad entre 0.0 y 1.0 a intervalos de 0.05 (repetiendo este proceso 30 veces cada vez para obtener una media estadística). Teniendo en cuenta que las funciones que hemos elegido tienen valores óptimos de sobra conocidos, la forma de evaluar el éxito o fracaso del AG consistirá en normalizar el resultado obtenido entre 0 y 1 a partir del valor óptimo conocido. Así, cuanto más próximo a 1 sea el resultado, mejor comportamiento habrá tenido y de esta forma podremos definir cuál es el valor óptimo de la probabilidad de cruce.

A este respecto, la teoría augura que el resultado debería ser un valor alto. Un valor bajo representaría que la población apenas evoluciona y que no avanza de manera útil. Como en casos anteriores, el resto de los parámetros los fijaremos a los valores teóricamente correctos. En la **figura 3.11** y la **tabla 3.1** siguientes podemos ver los resultados obtenidos.

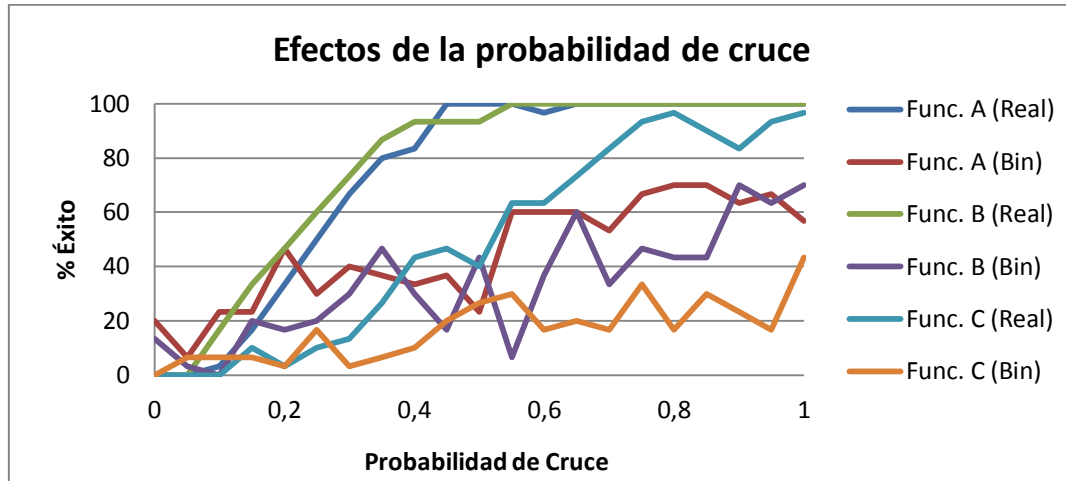


Fig.3.11 Efectividad del Algoritmo Genético en función de la probabilidad de cruce utilizada

Como podemos ver en el gráfico (**Fig. 3.11**) el porcentaje de éxito aumenta cuanto mayor es la probabilidad de cruce (aunque curiosamente los valores más efectivos no corresponden a una probabilidad de cruce igual a 1).

	A (Real)	A (Bin)	B (Real)	B (Bin)	C (Real)	C (Bin)	Media
P. Cruce	0,825	0,825	0,775	0,95	0,9	1	0,88

Tabla 3.1 Valores óptimos de la probabilidad de cruce

Analizando ahora la **tabla 3.1** podemos observar los que serían los valores de la probabilidad de cruce que ofrecerían un porcentaje de éxito más alto. A partir de ellos y con una simple operación alcanzamos el valor que buscábamos: en concreto la probabilidad de cruce óptima debería ser de 0.88, 0.9 para redondear. Es un valor alto como indicaba la teoría. Pero además de esto, este test ha puesto de nuevo de manifiesto la diferencia de resultados que hay entre la codificación real y la binaria, siempre quedando por detrás esta última.

3.3.5. Test de probabilidad de mutación

Este test guarda muchas similitudes con el anterior tanto en objetivos como en metodología. De hecho es prácticamente el mismo, con la salvedad de que en este caso lo que se busca es el valor óptimo de la probabilidad de mutación.

Como se puede ver en el gráfico inferior (**Fig. 3.12**) el porcentaje de éxito mayor se da cuando la probabilidad de mutación adopta valores bajos. Como en el caso anterior, si analizamos los datos numéricos de la tabla inferior (**Tabla 3.2**) podremos calcular el valor óptimo de la probabilidad de mutación.

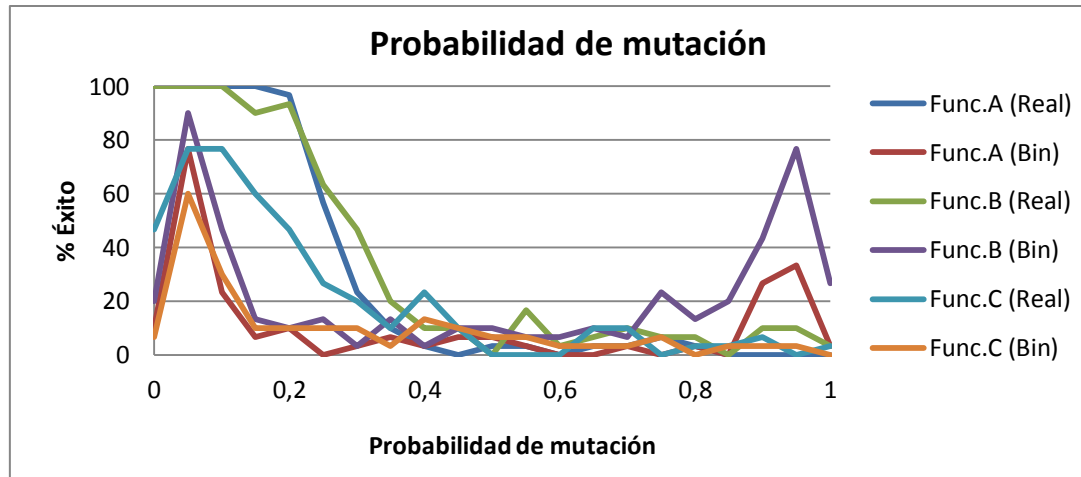


Fig.3.12 Efectividad del AG en función de la probabilidad de mutación utilizada

	A (Real)	A (Bin)	B (Real)	B (Bin)	C (Real)	C (Bin)	Media
P. Mutación	0,075	0,05	0,05	0,05	0,075	0,05	0,06

Tabla 3.2 Valores óptimos de la probabilidad de mutación

En este caso el valor obtenido estaría en torno a los 0.06 (un valor muy bajo como indicaba la teoría). Además, este test confirma la importancia de incluir inesperadas mutaciones en algunos individuos aunque esto no debe restarle protagonismo a la función de cruce que es, al fin y al cabo, la que lleva el peso de la evolución

3.3.6. Test de mutación y cruce simultáneamente

En los apartados anteriores hemos verificado la probabilidad de mutación y la de cruce dejando fijo en cada caso uno de los dos parámetros. Es decir, para calcular la probabilidad de mutación fijábamos el valor de la probabilidad de cruce al valor óptimo teórico y viceversa. De esta forma hemos podido comprobar que los datos teóricos se cumplen y hemos encontrado una combinación de probabilidades que resulta efectiva.

La idea de este test es demostrar que aunque el AG verifica los parámetros óptimos teóricos, no deja de ser un sistema libre donde existen otras combinaciones que también resultan efectivas. Para esto haremos variar la probabilidad de cruce y la de mutación simultáneamente en intervalos de 0,05 en 50 ocasiones cada vez para determinar el valor medio.

Como podemos ver en el gráfico (**Fig.3.13**) existen múltiples zonas donde el AG alcanza el óptimo global en todas las ocasiones. Por ejemplo, un valor muy alto de mutación convertiría el Algoritmo Genético en una simple búsqueda aleatoria que con una mínima ayuda de la probabilidad de cruce podría terminar alcanzando el objetivo buscado.

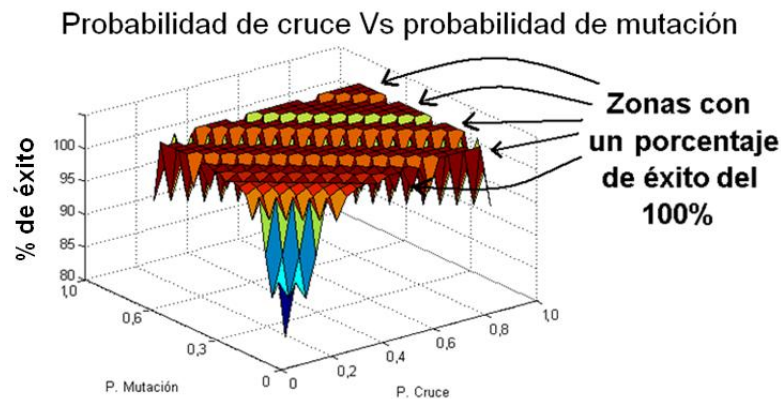


Fig.3.13 Efectividad del AG con diferentes combinaciones de probabilidad de cruce y probabilidad de mutación

La finalidad de esta prueba no es determinar las combinaciones de parámetros que son efectivas. Más bien es mostrar al lector que se trata de un sistema abierto y flexible que permite probar y probar moldeándolo a nuestra elección.

3.3.7. Test de *Sharing*

Llegados a este punto tenemos un AG con todos sus parámetros principales optimizados:

- Población de 44 individuos
- Criterio de parada fijado en 80 generaciones por variable
- Probabilidad de cruce de 0.9
- Probabilidad de mutación optimizada con valor de 0.06

Da la impresión de que habiendo obtenido estos datos ningún problema pueda resistirse, pero la verdad es que no es así. No hay que olvidar que los algoritmos genéticos padecen de problemas intrínsecos (Anexo A, apartado A.3) que requieren de técnicas específicas para intentar superarlos. Un ejemplo claro de estos problemas es el de quedar atrapado en máximos/mínimos locales sin posibilidad de alcanzar otras soluciones mejores.

Las técnicas de *Sharing* (Anexo A, apartado A.3.1) entre otras, intentan paliar estas debilidades, ayudando a mantener una mejor diversidad genética. Partiendo de esta base, y de la premisa de estar utilizando valores adecuados para otros parámetros importantes, intentaremos corroborar los efectos beneficiosos de utilizar estas técnicas.

El funcionamiento de este test es muy similar al de los casos anteriores: la idea es evaluar la función D (especialmente escogida para este fin por su topografía) sin utilizar las técnicas de *Sharing*. Después se evalúa la misma función aplicando dichas técnicas y haciendo variar el tamaño del nicho: en este caso comenzaremos con un tamaño de 1/10 del espacio búsqueda e iremos dividiéndolo por la mitad hasta que alcance los 1/5120 del espacio de búsqueda. Cada proceso se repetirá 100 ocasiones y como en el caso anterior (al ser el óptimo un valor conocido) se medirá el éxito del AG normalizando el resultado entre 0 y 1 con el valor esperado. A continuación podemos ver un gráfico (**Fig.3.14**) con los resultados obtenidos.

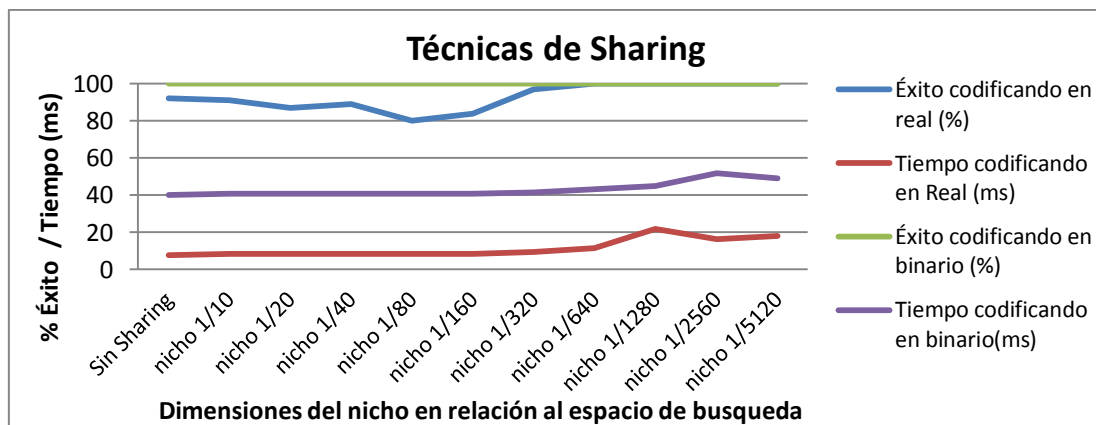


Fig.3.14 Efectividad y tiempo de proceso del AG en el test de *Sharing*

Viendo los resultados observamos un par de cosas destacables: por un lado el sorprendente éxito que presenta la codificación en binario y que, aparentemente, hace innecesario utilizar técnicas extras. Este efecto es producto de la propia codificación binaria en la que un cambio de un solo bit da lugar a un individuo radicalmente distinto, y por otro lado a que la función de mutación evalúa el cromosoma bit a bit, acentuando todavía más este efecto. Esto, como podemos ver, tiene aspectos positivos en cuanto a diversidad de soluciones y a la profundidad con la que se explora el espacio de búsqueda. Pero como veremos más adelante, hace del todo inútil utilizar criterios de parada basados en la diversidad genética.

Para luchar contra estos efectos se pueden utilizar codificaciones binarias del tipo *Código Grey* (donde cada individuo se separa de su vecino por un cambio en un único bit [Anexo A, apartado A.1.1]), aunque también sería necesario retocar la función de mutación de manera que sólo permitiera un cambio en un único bit por cromosoma (con la función actual podrían mutar todos los genes de un mismo cromosoma). Por otro lado, observando los resultados para la codificación real, vemos que en este caso las técnicas de *Sharing* sí que ofrecen una cierta mejoría en los resultados. Así pues, centrándonos en la evolución de los parámetros reales podemos concluir que las técnicas de *Sharing* son efectivas y que un valor óptimo para el tamaño del nicho (en relación al tiempo de proceso) sería de 1/640 del espacio de búsqueda aproximadamente en este problema concreto.

3.4. Mejoras

Más allá de implementar diferentes métodos de selección o de cruce (que tal vez mejorarían los resultados, pero que son más bien específicos de cada tipo de problema), optaremos por implementar un par de soluciones básicas y efectivas, en principio, para todo tipo de escenarios.

Por un lado implementaremos un criterio de convergencia basado tanto en el valor óptimo de generaciones obtenido en el test anterior como en la convergencia genética de la población. Definiremos un criterio tal, que se considerará que el AG habrá convergido (y por tanto daremos por finalizado el proceso), cuando más del 85% de su población se encuentre dentro de un cierto intervalo del espacio de búsqueda, definiendo este intervalo como una millonésima parte del espacio de búsqueda total. Además de esto, y teniendo presente el factor aleatorio propio de los AG's, esta convergencia deberá producirse en tres ocasiones consecutivas y sobre el mismo valor. De esta manera evitamos que una agrupación puntual y prematura de la población nos confunda y finalice el proceso.

Por otro lado implementaremos algunos cambios en la función de inserción para solucionar una particularidad que esta presenta: actualmente, el proceso de inserción consiste básicamente en no conservar ningún individuo de la generación anterior (método *Simplex*). Esto conlleva que a veces los mejores individuos se degraden por cruces desafortunados o mutaciones imprevistas perdiendo la posibilidad de explorar esa prometedora zona del espacio de búsqueda. El cambio que se propone es el de asegurar que los mejores individuos pasen a la siguiente generación independientemente de que se hayan reproducido previamente. Una vez implementados estos cambios, probaremos su efectividad con las tres funciones utilizadas anteriormente (A, B y C) y las evaluaremos con y sin las mejoras en cuanto a precisión y tiempo de proceso se refiere (**Fig.3.15** y **Fig.3.16**).

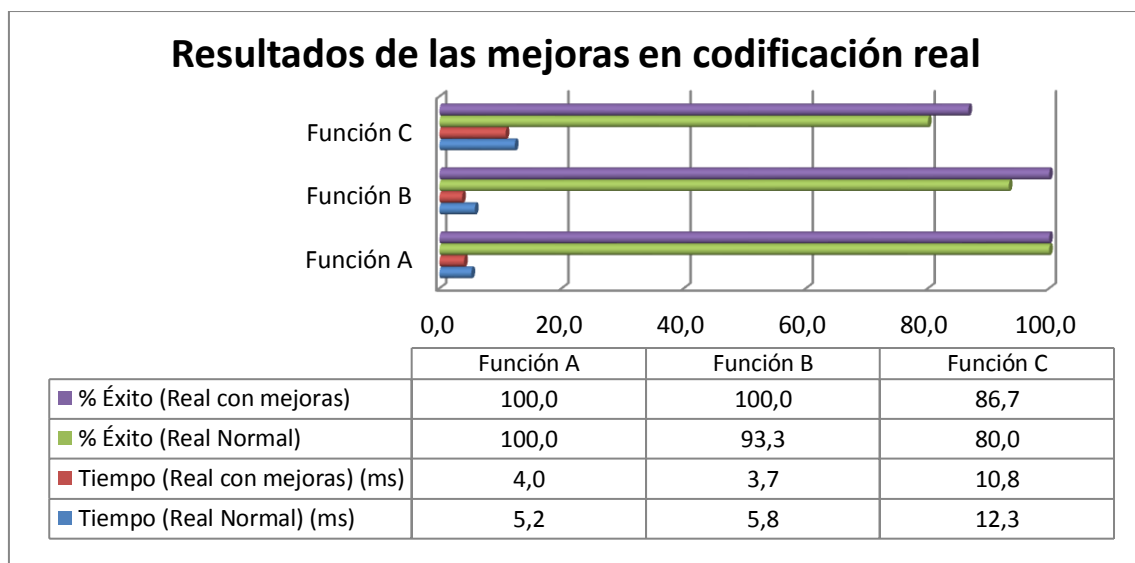


Fig.3.15 Resultados para las funciones A, B y C con y sin mejoras (real)

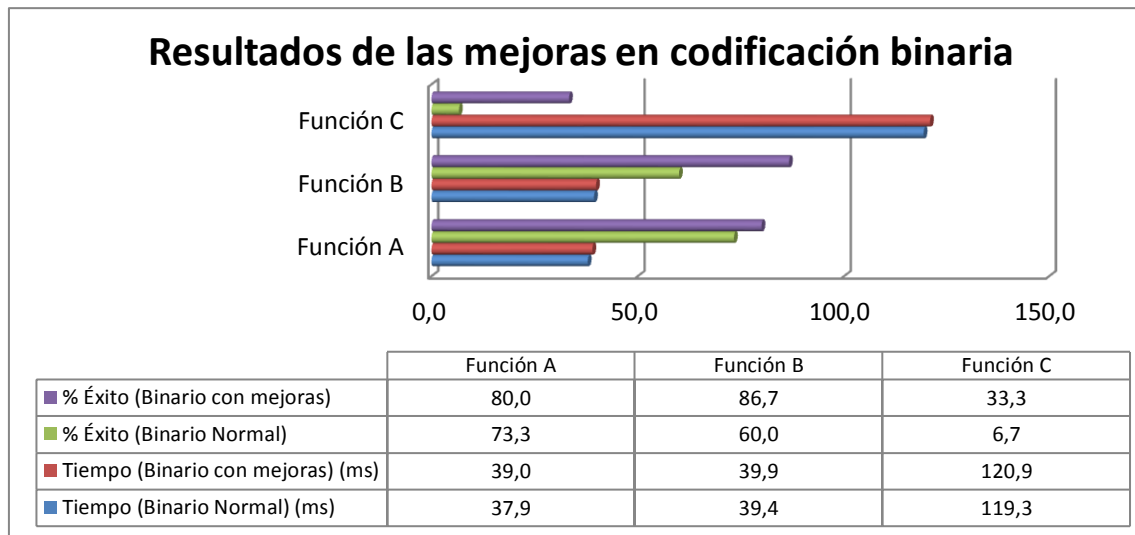


Fig.3.16 Resultados para las funciones A, B y C con y sin mejoras (binario)

En ambos casos puede verse una mejoría en los resultados obtenidos consiguiendo que éstos se acerquen cada vez más a los valores de los óptimos conocidos.

En lo que a tiempos se refiere, el resultado varía en función del tipo de codificación. Para el caso de la codificación real se observa una considerable reducción en los tiempos de proceso debido a que el criterio de convergencia da por finalizado el proceso de búsqueda antes de concluir las 80 generaciones.

En el caso de la codificación binaria el efecto es el contrario. El tiempo de proceso aumenta ligeramente debido a los cálculos necesarios para evaluar la diversidad genética de la población. Y en cambio el AG se ve obligado siempre a agotar los 80 ciclos impuestos por el criterio de parada debido a que la población binaria no acaba de converger hacia ningún valor concreto. Este efecto como explicamos en el apartado anterior, es provocado en parte porque la función de mutación se aplica a todos los bits por separado sin importar si otros han mutado ya o no. Pero sobre todo por la codificación binaria en sí misma puesto que un cambio de un bit puede dar lugar a un individuo radicalmente alejado de su antecesor.

Aun así podemos ver como los resultados mejoran tanto en una codificación como en otra, pero en el caso de la codificación binaria el criterio de parada basado en la diversidad debería ser suprimido puesto que solo aumenta el tiempo de proceso.

A lo largo de este capítulo hemos ido viendo como la selección apropiada de los parámetros determina en gran medida la efectividad del AG. Hemos optimizando todas estas características y hemos puesto en práctica algunos de los métodos específicos (*Sharing*) para mejorar su productividad en determinados escenarios. Incluso hemos implementado algunos métodos que mejoran su efectividad y, en ocasiones, reducen su tiempo de proceso. Aun

así, existen multitud de escenarios en los que estos métodos no son recomendables y su rendimiento es bastante deficiente (Anexo A, apartado A.3.2). De las funciones definidas en el capítulo 3.2 hemos estudiado en profundidad las cuatro primeras, y hemos dejado la última para el final puesto que se trata de uno de los múltiples casos en los que los AG's no son los métodos más indicados para resolver el problema.

Así pues, utilizaremos la función E para realizar el análisis final del AG. Analizaremos primero la función de forma simple (es decir sin ningún tipo de técnica o mejora). Seguidamente repetiremos el análisis utilizando las técnicas de *Sharing*, después con las mejoras implementadas anteriormente y finalmente con todo a la vez. En cada escenario repetiremos las pruebas en 50 ocasiones y normalizaremos los resultados a 1 (entendiendo como 1 la solución perfecta) para determinar finalmente los resultados medios que pueden verse en la **figura 3.17**.

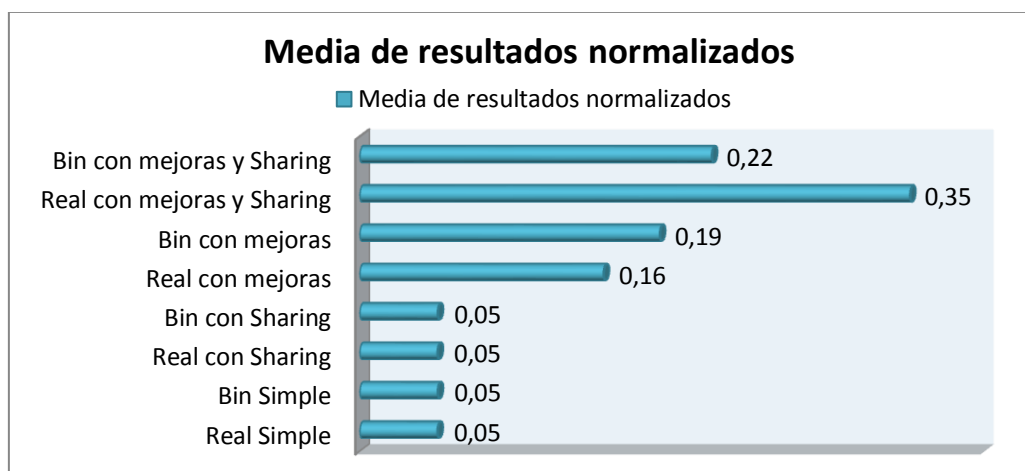


Fig.3.17 Resultados medios normalizados para la función E

Como podemos ver, al trabajar con problemas similares al de la función E ni la optimización de parámetros ni las mejoras incluidas consiguen que su rendimiento sea aceptable. De las 50 veces respectivamente que se repitió el proceso en el caso simple o con *Sharing*, ni una sola vez fue encontrada la zona del valor óptimo. Al aplicar la nueva función de inserción se consigue una leve mejoría (la zona es encontrada entre 7 y 10 veces dependiendo de la codificación). Pero ni siquiera aplicando las mejoras junto con las técnicas de *Sharing* (especialmente indicadas para diversificar la población ayudando a encontrar zonas de interés para explorar) se consigue alcanzar el 50% de efectividad.

Como podemos ver los algoritmos genéticos son métodos muy potentes pero no son la solución a todos los problemas. En este caso, como en muchos otros, su comportamiento hace que dependan enteramente de un golpe de suerte que los lleve a la zona adecuada y así su sistema evolutivo pueda llevarles a mejorar la solución hasta el valor óptimo.

CAPÍTULO 4: APLICACIÓN PRÁCTICA

4.1. Descripción del problema.

En los capítulos anteriores hemos visto las bases en las que se fundamentan los algoritmos genéticos, así como un conjunto de pruebas simples que intentaban corroborar de forma práctica lo descrito en la teoría. Hemos visto que los algoritmos genéticos son métodos efectivos en lo que a funciones continuas se refiere. Sería pues momento de ponerlos a prueba con otro tipo de problema y ver además si resultan unos métodos tan potentes como se les presupone en la teoría.

En este caso el problema en cuestión sería el de organizar la carga dentro de las bodegas de un avión. Un problema que puede parecer simple pero que como veremos más adelante tiene implicaciones importantes a la hora del vuelo.

4.1.1. Efectos negativos de la distribución de la carga

Independientemente del modelo o tamaño del avión elegido, la forma de incluir la mercancía en sus bodegas se realiza de la misma forma: dentro de contenedores. Es decir que la mercancía no se amontona de cualquier manera dentro de las bodegas, sino que va organizada y sujeta dentro de una serie de contenedores con diferentes formas y volúmenes dependiendo del tipo de avión o necesidades de la carga. En el anexo C, concretamente en el apartado C.1 podemos ver un listado de los contenedores existentes y sus características principales.

Así pues, tenemos que en un vuelo cualquiera, nuestras pertenencias y las de los otros ocupantes viajan encapsuladas dentro de estos contenedores que a su vez van distribuidos a lo largo de las diferentes bodegas del avión. Ahora bien, más allá de la pregunta de si “me habrán perdido otra vez las maletas” ¿existe algún otro problema que justifique nuestro interés y la utilización de métodos de optimización? La respuesta es que sí, y el culpable es el centro de gravedad del avión. No es que la carga sea el único factor que actúa sobre la posición del centro de gravedad: cualquier elemento (incluidos los ocupantes) tiene efectos sobre su localización final, pero en concreto la carga suele tener los efectos más determinantes.

Como en cualquier otro cuerpo, el centro de gravedad del avión es también su punto de equilibrio y como tal tiene una relación directa con la estabilidad del avión (**Fig.4.1**). Evidentemente si el avión fuera un cuerpo cerrado en el que no se pudiera introducir ni extraer nada, el centro de gravedad (c.g. a partir de ahora) sería un punto fijo, pero este no es el caso. El c.g. puede desplazarse tanto lateral como longitudinalmente (**Fig.4.1**) dentro de unos márgenes establecidos por el fabricante. Fuera de estos límites los efectos negativos de este desplazamiento pueden hacer que el avión se vuelva ingobernable.

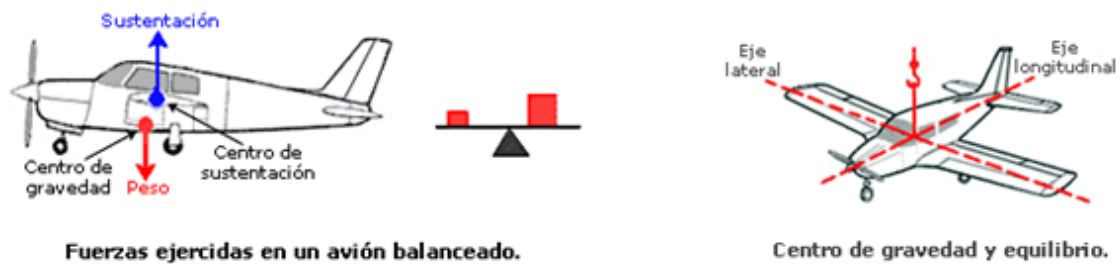


Fig.4.1 Diagrama de fuerzas básicas que actúan sobre el centro de gravedad (izquierda) y representación de los ejes del avión (derecha)

Antes de continuar con la exposición es importante remarcar que los temas de carácter aeronáutico en los que estamos entrando son sumamente complejos y cuentan con una amplísima terminología propia. De manera que, para hacer este TFC entendible a todos los niveles y no desviarnos demasiado del tema, se intentarán “traducir” y simplificar las explicaciones de forma que sean accesibles para todos los públicos, tanto para los expertos como para los recién llegados como es mi caso. Dicho esto, aquí podemos ver brevemente los efectos de un centro de gravedad desviado⁴.

- **Desplazamiento lateral del centro de gravedad:** Es menos perjudicial que el desplazamiento longitudinal, aunque implica un trabajo superior para el alerón del lado más cargado a la hora de equilibrar el vuelo. Incrementa la resistencia al viento haciendo más costoso el despegue y el aterrizaje y aumentando el consumo de combustible.
- **Desplazamiento longitudinal del centro de gravedad:** Un centro de gravedad adelantado implica que el avión resulte más pesado de morro necesitando un esfuerzo extra para levantar el mismo en el despegue. De entrada esto provoca que se necesite una mayor distancia de pista de despegue.

Además, en el intento de contrarrestar la tendencia a “picar de morro” del avión puede provocar que al recoger el tren de aterrizaje este entre en pérdida⁵. Dicho de otra forma, dificulta el despegue y puede llegar a frustrar el aterrizaje.

Un centro de gravedad retrasado es quizás la peor opción de todas. Provoca dificultades en el despegue así como dificultades para mantener el ángulo y velocidad de ascenso correctos. Durante el vuelo el avión se comporta de una manera inestable incrementando el peligro de entrar en *pérdida* en cualquier maniobra (resultado aún más complicado de recuperar puesto que entraría en *pérdida* la cola antes que las alas). También afecta negativamente al aterrizaje

⁴ www.manualdevuelo.com

⁵ [http://es.wikipedia.org/wiki/Entrar_en_pérdida](http://es.wikipedia.org/wiki/Entrar_en_p%C3%A9rdida)

debido a la tendencia del aparato a hundirse de la cola, en concreto en el momento de recoger el tren de aterrizaje.

Queda pues en evidencia la importancia de posicionar correctamente el centro de gravedad del aparato. Estos efectos antes descritos son para casos en los que el c.g. está claramente desviado de su posición adecuada, aunque son igualmente perjudiciales en casos de desviaciones más leves.

Como hemos comentado antes existen una serie de tolerancias definidas por el fabricante de forma que no es necesario un centrado perfecto del c.g. Además, existen unos mecanismos para compensar estas desviaciones y conseguir un vuelo más estable. El cálculo de estas contramedidas, así como del centro de gravedad, serán el núcleo del problema final y por tanto resultaría interesante conocer, ni que fuera brevemente, cómo se obtienen. El problema nuevamente es el limitado espacio del que disponemos. En el anexo C, apartado C.2 el lector encontrará una breve explicación de cómo se calcula el c.g., siempre sin entrar demasiado en detalles, pero que puede ayudar a entender mejor el proceso que se va a realizar.

Para no perder de vista el problema y no tener que recurrir a los anexos diremos que el c.g. depende de todas y cada una de las masas que entran dentro del avión. Tanto la carga, como el fuel, los pasajeros e incluso la comida del catering. Todo absolutamente modifica en mayor o menor medida la posición del c.g., y para simplificar el cálculo se utilizan unas hojas tabuladas donde introduciendo el peso de cada zona obtenemos la posición final del mismo.

Como es evidente después de contabilizar todas estas contribuciones el c.g. suele quedar desplazado. Si el desplazamiento es leve y se encuentra dentro de los márgenes que ofrece el fabricante puede ser compensado. Esta compensación se consigue mediante una superficie móvil llamada TRIM que tiene un rango de acción que va desde los $-2,5^\circ$ hasta los $2,5^\circ$ y que también se obtiene en la misma tabla antes mencionada (Anexo C, apartado C.2).

4.1.2. Espacio de búsqueda del problema

Como hemos dicho al inicio de este capítulo, nuestro objetivo es utilizar los algoritmos genéticos para crear un sistema que optimice la distribución de las mercancías que viajan en un avión (en nuestro caso un Airbus A320 pero podría ser cualquier otro).

La manera de hacer esto sería informatizar estas hojas tabuladas de las que hablábamos en el apartado anterior y cuyo proceso detallado se encuentra disponible en el anexo C, apartado C.3. Una vez informatizadas estas hojas podremos determinar, a partir de las diferentes masas introducidas, la posición del c.g. y del ángulo de compensación TRIM. Este último parámetro será el más importante para nosotros puesto que será el que intentaremos optimizar buscando que sea cero o lo más próximo posible a cero.

Por último es importante recordar que todas las masas introducidas en el avión afectan a la posición del c.g.. Para hacer el sistema lo más real posible el resto de los parámetros (pasajeros, fuel, etc.) serán libres y estarán generados como variables aleatorias de forma que se pueda probar el sistema en cualquier condición.

Con todos estos datos ya podemos hacernos una idea clara de cuál es el objetivo que buscamos, y por tanto del espacio de búsqueda en el que nos moveremos. Dejando a un lado esos otros parámetros que no podrán manipularse, lo que tendremos será un número indeterminado **B** de “bultos” a distribuir dentro de las bodegas del aparato. Una posible codificación sería definir cada solución como un vector de posiciones donde cada posición representa a un bulto. Dentro de cada posición se encontrará la ubicación del mismo, (o lo que es lo mismo, el identificador de uno de los **C** contenedores). En la figura siguiente (**Fig.4.2**) podemos ver un ejemplo de esta codificación para un sistema de 4 contenedores ($C = 4$) y 12 bultos ($B = 12$), y de su posterior descodificación dando lugar al desglose de bultos por contenedor.

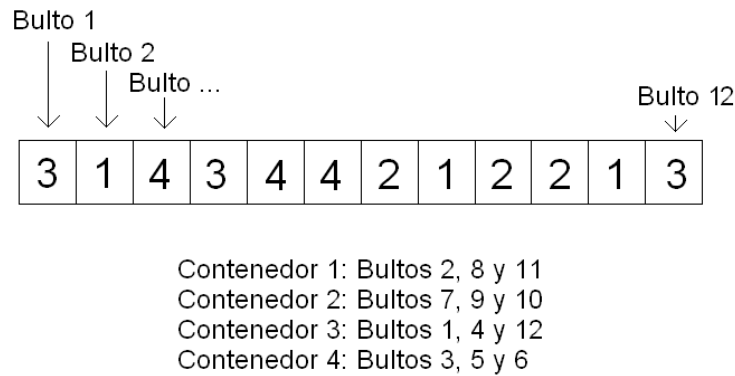


Fig.4.2 Ejemplo de codificación

A partir de esto, obtendremos que nuestro espacio de búsqueda se compone por todas las posibles combinaciones de estos **C** contenedores a lo largo de las **B** posiciones que tiene el vector (**ec.4.1**).

$$\phi(C^B) \quad (\text{ec. 4. 1})$$

Si además suponemos que es posible que toda la carga no pueda ser transportada (exceso de volumen, peso o imposibilidad física de obtener un vuelo seguro) deberemos incorporar un *contenedor virtual* más que represente la posibilidad de quedarse en tierra. De esta forma, el espacio de búsqueda de nuestro problema pasaría a definirse de la forma siguiente (**ec.4.2**).

$$\phi((C + 1)^B) \quad (\text{ec. 4. 2})$$

4.2. Por qué utilizar Algoritmos Genéticos

A la vista de lo propuesto sería natural preguntarse si los algoritmos genéticos son la opción más adecuada. Por un lado, la mayor parte de los métodos de optimización *tradicionales* no son viables puesto que este sistema no sigue ninguna ecuación (o como mínimo sería muy difícil de modelar) que pueda maximizarse o minimizarse. Otra posibilidad sería probar todas las posibles combinaciones valiéndonos de la potencia de los procesadores actuales.

Supongamos pues un caso particular: imaginemos que tenemos un vuelo normal Airbus A320 con una ocupación aproximada de un 70%. Teniendo en cuenta que el aforo máximo del aparato es de 180 personas, un 70% supondría 126 pasajeros. Supongamos un caso sencillo en que cada pasajero lleva una sola maleta (126 maletas) y supongamos además que ninguna se queda en tierra.

Por otra parte, en lo referente a contenedores el Airbus A320 permite cargar 7 contenedores AKH (**Fig.4.3**) y además dispone de otra bodega trasera que podría considerarse como otro contenedor con volumen y peso máximo diferentes.

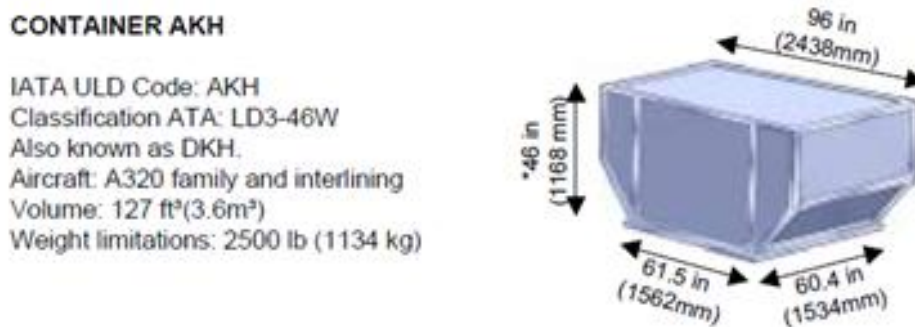


Fig.4.3 Detalle del contenedor AKH [15]

Así pues, en este caso simple tendremos 126 bultos distribuibles en 8 contenedores reales (7 AKH's más la bodega 5). Aplicando la fórmula anterior (**ec.4.1**) nos da el siguiente resultado (**ec.4.3**).

$$\phi C^B = 8^{126} \approx 6,16 \times 10^{113} \text{ soluciones posibles.} \quad (\text{ec. 4.3})$$

Evidentemente el resultado es un número enorme. Como también es evidente que dentro de esas $6,16 \times 10^{113}$ soluciones existen multitud de combinaciones que pese a no ser iguales, representan la misma distribución de carga a efectos prácticos. Esto es debido a que la carga viaja dentro de contenedores (y estos a su vez en bodegas) de forma que el orden de las maletas en el interior de estas bodegas no altera el resultado.

Dejando a un lado la complejidad a nivel de programación que implicaría el probar sólo las soluciones “no repetidas”, sería interesante comprobar en qué dimensiones deja esto a nuestro espacio de búsqueda y eso cambia en algo nuestra decisión.

$$\phi' = \dots (\text{ver Anexo C, apartado C.3}) \dots \approx 5,5 \times 10^{70} \text{ soluciones posibles.} \quad (\text{ec. 4.4})$$

El cálculo completo está disponible en el anexo C, apartado C.3, aunque viendo el resultado final (**ec.4.4**) sólo podemos decir que continúa siendo un conjunto enorme de posibilidades. Teniendo en cuenta que el cálculo del índice % MAC de una sola solución y su correspondiente ángulo TRIM tarda aproximadamente 35 nano segundos⁶ nos indica que serían necesarios nada menos que $1,92 \times 10^{63}$ segundos para evaluar todas las posibilidades.

Intentar pasar esta cifra a horas, días, meses o años es absurdo. Además de tener en cuenta que esta cifra crece exponencialmente con el número de bultos. Podemos concluir que es un tiempo demasiado elevado para este cálculo. Por el contrario los métodos *heurísticos* y en concreto los algoritmos genéticos sólo necesitarían unos cuantos millones de cálculos incluyendo todo el proceso, además de garantizar que encontrarán, sino la solución óptima global, una buena solución en un tiempo competitivo.

Bien es cierto que estos métodos tienen ciertas dificultades en algunos tipos de problemas (anexo A, apartado A.3). Pero también es cierto que han tenido grandes éxitos en campos a priori poco prometedores por su carácter aleatorio y poco predecible: por ejemplo la inteligencia artificial en juegos de damas, o incluso el diseño de antenas de radio donde cuentan con la primera patente no humana registrada⁷.

4.3. Especificaciones del código

Una vez conocidos los problemas que presenta una mala distribución de la mercancía y de cómo se calcula el centro de gravedad del aparato podemos pasar al problema propiamente dicho. La idea, ya comentada con anterioridad, es utilizar los AG's para distribuir de forma correcta la carga dentro de un Airbus A320.

⁶ Cálculos realizados con un procesador Core 2 Duo T7200 a 2,13GHz. Sólo se tiene en cuenta el tiempo necesario para calcular el valor del índice % MAC y del ángulo TRIM, no del proceso para asignar valores a todas las variables.

⁷ <http://www.popsci.com/scitech/article/2006-04/johnn-koza-has-built-invention-machine>

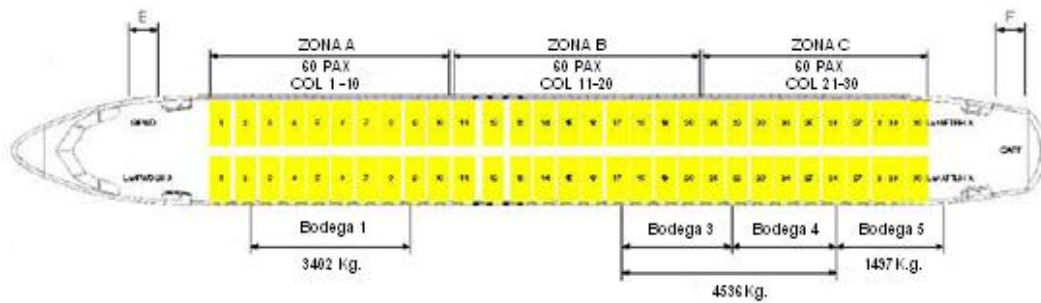


Fig.4.4 Esquema de un Airbus A320 [15]

Como puede verse en la **figura 4.4** el avión cuenta con 4 bodegas diferentes. En las bodegas 1, 3 y 4 la mercancía se introduce dentro de contenedores AKH que se distribuyen como se muestra en la figura siguiente (**Fig.4.5**). En la bodega cinco la mercancía viaja sin contenedor pero a efectos del código se puede considerar como un contenedor con un peso y volumen diferentes.

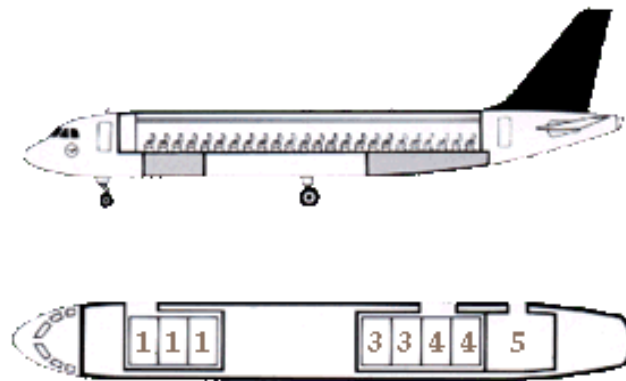


Fig.4.5 Detalle de las bodegas de un Airbus A320 y de los contenedores AKH

Además de estos 8 contenedores, definiremos un último contenedor sin límites de capacidad que representará la opción de dejar en tierra la carga. Como podemos ver se trata de un problema con una base diferente a los anteriores. No se trata de una función continua como la de los casos anteriores. Además, el número de variables es considerablemente más grande puesto que es igual al número de bultos.

Con todo esto vemos que es necesario utilizar un sistema de codificación específico y unos métodos de cruce adaptados al tipo de problema. Así pues, podemos hacernos una idea de la gran cantidad de modificaciones que serían necesarias para adaptar el código anteriormente utilizado a las nuevas necesidades. Es por esto que optaremos por implementar el código de cero, siguiendo el esquema del algoritmo canónico (apartado 2.2.1), y rescatando las pequeñas partes de código comunes en ambos problemas.

4.3.1. Codificación utilizada y métodos de selección, cruce y mutación

En lo que se refiere a la **codificación** utilizada implementaremos dos sistemas diferentes. Por un lado la primera opción, ya comentada en el apartado 4.2, será la de representar cada solución como un vector de enteros donde cada posición representará el número de contenedor al que será asignado (**Fig.4.6 - Izquierda**). La segunda opción presentará también cada solución como un vector de enteros, pero en esta ocasión las n primeras posiciones pertenecerán al contenedor 1, las n siguientes al contenedor 2 y así sucesivamente (**Fig.4.6 - Derecha**).

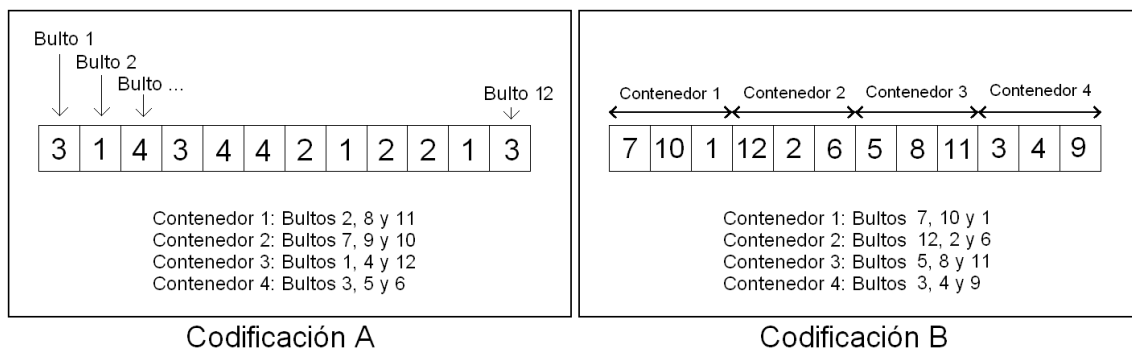


Fig.4.6 Ejemplo de la codificación A y de la codificación B

Además ambas codificaciones se realizarán sólo con números reales en base a los resultados obtenidos en las pruebas anteriores que mostraban una diferencia de tiempos notable en contra de la codificación binaria.

Referente a los **métodos de selección** utilizados, implementaremos los dos más típicos expuestos en el apartado 2.2.5. (*Selección por ruleta* y la *selección por torneo*) y observaremos los resultados que cada uno ofrece.

Como **métodos de cruce** utilizaremos el *cruce en un punto* (apartado 2.2.6) con algunas variaciones. En primer lugar ambos sistemas de codificación implementarán la opción de cruce en uno o dos puntos de manera que podremos ver cuál de ellos ofrece mejores resultados. En segundo lugar, al utilizar la codificación B el método de cruce no será exactamente igual al de *cruce en un punto* puesto que se ha de comprobar que cada bulto aparezca una sola vez evitando duplicados y desapariciones. A continuación veremos un ejemplo del *método de cruce* usando la codificación B descrito paso a paso para entender mejor cuáles son esas variaciones:

- Primero definimos un punto al azar del cromosoma y combinamos el material genético de ambos padres de forma normal (**Fig.4.7**). Esta parte es común en ambas codificaciones y de tratarse del modelo A, el proceso ya habría concluido. En el modelo B por el contrario cada posición del cromosoma representa el contenedor al que pertenece, mientras que el contenido de esas posiciones ha de ser un listado completo de todos los bultos del sistema. Si observamos con

detenimiento la **figura 4.7** veremos que es muy sencillo que se produzcan duplicados, y que cada duplicado lleva asociado la pérdida de otro paquete.

- Para corregir este problema el siguiente paso consiste en recorrer estos dos nuevos individuos en busca de repeticiones. De no encontrarse ninguna el proceso habría concluido, pero lo normal es que existan multitud de duplicados.
- De ser así, debemos tener en cuenta que los paquetes desaparecidos en uno de los hijos serán los que aparezcan duplicados en el otro. Para solucionar esto, lo que haremos será sustituir el primer bulto repetido del hijo 1 por el primer bulto repetido de hijo 2, y así sucesivamente (**Fig.4.8**). De esta forma resolvemos el problema a la vez que intentamos preservar al máximo la estructura heredada de sus progenitores.

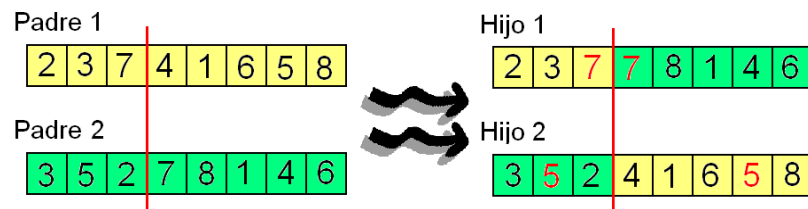


Fig.4.7 Ejemplo de cruce utilizando la codificación B con repeticiones

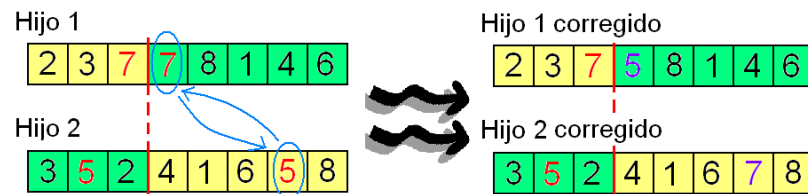


Fig.4.8 Ejemplo de cruce utilizando la codificación B una vez se ha resuelto el problema de los duplicados

Como podemos imaginar estos procesos secundarios de búsqueda y corrección dependerán directamente del número de bultos de sistema, y aumentarán significativamente el tiempo de proceso necesario. Será pues un factor importante a tener en cuenta y que puede decantar la balanza hacia una u otra codificación.

Por último, en cuanto al **método de mutación** utilizaremos la variante conocida como *permutación* que consiste en intercambiar de posición dos elementos del mismo individuo.

4.3.2. Sistema de *Fitness* y penalización

En lo que se refiere a la manera de evaluar cómo de buenas son las soluciones es necesario escoger entre dos posibilidades. Dando un vistazo a los métodos de cruce que se van a implementar es fácil observar un problema latente. Es muy posible, por no decir seguro, que aparecerán soluciones que violarán las condiciones del problema en lo referente al espacio físico disponible en los contenedores.

Estas soluciones posibles, aunque inaceptables, requieren un tratamiento especial. Una posibilidad sería “repararlas” para que cumplan con las limitaciones físicas de peso y volumen impuestas. Por otro lado, se podría pensar que estos individuos, aunque inviables, puedan ser los precursores de las soluciones óptimas que buscamos. En este caso quizás sería más efectivo conservar estas soluciones y dejarlas evolucionar para explorar todas las posibilidades.

En nuestro caso optaremos por esta última opción: estableceremos un sistema doble de *fitness* y penalización. De esta forma las soluciones inviables podrán seguir existiendo penalizándolas en función de cuánto incumplan las limitaciones **(ec.4.5)** y **(ec.4.6)**. Dicha penalización quedará definida en función del ángulo de TRIM (en valor absoluto) y de la cantidad de mercancía que se queda fuera del vuelo de manera innecesaria. Es decir, de cuánta mercancía que físicamente podría volar, se queda en tierra. Para ser justos en nuestra evaluación, existirán dos ecuaciones similares para medir esta penalización dependiendo del escenario en el que nos encontremos, y una vez elegida será utilizada la misma para todos los individuos del problema.

$$Penalización = |°TRIM| \times \frac{PESO\ NO\ CARGADO}{PESO\ OVERLOAD} \times \frac{VOLUMEN\ NO\ CARGADO}{VOLUMEN\ OVERLOAD} \quad (ec. 4.5)$$

$$Penalización = |°TRIM| \times \frac{PESO\ NO\ CARGADO}{PESO\ TOTAL} \times \frac{VOLUMEN\ NO\ CARGADO}{VOLUMEN\ TOTAL} \quad (ec. 4.6)$$

La primera de ellas **(ec.4.5)** será utilizada en el caso de que el conjunto de bultos generados supere los valores máximos de capacidad del avión (asignando a este exceso de volumen o peso el nombre de *VOLUMEN OVERLOAD* y *PESO OVERLOAD* respectivamente). De esta forma, penalizaremos a los individuos en función de la cantidad de carga que se deja en tierra teniendo en cuenta que ya existe una cierta cantidad que excedía la capacidad del aparato y que físicamente no podría volar.

En el caso de no existir este *OVERLOAD*, utilizaremos la segunda ecuación **(ec.4.6)** donde el factor de mercancía no cargada se calcula con respecto al total de la carga. De esta manera los individuos penalizan en función de cuanta mercancía dejan en tierra teniendo en cuenta que físicamente toda podía ser cargada.

En lo referente a la función de *fitness* evaluaremos todas las **soluciones válidas** en función del valor absoluto del ángulo de TRIM conseguido, así como de la cantidad total de mercancía que se deja en tierra mediante la **ecuación 4.7**. Este valor, normalizado entre 0 y 1, será mejor cuanto más próximo a cero sea el ángulo de TRIM. Asimismo también tendrá en cuenta la cantidad de carga introducida en el aparato, llegando a ser 1 cuando toda la carga esté dentro y se encuentre perfectamente equilibrado.

$$Fitness = \frac{\left(1 - \frac{\left(\frac{Volumen\ en\ tierra}{Volumen\ total} + \frac{Peso\ en\ tierra}{Peso\ total}\right)}{2}\right) * (2,5 - |^{\circ}TRIM|)}{2,5} \quad (ec. 4.7)$$

Por el contrario, para los individuos que incumplan alguno de los parámetros del problema se utilizará también la misma fórmula (**ec.4.7**), pero el ángulo de TRIM adoptará por defecto el valor máximo posible (2,5).

A partir de este sistema doble definido a imagen y semejanza del que se utilizaba en la implementación del capítulo 3, escogeremos a los individuos en función de su penalización en primer lugar (entendiéndose como mejor a los individuos que tengan una penalización más próxima a cero). Y sólo en caso de igualdad de penalización, la elección se hará en función de su valor de *fitness* (entendiéndose como mejor aquel individuo que ofrezca un valor de *fitness* más próximo a uno).

4.4. Primeros resultados

Una vez implementado el código (disponible en el anexo D, apartado D.2) procedemos a realizar las primeras pruebas del sistema. Para estas pruebas iniciales utilizaremos unos parámetros acordes con lo expuesto en los apartados anteriores sobre el tamaño de la población, el número de generaciones, la probabilidad de mutación y la probabilidad de cruce. Concretamente, los parámetros que definirán este escenario serán los siguientes:

Número de individuos: 100

Número de generaciones: 100

Número de bultos: 90

Probabilidad de cruce: 0.9

Probabilidad de mutación: 0.1

Número de ciclos: 50 (número de veces que se repetirá cada prueba)

Overload: 5458 Kg y 10.30 m³

A la vista de los resultados obtenidos en la **tabla 4.1**, que representa el valor medio calculado a partir de los mejores individuos de cada uno de los 50 ciclos, queda claro que un sistema en el que todas las soluciones se conserven (incluso las que no son válidas) no evoluciona como sería deseado.

	CODIFICACIÓN A	CODIFICACIÓN B
 TRIM 	2,5	2,5
Fitness	0	0
Ocupación (%)	99,20	99,66
Tiempo (s)	0,10	0,49
Soluciones válidas (%)	20	52

Tabla 4.1 Resultados obtenidos sin reparar las soluciones (problema final)

Como podemos ver, independientemente del tipo de codificación escogida, el AG no ofrece resultados aceptables. El número de soluciones inviables es elevadísimo (un 80% en la codificación A y un 48% en la B) y el ángulo medio que ofrecen estos mejores individuos es el peor posible. Es evidente que el AG se pierde en evoluciones sin sentido a partir de estos individuos defectuosos, que lejos de acercarse a soluciones cada vez mejores, nos alejan generación tras generación de las soluciones deseadas.

Es imprescindible pues implementar algún tipo de sistema que repare estos individuos corruptos y los convierta en soluciones aceptables. Para esto definiremos una función que revise a cada individuo contenedor por contenedor, marcando los que son correctos y los que no. En caso de que un individuo tenga contenedores “incorrectos” iremos extrayendo bultos al azar y reubicándolos en otros contenedores de forma aleatoria, pero controlando de no violar ningún parámetro, hasta que todos los contenedores sean correctos. En el primer caso intentaremos reubicarlos dentro de los contenedores “reales” (refiriéndonos a los 7 AKH’s y a la bodega 5) y en caso de no ser posible, destinaremos estos bultos elegidos al azar al contenedor virtual que representa la mercancía que se queda en tierra.

Una vez dispuesto el sistema repetimos las pruebas con los mismos parámetros y evaluamos los resultados (**Tabla 4.2**).

	CODIFICACIÓN A	CODIFICACIÓN B
 TRIM 	0,000034	0,002484
Fitness	0,40	0,48
Ocupación (%)	57,39	68,86
Tiempo (s)	0,11	0,50
Soluciones válidas (%)	100	100

Tabla 4.2 Resultados reparando las soluciones (problema final)

Como podemos ver en la tabla superior ahora disponemos de un 100% de soluciones viables en ambas codificaciones. Los resultados obtenidos en cuanto a TRIM se refiere son casi perfectos. El tiempo de proceso ha subido únicamente en una centésima de segundo al añadirle el proceso de reparación de soluciones. Y es interesante observar como la diferencia de tiempo de proceso se mantiene y sigue siendo superior en la codificación B con respecto

a la codificación A. El causante teórico es el sistema encargado de evitar duplicados y desapariciones a la hora de cruzar los individuos. Más adelante, cuando todo el sistema funcione correctamente, observaremos si realmente aumenta con el número de bultos y si ese aumento puede ser un motivo para descartar esta codificación.

Volviendo a los resultados de la **tabla 4.2.**, en este caso parece que está todo en su sitio: un buen TRIM, 100% de soluciones viables, tiempo de proceso mínimo. ¿Entonces por qué el valor de *fitness* es tan bajo? La respuesta se encuentra en la ocupación del vuelo.

Recordando los parámetros de la prueba vemos que existía un *Overload* de 5458 Kg y 10.30 m³. Dicho de otro modo, el peso y volumen de la carga generada no sólo llenaría el vuelo, sino que más de 5000 kg deberían quedarse fuera aunque éste estuviera lleno al 100%.

Nuestra solución ideal sería tal que el ángulo de TRIM fuera 0 y que la ocupación del vuelo fuese del 100%. Además, tenemos carga de sobras para llenar el aparato, y en cambio el sistema nos ofrece como mejores individuos aquellos que cargan poco más de la mitad de las bodegas. Es evidente que una ocupación del 100% real es improbable puesto que implicaría no dejar ni el más mínimo hueco. Pero un 57% y un 68 % respectivamente parecen más propios de un defecto del sistema que de tratarse de la mejor solución posible.

La posible causa de esta tendencia a equilibrar perfectamente el aparato a costa de la carga puede estar en el sistema de *penalización-fitness*. Fue definido para permitir que todas las soluciones (viables e inviables) coexistieran, pero ahora ha perdido su razón de ser puesto que todas las soluciones son correctas. Sería más práctico utilizar un sistema donde sólo un parámetro puntuara a los individuos. Así pues, eliminaremos la *penalización* del sistema y redefiniremos el parámetro de *fitness* para hacerlo un poco más efectivo (**ec.4.8**) y (**ec.4.9**).

$$Fitness = \frac{\left(\left(\frac{Volumen_{Cargado}}{Volumen_{Máximo\ del\ vuelo}} \right) + \left(\frac{Peso_{Cargado}}{Peso_{Máximo\ del\ vuelo}} \right) \right) + (2,5 - |TRIM|)}{3,5} \quad (ec. 4. 8)$$

$$Fitness = \frac{\left(\left(\frac{Volumen_{Cargado}}{Volumen_{Total\ de\ la\ carga}} \right) + \left(\frac{Peso_{Cargado}}{Peso_{Total\ de\ la\ carga}} \right) \right) + (2,5 - |TRIM|)}{3,5} \quad (ec. 4. 9)$$

De nuevo tenemos dos ecuaciones similares destinadas a evaluar el *fitness* de todos los individuos dependiendo del escenario en que nos encontremos. Por

un lado, en caso de existir *Overload*, evaluaremos a los individuos en función de la carga introducida respecto al máximo que es capaz de transportar el avión. De ahí que aparezcan dos parámetros nuevos como son el $Volumen_{Máximo\ del\ vuelo}$ y el $Peso_{Máximo\ del\ vuelo}$ que como su propio nombre indica hacen referencia a la suma de las capacidades máximas de las bodegas del aparato (**ec.4.8**).

De igual forma, en caso de que el total de bultos generados no supere el peso y/o volumen máximos del avión utilizaremos la **ecuación 4.9**, que evalúa el *fitness* sobre el total de peso y volumen de la carga. De esta manera el valor máximo de *fitness* sigue siendo 1, tanto para individuos que ocupen todo el espacio de las bodegas cuando existe exceso de mercancía, como para los que carguen todos los bultos cuando el conjunto de estos no alcance la ocupación total.

Como podemos observar el “peso” de este *fitness* lo lleva el valor del ángulo TRIM. La idea es conseguir una población rica en individuos con unos valores de TRIM óptimos e ir depurándola seleccionando aquellos que lo consigan con una mayor cantidad de material en sus bodegas. La mejor manera de ver si los cambios resultan beneficiosos es ponerlos a prueba. Así pues, repetiremos el experimento anterior con los mismos parámetros y dejemos que los resultados den su veredicto (**Tabla 4.3**).

	CODIFICACIÓN A	CODIFICACIÓN B
TRIM	0,013	0,015
Fitness	0,89	0,89
Ocupación (%)	78,36	78,54
Tiempo (s)	0,11	0,50
Soluciones válidas (%)	100	100

Tabla 4.3 Resultados del nuevo sistema sin penalización y con nuevo *fitness*

Como podemos ver los tiempos de proceso y el nivel de soluciones válidas es prácticamente el mismo. Y en lo referente a la ocupación y el *fitness*, hemos conseguido una notable mejora sin empeorar apenas el TRIM que sigue estando muy cercano a cero. Ahora alcanzamos una ocupación próxima al 80% de media. No cabe duda de que son unos buenos resultados, y en este punto podríamos decir que el sistema funciona, pero quizás se podría hacer algo más para mejorarlo.

Hasta este momento estamos suponiendo que los contenedores se colocan de forma secuencial dentro de las bodegas. Dicho de otro modo, hasta el momento estamos disponiendo los contenedores 1, 2 y 3 en la bodega 1, los contenedores 4 y 5 en la bodega 3, y los contenedores 6 y 7 en la bodega 4 sin excepción. Siguiendo este patrón, hacemos evolucionar las soluciones y, como hemos visto antes, alcanzamos resultados más que aceptables.

Pretender dar con la solución perfecta de esta forma es quizás pedir demasiado: no sólo pretendemos acertar la combinación ganadora, además queremos acertar el orden en el que saldrán los números. La idea sería aprovecharnos de la potencia de los AG para generar nuevos individuos cada vez mejores, y explotar al máximo cada uno de ellos probando todas las combinaciones diferentes de distribuir los contenedores en el interior del avión para ver cuál nos ofrece mejores resultados. De esta forma la idea de individuo evoluciona: pasando de estar formado únicamente por el cromosoma que define como están distribuidos los bultos, a añadir un campo nuevo que indique cuál es la combinación de contenedores que ofrece un mejor valor de *fitness*.

Con estos cambios que permiten probar cada solución con diferentes formas de disponer los contenedores aumentamos el campo de búsqueda y en teoría alcanzaremos mejores soluciones. Ahora bien, todo este proceso tendrá un coste computacional. Será cuestión de implementar el sistema y evaluar si los resultados compensan el teórico aumento de tiempo de proceso.

El primer paso para implementar este sistema consiste en determinar cuántas y cuáles son las combinaciones posibles. Para no abusar de la extensión de este TFC diremos que existen 210 combinaciones posibles diferentes, y dejaremos el cálculo completo para los anexos (anexo C, apartado C.4).

Una vez determinadas e introducidas en el sistema la función de *fitness* cambia de forma de actuar, y evalúa cada individuo válido de las 210 formas diferentes posibles, ofreciendo como *fitness* del individuo el valor mejor de los obtenidos. Implementado el nuevo sistema es hora de ponerlo a prueba, y ver si los resultados mejoran y si el tiempo de proceso añadido no es excesivo.

	CODIFICACIÓN A	CODIFICACIÓN B
[TRIM]	0,0077	0,0067
Fitness	0,91	0,91
Ocupación (%)	85,14	86,11
Tiempo (s)	0,12	0,50
Soluciones válidas (%)	100	100

Tabla 4.4. Resultados con el sistema de permutación de contenedores

Como podemos ver en la **tabla 4.4** el tiempo de proceso añadido al sistema es mínimo. Se trata de unas 2.100.000 operaciones por ciclo que suponen (recordando el tiempo calculado en el apartado 4.1.4) apenas 5 ms por ciclo. Mirando ahora los resultados vemos una significativa mejoría en todos los aspectos: el ángulo de TRIM se ha reducido aún más y la ocupación de las bodegas alcanza el 85% de media. Es importante recordar que todos estos resultados son valores medios (existen individuos por encima y por debajo de esta media) y que lo que nosotros buscamos es un único valor: el mejor de todos. De hecho, en la **tabla 4.5** podemos, ver el mejor individuo obtenido por codificación y comprobamos que se trata de resultados muy próximos al valor perfecto.

	TRIM	FITNESS	OCUPACIÓN (%)	TIEMPO (s)
Codificación A	0,0058	0,92	94,92	0,12
Codificación B	0,0055	0,91	94,11	0,50

Tabla 4.5. Mejores individuos por codificación con el sistema completo

4.5. Resultados finales

Una vez que hemos comprobado que el sistema funciona correctamente es hora de ponerlo realmente a prueba. Aumentaremos considerablemente el número de bultos (350 maletas), y evaluaremos los resultados del algoritmo y de sus diferentes métodos de selección y codificación.

Además del número de bultos, de su peso y de su volumen, existen otros parámetros, como por ejemplo el número de pasajeros y su disposición dentro del avión que tienen una importancia similar al de la carga misma a la hora de equilibrar el aparato. Para hacer el algoritmo lo más realista posible, estos y otros parámetros como el peso del carburante serán libres, y se generarán de forma aleatoria por el sistema.

Así pues, además de repetir cada simulación 25 veces por método y codificación para obtener unos valores medios, realizaremos las pruebas en 5 escenarios diferentes. Esto significa que cada vez existirá una colección de 350 bultos diferentes en peso y volumen a los anteriores. Así como un número y distribución de pasajeros y un volumen de carburante distinto cada vez. Con todo esto lo que se pretende es verificar que el AG es capaz de equilibrar el aparato en condiciones diferentes. Los parámetros que definen cada uno de los escenarios están recopilados en la **tabla 4.6** y los resultados obtenidos en las mismas, en las **tablas 4.7, 4.8 y 4.9**.

En concreto, cada fila de la **tabla 4.7** representa el valor medio de los mejores individuos de cada uno de los 25 ciclos en los que se ha evaluado esa combinación de métodos de cruce, selección y codificación.

Por lo que respecta a las **tablas 4.8 y 4.9**, cada fila representa al mejor individuo escogido entre los mejores de cada ciclo (25 por cada combinación de métodos de cruce, selección y codificación) en función de su *fitness* o su ocupación respectivamente.

	Pasajeros Zona A	Pasajeros Zona B	Pasajeros Zona C	Fuel (Kg.)	Overload Peso (Kg.)	Overload Vol. (m ³)
Prueba 1	44	41	43	16000	460	2,6
Prueba 2	54	35	19	16500	6	2,97
Prueba 3	22	5	5	13000	861	2,66
Prueba 4	50	25	22	17500	421	3,2
Prueba 5	27	36	25	18500	485	1,37

Tabla 4.6 Detalles de los diferentes escenarios de pruebas

Codificación y métodos	TRIM	<i>Fitness</i>	Ocupación (%)	Tiempo (s)
Torneo - 1 punto (A)	0,0020	0,96	87,06	0,35
Torneo - 2 puntos (A)	0,0016	0,96	87,06	0,35
Ruleta - 1 punto (A)	0,0130	0,92	87,33	0,35
Ruleta - 2 puntos (A)	0,0446	0,93	87,13	0,36
Torneo - 1 punto (B)	0,0020	0,94	88,56	5,59
Torneo - 2 puntos (B)	0,0027	0,94	88,38	5,60
Ruleta - 1 punto (B)	0,0079	0,91	89,49	5,60
Ruleta - 2 puntos (B)	0,0044	0,91	89,27	5,60

Tabla 4.7 Valores medios de las diferentes pruebas al sistema completo

Codificación y métodos	TRIM	<i>Fitness</i>	Ocupación (%)	Tiempo (s)
Torneo - 1 punto (A)	0,0003	0,96	92,20	0,35
Torneo - 2 puntos (A)	0,0016	0,96	90,73	0,35
Ruleta - 1 punto (A)	0,0681	0,94	87,18	0,36
Ruleta - 2 puntos (A)	0,0557	0,94	86,54	0,36
Torneo - 1 punto (B)	0,0261	0,95	89,70	5,57
Torneo - 2 puntos (B)	0,0253	0,95	87,45	5,61
Ruleta - 1 punto (B)	0,1189	0,92	88,77	5,62
Ruleta - 2 puntos (B)	0,1287	0,93	89,07	5,60

Tabla 4.8 Individuos con mejor *fitness* obtenidos con el sistema completo

Codificación y métodos	TRIM	<i>Fitness</i>	Ocupación (%)	Tiempo (s)
Torneo - 1 punto (A)	0,0174	0,96	93,24	0,35
Torneo - 2 puntos (A)	0,0015	0,96	93,53	0,35
Ruleta - 1 punto (A)	0,1439	0,92	91,41	0,35
Ruleta - 2 puntos (A)	0,1368	0,92	90,97	0,36
Torneo - 1 punto (B)	0,0679	0,94	91,26	5,57
Torneo - 2 puntos (B)	0,0867	0,93	91,65	5,59
Ruleta - 1 punto (B)	0,1762	0,91	92,37	5,60
Ruleta - 2 puntos (B)	0,1723	0,91	92,34	5,61

Tabla 4.9 Individuos con mejor ocupación obtenidos con el sistema completo

Observando los resultados de las tablas superiores lo primero que destaca es la diferencia de tiempos entre la codificación A y la B. En el primer caso el tiempo ha aumentado ligeramente (un factor inferior a 3) al cuadruplicar el número de bultos. Por el contrario, el tiempo de proceso de la codificación B se ha multiplicado por un factor 11 aproximadamente.

Como ya se apuntaba en este mismo capítulo la codificación B necesita más operaciones para asegurar que la lista de bultos no se vea alterada (duplicados o desapariciones) al cruzar a los individuos. Este aumento de tiempos es proporcional al número de bultos pero como podemos ver en el gráfico de la

figura 4.9. En el caso de la codificación A el aumento es de tipo lineal, mientras que para la codificación B el aumento es de tipo exponencial.

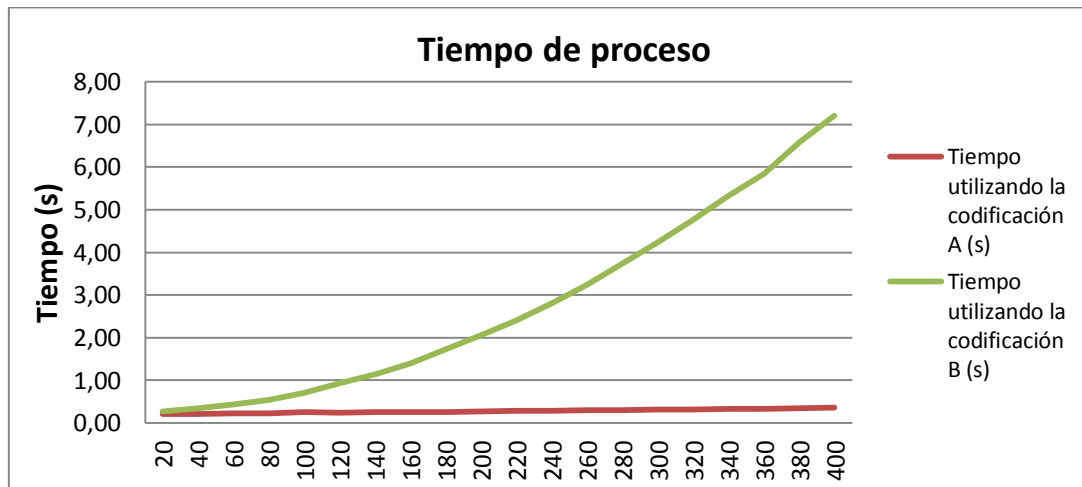


Fig.4.9 Tiempo de proceso en función del número de bultos

No hay que olvidar que estas pruebas se están realizando con un avión “pequeño”. En un gran carguero el número de bultos sería considerablemente más grande y por tanto el tiempo de proceso se dispararía. Además de esto, comparando los resultados obtenidos en una y otra codificación vemos que el modelo A obtiene siempre mejores individuos (aunque sea ligeramente) que el modelo B. Con todo esto podemos concluir que es adecuado descartar la codificación B y utilizar únicamente la A que ofrece unos resultados y tiempos de proceso mucho mejores.

Por lo que respecta a los métodos utilizados: viendo las **tablas 4.7, 4.8 y 4.9** observamos que en general el método de selección por torneo ofrece mejores resultados que el método de selección por ruleta.

En lo referente al método de cruce la respuesta es un poco más ajustada. No es fácil decidir si la mejor elección es el cruce en un punto o en dos puntos. Si nos centramos únicamente en los valores medios obtenidos, tenemos que dar por vencedor al método de cruce en dos puntos. Aunque a la vista de los valores tan ajustados que presentan ambos métodos, utilizar el otro método tampoco comprometería la eficacia del algoritmo.

Así pues, después de realizar todas estas pruebas podemos concluir que el sistema funciona, quedando demostrada la eficacia de los AG's como métodos de optimización en problemas con espacios de búsqueda un tanto particulares.

Y en segundo lugar, que la mejor combinación (de las que tenemos disponibles) sería utilizar la codificación A junto con el sistema de selección por torneo. En lo referente al método de cruce la decisión no está demasiado clara: sería aconsejable escoger el cruce en dos puntos, aunque en este último caso la elección sería quizás irrelevante.

CAPÍTULO 5: GESTIÓN DEL PROYECTO

5.1. Diagrama de Gantt

En principio, la duración inicial de este proyecto era de unos dos meses y medio a tiempo completo (**Fig.5.1**). Nada más lejos de la realidad puesto que al final los objetivos propuestos se han cumplido pero el tiempo necesario se ha alargado considerablemente.

		Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		Planificación general del proyecto	5 días	lun 04/05/09	vie 08/05/09	
2		 Búsqueda bibliográfica	25 días	lun 11/05/09	vie 12/06/09	1
3		Actividades de bibliografía sobre optimización y AG	15 días	lun 11/05/09	vie 29/05/09	
4		Redactar capítulo 0 - Introducción (Preliminar)	5 días	lun 25/05/09	vie 29/05/09	
5		Análisis sobre el funcionamiento teórico de los AG (preliminar)	10 días	lun 01/06/09	vie 12/06/09	4;1
6		Redacción del capítulo 1- Algoritmos genéticos (Preliminar)	5 días	lun 01/06/09	vie 05/06/09	4;1
7		 Análisis de las implantaciones de los AG	35 días	lun 15/06/09	vie 31/07/09	2;6;5
8		Análisis de las implantaciones de los AG	35 días	lun 15/06/09	vie 31/07/09	
9		Redactar capítulo 2 - Implementaciones (Preliminar)	30 días	lun 22/06/09	vie 31/07/09	
10		 Aplicación demostrativa	26 días	lun 03/08/09	lun 07/09/09	7;8;9
11		Definir aplicación practica par la última parte del proyecto	2 días	lun 03/08/09	mar 04/08/09	7;9
12		Modelar aplicación	4 días	mié 05/08/09	lun 10/08/09	11
13		 Implementación	10 días	mar 11/08/09	lun 24/08/09	12
14		Implementación de los diferentes bloques	10 días	mar 11/08/09	lun 24/08/09	12
15		 Integración de las partes	10 días	mar 25/08/09	lun 07/09/09	13
16		Tareas de integración	5 días	mar 25/08/09	lun 31/08/09	14
17		Pruebas	2 días	mar 01/09/09	mié 02/09/09	16
18		Análisis de resultados	3 días	jue 03/09/09	lun 07/09/09	17
19		Redacción de conclusiones (Preliminar)	10 días	mar 25/08/09	lun 07/09/09	
20		 Memoria y presentación	15 días	mar 08/09/09	lun 28/09/09	18;19;15;10
21		Corrección de los documentos preliminares	5 días	mar 08/09/09	lun 14/09/09	
22		Conclusiones	5 días	mar 15/09/09	lun 21/09/09	21
23		Preparar presentación	5 días	mar 22/09/09	lun 28/09/09	22
24		Semana de imprevistos	5 días	mar 29/09/09	lun 05/10/09	20
25		PRESENTACIÓN	1 día	mar 06/10/09	mar 06/10/09	24

Fig.5.1 Planificación inicial del proyecto

Uno de los primeros problemas surgió durante la búsqueda bibliográfica debido a la enorme cantidad de bibliografía y temas relacionados que iban apareciendo sobre métodos de optimización global y sobre los propios algoritmos genéticos. Esto junto con la redacción de la introducción y el primer capítulo de la memoria agotó no sólo la extensión permitida para el TFC, sino también todo el tiempo que nos habíamos propuesto.

Pasados esos tres meses y dispuestos a empezar con el núcleo de este proyecto surgió el verdadero problema: las horas de dedicación se vieron drásticamente reducidas hasta que el proyecto quedó parado durante 8 meses por causas que nada tienen que ver con el proyecto.

Transcurrido ese tiempo el proyecto resurgió y hubo que replantear y reorganizar el tiempo disponible intentando hacerlo de una forma más madura y realista en base a las horas disponibles (**Fig.5.2**)






		Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		Reorganización del proyecto	5 días	lun 15/03/10	vie 19/03/10	
2		Redactar capítulo III: Codificación del AG	25 días	lun 22/03/10	vie 23/04/10	1
3		Pruebas simples para evaluar efectividad del algoritmo.	20 días	lun 22/03/10	vie 16/04/10	1
4		Análisis de los resultados	10 días	lun 05/04/10	vie 16/04/10	
5		Conclusiones	5 días	lun 19/04/10	vie 23/04/10	4
6		Redactar capítulo IV: Aplicación practica	42 días	lun 26/04/10	mar 22/06/10	2
7		Planteamiento del problema	5 días	lun 26/04/10	vie 30/04/10	5
8		Modificación de del código	15 días	lun 03/05/10	vie 21/05/10	7
9		Pruebas	10 días	lun 24/05/10	vie 04/06/10	8
10		Análisis de los resultados	7 días	lun 07/06/10	mar 15/06/10	9
11		Conclusiones	5 días	mié 16/06/10	mar 22/06/10	10
12		Redactar Capítulo IV: Conclusiones	5 días	mié 23/06/10	mar 29/06/10	6
13		Revisar el contenido de los capítulos anteriores	5 días	mié 23/06/10	mar 29/06/10	
14		Preparar Presentación	5 días	mié 30/06/10	mar 06/07/10	13
15		PRESENTACION	1 día	mié 07/07/10	mié 07/07/10	14

Fig.5.2 Reorganización del proyecto

En este caso los tiempos asignados a cada tarea se fueron respetando en la medida de lo posible hasta llegar al capítulo cuarto (Aplicación práctica). En este capítulo estaba previsto servirnos del código utilizado en las pruebas básicas pero fue imposible. El problema era demasiado diferente y los cambios sólo daban errores tras errores en el proceso de compilación. Finalmente se decidió abandonar la idea de modificar el antiguo código e implementar uno desde cero. Esto implicó asumir más tiempo perdido durante los intentos de modificación, y que se añadiera una nueva tarea considerablemente grande que multiplico el tiempo previsto para esta parte del proyecto.

Para finalizar, el último gran problema fue la redacción de la memoria y la extensión que iba tomando a lo largo del tiempo. Intentar respetar el límite establecido llevó a reescribir gran parte del TFC intentando que el cuerpo del proyecto fuera independiente y no fuera necesario recurrir a los anexos si no era con intención de ampliar conocimientos.

Así pues, el desarrollo de este proyecto ha estado plagado de contratiempos: algunos como el sufrido en el último capítulo o como la planificación inicial tan inocente son propios de este tipo de trabajos. Otros en cambio no tienen nada que ver, aunque son los que provocan los retrasos más elevados.

5.2. Costes del proyecto

La duración de este proyecto ha sido de más de un año como se ha expuesto en el apartado anterior, pero la suma de horas que se le han podido dedicar ha sido aproximadamente de unas 500 horas.

Teniendo en cuenta que este proyecto ha sido realizado por un ingeniero técnico novel y que según el mercado el precio por hora asciende a unos 30€, podemos calcular el precio neto que asciende a unos 15.000€.

A esta cantidad será necesario añadir los costes del material utilizado que ascienden únicamente a 60€ en concepto de gastos por las copias del documento final (si no tenemos en cuenta el equipo con el que se han realizado las pruebas). Con todo esto y añadiendo el 18% de IVA, el coste final del proyecto asciende a 17.770€.

5.3. Ambientalización del proyecto

El proyecto en sí no está pensado como una aplicación práctica destinada a un uso. Es más bien una introducción teórica al mundo de los algoritmos genéticos y a sus posibilidades. Ahora bien, durante el transcurso del capítulo final hemos podido comprobar que si bien existen algunos programas creados para determinar el centro de gravedad de los aviones, no existe (o por lo menos no lo hemos encontrado) ningún software que indique cómo deben distribuirse las mercancías para alcanzar una posición óptima.

En la actualidad estas tareas se realizan de una forma manual por las terminales y bajo la supervisión de los pilotos que pueden decidir no volar en ciertas condiciones o con ciertas mercancías. Cualquier acto de optimización supone un ahorro de recursos y una mejora de la eficiencia. Así pues, con un sistema similar se podría mejorar la seguridad en los vuelos y tal vez reducir el consumo de carburante mejorando las condiciones de la velocidad de crucero y durante el despegue y aterrizaje.

CONCLUSIONES

Una vez concluidas todas las pruebas y habiendo superado con éxito el problema final podemos afirmar que los algoritmos genéticos realmente son métodos efectivos, y que cumplen con la premisa expuesta allá por el capítulo 1 de garantizar que alcanzarán el valor óptimo global, o por lo menos una buena solución en un tiempo competitivo.

Evidentemente no son la solución a cualquier problema. De hecho les hemos visto fracasar por ejemplo al intentar optimizar la función E (con un máximo aislado y el resto del espacio de búsqueda sin apenas información útil). No en vano esta función fue escogida para eso ya que representaba a la perfección uno de los talones de Aquiles de estas técnicas.

Como ya se ha dicho con anterioridad, no son métodos perfectos. Tienen debilidades como todos, pero sus puntos fuertes están encarados hacia problemas donde el resto de las técnicas digamos “normales” no tendrían éxito. Es por esto que estas técnicas heurísticas (entre las que se encuentran los algoritmos genéticos) resultan ser herramientas muy útiles, ya sea como métodos de optimización directa o como pioneros que exploren el terreno en busca de zonas interesantes para otros métodos más específicos.

Pero quizás eso no sea lo más importante que hemos podido comprobar en este TFC. Más allá de defectos y virtudes (ya de sobras conocidos y expuestos con anterioridad en la parte teórica) existe otro aspecto de igual importancia y que quizás haya pasado desapercibido: nos estamos refiriendo a la enorme flexibilidad que presentan estos métodos y que permite configurar el sistema a nuestras necesidades, pero que también implica que existan un montón de maneras diferentes de cómo no modelar un algoritmo genético.

Recordemos por ejemplo lo ocurrido en el capítulo 4 con el problema final: en primer lugar es necesario escoger el alfabeto y el método de codificación que vamos a utilizar. Además es necesario “calibrar” correctamente los diferentes parámetros como por ejemplo el número de generaciones, número de individuos, probabilidad de mutación y de cruce, etc. y es necesario definir un método que evalúe cómo de buenos son los individuos. En ese caso nosotros decidimos utilizar un sistema doble de penalización y *fitness* y definimos unos parámetros que fueron un fracaso. Hubo que hacer cambios, pruebas y tomar decisiones que tanto podían haber sido correctas como acabar empeorando aún más las cosas. Finalmente conseguimos el objetivo y modelamos un sistema capaz de distribuir la carga tal y como queríamos en un principio, pero qué hubiera pasado si hubiéramos fracasado un par de veces más. Quizás habríamos concluido que los algoritmos genéticos no son los más adecuados para este tipo de problemas.

Resumiendo, los algoritmos genéticos han demostrado su potencia pero también su complejidad, dejando claro que para que tengan éxito hay que dedicarles mucho más tiempo de planificación y preparación inicial que a otros métodos más tradicionales.

BIBLIOGRAFÍA

- [1] Jesús Ramón Pérez López. *Contribución a los métodos de optimización basados en procesos naturales y su aplicación a la medida de antenas en campo próximo*. PhD thesis, Universidad de Cantabria, Departamento de Ingeniería de comunicaciones, 2005.
- [2] Thomas Weise. Global optimization algorithms - theory and application. PDF, Mayo 2009.
- [3] Byron S. Gottfried; Joel Weisman. *Introduction To Optimization Theory*. Prentice-Hall, 1978.
- [4] Urmila M. Diwekar. *Introduction To Applied Optimization*. Kluwer Academic Publishers, 2003.
- [5] De Luigi F. Maniezzo V, Gambardella L.M. *New Optimization Techniques in Engineering*. Onwubolu, G. C., and B. V. Babu, 2004.
- [6] Marcos Gestal Pose. Introducción a los algoritmos genéticos. PDF, 2001.
- [7] Enrique J. Carmona Suárez. Algoritmos evolutivos para minería de datos. Apuntes de UNED.
- [8] Michael J. Walsh Lawrence J. Fogel, Alvin J. Owens. *Artificial intelligence through simulated evolution*. John Willey & Sons, New York, 1967.
- [9] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press., 1992.
- [10] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog Verlag, Stuttgart, Germany., 1970.
- [11] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
- [12] Moujahid, A.; Inza, I.; Larrañaga P. Tema 2: Algoritmos genéticos. PDF, Febrero 2008.
- [13] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems*. University of Michigan, Ann Arbor, MI, USA, 1975.
- [14] J.T. Alander. On optimal population size of genetic algorithms. In *Computer Systems and Software Engineering, 6th Annual European Computer Conference*, 1992.
- [15] CMD Urs Oetiker. *A320 Line Training Summary*. Air Berlin, Revision 4.1.

- [16] A.A. Hopgood. (B - Book) *Intelligent Systems for Engineers and Scientists, 2nd edition*. CRC Press, 2 edition, 2001. ISBN 0-8493-0456-3 (First edition published in 1993 as Knowledge-Based Systems for Engineers and Scientists, ISBN 0-8493-8616-0.).
- [17] Stephanie Forrest. Genetic algorithms: Principles of natural selection applied to computation. *Science*, Vol. 261:872–878, Aug. 13, 1993.
- [18] Kalyanmoy Deb. An introduction to genetic algorithms. PDF.
- [19] Nathyhelem Gil Londoño. Algoritmos genéticos. PDF, Noviembre 2006.
- [20] Eduardo. Morales. Búsqueda, optimización y aprendizaje. PDF, Noviembre 2004.
- [21] Gilbert Syswerda. *Simulated Crossover in Genetic Algorithms*. 1992.
- [22] Nicolas Durand Jean-Marc Alliot. Algorithmes génétiques. Pdf, Marzo 2005.
- [23] J. David Schaffer, Richard A. Caruana, Larry J. Eshelman, and Rajarshi Das. *A study of control parameters affecting online performance of genetic algorithms for function optimization*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.
- [24] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (2nd, extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1994.
- [25] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
- [26] Eckart Zitzler. Evolutionary algorithms for multiobjective optimization: Methods and applications. Pdf, Noviembre 1999.
- [27] Yaseer Arafat Durrani. *Macromodelado del consumo de sistemas digitales basados en IPs descritos a nivel de transferencia de registros*. PhD thesis, Universidad Politécnica de Madrid, Escuela Técnica Superior de Ingenieros Industriales., 2008.
- [28] A. Mierzejewska A. A. Hopgood. Transform ranking: a new method of fitness scaling in genetic algorithms. .
- [29] Ángel Murias Rodríguez. *Estudio de bloques constructivos en algoritmos genéticos*. PhD thesis, Universidad Politécnica de Cataluña, 2007.
- [30] Joachim Stender. *Parallel Genetic Algorithms: Theory and Applications*. IOS Press, 2003.

- [31] Darrell Whitley. A genetic algorithm tutorial. PDF.
- [32] Michael Allen Barry Wilkinson. *Parallel programming*. Prentice-Hall, 1999.
- [33] Martina Gorges-Schleuter. *Explicit Parallelism of Genetic Algorithms through Population Structures*. Springer-Verlag, London, UK, 1991.
- [34] P. Moscato. *On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms*. Technical Report Caltech Concurrent Computation Program., 1989.
- [35] Stephen Frederick Smith. *A learning system based on genetic adaptive algorithms*. University of Pittsburgh, Pittsburgh, PA, USA, 1980.
- [36] Alma Martínez. Definición de una red neuronal para clasificación por medio de un programa evolutivo. *Revista Mexicana de Ingeniería Biomédica*, XXII: pp. 4 – 11, 2001.
- [37] Airbus. *Getting to Grips with Aircraft Weight and Balance*, Airbus, 2008, PDF.
- [38] Airbus. *Flight Crew Operating Manual*, Airbus, 2007, PDF.

ANEXO A: ANÁLISIS EN PROFUNDIDAD DE LOS DIFERENTES ASPECTOS QUE DEFINEN UN ALGORITMO GENÉTICO

A.1. Aspectos básicos de un algoritmo genético

A.1.1. Codificación

En primer lugar, y quizás uno de los factores más determinantes a la hora de que el AG funcione correctamente, está en determinar la codificación más adecuada.

Como ya se ha dicho con anterioridad los individuos que componen la población de un AG son representaciones de soluciones posibles al problema, pero, en general, no son los valores de dichas soluciones. La explicación más sencilla a esto es que raramente se trate de un problema de optimización con una única variable o restricción. Los AG's son aplicables tanto en problemas con una única restricción como con múltiples restricciones en cuyo caso muchas de las soluciones generadas serán posibles, y quizás serían soluciones óptimas para algunas de las variables, pero no serán válidas porque violarán algunas de las restricciones.

Representar esto mediante un único valor sería del todo imposible ya que provocaría que se perdiera gran cantidad de información. De manera que en general, cada uno de los individuos estará definido por un conjunto de valores representativos de cada una de las restricciones o variables del problema a los que denominaremos genes. Así pues, a imagen y semejanza de los seres vivos, cada uno de los individuos estará compuesto por un conjunto de genes que darán lugar a los cromosomas que definirán las características del individuo.

Durante los primeros años de vida de los AG's, el alfabeto escogido para representar estos genes era el binario ya que se adapta muy bien al tipo de operaciones que se han de realizar dentro de los AG's. A pesar de esto, en ocasiones provoca problemas (sobre todo a la hora de introducir las mutaciones, ya que una pequeña variación de un solo bit provoca una gran variación del individuo en sí). Esto puede desplazarte muchísimo del espacio de búsqueda de forma que pasas de acercarte a una solución óptima a estar explorando una sección totalmente nueva.

En sí mismo este efecto no tiene por qué ser siempre malo ya que puede ayudar a escapar de óptimos locales, pero es algo que debe darse en contadas ocasiones. Para evitar, en la medida de lo posible, estos efectos se suele utilizar el código Gray⁸ [16, pp.185] para representar los genes en binario [4]. Esta codificación tiene la particularidad de que su distancia de *Hamming*⁹ entre dos números consecutivos es 1 (**Fig.A.1**), o dicho de otra forma: que para

⁸ http://es.wikipedia.org/wiki/Codigo_Gray

⁹ http://es.wikipedia.org/wiki/Distancia_de_Hamming

pasar de la secuencia de bits que representan un valor dado, a la que representa el valor siguiente sólo es necesario modificar un bit.

Decimal	Gray	Binario
0	000	000
1	001	001
2	011	010
3	010	011
4	110	100
5	111	101
6	101	110
7	100	111

Fig.A.1 Representación decimal, binaria y en codificación Gray de los 8 primeros dígitos.

A parte de esto, también es posible representar los genes mediante números enteros, reales (*real-encoded*) [2], [17], cadenas de letras o cualquier otro tipo de estructura. De esta forma, la longitud l de las cadenas que conforman los cromosomas de un individuo dependerá del alfabeto escogido y del número de genes que lo componen.

Es importante destacar que el número de genes puede ser fijo o variable; es decir, puede tratarse de un problema en el que el número de variables a tener en cuenta no cambia. O por el contrario, puede tratarse de un problema en el que el sistema va *aprendiendo* por sí mismo de manera similar a lo que ocurre con las redes neuronales, en cuyo caso el número de estados del sistema va variando.

A modo de resumen y al mismo tiempo para definir más claramente algunos conceptos importantes utilizaremos un ejemplo simple de optimización [18]. Supongamos que queremos optimizar el coste de fabricar un envase en función de la cantidad de aluminio que se necesita para construirlo. Así pues, los *genes* de estos individuos serán dos: la altura del envase y el diámetro del mismo. Los cromosomas de estos individuos estarían formados por estos genes codificados en binario (**Fig.A.2**).

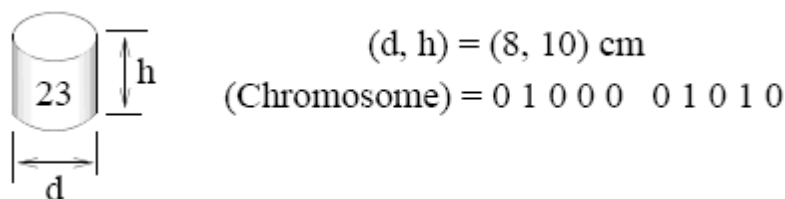


Fig.A.2 Representación en forma real y binaria de los cromosomas de un problema de optimización [18]

A la posición concreta de cada gen dentro del cromosoma se le llama *locus* y a su valor concreto se le denomina *alelo*. El conjunto de estos dos cromosomas sería lo que denominaríamos *genotipo*, a partir de los cuales se podría calcular fácilmente el coste de construir el envase ya que mediante sus genes (altura y radio) se puede determinar sin problemas la superficie del mismo. Este coste del envase sería lo que denominaríamos *fenotipo*, y vendría a ser una representación “real” de la combinación de genes que describen al individuo (**Fig.A.3**).

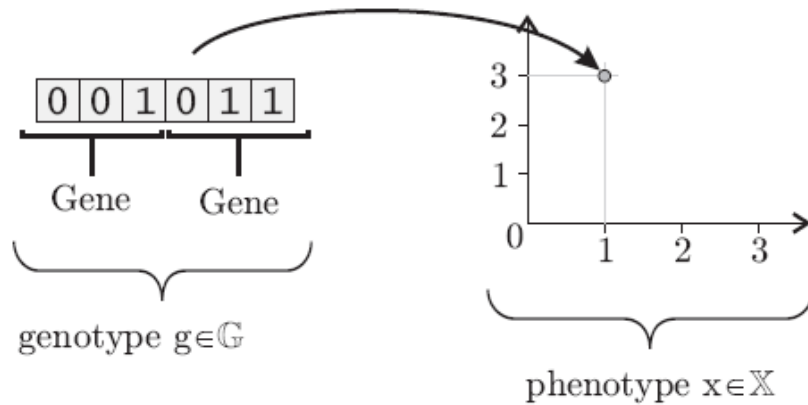


Fig.A.3 Ejemplo de la relación directa entre genotipo y fenotipo [2]

Durante el proceso propio del AG veremos cómo la población evoluciona y van apareciendo todo tipo de individuos diferentes como se observa en la figura de abajo (**Fig.A.4**) hasta que, si el AG se ha planteado correctamente, se alcance el valor óptimo deseado.

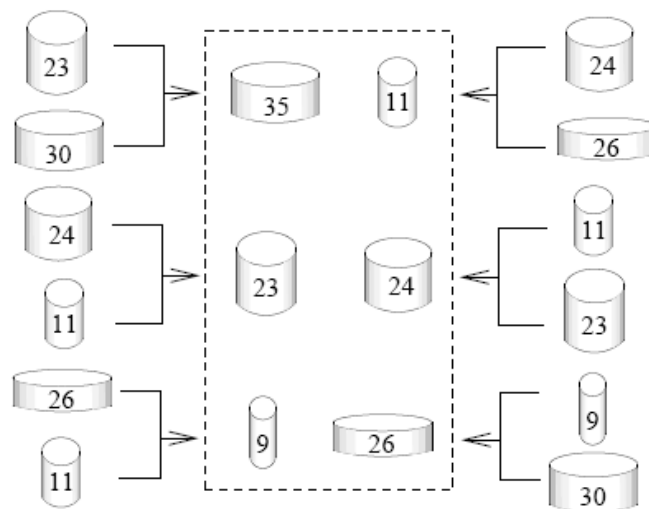


Fig.A.4 Ejemplos de los diferentes individuos que podrían aparecer [18]

A.1.2. Población inicial

Partiendo del esquema del algoritmo canónico (ver apartado 2.2.1) y una vez decidida la codificación más adecuada, el primer paso a realizar es la generación de la población inicial. Este proceso sencillo a simple vista, tiene toda una serie de implicaciones importantes a tener en cuenta.

El primer problema que se ha de plantear a la hora de generar la población inicial es el tamaño de la misma. Parece evidente que una población demasiado pequeña no podrá ser capaz de cubrir de forma correcta el espacio de búsqueda. Por otro lado, una población demasiado grande aumentaría el coste computacional del AG haciéndolo poco competitivo frente a otros métodos de optimización.

De Jong [13] aconseja utilizar un número prudente ($n = 50$). Por otro lado, estudios empíricos publicados por [14] sugieren que, con una población codificada en forma binaria y de longitud L , es suficiente con tomar una población de entre L y $2L$ individuos para atacar la mayoría de los problemas propuestos de forma satisfactoria.

Por último, una vez decidido el alfabeto, la codificación de los genes y el tamaño de la población inicial sólo cabe preguntarse el método correcto para escoger esos primeros elementos. Generalmente se escogen de forma aleatoria con probabilidad uniforme, aunque también sería posible escogerlos como el resultado de alguna técnica previa de optimización local. ¿Cuál es el método más efectivo? En el caso de escoger individuos de forma no aleatoria se comprueba que el AG converge de forma más rápida. Pero en ocasiones esta *convergencia prematura* no es beneficiosa porque se realiza sobre óptimos locales [19]

A.1.3. Función de adaptación o *Fitness function*

La función de adaptación o *fitness function* es la encargada de valorar a los individuos que forman la población e indicarnos cómo de buena es la solución que nos ofrecen. Así nosotros podemos escoger a los más aptos para que se reproduzcan y den lugar a las siguientes generaciones. Mientras tanto, los individuos menos *adaptados* encontrarán mayores dificultades para que su material genético se propague en las sucesivas generaciones hasta que al final acaben desapareciendo.

No es necesario recalcar la importancia que tiene esta función de *fitness*. Es junto con la codificación escogida los dos factores más decisivos para que un AG funcione correctamente. Lo ideal sería definir funciones de *fitness* que reflejen el valor del individuo de una marea “real” y que verifiquen que para dos individuos que estén próximos en el espacio de búsqueda, el valor de la función de adaptación de éstos esté próximo entre sí también [12].

Basándonos en lo descrito por Koza, J. R. en [9] se pueden definir cuatro tipos de valores de adaptación o *fitness*:

A.1.3.1. Fitness puro [r(i,t)]

Consiste en asignar un valor real que refleje cómo de bueno es el individuo. Esto se consigue comparando el valor que ofrece el individuo con el deseado donde, el valor deseado, es el mejor obtenido hasta la fecha. Así pues, en problemas de maximización el individuo mejor adaptado será el que ofrezca los valores más altos. Mientras tanto, en problemas de minimización, el mejor adaptado ofrecerá los valores más bajos.

A.1.3.2. Fitness estandarizado [s(i,t)]

Parte de la base del *fitness puro* pero añade ciertas variaciones para que, tanto en problemas de maximización como de minimización, el individuo mejor adaptado sea el que ofrezca un valor más próximo a cero. Para esto modifica el *fitness puro* mediante la ecuación siguiente (**ec.A.1**)

$$s(i, t) = \begin{cases} r(i, t) & \text{minimización} \\ r_{\max} - r(i, t) & \text{maximización} \end{cases} \quad (\text{ec. A. 1})$$

En el caso de problemas de minimización utilizaremos el valor del *fitness puro*. Para problemas de maximización utilizaremos el valor del *fitness puro* restado de una cota máxima r_{\max} que estará representada por el mejor valor obtenido hasta la fecha. Así, en cualquiera de los dos casos, el individuo mejor adaptado será el que ofrezca valores más próximos a cero.

A.1.3.3. Fitness ajustado [a(i,t)]

De nuevo se trata de una mejora del modelo anterior. Mediante la ecuación siguiente (**ec.A.2**) se consigue que los valores obtenidos a partir del *fitness estandarizado* tomen valores entre 0 y uno.

$$a(i, t) = \frac{1}{1 + s(i, t)} \quad (\text{ec. A. 2})$$

En este caso, el mejor grado de adaptación se conseguirá con valores próximos a 1.

A.1.3.4. Fitness normalizado [n(i,t)]

En los tipos de *fitness* antes presentados sólo se tenía en cuenta el valor de adaptación del individuo en cuestión. Al igual que el *fitness estandarizado*, el *fitness normalizado* utiliza un rango de valores entre 0 y 1, pero en este caso lo hace en relación al resto de las soluciones que ofrece la población (**ec.A.3**)

$$n(i, t) = \frac{a(i, t)}{\sum_{k=1}^N a(k, t)} \quad (\text{ec. A.3})$$

De esta forma, un valor de *fitness* próximo a 1 no sólo indica que se trata de una buena solución. Además indica que es mucho mejor que las otras soluciones ofrecidas por los demás individuos.

Ni que decir tiene que en la mayoría de los problemas, donde existen multitud de restricciones, definir cómo de bueno es un individuo no resulta tarea fácil. Para empezar una gran parte de los puntos del espacio de búsqueda hacen referencia a posibles soluciones que violan alguno de estos criterios y por tanto no son válidas. En estos casos existen diferentes opciones para tratar de obtener individuos válidos y poder continuar con el proceso como se muestra en [19].

- Una primera opción que podríamos definir como más *absolutista*, consistiría en asignarles un valor de función de adaptación igual a cero.
- Otra opción sería ignorar que se trata de individuos no válidos y continuar con el proceso de cruce y mutación hasta obtener algún individuo válido.
- Otra posibilidad sería la de penalizar a los individuos que violen alguna de las restricciones con algún factor que estaría definido en función del número de restricciones violadas o al coste de reconstruir dicho individuo de manera que no viole ninguna restricción.
- Por último, en caso de que la función de adaptación sea muy compleja, se podría optar por una *evaluación aproximada* de dicha función. Es decir, se trataría de definir n funciones aproximadas de la función objetivo original, siempre y cuando la evaluación de una función aproximada (más simple que la función original) resulte como mínimo n veces más rápida y por tanto resulte rentable computacionalmente hablando.

Así mismo existen otros problemas intrínsecos de los AG's, que trataremos en mayor profundidad en el apartado A.3 y que se pueden solventar en parte mediante la redefinición de la función de *fitness*. Es el caso de los problemas derivados de la velocidad de convergencia. Generalmente suele denominarse *convergencia prematura* y surge generalmente porque existen lo que podríamos definir como *súper individuos* (óptimos locales muy superiores al resto de sus valores vecinos), es decir, individuos cuya adaptación al medio es muy superior al resto y que acaban dominando la población puesto que siempre son escogidos para los procesos de cruce.

En el caso de la convergencia prematura se pueden reducir los valores de aptitud que se otorgan a un individuo cuando en la población se observen ciertas cantidades de individuos parecidos (*sharing*). En estos casos la solución no radica únicamente en modificar la función de adaptación. Es interesante

también replantearse utilizar un método de selección que no sea meramente elitista como por ejemplo la *selección por ruleta* o algún otro de los que veremos a continuación [20]

A.1.4. Función de selección

La función de selección será la encargada de elegir a los individuos que serán utilizados para la reproducción y cuáles no. En principio la idea más intuitiva sería elegir siempre a los más aptos (es decir, a los que ofrezcan unos valores de función de adaptación más altos) de entre todos los de la población. Pero eso provocaría que en pocas generaciones la población se volviera homogénea; es decir, que en pocas generaciones los genes de los individuos más aptos acabarían predominando en casi todos los individuos.

Así pues, no se debe eliminar del todo la posibilidad de que los individuos menos aptos puedan reproducirse puesto que esto reduciría la diversidad de la población lo que se traduciría en espacios de búsqueda sin explorar. La opción más común es escoger previamente a los individuos participantes en el cruce mediante algún método de los que veremos a continuación y después, de entre los individuos menos aptos, realizar una selección aleatoria de algunos progenitores más [6].

A continuación podemos encontrar un listado de los métodos más comunes mediante los que un AG puede escoger a los progenitores de su siguiente generación. Algunos de estos métodos son excluyentes entre sí, pero en general los AG's no utilizan un solo método sino combinaciones de varios a la vez.

A.1.4.1. Selección puramente elitista.

En este caso la idea es muy simple, se trata de evaluar a los individuos y escoger como progenitores a los más aptos, es decir, los que ofrezcan unos valores más altos. Es una idea muy simple e intuitivamente parece la más correcta, pero como hemos visto antes genera problemas de convergencia debido a una reducción rápida de la diversidad de la población.

A.1.4.2. Selección por ruleta.

La *selección por ruleta* o *método Montecarlo* [6] o *muestreo universal estocástico* [20] es quizás el método más utilizado junto con la *selección por torneo* que veremos más adelante. Su funcionamiento es simple: consiste en simular una ruleta donde cada individuo x tiene asignado una sección de tamaño $P(x)$ que representa la probabilidad que tiene ese elemento de ser escogido. Esta $P(x)$ se calcula en función de su nivel de adaptación; es decir, del valor que nos devuelve la función de *fitness*, y que se normaliza entre 0 y 1 mediante la expresión **A.4.**

$$P(x) = \frac{\text{Adaptación}(x)}{\sum_{j=1}^N \text{Adaptación}(x_j)} \quad (\text{ec. A. 4})$$

Una vez construida la ruleta (**Fig.A.5**) se trata de obtener una serie de valores entre cero y uno que nos indicarán los elementos seleccionados como progenitores de la nueva generación. La posición inicial para la primera elección es el cero, pero las siguientes posiciones se obtienen a partir de sumar la posición actual con el valor aleatorio obtenido en el módulo 1. De esta forma se intenta evitar que se escoja dos veces seguidas al mismo individuo en la medida de lo posible.

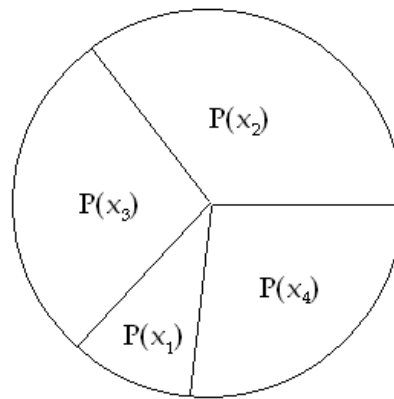


Fig.A.5 Ruleta con las probabilidades de los cuatro elementos que componen la población

Así pues, este método sigue una pauta elitista en tanto que, cuanto mejor adaptado esté un individuo, mayor será la sección de ruleta que se le asignara, y por tanto mayor probabilidad de ser escogido. Aun así no es un método puramente elitista. Ningún individuo tiene una probabilidad cero de ser escogido, de forma que la diversidad de la población está garantizada y depende únicamente del azar. No es un método perfecto, puesto que se pueden escoger sólo los individuos de peor calidad. Además de que su complejidad computacional crece a medida que aumenta la población de manera que el AG puede acabar volviéndose ineficiente.

A.1.4.3. Selección por rango.

Este método mejora la efectividad contra los problemas de *convergencia prematura* con respecto al método anterior. En este caso, la probabilidad de elegir a un individuo no se asigna en base a la función de *fitness*, sino al rango que ocupa el valor de dicha función. Como se explica en [12] el primer paso para utilizar este método de selección consiste en ordenar de menor a mayor

los individuos que componen la población. Evidentemente, dicha ordenación se realiza a partir de la del valor de la función de *fitness*.

Así pues, el individuo menos adaptado tendría asignado rango 1 mientras que el más adaptado de los n individuos tendría asignado rango n . A partir de esto, la probabilidad de que un individuo i de la generación t , sea escogido como padre ($P_{i,t}$) se calcula mediante la expresión siguiente **(ec.A.5)**.

$$P_{i,t} = \frac{\text{rango}(f_{\text{fitness}}(i, t))}{n(n+1)/2} \quad (\text{ec. A. 5})$$

Donde la suma de los rangos $n(n+1)/2$ se comporta como una constante de normalización que, como podemos ver en la **figura A.6**, da como resultado una distribución mucho más uniforme de las probabilidades de selección en comparación con el método anterior y con otros diferentes [12].

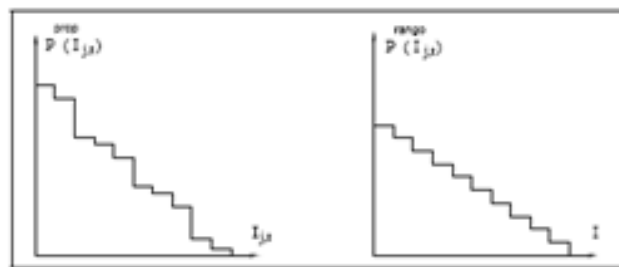


Fig.A.6 Representación de la distribución de probabilidad que ofrece la selección por ruleta (izquierda) y la selección por rango (derecha) [12].

Aquí todos los individuos tienen las mismas probabilidades de ser elegidos lo que hace realmente complicado que un individuo colapse la población de forma rápida y permite una mayor efectividad del AG. En contrapartida, el AG sufre una ralentización en su proceso de cálculo.

A.1.4.4. Selección por torneo.

La idea de este método es bastante simple: se trata de escoger al azar un número determinado de individuos (en general 2) y hacerlos competir entre sí para decidir cuál es el que se convertirá en uno de los progenitores de la nueva generación. En este punto el camino se bifurca.

Por un lado existe una versión más determinista en cuyo caso el individuo seleccionado es el que posea un nivel de adaptación mayor [20]. Es decir, el que ofrezca unos valores más altos de la función de adaptación.

Opuestamente, existe una versión más estocástica donde la elección del progenitor se realiza al azar [19] eligiendo un número entre 0 y 1. En el caso de que el valor sea más alto que un cierto umbral k definido previamente, el progenitor será el individuo con mejor adaptación. En caso contrario el progenitor será el menos adaptado, diversificando así mucho mejor los valores de la población.

En todos los casos lo más destacable de este método es su sencillez que se traduce en coste computacional muy bajo. Y el hecho de que permite suavizar los efectos propios de la presión selectiva, ya sea mediante la utilización de un criterio más bien estocástico, o de un método más determinista. En ambos casos, tomando pocos individuos para realizar el torneo, se aumentan las opciones globales de los individuos menos adaptados, mientras que en torneos con muchos participantes sus opciones son más reducidas [6].

A.1.4.5. Selección del valor esperado.

Otra posible solución a los problemas derivados de súper individuos es la que ofrece el método de *selección del valor esperado* [12]. Este sistema asigna un contador a cada individuo cuyo valor inicial es directamente proporcional a su función de *fitness* e inversamente proporcional a la media de dicha función en el instante en que aparece dicho individuo (**ec.A.6**).

$$Contador = \frac{f_{fitness}(x)}{\bar{f}_{fitness}} \quad (ec. A. 6)$$

Este contador decrecerá en un valor comprendido entre 0,5 y 1 cada vez que el individuo sea seleccionado para ser progenitor. De esta forma el número de veces que un mismo individuo puede ser seleccionado está limitado y será descartado cuando el contador pase a ser negativo.

Como hemos podido comprobar existen muchos y muy variados métodos de selección. En esta lista no están ni mucho menos todos, pero sí los más comunes o significativos. Además del hecho de que estos u otros métodos pueden utilizarse combinados para mejorar su efectividad siempre y cuando eso no suponga un coste computacional demasiado elevado.

La elección de uno u otro método de selección definirán la estrategia de búsqueda del AG: Utilizando métodos más elitistas la búsqueda se centrará en los entornos próximos a las mejores soluciones. Utilizando métodos donde la presión de la selección sea menor se está dejando abiertas las puertas a la exploración de zonas desconocidas que pueden reportar mejores soluciones o simplemente tiempo de cálculo perdido.

A.1.5. Función de cruce

Una vez seleccionados los individuos que harán las funciones de padres llega el momento de la reproducción. A imagen y semejanza de lo que ocurre en los seres vivos la reproducción de los individuos de una población puede darse de un modo *sexual* o *asexual*.

En el modo *sexual* se produce un intercambio de material genético entre los progenitores y es precisamente la *función de cruce* la encargada de realizar dicha tarea. A diferencia del anterior, en el modo *asexual*, los hijos son meras copias de los padres de forma que no hay evolución más allá de lo puedan provocar las mutaciones inesperadas. El proceso a seguir para conseguir la nueva generación de individuos es flexible y se puede realizar de diferentes maneras.

Por lo general la reproducción *asexual* no es un método recomendable por sí mismo. Aun así, como se explica en [6], tampoco ha de ser descartada de forma tajante. Se puede considerar la posibilidad de permitir un pequeño margen de individuos reproducidos de manera *asexual* y cuyas únicas diferencias con sus padres sean las provocadas por la *función de mutación*.

La reproducción *sexual*, de la que se encarga la *función de cruce*, es la base del funcionamiento de los AG's. Mediante los diferentes métodos que veremos a continuación la población se diversifica y evoluciona en todas direcciones de manera que obtendremos tanto individuos mejor dotados como menos adaptados que sus progenitores. Esto se traduce en una nueva lista de posibles soluciones y de nuevos vectores de búsqueda que tanto pueden llevar al éxito como al fracaso, así que es importante tener claro que la función de cruce sólo hace evolucionar a la población sin tener en cuenta si esto nos acerca o nos aleja de la solución deseada.

A.1.5.1. Operador de cruce basado en un punto (SPX).

El operador de cruce basado en un punto o SPX (*Single Point Crossover*) es quizás el proceso más simple y el más utilizado en los AG's. Como puede verse en la **figura A.7**, una vez escogidos los padres se elige al azar un punto y se divide el cromosoma en dos secciones. A continuación se intercambian las secciones entre uno y otro progenitor dando lugar a los descendientes [12].

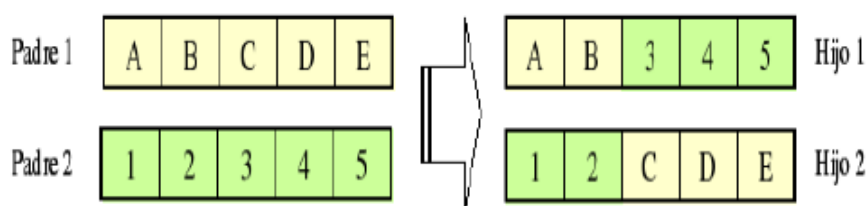


Fig.A.7 Ejemplo del funcionamiento del operador de cruce en un punto. [6]

A.1.5.2. Operador de cruce basado en n puntos

Partiendo del operador anterior era cuestión de tiempo que los investigadores se aventuraran a probar de dividir el cromosoma en n secciones para intentar que el intercambio fuera más profundo. Las conclusiones obtenidas por De Jong [13] son que aumentar el número de puntos de corte a más de dos no beneficia al AG.

El problema principal de utilizar más de dos puntos de corte consiste en que al generar multitud de pequeños segmentos, resulta mucho más fácil perder las buenas características de los progenitores, a favor de una evolución mucho más caótica que, como aspecto positivo, permite una exploración más a fondo del espacio de búsqueda [6].

El *operador de cruce basado en dos puntos* o DPX (Double Point Crossover) tendría un funcionamiento muy similar al SPX. De hecho son dos casos particulares de la misma idea. En el caso del DPX (**Fig.A.8**) se definirán al azar dos puntos de corte dentro del cromosoma. La sección comprendida entre ambos puntos de corte será la que se intercambiarán los progenitores.

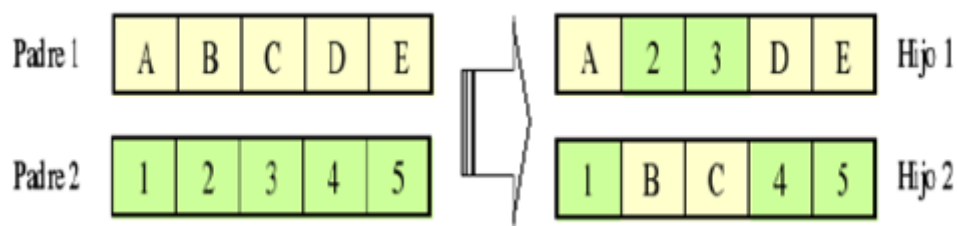


Fig.A.8 Ejemplo del funcionamiento del operador de cruce en dos puntos [6]

A.1.5.3. Operador de cruce uniforme (UPX)

El *operador de cruce uniforme* o UPX (*Uniform Point Crossover*) utiliza un sistema diferente a los anteriormente explicados. La idea, presentada por primera vez en 1992 por Gilbert Syswerda [21], consiste en definir de forma aleatoria una máscara, de tipo binario con ceros y unos, que nos indique de qué progenitor debemos tomar el elemento del cromosoma que ocupa dicha posición.

Así pues, para una posición determinada, si en la máscara hay un uno, el bit (o el elemento del alfabeto utilizado) que debemos tomar procederá del primer progenitor. En caso de que encontremos un cero, el bit (o el elemento del alfabeto utilizado) procederá del segundo progenitor. En la **figura A.9** podemos ver un ejemplo muy claro de su funcionamiento.

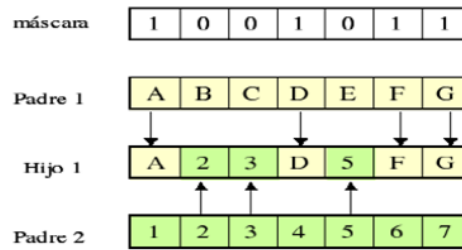


Fig.A.9 Funcionamiento del operador de cruce uniforme [6]

Como podemos ver en la figura superior, obtener el cromosoma del primer hijo es una cuestión muy sencilla y ara conseguir el segundo descendiente basta con intercambiar los padres o invertir la máscara de cruce. Queda por tratar el tema de cómo obtener de forma aleatoria los valores de la máscara de cruce: en este caso la máscara de cruce estaría compuesta por una muestra aleatoria extraída de una distribución de Bernouilli de parámetro $\frac{1}{2}$ [6].

A.1.5.4. Operador de cruce basado en la función de *fitness*

El *operador de cruce basado en la función de adaptación* sería un caso particular del *operador de cruce uniforme* (UPX) donde se tendría en cuenta el valor de la función de *fitness* para generar la máscara de cruce [12]. El funcionamiento es exactamente el mismo en cuanto a la generación de los descendientes. La diferencia radica en que la máscara se generara a partir de una distribución de Bernouilli de parámetro p que se obtiene a partir de la expresión siguiente (**ec.A.7**).

$$p = \frac{f_{fitness}(i, t)}{f_{fitness}(i, t) + f_{fitness}(j, t)} \quad (\text{ec. A. 7})$$

Donde $f_{fitness}(i, t)$ y $f_{fitness}(j, t)$ serían los valores de la función de *fitness* de los individuos i y j respectivamente en la generación t . De esta forma los genes de los individuos mejor dotados tienen mayores posibilidades de propagarse a las futuras generaciones, y aun así, ningún gen tiene probabilidad cero, de manera que la diversidad de la población se mantiene.

A.1.5.5. Operador de cruce inspirado en el *Simulated Annealing*

En este caso el operador de cruce no recurre a puntos de corte ni a máscaras como en casos anteriores. Se trata cada posición del cromosoma por separado y se define una probabilidad de que dicho elemento provenga de uno u otro progenitor siguiendo una analogía del proceso utilizado en el endurecimiento de los metales. Se define un umbral de energía θ_c y una temperatura T que irá descendiendo con el paso del tiempo mediante un proceso de enfriamiento.

Una vez hecho esto el funcionamiento es simple: se define una probabilidad p (**ec.A.8**)

$$P = e^{-(\theta c/T)} \quad (\text{ec. A. 8})$$

Este parámetro P será la probabilidad de que el padre del que proviene el elemento $i+1$ del cromosoma sea el opuesto al progenitor del elemento i . Cuando la temperatura T es alta el sistema se comporta como el *operador de cruce uniforme* de manera que la diversidad en la población es elevada. Por el contrario al enfriarse, los descendientes resultantes acabarán siendo muy parecidos a alguno de sus padres [12].

A.1.5.6. Operador de cruce baricéntrico

Hasta ahora, los operadores vistos estaban más indicados para problemas discretos. El *operador de cruce baricéntrico* [22] a diferencia de los anteriores es más eficaz en problemas continuos. El funcionamiento es sencillo: una vez seleccionados los individuos progenitores sus genes se combinan linealmente mediante la expresión siguiente (**ec.A.9**).

$$\begin{cases} C_1(i) = \alpha P_1(i) + (1 - \alpha)P_2(i) \\ C_2(i) = (1 - \alpha)P_1(i) + \alpha P_2(i) \end{cases} \quad (\text{ec. A. 9})$$

Donde el *coeficiente de ponderación* α nos permite tener cierto control sobre el resultado del cruce adoptando valores que van desde -0,5 hasta 1.5. De esta forma podemos favorecer que los genes de los descendientes estén más próximos a uno u otro progenitor en función del valor de *fitness* de sus padres, de la diversidad poblacional, del grado de saturación del sistema, etc. En la figura siguiente (**Fig.A.10**) podemos ver a grandes rasgos los efectos del *coeficiente de ponderación* sobre los descendientes.

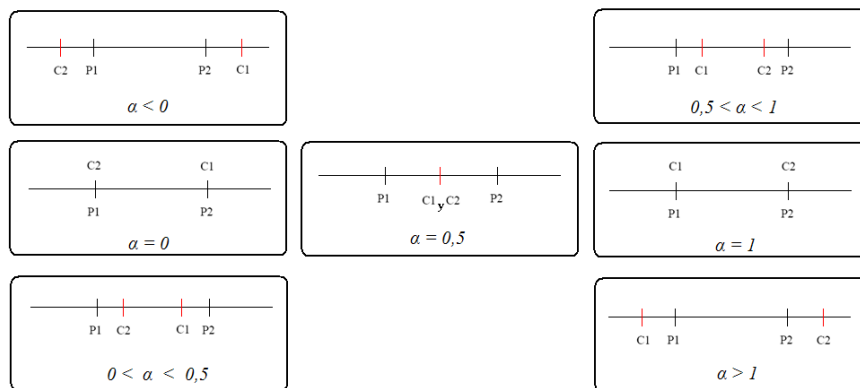


Fig.A.10 Efectos del coeficiente α en el operador de cruce baricéntrico

Como podemos ver, en función del valor de α escogido los descendientes se parecerán más a uno u otro progenitor según nos interese en cada momento o incluso serán copias exactas de sus padres como ocurre para $\alpha = 0$ y $\alpha = 1$. De esta forma el cruce baricéntrico se revela como un método muy flexible para controlar problemas de diversidad poblacional.

Como ha podido comprobarse, el sistema para realizar los cruces de los individuos es flexible y permite multitud de opciones. En este apartado se han visto representados sólo algunos de los métodos más genéricos. Además, cada tipo de problema puede dar lugar a métodos específicos para realizar el proceso de cruce siempre basándose en unos principios muy concretos: por un lado, mantener la diversidad de la población, en la medida de lo posible, para explorar la mayor cantidad del espacio de búsqueda. Y por otro preservar los valores “buenos” que van apareciendo en la población y que son el camino a seguir para alcanzar las soluciones óptimas.

A.1.6. Función de mutación

La función de mutación es otro de los pilares básicos del funcionamiento de los AG's. Una vez más, la intención es imitar lo que ocurre en el mundo real. Se trata de insertar una ínfima posibilidad (menos del 1%) de que un gen (o un elemento del alfabeto del cromosoma) varíe de forma inesperada. De esta forma imitamos esos “errores” que se producen al copiar el material genético en el proceso reproductivo y que, aunque insignificantes a primera vista, tienen unas consecuencias impredecibles. No en vano estas mutaciones nos han permitido pasar de organismos unicelulares a lo que somos actualmente.

En los AG's ocurre lo mismo. Existen diferentes formas de realizar esta mutación: desde la más simple donde cada gen o cada elemento del cromosoma muta de forma independiente, hasta soluciones más complejas donde la estructura del problema y la relación entre los genes juegan un papel importante [19].

Si bien es cierto que es la función de cruce la que recorre el espacio de búsqueda, la función de mutación garantiza que ningún punto del espacio tenga una probabilidad cero de ser explorado. Un factor que va ganando importancia a medida que la población va convergiendo [6].

Más aun, investigadores como Schaffer [23] sugieren que los efectos de la mutación son incluso superiores a los del cruce. Para ello simulan dos procesos evolutivos sobre una misma población. En un primer caso mediante un proceso de selección y cruce únicamente. En el otro caso mediante un proceso de selección y mutación.

Más allá de todo esto, el problema principal de la función de mutación es el de encontrar su valor adecuado. Un valor demasiado bajo provocaría una reducción de la diversidad de la población pudiendo producirse una convergencia prematura sobre algún óptimo local. Un valor demasiado alto

añadiría demasiada diversidad a la población. Esto repercutiría positivamente en cuanto a la cantidad de espacio de búsqueda explorado, pero dificultaría la convergencia del AG.

Estudios teóricos publicados por De Jong en [13] sugieren que se utilice una probabilidad de mutación inversa a la longitud del cromosoma. Por otro lado, otros investigadores como Schaffer en [23] sugieren empíricamente que el valor óptimo de la probabilidad de mutación P_m se debe calcular mediante la expresión siguiente (**ec.A.10**)

$$P = \frac{1}{n^{0,9318} \cdot l^{0,5435}} \quad (\text{ec. A. 10})$$

Donde n es el número de individuos que tiene la población y l la longitud de sus cromosomas. Por último otros investigadores sugieren que no se utilice un parámetro fijo para P_m , sino que esta disminuya conforme aumenta el número de iteraciones ya que es en esos momentos donde su efecto resulta más crítico [12].

Existe también una variante de esta mutación denominada *permutación* [3, pp.148] que consiste en variar la posición de los genes dentro de un mismo cromosoma (**Fig.A.11**). Este método sólo es aplicable para problemas de optimización donde lo que se busca es una secuencia de elementos.

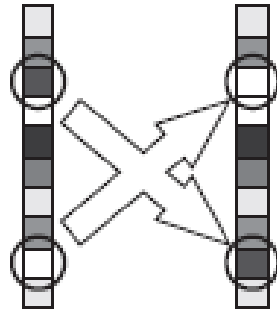


Fig.A.11 Permutación de genes dentro de un cromosoma [12]

A.1.7. Función de inserción

Una vez generados y evaluados los nuevos individuos, es decir, después de seleccionar a los progenitores, cruzarlos, aplicarles la función de mutación y evaluarlos mediante la función de *fitness*, nos encontramos con un nuevo problema: la población ha crecido por encima del máximo permitido. Sobran un determinado número de individuos dando lugar a un nuevo parámetro del AG llamado *tasa de reemplazamiento generacional* (t_{gr}), que representa el

porcentaje de hijos generados con respecto al tamaño de la población. Llega pues, el momento de “recortar” la población. La idea original de los primeros AG's era realizar este reemplazamiento de uno en uno ($t_{gr} = n^{-1}$). Actualmente se suele realizar en bloque ($t_{gr} = 1$)[12].

Esto no tiene por qué ser necesariamente así y se debe a que estamos trabajando con una única población de tamaño fijo. Más adelante veremos otras implementaciones donde el tamaño de la población puede ser variable o incluso cada nueva generación puede explorarse de forma independiente como si de otra población se tratase (Anexos - A.5).

En el caso genérico, el resultado del cruce y la mutación es una población demasiado grande. En este momento está formada por los individuos de la anterior generación más los nuevos hijos. Es necesario insertar estos nuevos individuos y para ello es imprescindible eliminar algunos otros mediante algún tipo de criterio.

Una posibilidad sería utilizar una *reducción simple* consistente en no conservar individuos entre generaciones. Así pues, los descendientes generados sustituyen a sus progenitores convirtiéndose en la nueva generación. Otra posibilidad sería utilizar un *criterio de reducción elitista*, donde se escogen los n individuos más adaptados para que formen la nueva población descartando al resto. Otra opción sería tomar un camino intermedio, es decir, si el tamaño de la población es n , tomar los m individuos más adaptados (donde m es siempre menor que n) de entre los descendientes y progenitores. Y posteriormente tomar los $(n-m)$ que faltan de entre los individuos que no fueron seleccionados para la reproducción.

Otros AG's como por ejemplo el *Algoritmo Genético Modificado* [24] o MOD_{GA} proponen un reemplazamiento generacional basado en la elección de m_1 individuos para hacer de progenitores, y m_2 individuos destinados a morir. El resto de los individuos de la población, es decir $(n - (m_1 + m_2))$, son considerados como neutros y pasan directamente a la nueva generación. En este MOD_{GA} el criterio para elegir a estos individuos se basa en la función de *fitness*. De esta manera los individuos mejor adaptados tienen mayor posibilidad de ser escogidos para formar parte de los progenitores, y los individuos menos dotados, tienen mayor probabilidad de ser escogidos para morir.

De nuevo existen multitud de implementaciones diferentes, cada una con aspectos positivos y negativos. En unos casos la utilización de criterios elitistas aumenta la velocidad de convergencia reduciendo la diversidad. El AG parece más eficiente pero puede quedar atrapado más fácilmente en óptimos locales. La otra cara de la moneda serían los criterios más probabilísticos que mejoraran la exploración del espacio de búsqueda a costa de un tiempo de convergencia mayor. Una vez más se trata de encontrar un equilibrio adecuado que dependerá en gran medida del tipo de problema que estemos tratando.

A.1.8. Condición de parada

La idea en la que se basan los AG's consiste en iterar los procesos anteriormente descritos sobre una población de individuos hasta que finalmente converjan en la solución óptima al problema. Así pues, el *criterio de parada* debería hacer referencia a la convergencia de la población. Pero ¿cómo definir esta convergencia?

Este término debería hacer referencia a una pérdida de diversidad en la población. Por definición un gen se dice que ha convergido cuando el 95% de la población comparte el mismo valor para dicho gen [12]. De esta forma el *criterio de parada* puede establecerse de manera que el proceso termine cuando todos los genes, o al menos un alto porcentaje de los mismos, hayan convergido.

Otra forma de evaluar esta misma condición es a partir del valor medio de la función de *fitness*. Partiendo de esta idea, se podría establecer un valor límite a partir del cual la solución es suficientemente buena como para ser aceptada, pero esto implicaría un conocimiento de los valores máximos que se pueden alcanzar en el problema (cosa que no siempre tendremos) [20].

Ahora bien, esta no debería ser la única variable a tener en cuenta. Si bien es cierto que la función principal del AG es alcanzar esta convergencia, puesto que eso significa que se ha alcanzado alguna solución óptima. El tiempo necesario para conseguirlo es igualmente importante. Los AG's han de ser competitivos. No se puede esperar indefinidamente a que los cruces y mutaciones nos lleven a la solución global. Lo más sencillo sería combinar el criterio de convergencia con un número máximo de iteraciones de forma que se limite el tiempo de proceso del AG.

A.2 Evaluación de los Algoritmos Genéticos

Una pregunta importante que nos puede asaltar es: ¿está funcionando bien nuestro AG? La respuesta más simple sería esperar a ver los resultados finales y comprobar si los valores son aceptables. Introducidos por De Jong [13], los métodos que se exponen a continuación permiten evaluar la progresión que está siguiendo el AG, de manera que se pueda valorar si está funcionando correctamente.

A.2.1. Evaluación On-Line

Utilizando el método *On-Line* a lo largo de las diferentes generaciones, podemos observar la evolución media de todos los individuos hasta la generación actual T (**ec.A.11**).

$$E_{On-Line}(T) = \frac{\sum_{t=1}^T \sum_{i=1}^N f_{fitness}(i, t)}{NT} \quad (\text{ec. A. 11})$$

Donde $f_{fitness}(i, t)$ es el valor de la función de *fitness* para un determinado individuo de la generación t , y N es el número total de individuos que componen la población

A.2.2. Evaluación Off-Line:

Sirviéndonos del método *off-line* (ec.A.12), observaremos cómo va evolucionando el valor óptimo alcanzado en cada generación a lo largo de T generaciones.

$$E_{Off-Line}(T) = \frac{\sum_{t=1}^T f_{\max fitness}(t)}{T} \quad (\text{ec. A. 12})$$

Donde $f_{\max fitness}(t)$ representa el valor más alto de *fitness* alcanzado por un individuo dentro de la generación t .

A.3. Problemas específicos

Como hemos explicado hasta ahora los AG's son métodos de optimización muy potentes pero no están exentos de problemas. En primer lugar hay que tener muy claro que no son la mejor solución para todos los problemas. Hay entornos en los que los AG's no trabajan demasiado bien (**Fig.A.12**). No hay que olvidar que son algoritmos *ciegos* que se guían únicamente por las tendencias evolutivas (gradientes) y por el azar.

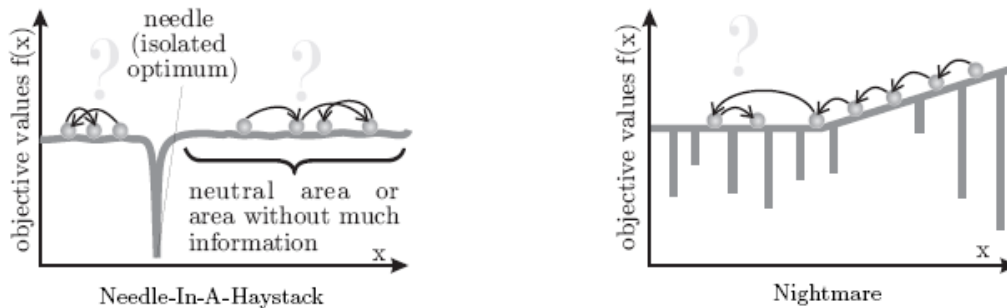


Fig.A.12 Ejemplo de espacios de búsqueda particulares donde los AG's no funcionan correctamente. [2]

Como podemos ver en la **figura A.12** hay entornos en los que resulta francamente difícil encontrar una progresión lógica que nos lleve hasta donde queremos ir. Dependemos básicamente de un golpe de suerte que nos sitúe en el punto correcto.

En otras ocasiones la orografía del espacio de búsqueda hace que los gradientes de búsqueda cambien continuamente (*ruggedness* [2], **Fig.A.13**). En estos casos la evolución que siguen los individuos es del todo inesperada. Tan pronto una pequeña variación en sus genes puede provocar grandes cambios en su fenotipo (*Strong Causality*) como todo lo contrario, y grandes cambios en sus cromosomas dejar a los individuos prácticamente igual (*Weak Causality*).

En estos casos no hay soluciones buenas por las que optar. Utilizar poblaciones grandes y mantener una diversidad alta es prácticamente la única posibilidad [2], pero el resto depende de dónde el azar nos quiera llevar.

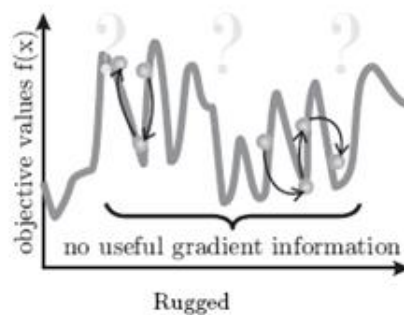


Fig.A.13 Función demasiado irregular (*Ruggedness*) [2]

En otras ocasiones el espacio de búsqueda no resulta tan “inhóspito”, pero eso no significa que no pueda dar problemas. Por ejemplo en la figura siguiente (**Fig.A.14**) la mayor parte del espacio de búsqueda no sugiere ninguna ruta a seguir. El número de generaciones máximas podría agotarse sin que se hubiera alcanzado ninguna zona de interés. En estos casos la utilidad de los AG’s queda en entredicho y resulta más efectivo utilizar métodos de búsqueda más exhaustivos.

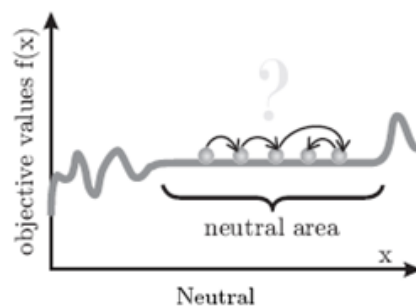


Fig.A.14 Espacio de búsqueda con una gran sección neutral. [2]

Más allá de todo esto, el problema por excelencia de los AG's es la pérdida de diversidad. Está presente en casi todos los contratiempos que se nos presentan al utilizar estos métodos. A veces se presenta como mera repercusión de otro problema más importante y en ocasiones es provocada por la implementación del propio AG. Sea como fuere, sus efectos suelen ser demoledores: reducción del espacio de búsqueda explorado, aparición de súper individuos que colapsan la población, imposibilidad de escapar de óptimos locales y finalmente *convergencia prematura* del algoritmo de búsqueda.

Para luchar contra este enemigo no es suficiente con una única estrategia. A lo largo de este capítulo hemos visto como la implementación de las funciones básicas del AG favorece en mayor o menor medida que se mantenga esta diversidad (métodos de *selección por torneo* (apartado A.1.4.4), *fitness normalizado* (apartado A.1.3.4), *función de cruce inspirada en el simulated annealing* (apartado A.1.5.5), etc.). Además de todas estas pequeñas contribuciones existen métodos creados específicamente para reducir los efectos de la pérdida de diversidad en la población. A continuación veremos algunos de los más importantes como son las *técnicas de nicho*, *sharing*, *scaling*, etc.

A.3.1. Sharing

La técnica de *sharing* o *fitness sharing* es una de las más conocidas y utilizadas para intentar mantener la diversidad de la población. Introducida por Goldberg [25], la función de *sharing* penaliza a los individuos en función de la densidad poblacional de la zona. Es decir, que cuanto más concentrados estén los individuos en una zona o *nicho*, mayor será la penalización que sufrirán.

De esta forma, individuos situados en zonas poco pobladas (que con los métodos tradicionales estarían condenados a desaparecer) no desaparecen completamente y los individuos de zonas superpobladas, no siguen reproduciéndose indiscriminadamente [26].

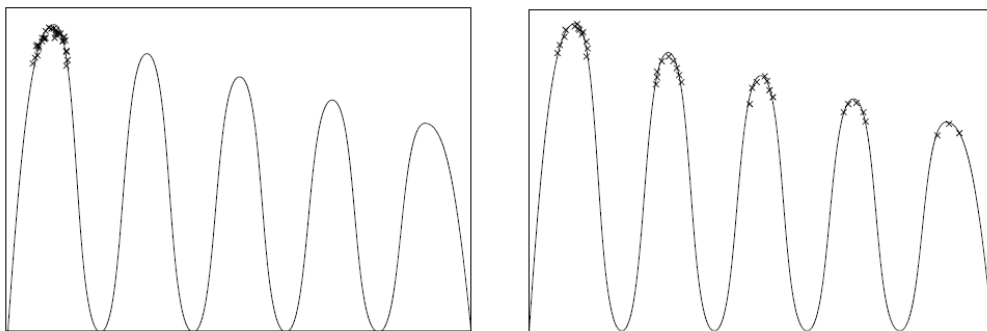


Fig.A.15 Efectos de utilizar *sharing*: a la izquierda distribución de la población en una función normal, a la derecha distribución de la población utilizando técnicas de *sharing* [22]

Como podemos ver en la **figura A.15**, la función de *sharing* permite una mejor exploración del espacio de búsqueda. Al evitar las concentraciones de población en zonas concretas. Existen múltiples variantes de la función de *sharing* [2] **(ec.A.13)** **(ec.A.14)** **(ec.A.15)** **(ec.A.16)** aunque todas se basan en los mismos principios básicos:

- En primer lugar hemos de definir *radio de nicho* σ .
- Tomamos dos individuos s_1 y s_2 pertenecientes al mismo *nicho* y calculamos la distancia d que los separa. La forma de calcular la distancia entre los individuos depende de la codificación utilizada. En el caso de estar trabajando con un alfabeto binario la distancia se medirá en términos de *distancia de Hamming*. En el caso de tratarse de vectores codificados con números reales la distancia será la distancia Euclídea [2]. Dicha distancia no sólo puede calcularse en base al espacio de búsqueda, también puede ser medida en el *espacio de soluciones*. Sea como fuere, una vez definida la distancia d se aplica la función de *sharing* elegida **(ec.A.13)** **(ec.A.14)** **(ec.A.15)** **(ec.A.16)**.

$$Sharing_{tri_{\sigma}}(d) = \begin{cases} 1 - \frac{d}{\sigma} & \text{si } 0 \leq d < \sigma \\ 0 & \text{en el resto de los casos} \end{cases} \quad (\text{ec. A. 13})$$

$$Sharing_{cvex_{\sigma,p}}(d) = \begin{cases} \left(1 - \frac{d}{\sigma}\right)^p & \text{si } 0 \leq d < \sigma \\ 0 & \text{en el resto de los casos} \end{cases} \quad (\text{ec. A. 14})$$

$$Sharing_{ccav_{\sigma,p}}(d) = \begin{cases} 1 - \left(\frac{d}{\sigma}\right)^p & \text{si } 0 \leq d < \sigma \\ 0 & \text{en el resto de los casos} \end{cases} \quad (\text{ec. A. 15})$$

$$Sharing_{exp_{\sigma,p}}(d) = \begin{cases} 1 & \text{si } d \leq 0 \\ 0 & \text{si } d \geq \sigma \\ \frac{e^{-\frac{pd}{\sigma}} - e^{-p}}{1 - e^{-p}} & \text{en el resto de los casos} \end{cases} \quad (\text{ec. A. 16})$$

- Donde p es un valor entero positivo. Con estos valores de *sharing* podemos definir el último concepto que nos hace falta: la cuenta de nicho (*niche count*) **(ec.A.17)** que se define como la suma del valor de *sharing* calculado para todos los individuos pertenecientes al nicho.

$$\forall p \in P \Rightarrow m(p, P) = \sum_{i=0}^{len(P)-1} Sharing_{\sigma} \left(dist(p, P_{[i]}) \right) \quad (\text{ec. A. 17})$$

Una vez hecho todo esto ya se puede proceder a calcular el *fitness* de cada individuo y dividirlo después por el *niche count*. De esta forma, los individuos que pertenezcan a zonas muy pobladas verán disminuidas sus posibilidades de reproducción, mientras que individuos más aislados no serán condenados a desaparecer [2].

A.3.2. Scaling

Los AG's seleccionan a los progenitores con una probabilidad proporcional al valor de *fitness*. Si nos servimos únicamente de estos valores de *fitness* sin tratarlos, los resultados esperables son básicamente dos: por un lado, si los valores de *fitness* están muy próximos entre sí, el AG podría seleccionar a cualquier individuo sin que ninguno llegase a destacar. Así pues, los cambios en la población serían mínimos y el AG quedaría reducido a una mera búsqueda aleatoria.

Si por el contrario los valores de *fitness* están muy alejados el efecto sería el contrario. Los individuos más adaptados serían seleccionados una y otra vez para reproducirse saturando la población (*súper individuos*), mientras que los menos adaptados desaparecerían sin dejar rastro. El AG convergería prematuramente sin haber explorado correctamente el espacio de búsqueda sobre la primera solución buena que hubiera encontrado [27].

Como es evidente, ninguno de estos efectos nos interesa en absoluto. Las técnicas de *Scaling* se encargan de suavizar estas diferencias entre los valores de *fitness* favoreciendo a los individuos más débiles cuando la presión selectiva sea alta, de forma que se preserve la diversidad en la mayor medida posible.

Lo métodos más comunes son el *Scaling lineal* y el *Scaling* exponencial [22]. Evidentemente no son los únicos, existen otros tipos de *Scaling* [28] y si nos decidiéramos a usarlo es importante elegir correctamente la técnica a utilizar, de ello dependerá después el correcto funcionamiento del AG.

A.3.2.1. Scaling lineal

Este método de *Scaling* es el más simple de todos. Es meramente un escalado lineal de los valores de *fitness* mediante la expresión siguiente (ec.A.18).

$$f_{scaling_lineal} = a \cdot f_{fitness} + b \quad (\text{ec. A. 18})$$

Generalmente el coeficiente a es inferior a uno, esto permite reducir los descartes de *fitness* y así favorecer la exploración del espacio. Es un método que no tiene en cuenta el número de generaciones y que penaliza el fin de la convergencia cuando queremos favorecer a los individuos dominantes [22].

A.3.2.2. Scaling exponencial

Se define mediante las expresiones siguientes **(ec.A.19)** y **(ec.A.20)**. Es un método más efectivo que el anterior de forma que su uso está más extendido.

$$f_{scaling_exponencial} = (f_{fitness}) \cdot k(n) \quad (\text{ec. A. 19})$$

$$k = \left(\tan \left[\left(\frac{n}{N+1} \right) \frac{\pi}{2} \right] \right)^p \quad (\text{ec. A. 20})$$

Donde n es la generacion actual, N es el numero de generaciones y p es un parametro fijado previamente (en general $p=0,1$). Podemos variar los valores de k **(ec.A.20)** para conseguir los efectos más deseables en cada momento.

- Cuando k tiene un valor próximo a 1 el *scaling* es inoperativo.
- Cuando k adopta un valor cercano a cero ningún individuo se ve realmente favorecido. El AG se comporta como un algoritmo de búsqueda aleatoria lo que permite una mejor exploración del espacio de búsqueda aunque por otro lado se dificulta la convergencia sobre un valor óptimo.
- Cuando $k > 1$ los descartes son exagerados y solamente los buenos individuos son seleccionados.

A.3.3. Otros

Sharing y *Scaling* son los métodos más extendidos para preservar la diversidad genética de la población pero no son los únicos. Existen una gran variedad de métodos entre los que podemos destacar los citados a continuación:

A.3.3.1. Crowding

Este método hace referencia a la función de inserción y presupone que un descendiente sólo podrá ocupar el lugar de un individuo de similares características [2] [26].

A.3.3.2. Restarting

Este sistema está pensado para luchar contra la *convergencia prematura* del AG. Pretende reiniciar partes de la población cada cierto periodo de tiempo [2], [26].

A.3.3.3. Restricted mating

Este método se basa en permitir únicamente que dos individuos se crucen si están separados, como mínimo, por una cierta distancia. De esta forma se dificulta la aparición de súper individuos que acaben con la diversidad [26].

A.3.3.4. Isolation by distance:

La opción propuesta en este caso es la de mantener diferentes núcleos de poblaciones separadas y que cada cierto tiempo se produzca algún tipo de intercambio poblacional [26]. Es una opción propia de los sistemas distribuidos como veremos más adelante en el apartado A.5.

A.4. Bases matemáticas de los Algoritmos Genéticos

Como hemos podido ver hasta el momento, los AG's resultan ser métodos de los más efectivos para la exploración y búsqueda de valores óptimos. Pero, ¿en qué reside su efectividad? La respuesta se basa en un *paralelismo implícito* que presentan los AG's, inherente a su funcionamiento, y que no depende de su implementación.

Para poder explicar correctamente este *paralelismo implícito* es necesario profundizar en las bases matemáticas de los AG's e introducir una serie de conceptos nuevos hasta ahora como son los *modelos de esquemas* [2], [11], [25], y la teoría de *bloques constructivos* [29], [31].

Según Goldberg [25], un esquema es un modelo de similitud que comprende un conjunto de cadenas con símbolos iguales en algunas posiciones. Es decir, que diríamos que dos individuos pertenecen al mismo esquema si tienen los mismos valores en determinadas posiciones (**Fig.A.16**). Esto no quiere decir que sean igual completamente, sólo que tienen determinados valores en común.



Fig.A.16 Ejemplo de esquema

Antes de proseguir adelante, nos serviremos del esquema de la figura anterior (**Fig.A.16**), para definir un par de conceptos importantes:

- **Orden del esquema:** Se define como *orden del esquema* al número total de valores fijos que tiene [25]. Ej. dado el esquema de la figura anterior (**Fig.A.16**) $H=1*0*0*$, el orden de este esquema sería 3.
- **Longitud del esquema:** Se define como *longitud del esquema* como la distancia que hay desde el primer valor fijo hasta el último [25]. Ej. En el esquema anterior $H=1*0*0*$ la longitud sería 4.

Así pues, en la figura anterior (**Fig.A.16**) tenemos dos individuos diferentes pero pertenecientes al mismo esquema. Ahora bien, estos dos individuos también pertenecen a un sinnúmero de esquemas más. En realidad, dentro de una población de individuos codificados en binario y de longitud l pueden generarse $(n+1)^l$ esquemas diferentes.

Estos esquemas representan patrones comunes de los individuos. Patrones que se evalúan indirectamente en la función de *fitness* y que evolucionan, se cruzan, mutan y se combinan como características de los individuos que son. De manera que, igual que va convergiendo la población del AG hacia el valor óptimo, la diversidad de esquemas a los que pertenecen los individuos también va convergiendo hacia los *esquemas* más “óptimos” (también llamados *bloques constructivos*).

Según el **teorema fundamental**, formulado por Holland en 1975 [11], estos *bloques constructivos* (que son esquemas cortos, de bajo orden y que ofrecen un *fitness* superior a la media), son los que tienen más probabilidades de propagarse a lo largo del proceso evolutivo del AG. Y aquí es donde aparece el *paralelismo implícito* de los AG's puesto que, según los trabajos del mismo Holland [11] y de Goldberg [25], al analizar una determinada generación de una población de tamaño n , se están analizando simultáneamente del orden de n^3 esquemas diferentes.

De esta forma durante el proceso de búsqueda del AG no sólo se están analizando n individuos de una población. También se están analizando n^3 patrones distintos que, según la *teoría de bloques constructivos*, tenderán a forma cadenas cada vez mejores a partir de las encontradas en pasados muestreos [29].

De esta forma, en cada generación el espacio de búsqueda está siendo explorado en muchas más direcciones de las que se ven a aparentemente a simple vista. Esto deja entre ver la potencia y la sencillez del modelo que utilizan los AG's. Pero no los convierte en la solución perfecta a todos los problemas.

Precisamente estas características determinan las propias limitaciones del AG: es necesario que los problemas a los que se les pretende aplicar este método verifiquen esta *hipótesis de bloques constructivos* como por ejemplo en los problemas de optimización combinatoria. De otro modo es muy fácil que los AG's resulten ineficientes como por ejemplo en problemas con óptimos aislados (**Fig.A.17**).

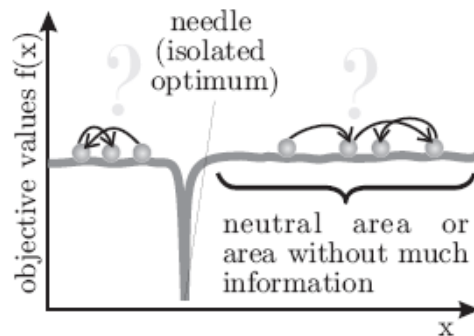


Fig.A.17 Ejemplo de espacio de búsqueda con un óptimo aislado donde difícilmente daremos con el utilizando un *Algoritmo Genético* [2]

A.5. Mejoras

A.5.1. Algoritmos en paralelo (Modelos de islas)

Una de las grandes propiedades de los AG's es el paralelismo implícito que ofrecen como ha quedado expuesto antes en el apartado A.4. y que es inherente al propio algoritmo y no depende de la implementación que se haga del mismo. En este apartado trataremos otro tipo de paralelismo. Uno más inclinado hacia el correcto aprovechamiento de los recursos para maximizar más si cabe la efectividad de los AG.

Para poder entender mejor la aparición de estos modelos es necesario mirar atrás para recordar un concepto importante mencionado en el apartado A.1.2 sobre el tamaño de la población. Como se dijo anteriormente el tamaño de la población es un factor determinante en la efectividad y en el tiempo de convergencia del AG. Cuanto mayor sea la población mejor será la exploración del espacio de búsqueda, pero mayor será el tiempo de convergencia necesario. Parece que lo ideal sería encontrar la forma de aumentar la población de búsqueda sin perjudicar el tiempo de convergencia, y es en este punto donde entran en juego los llamados *modelos de islas* [30].

La idea es que en lugar de ejecutar un único AG sobre una gran población de tamaño M , resulta mucho más efectivo ejecutar n AG's, de forma simultánea, sobre una sub-población de tamaño M/n individuos. De ahí el concepto de *modelo de islas*, ya que tendríamos una serie de núcleos o islas aisladas, donde se ejecutaría un AG de forma independiente a las demás.

Para mantener la diversidad y evitar que las islas converjan hacia óptimos locales es necesario introducir un nuevo concepto llamado *migración*. Esta *migración* supone que cada determinado número de generaciones, las islas intercambiaran algunos de sus individuos entre sí, a razón de una *tasa de migración* previamente establecida. Esta *tasa de migración* es, al igual que la *tasa de mutación*, un factor determinante para la convergencia del AG y ha de escogerse con cuidado. Según Stender en [30], podemos diferenciar diferentes *modelos de islas* en función del tipo de arquitectura de intercambio genético que presentan.

A.5.1.1. Arquitectura en estrella

Para este tipo de arquitectura definimos una población maestra y una serie de poblaciones esclavas. La evolución dentro de cada una de estas islas se produce de forma independiente hasta el momento de la migración. Llegado este momento, las poblaciones esclavas seleccionan k individuos a partir de alguno de los métodos antes descritos [apartado A.1.4] y se los envían a la población maestra. A su vez, esta población maestra selecciona sus k individuos para ser enviados a todas las poblaciones esclavas.

A.5.1.2. Arquitectura en red

De metodología muy similar a la anterior, sólo se diferencia en que no existe una población maestra. Todas las poblaciones tienen la misma categoría y, llegado el momento de la migración, el intercambio se produce desde cada población hacia todas las demás (**Fig.A.18**).

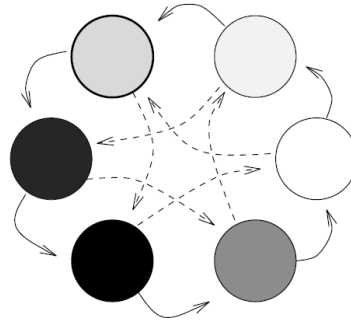


Fig.A.18 AG's en paralelo unidos por una arquitectura en red [31]

A.5.1.3. Arquitectura en anillo

El modelo de isla en anillo, también conocido como el modelo de la pasarela (*Stepping Stone*) [32] es muy similar al expuesto anteriormente. En este caso tampoco existe ninguna población con jerarquía superior a las demás. De nuevo el AG interno de cada sub-población se ejecuta de forma independiente y, cada determinado número de generaciones, se produce la *migración* para asegurar la diversidad de las poblaciones (**Fig.A.19**).

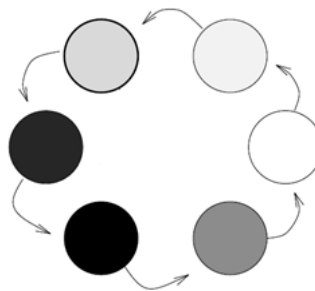


Fig.A.19 AG's conectados en paralelo mediante arquitectura en anillo [31]

La diferencia reside en que los k individuos seleccionados por cada población se intercambian únicamente con su población vecina y siempre en el mismo sentido giratorio.

A.5.2. Algoritmos celulares

Este tipo de implementación se podría considerar un caso particular los modelos de islas anteriormente expuestos, pero trabajado con un número elevado de procesadores [2] [31] [32].

Supongamos que disponemos de 2500 procesadores dispuestos en una matriz de 50 x 50. Cada uno de estos procesadores se encargará de un único individuo y tendrá limitadas las comunicaciones únicamente a sus vecinos adyacentes (es decir, solo a los que estén en contacto directo en lo que a su disposición matricial se refiere).

Así pues, cada procesador observará a los individuos de sus vecinos eligiendo a uno de ellos (a partir de alguno de los métodos de selección antes expuestos [A.1.4]) para combinarlo con el suyo. Al cabo de un cierto número de iteraciones se empezarán a formar núcleos de población con características similares (**Fig.A.20**).

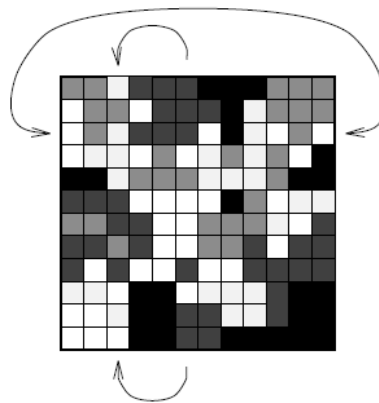


Fig.A.20 Ejemplo de *algoritmo genético Celular* [31]

Esto crea una similitud con los modelos de islas. Sí que es cierto que no existen poblaciones aisladas “físicamente”, pero los individuos que se encuentren a una distancia superior a 20 o 25 procesadores unos de otros difícilmente podrán interactuar entre ellos. A esto es a lo que se conoce como *aislamiento por distancia* [33].

A.5.3. Procesos en paralelo (Balanceado de carga de procesamiento)

El objetivo de este apartado es aplicar un tercer tipo de paralelismo a los AG's de manera que aún se potencie más su eficiencia. Hasta el momento hemos visto el *paralelismo implícito* de los AG's codificados en binario [Apartado A.4]. Posteriormente hemos introducido el concepto de aplicar más de un AG de manera simultánea sobre una población distribuida (*modelos de islas* [A.5.1] y *algoritmos genéticos celulares* [A.5.2]).

De esta forma se explora más profundamente el espacio de búsqueda sin perjudicar demasiado los tiempos de proceso. Además de esto, aún podemos mejorar más la eficiencia de los AG's balanceando la carga de procesamiento sobre diferentes *threads* o sobre diferentes procesadores. Para esto sólo hemos de ser conscientes de que la mayor parte del tiempo de proceso que genera un

AG lo produce la *función de fitness*. A partir de aquí una sencilla opción sería la que observamos en la figura siguiente.

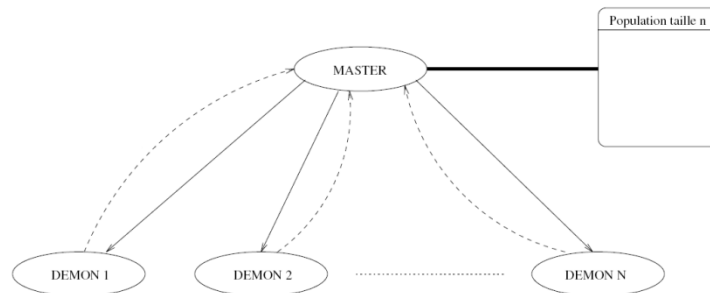


Fig.A.21 Distribución de la carga de procesamiento del *algoritmo genético* [22]

En este caso tenemos definido un procesador maestro que se encarga prácticamente de realizar todas las funciones a excepción de la *función de fitness*. Este proceso se encuentra distribuido entre el resto de los procesadores que enviarán sus resultados al procesador maestro. Este, con los valores de *fitness* calculados por los esclavos, continuará sólo con el proceso hasta que se tenga que evaluar a una nueva generación. Esta idea, en principio simple y ventajosa, ofrece algunos aspectos a tener en cuenta:

- En primer lugar, el hecho de disponer de un procesador maestro que efectúe la totalidad de las funciones puede generar cuellos de botella indeseados.
- En segundo lugar, este sistema implica un alto intercambio de mensajes. De manera que este tipo de arquitectura no es aconsejable para todos los problemas. Únicamente para aquellos problemas en los que el tiempo de procesamiento de la *función de fitness* sea considerablemente superior al tiempo necesario para el intercambio de los mensajes [31], [33].

A.5.4. Algoritmos Genéticos Híbridos

El último paso sería combinar la eficacia de los AG en encontrar zonas prometedoras, con la exhaustividad de exploración de algunos de los métodos expuestos en el capítulo 1. A este tipo de AG's que se combinan con otros métodos más específicos de búsqueda se les denomina *Algoritmos Genéticos Híbridos* o *Algoritmos Meméticos* y fueron introducidos por P. Moscato en 1989 [34].

Quizás estos AG's *Híbridos* son el máximo exponente de la versatilidad que presenta esta rama de los *algoritmos evolutivos* ya que, a pesar de que los AG's no son el método definitivo para resolver todos los problemas, aún pueden resultar eficaces como orientación inicial para otros métodos más específicos.

A.6. Cromosomas de longitud variable

Todo lo expuesto hasta ahora gira en torno a la idea de que los cromosomas de los individuos tienen una longitud fija. Esto supone problemas en los que el número de parámetros es conocido e invariable. Esto no siempre es así, existen multitud de problemas como por ejemplo los de optimización de trayectorias o las redes neuronales, donde el número de estados por los que puede pasar un individuo es desconocido. Es decir, los cromosomas de dichos individuos pueden tener diferentes longitudes, siempre limitadas a un máximo preestablecido por el usuario.

Estos *cromosomas de longitud variable* fueron propuestos por primera vez por Smith en 1980 [35] y en esencia no suponen un cambio en el funcionamiento básico del AG, aunque sí que incorporan algunas modificaciones en lo que a la generación de la *población inicial* se refiere y también en las funciones de mutación y de cruce como se explica en [2] [33].

A.6.2. Población inicial con cromosomas de longitud variable

Igual que en el caso de los AG's "normales", los individuos iniciales se han de generar de alguna forma. Además de todo lo expuesto anteriormente [Apartado A.1.2] habría que plantearse una nueva pregunta: ¿Cuál es la longitud adecuada para estos individuos? la opción más utilizada consiste en generar de forma aleatoria la longitud de dichos cromosomas manteniendo un par de condiciones.

- La longitud l ha de ser inferior a un máximo definido previamente, que puede ser el fruto de algún algoritmo de optimización previamente utilizado [33].
- La longitud l ha de contener un número entero de genes.

A partir de aquí el proceso de creación de sus respectivos genes sigue el mismo proceso que para los de longitud fija. Es decir, rellenar los cromosomas con valores del alfabeto escogidos aleatoriamente.

A.6.3. Función de mutación con cromosomas de longitud variable

En este caso, la *función de mutación* no sólo puede variar un gen existente dentro del cromosoma. Además, la mutación puede darse en forma de adición o eliminación de un gen del cromosoma. Por este motivo es necesario añadir dos nuevas funciones al AG:

- *Función insertar*: Que tal como su propio nombre indica, inserta un gen al cromosoma. La forma de hacerlo puede variar. El método más común consiste en duplicar un gen ya existente dentro del cromosoma, mutarlo (con una probabilidad mínima como en casos anteriores) e insertarlo, generalmente, en la última posición del cromosoma [19].
- *Función eliminar*: Como su propio nombre indica esta función elimina uno de los genes del cromosoma. La elección del cromosoma a eliminar suele ser aleatoria, aunque también puede estar basada en el nivel de saturación de ese cromosoma dentro de la población.
-

A.6.3. Función de cruce con cromosomas de longitud variable

De igual forma que en casos anteriores, la función de cruce de cromosomas de longitud variable sigue los mismos patrones que para cromosomas de longitud fija. Con la salvedad de que el resultado de cruzar dos individuos de longitudes diferentes suele dar lugar a hijos de longitudes distintas a las de sus padres (**Fig.A.22**).

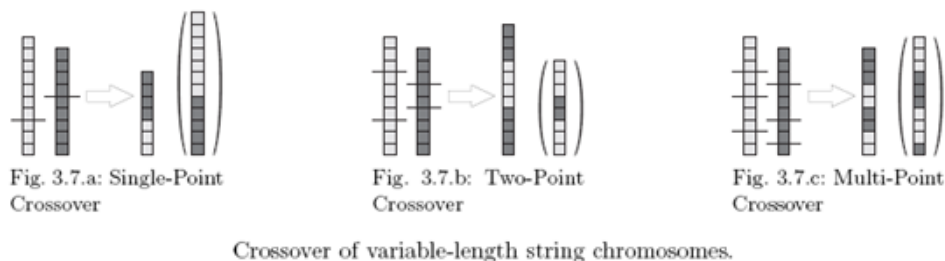


Fig.A.22 Ejemplos de cruces con cromosomas de longitud variable [2]

Por último, es importante recordad que todas estas funciones (*cruce*, *mutación*, *eliminar* e *insertar*) pueden modificar la longitud de los cromosomas que definen a los individuos de la población, pero respetando siempre las dimensiones propias de los genes que los componen. Es decir, que no se deben construir cromosomas con dimensiones que no sean múltiplos enteros de un gen.

ANEXO B: FUNCIONES ESCOGIDAS

Para poder poner a prueba los diferentes parámetros del AG es necesario definir algunos escenarios de búsqueda. En este caso no se trata de funciones excesivamente complicadas, más bien lo contrario. Se trata de un conjunto de cinco funciones simples, con valores óptimos conocidos de forma que se puedan evaluar fácilmente los resultados.

B.1. Función A

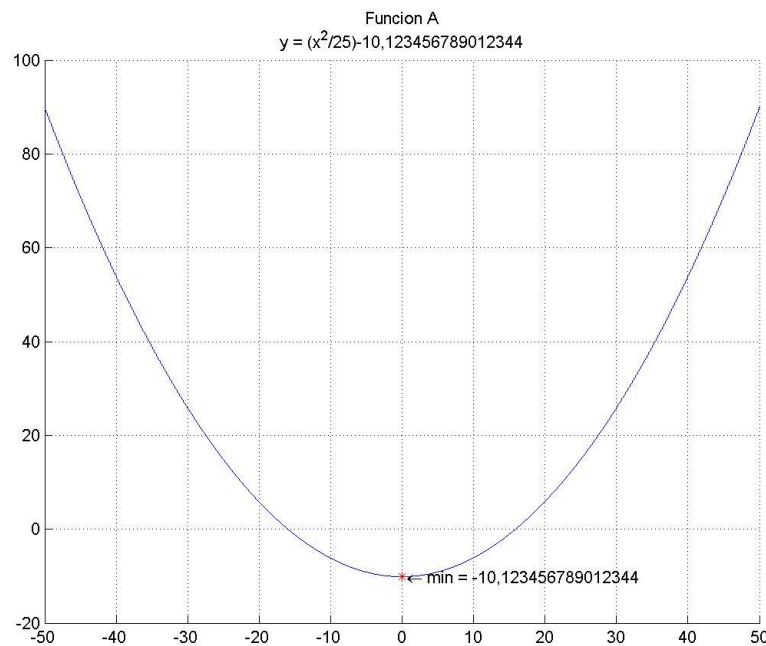


Fig.B.1 Gráfica de la función A.

$$y = \frac{x^2}{25} - 10,12345678901234 \quad \forall x \in [-50, 50] \quad (\text{ec. B. 1})$$

Una función muy simple pero perfecta para comprobar el funcionamiento del AG en lo que a minimización se refiere. Es muy simple y la solución óptima es visible a simple vista. Además el utilizar tantos decimales permite comprobar el grado de precisión que puede alcanzar el AG en un problema simple. El rango de búsqueda está definido entre -50 y 50 pero puede ser muchísimo más extenso.

B.2. Función B

Una función un poco más compleja: el objetivo será alcanzar el máximo que tiene en el origen y ver como el AG evita los máximos locales que tiene alrededor. En este caso el rango ha de ser el establecido puesto que luego la función se torna más estable con muchos máximos de valor 1 que no permitirían demostrar nada.

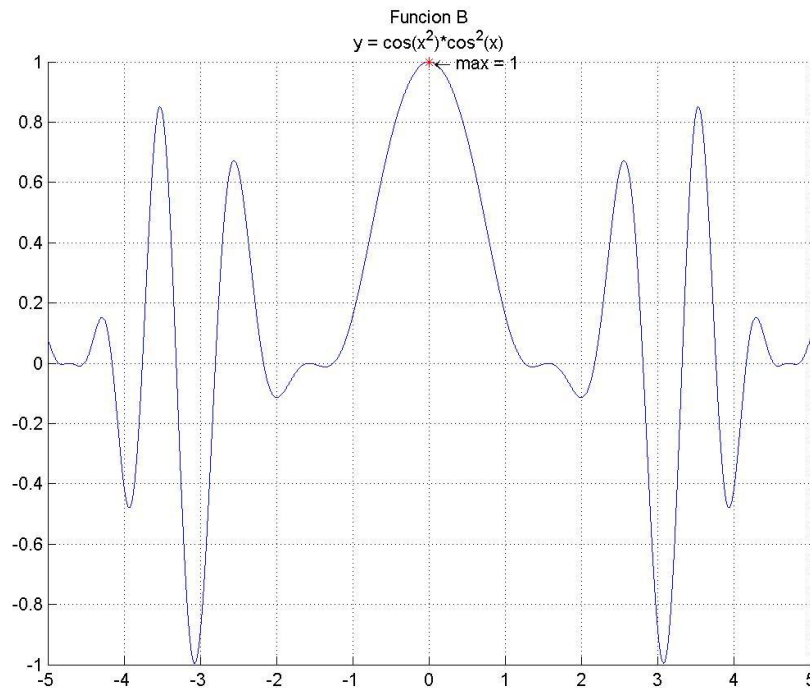


Fig.B.2 Gráfica de la función B

$$y = \cos x^2 \times \cos^2 x \quad \forall x \in [-5,5] \quad (\text{ec. B. 2})$$

B.3. Función C

Esta función con dos variables definida en este rango permite tanto la maximización como la minimización, de manera que se pueda comprobar el correcto funcionamiento del AG en problemas de más de una variable.

$$Z = (\cos x * \cos y) * (x + y) \quad \forall x, y \in [-5,5] \quad (\text{ec. B. 3})$$

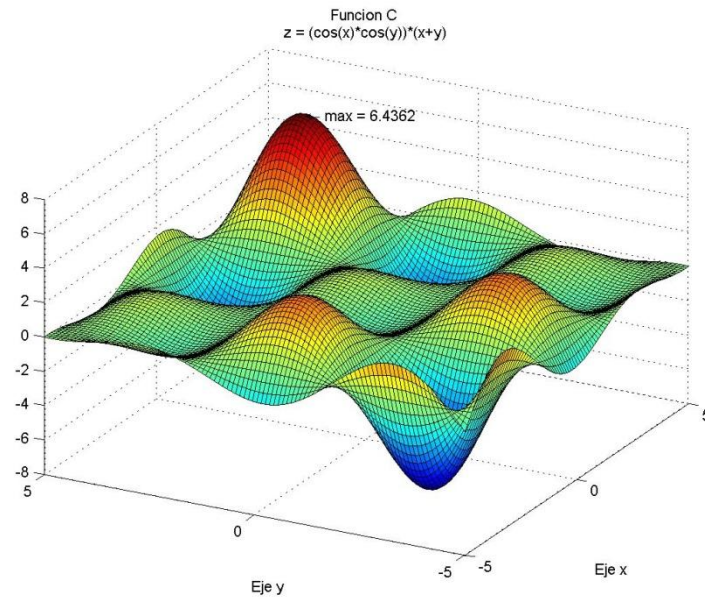


Fig.B.3 Gráfica de la función C

B.4. Función D

Esta función con multitud de máximos y mínimos locales en los que quedar atrapado y que, si bien corresponden a buenas soluciones para este problema, no permiten alcanzar otras soluciones mejores y que suponen uno de los grandes problemas que presentan los AG. En este caso, esta función será el terreno perfecto para ver *fracasar* al AG y ver los efectos beneficiosos que ofrecen las técnicas de *Sharing*.

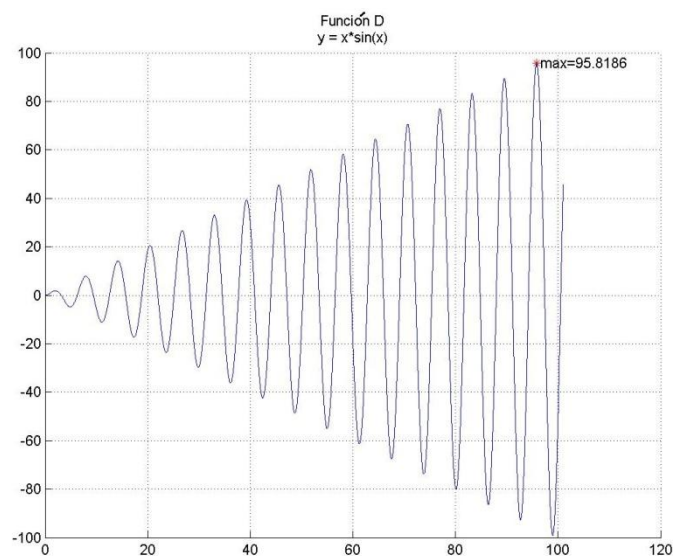


Fig.B.4 Gráfica de la función D

$$y = x * \sin x \quad \forall x \in [0,100] \quad (\text{ec.B.4})$$

B.5. Función E

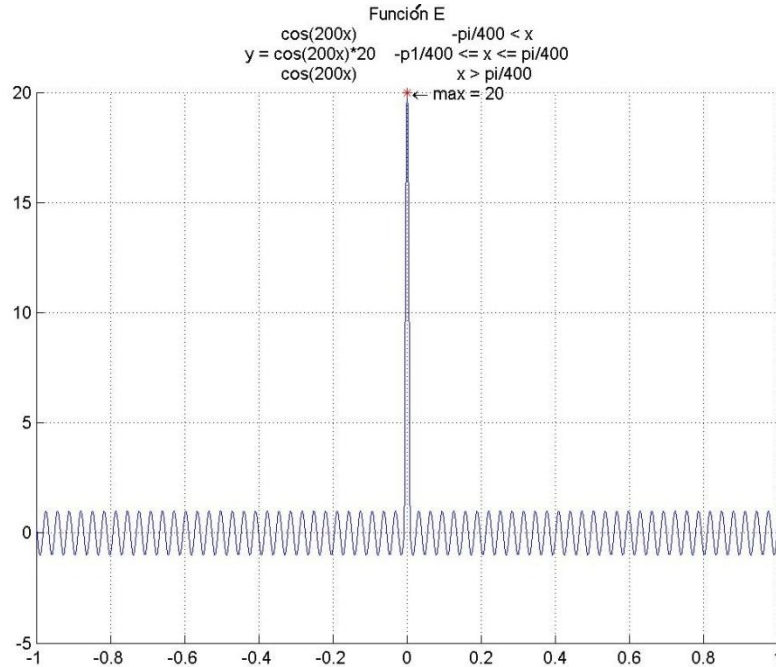


Fig.B.5 Gráfica de la función E

$$y = \begin{cases} \cos 200x & \text{si } x > -\frac{\pi}{400} \\ \cos 200x * 20 & \text{si } x \in \left[-\frac{\pi}{400}, \frac{\pi}{400}\right] \\ \cos 200x & \text{si } x < \frac{\pi}{400} \end{cases} \quad (\text{ec. B. 5})$$

Esta función está especialmente definida para dejar patente otro de los puntos débiles de los AG's: los óptimos aislados. Durante las pruebas se podrá comprobar que no es imposible que el algoritmo de con el óptimo asilado, pero que en un gran número de ocasiones pasará desapercibido. Además permitirá poner a prueba los valores óptimos que hemos determinado con los diferentes tests anteriores y ver si aun así el AG sigue sin dar con la respuesta correcta en un porcentaje razonable de ocasiones. Referente al rango de búsqueda: cuanto mayor sea el espacio de búsqueda más patente será la debilidad del AG, pero también se comprobará que espacios pequeños como el de la figura no garantizan el éxito de la búsqueda.

ANEXO C: PROBLEMA FINAL

C.1. Tipos de contenedores

A. CARGO SYSTEMS



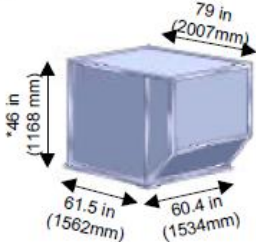
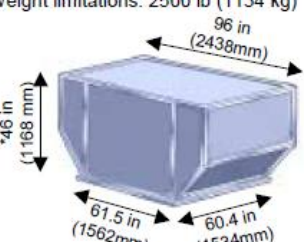
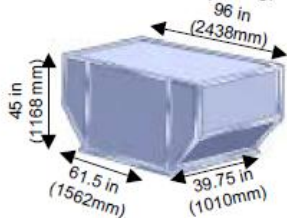
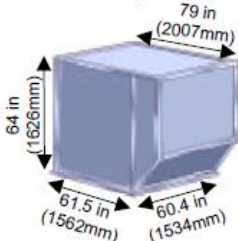
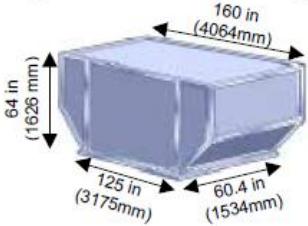
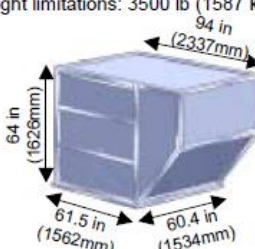
<p>CONTAINER AKG</p> <p>IATA ULD Code: AKG Classification ATA: LD3-46 Aircraft: A320 family and interlining Volume: 110 ft³(3.1m³) Weight limitations: 2500 lb (1134 kg)</p> 	<p>CONTAINER AKH</p> <p>IATA ULD Code: AKH Classification ATA: LD3-46W Also known as DKH. Aircraft: A320 family and interlining Volume: 127 ft³(3.6m³) Weight limitations: 2500 lb (1134 kg)</p> 
<p>Additional CONTAINER A319</p> <p>IATA ULD Code: H Classification ATA: LD3-40/45 Aircraft: A319 Volume: 80 ft³(2.25m³) Certified MGW limit: 1660 lb (753 kg)</p> 	<p>CONTAINER AKE</p> <p>IATA ULD Code: AKE Classification ATA: LD3 Also known as AVA, AVB, AVC, AVK, DVA, DVE, DVP, XKS, XKG. Aircraft: A310, A300, A330, A340. Volume: 145 to 158 ft³(4.1 to 4.47m³) Weight limitations: 3500 lb (1587 kg)</p> 
<p>CONTAINER ALF</p> <p>IATA ULD Code: ALF. Classification ATA: LD6 Aircraft: A310, A300, A330, A340. Volume: 316 ft³(8.9m³) Weight limitations: 7000 lb (3175 kg)</p> 	<p>CONTAINER AKC</p> <p>IATA ULD Code: AKC Classification ATA: LD1 Also known as AVC, AVD, AVK, AVJ. Aircraft: A310, A300, A330, A340. Volume: 159 to 173 ft³(4.5 to 4.9m³) Weight limitations: 3500 lb (1587 kg)</p> 

Fig.C.1 Tabla con diferentes tipos de contenedores (1/2) [15]



A. CARGO SYSTEMS

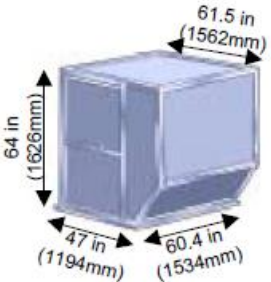
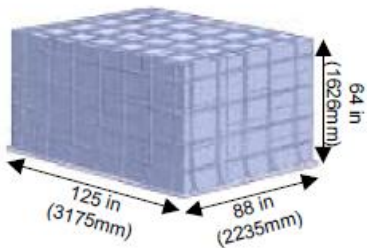
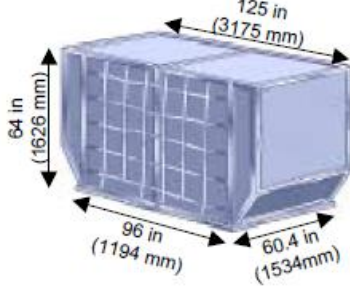
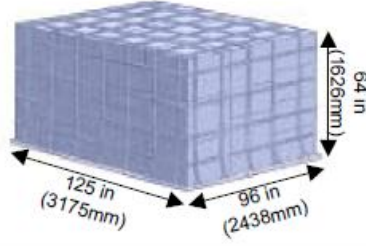
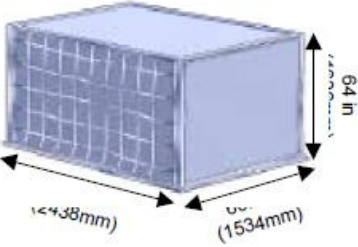
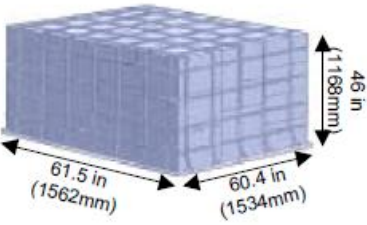
<p>CONTAINER DPE</p> <p>IATA ULD Code: DPE Classification ATA: LD2. Also known as APA, DPA. Aircraft: A330, A340. Volume: 120 ft³(3.4m³) Weight limitations: 2700 lb (1224 kg)</p> 	<p>PALLET PAx</p> <p>IATA ULD Code: Pax Classification ATA: LD7 Also known as PAA, PAG, PAJ, PAP, PAX, P1A, P1C, P1D, P1G. Aircraft: A310, A300, A330, A340. Volume: 372 ft³(10.5m³) Weight limitations: 10200 lb (4626 kg)</p> 
<p>CONTAINER DQF</p> <p>IATA ULD Code: DQF. Classification ATA: LD8 Also known as ALE, ALN, DLE, DLF, DQP. Volume: 245 ft³(6.9m³) Weight limitations: 5400 lb (2449 kg)</p> 	<p>PALLET PMx</p> <p>IATA ULD Code: PMx Classification ATA: LD9 Also known as P6P, P6A, P6Q, PMA, PMC, PMP, PQP. Aircraft: A310, A300, A330, A340. Volume: 407 ft³(11.5m³) Weight limitations: 11250 lb (5103 kg)</p> 
<p>CONTAINER DQP</p> <p>IATA ULD Code: DQP Classification ATA: LD4 Volume: 202 ft³(5.7m³) Weight limitations: 5400 lb (2449 kg)</p> 	<p>PALLET PKx</p> <p>IATA ULD Code: PKx Volume: 85 ft³(2.4m³) Aircraft: A320 family Weight limitations: 2500 lb (1134 kg) Certified MGW limit: 3500 lb (1587 kg)</p> 

Fig.C.2 Tabla con diferentes tipos de contenedores (2/2) [15]

En estas hojas están parametrizados los efectos de las diferentes masas que pueden introducirse en el avión en función de la posición donde se sitúen. Resulta una herramienta imprescindible puesto que sustituye a toda la serie de cálculos complejos necesarios para localizar el c.g. a costa, eso sí, de permitir unas ciertas simplificaciones como por ejemplo agrupar pasajeros por zonas o la carga por bodegas.

Sin entrar demasiado en profundidad sobre su funcionamiento diremos que los campos (1) y (2) hacen referencia a su masa en vacío, masas máximas, etc.

La corrección aplicada en el punto (3) hace referencia al peso (de más o de menos) propio de los tripulantes o del catering del vuelo [37].

Son los apartados (4) y (5) los más interesantes, puesto que en ellos se ve claramente los efectos que provoca el cargar más o menos peso en uno u otro lugar del aparato.

En la figura siguiente podemos ver un ejemplo de cómo se ve afectado el c.g. del Airbus A320 al repartir la carga por sus bodegas e introducir el peso de los pasajeros y el combustible (**Fig.C.4**).

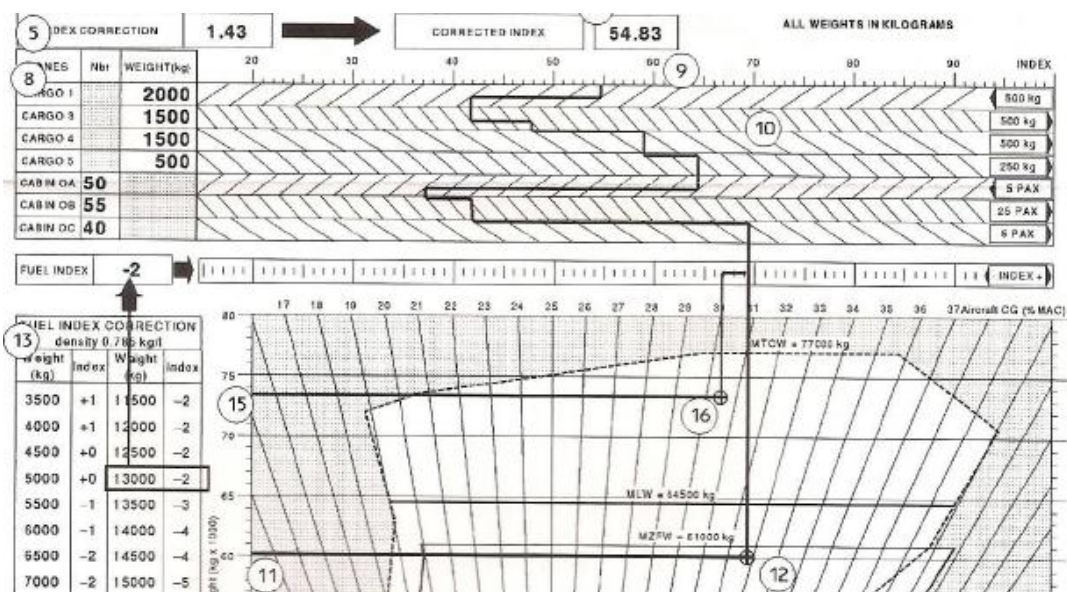


Fig.C.4 Ejemplo del cálculo del centro de gravedad del Airbus A320 mediante las hojas tabuladas de equilibrio [38]

Una vez introducidos todos los parámetros obtenemos dos valores finales: MTOW (*Maximum Take Off Weight*) que representa el “peso” del avión en el momento del despegue (cuando está cargado completamente de combustible).

Y el MZFW (*Minimum Zero Fuel Weight*) que representa el “peso” del aparato cuando ha agotado los depósitos de combustible (a excepción de una pequeña reserva).

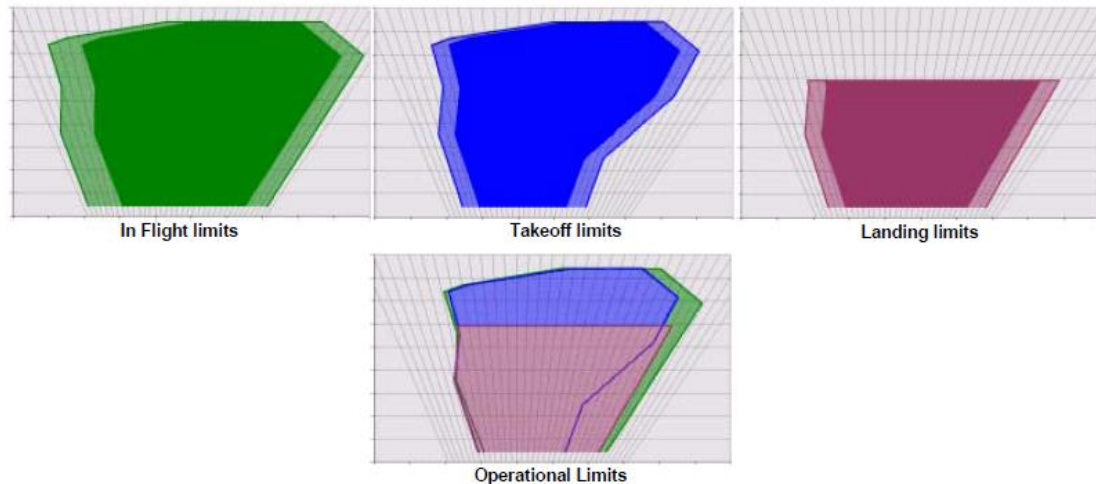


Fig.C.5 Limitaciones físicas aportadas por el fabricante del Airbus A320 [15]

Ambos parámetros han de estar dentro del gráfico (6) que representa las limitaciones propias del modelo y que son aportadas por el fabricante (**Fig.C.5**). Si el resultado estuviera fuera de dicho gráfico no sería aceptable puesto que representaría una violación de las limitaciones físicas del aparato e implicaría redistribuir o incluso dejar en tierra parte de la carga.

Este sería el caso más extremo. Lo normal es que el resultado se encuentre dentro de las limitaciones propias del aparato, aunque desviado de la posición de equilibrio. En este punto es donde entra en juego el ángulo de TRIM anteriormente comentado, puesto que su función es compensar estas desviaciones para permitir que el aparato siga siendo gobernable.

El ángulo TRIM está directamente ligado a la posición del c.g. y se obtiene de forma simultánea como podemos ver en la **figura C.6**. En esta imagen vemos que para cada valor aceptable, que no posible, del centro de gravedad existe un ángulo de compensación. El rango de dicho ángulo va de los -2.5° a 2.5° (de manera que no puede compensar desviaciones fuera de las limitaciones aportadas por el fabricante), y su posición cero está asociada a un índice aproximado de 61,5.

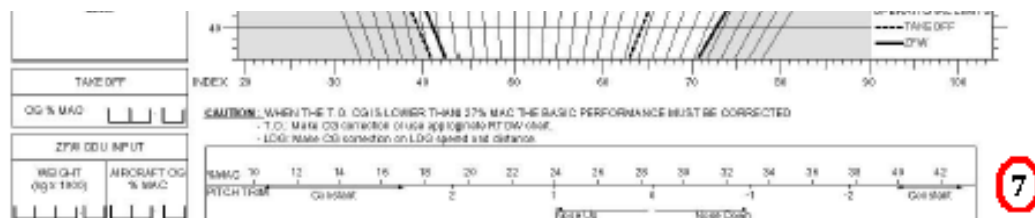


Fig.C.6 Parte final de la hoja de equilibrado donde aparece el valor del ángulo TRIM asociado al índice obtenido [38]

C.3. Cuantificación de los parámetros utilizados en las hojas de carga y equilibrado del avión

Como hemos explicado con anterioridad, mediante estas hojas de equilibrado podemos ver cómo las diferentes masas del avión afectan y desplazan el centro de gravedad, haciendo necesario compensarlo mediante el ángulo TRIM. Este ángulo es el centro del sistema optimización que hemos definido puesto que nuestro objetivo es conseguir que sea lo más próximo a cero posible. Para conseguir esto, el primer paso es cuantificar todas las contribuciones posibles y presentes en las tablas para pasar de este sistema gráfico a uno numérico simple.

Así pues, en primer lugar hemos de determinar el índice DOW (*Dry Operating Weight*) que no es otra cosa que el centro de equilibrio del avión cuando está vacío (**Fig.C.7**). Este parámetro es diferente para cada modelo, incluso para cada aparato concreto, pero para nosotros será el menor de los problemas puesto que sólo trabajaremos con un modelo fijo. De esta forma aunque podemos calcular el índice DOW de cualquier avión mediante la fórmula que aparece en la cabecera de la hoja (siempre y cuando tengamos sus datos técnicos), nos limitaremos a designar como valor fijo el índice DOW en 53,4.

Fig.C.7 shows two forms. The left form (red box) is titled 'DRY OPERATING WEIGHT CONDITIONS' and contains a table with the following data:

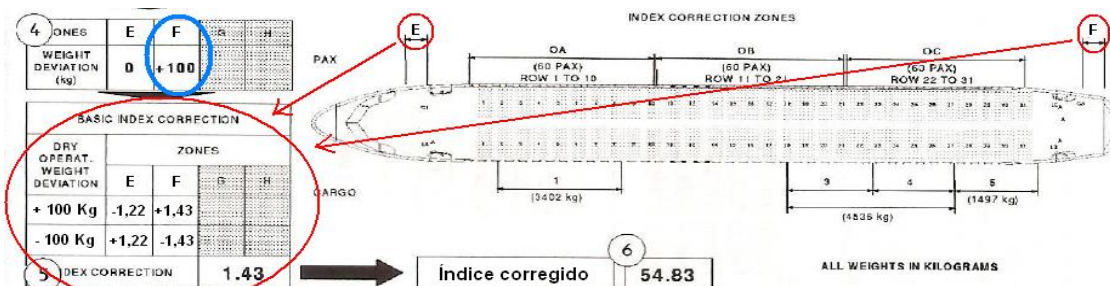
	WEIGHT (kg)	lt-arm (m)
1	42 500	18.93
2	$\frac{(lt-arm - 18.8199) \times W}{1000} = 50$	
3	RY OPERATING WEIGHT INDEX	53.4

The right form (green box) is titled 'AIRCRAFT REGISTER' and contains the following data:

DRY OPERATING WEIGHT	42 500
WEIGHT DEVIATION (PANTRY)	+ 100
CORRECTED DRY OPERATING WEIGHT	42 600
CARGO	5 500
PASSENGERS	145 x 84 = 12 180
ZERO FUEL WEIGHT	60 280
TOTAL FUEL ONBOARD	13 000
TAKEOFF WEIGHT	73 280

Fig.C.7 Datos técnicos del avión y el vuelo (recuadro verde), así como la fórmula para determinar su índice DOW (recuadro rojo) [38]

El siguiente paso es cuantificar la contribución del *catering* y de los propios tripulantes de más o de menos que viajen en el vuelo. A este conjunto de “pesos” se le denomina genéricamente *Pantry*. Como podemos ver en la **figura C.8** dependiendo de la zona y del peso de más o de menos que se inserte se obtiene un factor de corrección para el índice DOW (círculos rojos).



Como podemos ver en la **figura C.8**, en este ejemplo existe un incremento de 100 kg en la zona F (círculo azul), de manera que se ha de aplicar un factor de corrección de +1,43. Mediante una división simple podemos determinar el factor de corrección por kg de carga en cada zona (**tabla C.1**) y luego sería tan sencillo como multiplicar por la cantidad de kg de más o de menos (signo positivo o signo negativo).

PANTRY_E	-1,22/100
PANTRY_F	1,43/100

Tabla C.1 Contribuciones cuantificadas del peso de los tripulantes, catering, etc

Una vez disponemos del índice corregido habríamos de repetir el proceso con el grueso de la hoja de equilibrado (**figura C.9**), donde se encuentran las contribuciones de las diferentes de las bodegas, zonas de pasajeros y combustible.

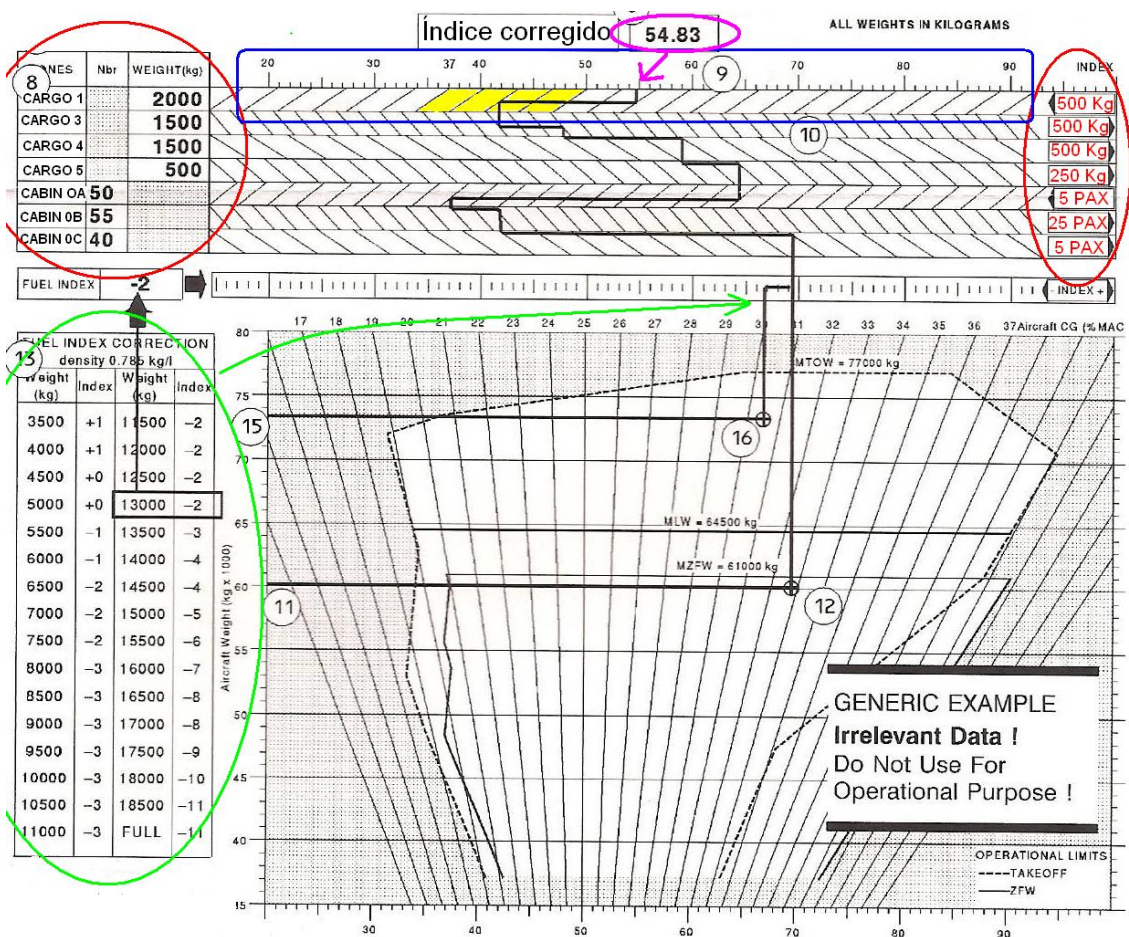


Fig.C.9 Contribuciones de la carga en las bodegas, zonas de pasajeros y combustible [38]

Como podemos ver en la **figura C.9** partiremos del índice corregido obtenido en el punto anterior. A partir de esta posición inicial, iremos produciendo desplazamientos a izquierda y derecha según la cantidad de kilos y la zona concreta donde se hayan introducido.

Si nos fijamos en las zonas rodeadas por los círculos rojos de la **figura C.9** observaremos que cada sección contribuye de una manera diferente, siendo las bodegas 1 y 4 y la zona de pasajeros OC (zona trasera) las que más desplazan nuestro centro de gravedad. Nuevamente se trataría de cuantificar este desplazamiento para cada una de las zonas sobre la escala graduada (recuadro azul).

No hace falta recordar que este proceso se ha de realizar bajo el criterio de cada uno, de manera que será inevitable introducir un cierto “error humano” de apreciación. Para intentar reducir al máximo este factor intentaremos seleccionar en la medida de lo posible contribuciones de una casilla y a ser posible que coincidan claramente con alguna de las marcas de la escala. Por ejemplo para determinar la contribución de la bodega 1 (CARGO 1) seleccionaremos la zona coloreada de amarillo. Así pues, cada 2000 Kg (4 casillas * 500 Kg) se desplaza el índice 13 posiciones a la izquierda (50 - 37). Siguiendo este proceso, y tomando los desplazamientos a la derecha como positivos, podemos cuantificar todas las contribuciones por kilo de peso obteniendo la **tabla C.2**.

FACTOR_CARGO_1	-3.16/500
FACTOR_CARGO_3	1.86/500
FACTOR_CARGO_4	3.69/500
FACTOR_CARGO_5	2.89/250
FACTOR_CABIN_0A	-2.72/5
FACTOR_CABIN_0B	2.08/25
FACTOR_CABIN_0C	3.43/5

Tabla C.2 Contribuciones por kilo de peso de las bodegas y zonas de pasajeros

A continuación sólo nos resta introducir la contribución producida por el peso del combustible. Por suerte ésta se encuentra ya cuantificada en la misma **figura C.9** (circulo verde) de manera que sólo es necesario crear una variable llamada **FACTOR_FUEL** y asignarle el valor de corrección correspondiente para cada cantidad de combustible.

Llegados a este punto ya disponemos del valor final del índice DOW y podemos determinar el valor del ángulo TRIM que necesitamos. Observando la **figura C.10** podemos ver las relaciones que existen entre el valor del índice DOW y el TRIM: Por un lado vemos que la posición cero del ángulo TRIM corresponde a un índice aproximado de 61,5 unidades. Por otro lado vemos que el rango completo de acción del ángulo de TRIM (los cinco grados completos) equivale a unas 77 unidades del índice DOW.

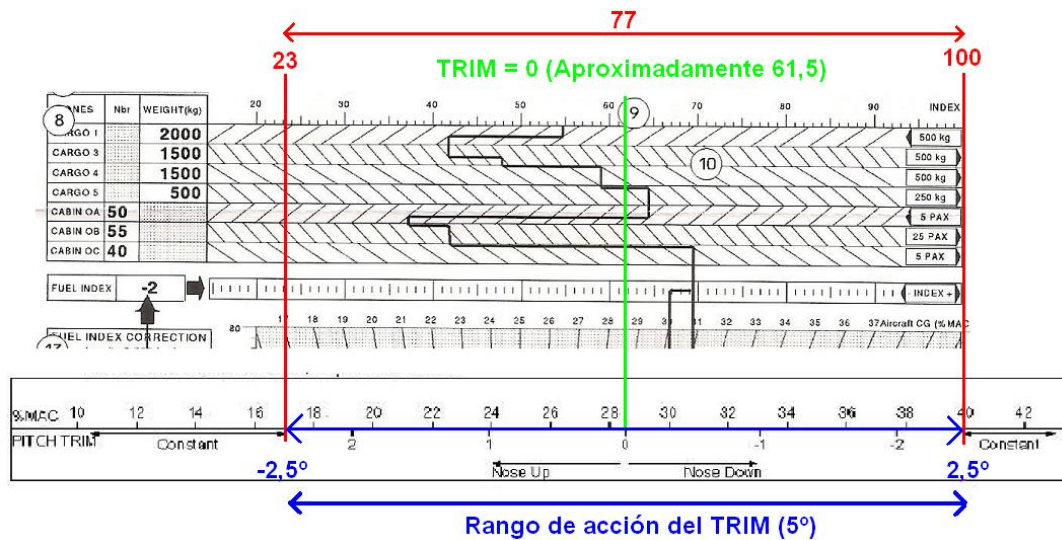


Fig.C.10 Relaciones entre los valores del índice DOW y el ángulo TRIM [38], [15]

Con todos estos datos ya podemos definir los dos últimos parámetros que nos quedan para poder calcular el ángulo TRIM (**Tabla C.3**).

CERO_TRIM	61.5
FACTOR_TRIM	5.0/77

Tabla C.3 Parámetros CERO_TRIM y FACTOR_TRIM

A partir de este punto el cálculo del ángulo de TRIM es un proceso sencillo en el que sólo es necesario aplicar las dos fórmulas siguientes (**ec.C.1** y **ec.C.2**) en las que utilizamos todos los parámetros antes descritos.

$$\begin{aligned}
 \text{DOW} = & \text{INDICE_DOW} + \\
 & \text{pantry_e} * \text{PANTRY_E} + \\
 & \text{pantry_f} * \text{PANTRY_F} + \\
 & b1 * \text{FACTOR_CARGO1} + \\
 & b3 * \text{FACTOR_CARGO3} + \\
 & b4 * \text{FACTOR_CARGO4} + \\
 & b5 * \text{FACTOR_CARGO5} + \\
 & \text{pasajeros_A} * \text{FACTOR_CABIN_0A} + \\
 & \text{pasajeros_B} * \text{FACTOR_CABIN_0B} + \\
 & \text{pasajeros_C} * \text{FACTOR_CABIN_0C} + \\
 & \text{fuel} * \text{FACTOR_FUEL}.
 \end{aligned}
 \tag{ec.C.1}$$

Donde los parámetros *pantry_e*, *pantry_f*, *b1*, *b3*, *b4*, *b5*, *pasajeros_A*, *pasajeros_B*, *pasajeros_C* y *fuel* hacen referencia al peso de introducido en las respectivas zonas

$$\text{Ángulo de TRIM} = (\text{DOW} - \text{CERO_TRIM}) * \text{FACTOR_TRIM}
 \tag{ec.C.2}$$

C.4. Número de combinaciones posibles al introducir los paquetes en contenedores.

Antes de nada es importante saber que para realizar este cálculo es necesario hacer algunas suposiciones que no tienen por qué cumplirse en todos los casos. La idea es simplificar al máximo el número de combinaciones posibles para ver si de esta forma resultaría factible un ataque directo (probándolas una detrás de otra) o si ni aun reduciéndolas al máximo es aceptable el proceso.

Así pues, como hemos dicho en múltiples ocasiones la mercancía (maletas, paquetes, etc.) viaja dentro de contenedores y estos a su vez en bodegas. En cambio para calcular la posición del c.g. del aparato sólo se tiene en cuenta el peso de la bodega al completo de forma que la posición interna de los bultos no tiene importancia (ver apartado Anexos - C.2). Esto elimina infinidad de permutaciones que representan una misma solución. Partiendo de esta base, el cálculo varía y pasamos de las n permutaciones de las **ecuaciones C.3 y C.4**, a un número más reducido de combinaciones que aparece en la **ecuación C.5**.

$$Soluciones_{sin\ mercancía\ en\ tierra} = (C)^B \quad (ec. C. 3)$$

$$Soluciones_{con\ mercancía\ en\ tierra} = (C + 1)^B \quad (ec. C. 4)$$

$$Soluciones' = \binom{B}{3m} x \binom{B-3m}{2m} x \binom{B-5m}{2m} x \binom{B-(7m)}{m'} x \binom{B-(7m+m')}{m''} \quad (ec. C. 5)$$

Donde

- C es el número total de contenedores (incluyendo la bodega 5 que vendría a ser otro contenedor pero con un peso y volumen especial).
- B es el número total de bultos.
- m representa la capacidad (en número de bultos) de un contenedor AKH.
- m' sería la capacidad de la bodega 5.
- m'' la capacidad del contenedor virtual que representa la posibilidad de dejar la mercancía en tierra.

Observando la **ecuación C.5** vemos que está formada por seis factores. Cada uno de ellos hace referencia al número de combinaciones (sin repetición) que se pueden realizar en cada una de las bodegas del aparato exceptuando el último factor que hace referencia a la mercancía que se queda en tierra.

Llegados a este punto sería cuestión de imaginar un caso práctico para poder comparar los resultados. Prosigamos con el mismo ejemplo descrito en el cuerpo del trabajo y supongamos que trabajamos con un vuelo en el que hay 126 pasajeros y por tanto 126 maletas ($B=126$). Además imaginemos un caso

muy sencillo en el que ninguna maleta se deja en tierra ($m'' = 0$). Es decir que el último factor no aparecerá y $B - (7m) = m'$.

Por último definiremos $m = 15$ de forma que en cada contenedor AKH colocaremos 15 maletas, y dejaremos el resto (21 maletas) para la bodega número 5 ($m' = 36$).

Aplicando todo lo dicho anteriormente ya podemos ofrecer un resultado a cada una de las posibilidades y comparar los resultados **(ec.C.6)** y **(ec.C.7)**.

$$\text{Soluciones} = (C)^B = 8^{126} \approx 6,16 \times 10^{113} \quad (\text{ec. C. 6})$$

$$\text{Soluciones}' = \binom{126}{45} \times \binom{81}{30} \times \binom{51}{30} \times \binom{21}{21} =$$

$$\frac{126!}{126!(126-45)!} \times \frac{81!}{81!(81-30)!} \times \frac{51!}{51!(51-30)!} \times \frac{21!}{21!(21-21)!} \approx$$

$$3,42 \times 10^{34} \times 1,41 \times 10^{22} \times 1,14 \times 10^{14} \times 1 \approx 5,5 \times 10^{70} \text{ soluciones} \quad (\text{ec. C. 7})$$

Como podemos ver el número de posibles soluciones se ha reducido a la mitad, pero aun así continua siendo un número enorme. Todo esto teniendo en cuenta que se trata de un caso de los más sencillos: con pocos bultos, sin posibilidad de dejar mercancía en tierra y suponiendo fijo el número de maletas que colocaremos en cada contenedor.

Es evidente que este método directo no resulta viable en ninguno caso y es necesario probar algún otro método alternativo para poder atacar este tipo de problemas

C.5. Cálculo de las diferentes combinaciones posibles al introducir los contenedores en las bodegas.

Como hemos dicho con anterioridad la carga del Airbus A320 se encuentra distribuida en las bodegas del mismo siguiendo el esquema de la **figura C.11**. De esta forma, 3 contenedores AKH viajarán en la bodega 1, 2 en la bodega 3 y los dos últimos lo harán en la bodega 4. Por último quedaría la bodega número 5 donde la mercancía viajara sujeta por otros medios sin utilizar contenedor. Este último compartimento se considera a efectos virtuales como un contenedor más, pero al tener un tamaño diferente no se puede cambiar de posición de manera que no participa en este pequeño baile de combinaciones.

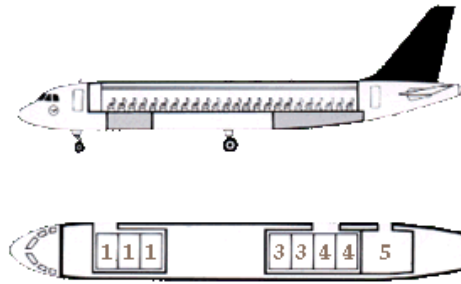


Fig.C.11 Distribución de los contenedores AKH dentro de las bodegas

Además de esto, para acabar de comprender la utilidad de este cálculo, es necesario recordar que para determinar la posición del centro de gravedad del aparato se tienen en cuenta la contribución de todas y cada una de las masas que viajan dentro del mismo, aunque de forma aproximada.

Así pues, se tiene en cuenta el peso de las bodegas, pero no el de los contenedores por separado, y por tanto el orden de los contenedores dentro de las mismas no es importante.

De no ser así, el total de posibles soluciones sería el número de permutaciones (sin repetición) que se pueden hacer con los 7 contenedores sin repetir ninguno **(ec.C.8)**.

$$\text{Combinaciones} = \frac{n!}{(n-r)!} = \frac{7!}{(7-7)!} = 7! = 5040 \quad (\text{ec. C. 8})$$

Teniendo en cuenta lo antes explicado el número de combinaciones se reduce. Se pueden eliminar un gran número de soluciones porque a efectos prácticos, representan la misma distribución de contenedores. De esta forma, el cálculo de las soluciones se compone de las combinaciones (sin repetición) que se pueden dar en la primera bodega, por las que se puedan dar en la tercera bodega y por las de la cuarta bodega (teniendo en cuenta que cada vez se van reduciendo el número de contenedores disponibles) **(ec.C.9)**.

$$\begin{aligned} \text{Combinaciones}' &= \binom{7}{3} \times \binom{4}{2} \times \binom{2}{2} = \\ &= \frac{7!}{3!(7-3)!} \times \frac{4!}{2!(4-2)!} \times \frac{2!}{2!(2-2)!} = 210 \quad (\text{ec. C. 9}) \end{aligned}$$

Como podemos ver el número de combinaciones se ha reducido considerablemente una vez eliminamos las repeticiones. Ahora sería momento de determinar cuáles son estas combinaciones que como hemos comentado

con anterioridad se ha de realizar de forma manual. A continuación podemos ver un listado en forma de tabla los 210 resultados (**Tabla C.4**).

1234567	1342567	1462537	2341567	2461537	3451267	3674512
1234657	1342657	1462357	2341657	2461357	3451627	3674152
1234765	1342765	1462735	2341765	2461735	3451762	3674215
1235647	1345627	1465327	2345617	2465317	3452617	3675142
1235746	1345726	1465723	2345716	2465713	3452716	3675241
1236745	1346725	1463725	2346715	2463715	3456712	3671245
1243567	1354267	1472563	2354167	2471563	3461527	4561237
1243657	1354627	1472653	2354617	2471653	3461257	4561327
1243765	1354762	1472365	2354761	2471365	3461725	4561732
1245637	1352647	1475623	2351647	2475613	3465217	4562317
1245736	1352746	1475326	2351746	2475316	3465712	4562713
1246735	1356742	1476325	2356741	2476315	3462715	4563712
1254367	1364527	1564237	2364517	2564137	3471562	4571263
1254637	1364257	1564327	2364157	2564317	3471652	4571623
1254763	1364725	1564732	2364715	2564731	3471265	4571362
1253647	1365247	1562347	2365147	2561347	3475612	4572613
1253746	1365742	1562743	2365741	2561743	3475216	4572316
1256743	1362745	1563742	2361745	2563741	3476215	4576312
1264537	1374562	1574263	2374561	2574163	3564127	4671523
1264357	1374652	1574623	2374651	2574613	3564217	4671253
1264735	1374265	1574362	2374165	2574361	3564721	4671325
1265347	1375642	1572643	2375641	2571643	3561247	4675213
1265743	1375246	1572346	2375146	2571346	3561742	4675312
1263745	1376245	1576342	2376145	2576341	3562741	4672315
1274563	1452367	1674523	2451367	2674513	3574162	5674123
1274653	1452637	1674253	2451637	2674153	3574612	5674213
1274365	1452763	1674325	2451763	2674315	3574261	5674321
1275643	1453627	1675243	2453617	2675143	3571642	5671243
1275346	1453726	1675342	2453716	2675341	3571246	5671342
1276345	1456723	1672345	2456713	2671345	3576241	5672341

Tabla C.4 Tabla con las 210 combinaciones diferentes que se pueden formar con los 7 contenedores AKH dentro de las bodegas del Airbus A320

ANEXO D: CODIFICACIONES

D.1. Análisis de la primera codificación utilizada proveniente del *Kanpur Genetic Algorithm's Laboratory*

El código en sí pertenece a la colección de códigos libres disponibles en el *Kanpur Genetic Algorithms Laboratory* de la India y cuyo director es Dr. Kalyanmoy Deb. El código fuente es libre y puede descargarse de forma gratuita de su página web¹¹. En el esquema siguiente (**Fig.D.1**) se puede ver, muy simplificado, el proceso que sigue el código para completar su búsqueda de soluciones.

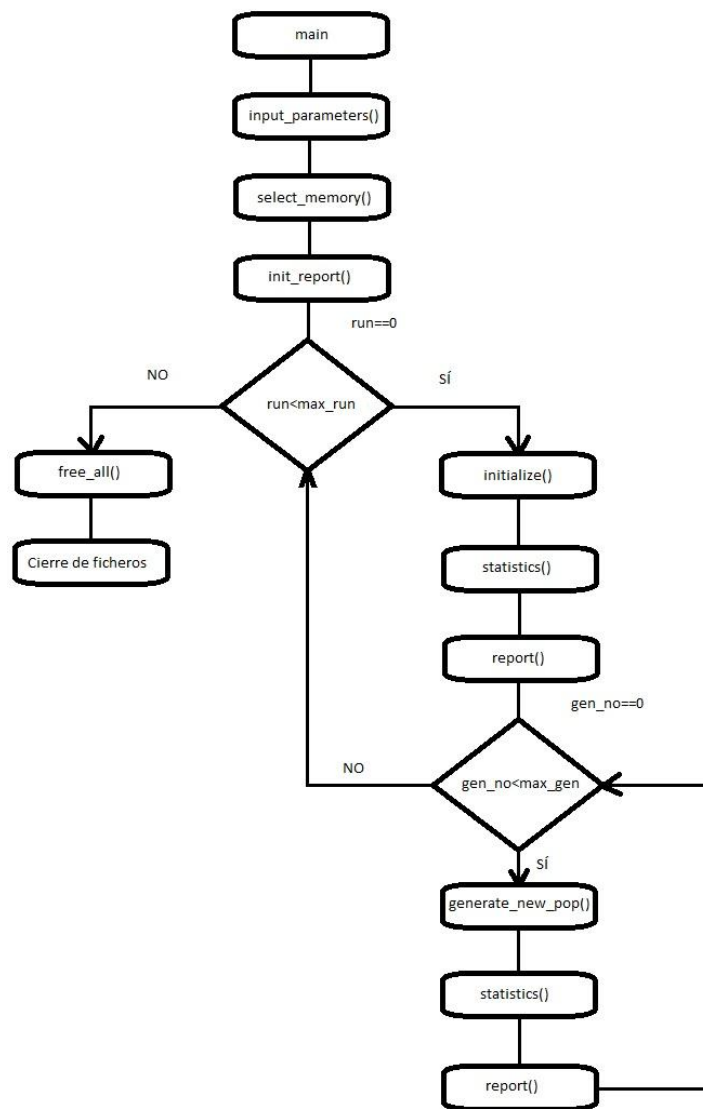


Fig.D.1 Diagrama de bloques con el esquema de funcionamiento del AG inicial

¹¹ <http://www.iitk.ac.in/kangal/index.shtml>

A continuación se puede ver un análisis más detallado de las funciones más importantes utilizadas durante el proceso.

D.1.1. *Input_parameters()*

El código permite introducir una serie de parámetros al inicio del programa que son los siguientes:

- Número de generaciones (*max_gen*)
- Tamaño de la población (*pop_size*)
- Número de variables codificadas en binario (*nvar_bin*)
 - o Longitud del gen/variable binaria (*chr_len*)
 - o Valor máximo que puede alcanzar dicho gen/variable (*xbin_upper*)
 - o Valor mínimo que puede alcanzar dicho gen/variable (*xbin_lower*)

NOTA: Esto para cada una de las variables binarias que hay.

- Número de variables codificadas con números reales (*nvar_real*)
 - o Valor máximo que puede alcanzar dicho gen/variable (*xreal_upper*)
 - o Valor mínimo que puede alcanzar dicho gen/variable (*xreal_lower*)

NOTA: Esto para cada una de las variables binarias que hay.

- Para las variables reales es necesario saber si se aceptan valores que excedan el máximo y mínimo o si por el contrario son límites firmes. (*boolean RIGID*)
- Utilizando *Sharing* o no (*boolean SHARING*)
 - o En caso de respuesta afirmativa es necesario definir el tamaño del nicho (*sigma_share*)
- Si deseamos que se impriman reportes intermedios de la actividad. (*boolean REPORT*)
- Cuántas veces se repetirá el proceso con todas sus generaciones (*maxrun*)
- Probabilidad de cruce (entre 0 y 1): (*p_xover*)
- Probabilidad de mutación:

- Para variables binarias (*p_mutation_bin*)
- Para variables reales (*p_mutation_real*)
- Si existen variables reales es necesario introducir los parámetros para la SBX que es una distribución de probabilidad que se utiliza para determinar la mutación en caso de variables reales.
 - Índice SBX: (*n_distribution_c*)
 - Mutación SBX: (*n_distribution_m*)
- Por último es necesario introducir un número real entre 0 y 1.0 que será el generador de los otros números aleatorios necesarios para el proceso de creación de la población inicial.
 - Seed inicial: (*basic_seed*)

En este caso, parámetros importantes como el tamaño del torneo o si se desea optimizar mediante maximización o minimización no son definibles mediante el menú.

- Tamaño del torneo: *tourneysize*
- Maximizar o minimizar: MINM (1 minimizar, -1 maximizar)

D.1.2. *Select_memory()*

Una función muy simple encargada de comprobar si el tamaño de la población es suficiente para realizar los torneos.

D.1.3. *init_report()*

Función simple que imprime en un fichero los datos introducidos por teclado dejando definido así el proceso que se va a realizar.

D.1.4. *initialize()*

Esta función reserva la memoria necesaria en función de los parámetros introducidos por teclado y pone a cero todos los parámetros.

D.1.5. *statistics()*

A grandes rasgos esta función realiza las funciones siguientes:

- Decodifica los cromosomas binarios si los hubiera.
- Calcula la función objetivo para todos los individuos.
- Selecciona el mejor individuo de la población en función de dos parámetros: la penalización y la función objetivo. En primer lugar selecciona como mejor individuo el que tenga menos penalización (siempre que ésta sea positiva). En caso de ser cero o negativa, entre dos individuos escogerá como “mejor” el que tenga un valor de función objetivo mejor que el otro.
- Guardará el mejor individuo de esta generación y el mejor de todas las generaciones así como una suma global de los valores de la función objetivo de la generación en curso.

D.1.6. *report()*

Imprime los datos estadísticos de la generación en el fichero de datos de salida.

D.1.7. *generate_new_pop()*

Esta función es la encargada de crear la nueva generación de individuos a partir de los progenitores de la generación anterior. A grandes rasgos, podríamos dividir esta función en tres etapas diferentes: proceso de selección, cruce y mutación.

D.1.7.1. Proceso de selección

En esta parte del proceso se selecciona que individuos de la población serán los progenitores de la nueva población mediante el único método disponible: el torneo (por defecto el tamaño del torneo está fijado a 2 individuos). Para esto primero se crea una lista de enfrentamientos posibles mediante la función *preselect_tour()*. A continuación se hacen competir ambos individuos y se selecciona al ganador en función de los siguientes parámetros:

- *Sin SHARING:*

En este caso se utiliza la función *tour_select()* y se decide cuál de los dos es el elegido de la forma siguiente: En primer lugar se escogerá como vencedor al individuo que tenga una penalización menor. En caso de que ambos individuos tengan una penalización negativa o igual a cero, se escogerá como vencedor al individuo cuyo valor de función de *fitness* sea “mejor” (teniendo en cuenta que puede tratarse de una maximización o una minimización).

- *Con SHARING:*

En este caso se utilizará la función *tour_select_constr()*. El proceso es muy similar al anterior, sólo que varía en el momento de decidir qué individuo es el ganador en caso de que ambos tengan una penalización igual o menor que cero. En este caso pueden pasar dos cosas:

- a) Si están dentro del mismo nicho, se dará como vencedor el que tenga una función de *fitness* “mejor”.
- b) Si no están dentro del mismo nicho, se escogerá al azar un individuo que esté dentro de su mismo nicho y que tenga una penalización igual o menor que cero. Una vez encontrado, se le hará competir contra el mejor de los dos individuos iniciales y se dará como vencedor al que mejores valores de función de *fitness* tenga.

D.1.7.2. Proceso de cruce

Una vez se ha realizado dos veces el proceso de selección y tenemos a los dos progenitores ya podemos cruzarlos para obtener dos nuevos individuos. En primer lugar se ha de decidir si el cruce se realiza o no. Si no se realiza el cruce los progenitores pasan a formar parte de la nueva generación sin cambios (reproducción asexual). En caso de que se produzca el cruce (reproducción sexual), hay un proceso diferente si la codificación es binaria o real:

- a) Codificación binaria: En este caso se utiliza la función *binary_xover()*, que en definitiva lo que hace es escoger aleatoriamente un punto de cromosoma y cruzar ambos individuos dando lugar a dos nuevos.
- b) Codificación real: en este caso el proceso es un poco diferente. En primer lugar hay un 50% de posibilidades de que se produzca el cruce de esa variable (una probabilidad extra además de la anterior). En caso de producirse el proceso es similar al del caso binario, con la diferencia de que se utilizan los parámetros de la *distribución binaria (SBX)* para calcular los parámetros de los hijos resultantes (función *create_children()*).

D.1.7.3. Proceso de mutación:

Es la fase final para obtener la nueva generación de individuos. Un vez más el proceso en caso de variables binarias o reales es diferente: por un lado, las variables binarias (al tener limitadas el número de bits) tienen unos máximos y mínimos perfectamente acotados. La mutación de uno o varios bits no provocaría una salida de rango así que ese es un parámetro que no hace falta ser controlado. Así pues, la mutación binaria se realiza mediante la función *binmutation()*, que consiste en recorrer el cromosoma testeando cada bit, mediante la probabilidad de mutación, para construir una máscara de ceros

(bits no mutados) y unos (bits mutados). Una vez definida esta mascara se aplicará al cromosoma, de manera que pueden cambiar varios bits de una variable por cada generación, de forma que los cambios pueden ser más significativos de lo esperado.

Por otra parte, en lo que a variables reales se refiere, estas sí que pueden salirse del rango definido, y de hecho al principio del código se da la opción de escoger si las variables deben estar perfectamente acotadas o si se permite exceder los límites (parámetro RIGID). En ambos casos el proceso es similar: consiste básicamente en generar aleatoriamente unas *deltas* de desplazamiento a partir de un punto central que sería el valor de la variable.

En caso de tratarse de valores acotados, se ha de controlar que estas deltas no excedan los rangos definidos. En el otro caso, las *deltas* suponen un rango completo hacia arriba y hacia abajo. Una vez definidos los posibles márgenes, se procede a buscar aleatoriamente un valor comprendido entre esos márgenes que será el que adoptará la variable del individuo.

Una vez concluido todo este proceso ya disponemos de una *nueva* generación de individuos que conserva, de la antigua, aquellos individuos que no han sido cruzados ni han mutado después. Eso supone una minúscula parte si hemos definido una probabilidad de cruce alta como suele hacerse, de manera que debería ser tenido en consideración.

D.1.8. *free_all()*

Esta función sencillamente libera toda la memoria reservada.

D.2. Codificación final para la aplicación práctica

```

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>

#define TRUE 1
#define FALSE 0

//Existen 210 formas diferentes de disponer los 7 contenedores dentro de las bodegas 1 , 3 y 4
#define NUM_COMBINACIONES 210
#define MAX_BULTOS_POR_VUELO 650

/*MAX_BULTOS_CONTENEDOR es el valor maximo de los paquetes que caben en un contenedor y
de los paquetes que caben en la bodega...
- 3.5M3 ENTRE 0.05 (VOL MINIMO) == 70 BULTO
- 7.7M3 ENTRE 0.05M3 (VOL MINIMO) == 154 BULTOS
*/
#define MAX_PESO_POR_VUELO 8847.0
#define MAX_VOLUMEN_POR_VUELO 32.2
#define MAX_BULTOS_POR_CONTENEDOR 154
#define MAX_INVIDUOS_EN_POBLACION 200
#define VOL_MAXIMO_CONTENEDOR 3.5
#define VOL_MAXIMO_BODEGA_5 7.7
#define PESO_MAXIMO_CONTENEDOR 1050
#define PESO_MAXIMO_BODEGA_5 1497
#define NUM_CONTAINERS 8
#define VOL_MIN_BULTO 0.05
#define VOL_MAX_BULTO 0.15
#define PESO_MIN_BULTO 10
#define PESO_MAX_BULTO 45
#define MAX_CICLOS 1000
#define MAX_GENERACIONES 200

/*PARAMETROS PARA MALETAS (1 POR PASAJERO)
#define VOL_MIN_BULTO 0.05
#define VOL_MAX_BULTO 0.2
#define PESO_MIN_BULTO 10
#define PESO_MAX_BULTO 40
*/

/*PARAMETROS PARA 300 BULTOS
#define VOL_MIN_BULTO 0.05
#define VOL_MAX_BULTO 0.15
#define PESO_MIN_BULTO 10
#define PESO_MAX_BULTO 40
*/

/*PARAMETROS PROPIOS DEL AIRBUS A320
- PESO 42500kg
- CG 27 (%MAC)
- H-ARM 18.93 m (H-arm -18.8499) * W
INDEX-DOW= ----- + 50
1000
*/

#define INDICE_DOW 53.4 //DRY OPERATION WEIGHT INDEX
#define FACTOR_TRIM (5.0/77) // %MAC
#define CERO_TRIM 61.5
#define MAX_TRIM 100
#define MIN_TRIM 23
#define FACTOR_CARGO_1 (-3.16/500) //Valor por Kg de carga...

```

```

#define FACTOR_CARGO_3 (1.86/500) //Valor por Kg de carga...
#define FACTOR_CARGO_4 (3.69/500) //Valor por Kg de carga...
#define FACTOR_CARGO_5 (2.89/250) //Valor por Kg de carga...
#define FACTOR_CABIN_0A (-2.72/5) //Valor por pasajero...
#define FACTOR_CABIN_0B (2.08/25) //Valor por pasajero...
#define FACTOR_CABIN_0C (3.43/5) //Valor por pasajero...
#define PANTRY_E -1.22/100 //Valores para cada 100Kg de mas que se carguen (Si es al revés se
cambia el signo)
#define PANTRY_F 1.43/100 //Valores para cada 100Kg de mas que se carguen (Si es al revés se
cambia el signo)
#define PESO_PASAJERO 84 //75 + EQUIPAJE DE MANO (NO SE DISTINGUE ENTRE NIÑOS y
ADULTOS)
#define MAX_PASAJEROS_ZONA 60
#define VOL_CARGO_5 7.7
#define PESO_CARGO_5 1497

/*****
***** ESTRUCTURAS PRINCIPALES *****/
*****/

struct paquete{
    int peso;
    double volumen;
};
typedef struct paquete BULTO;

struct container{
    int contenido [MAX_BULTOS_POR_VUELO];
    /*Indica la posición que ocupan los bultos dentro del listado del
    cargamento total del vuelo */
    int num_bultos;
    int peso_total;
    double vol_total;
    int completo;
    int correcto; //Si esta en FALSE significa que el volumen, peso y/o num_bultos es incorrecto...
};
typedef struct container CONTENEDOR;

struct combinacion{
    int distribucion[NUM_CONTAINERS-1];
};
typedef struct combinacion SECUENCIA;

struct solucion{
    int detalle [MAX_BULTOS_POR_VUELO];
    CONTENEDOR contenedores [NUM_CONTAINERS+1];
    SECUENCIA posicion; //Distribución dentro de las bodegas de los contenedores...
    double TRIM;
    double fitness;
    double vol_cargado;
    int peso_cargado;
};
typedef struct solucion INDIVIDUO;

typedef INDIVIDUO *POBLACION ;

/*****
***** VARIABLES GLOBALES *****/
*****/
CONTENEDOR contenedores [NUM_CONTAINERS];
BULTO cargamento [MAX_BULTOS_POR_VUELO];
SECUENCIA combinaciones[NUM_COMBINACIONES];
POBLACION actual, nueva, temporal;
INDIVIDUO mejor_absoluto, mejores_ciclo [MAX_CICLOS],
mejores_generacion[MAX_GENERACIONES];

```



```

int pasajeros_zona_A, //Pasajeros cabina A
    pasajeros_zona_B, //Pasajeros cabina B
    pasajeros_zona_C, //Pasajeros cabina C
CONTROL_BOOKING, /*Controla que no se produzcan situaciones imposibles como por
                    ejemplo...
                    1 Pasajero en la zona A
                    1 en la zona B
                    60 en la zona C

                    */
fuel[12] = {-2,-3,-4,-4,-5,-6,-7,-8,-8,-9,-10,-11}, //MIN = 13000 <==> MAX = 18500
peso_fuel, //Peso del carburante
pantry_e, //Peso de mas o de menos en la zona e
pantry_f, //Peso de mas o de menos en la zona f
FACTOR_FUEL, /*Factor de correccion (de los de arriba) que se aplica en
              funcion de la cantidad de fuel generado aleatoriamente*/
peso_total, //Peso total de todos los bultos..
num_individuos, //Numero de individuos de la poblacion...
num_bultos, //Numero de bultos que se intentan cargar...
num_ciclos, //Numero de ciclos que ejecutara el codigo
num_generaciones, //Numero de generaciones por ciclo
op_seleccion, //Metodo de seleccion usado (0)Torneo (1)Ruleta
op_codificacion, /*Codificacion utilizada...
(0)- Cada cromosoma esta presentado por una lista de enteros donde cada posicion
      guarda el numero de container al que pertenece.

(1)- Cada cromosoma esta representado por una lista de enteros donde las n primeras
      posiciones contienen los bultos del contenedor 0, las n siguientes las del contenedor
      1 y asi sucesivamente.*/

num_puntos_cruce, //Determina si se utilizar la funcion de cruce en uno o dos puntos...
op_envio, //Define si permite exceso de carga (1) o no (0)
OVERLOAD, //Excede el peso o volumen maximos...
PESO_OVERLOAD, //Peso de mas que se ha generado...
bultos_aux [MAX_BULTOS_POR_VUELO], //Vector auxiliar para generar las soluciones iniciales
lista_enfrentamientos[ MAX_INVIDUOS_EN_POBLACION*2], //Vector donde se defieniran los
                                                       enfrentamientos

pos_torneo, //posicion en la que se quedo la lista de torneos...
pos_ruleta, //Posicion en la que quedo la ruleta tras el ultimo giro...
ciclo_actual, //ciclo actual de ejecucion.
gen_actual, //Generacion actual del ciclo.
generacion_del_mejor, //Generacion en la que aparece el mejor individuo del ciclo...
REPARAR, /*Define si se reparan las soluciones imposibles que violan
          los parametros del vuelo...
          TRUE --> Se reparan, FALSE --> No se reparan.*/
IMPRIME_DATOS, //Si esta en TRUE se imprime el fichero de datos, si es
               FALSE no...

MAXIMIZAR_CARGA, //Si esta en TRUE utilizamos la funcion de carga_maxima...
PROBAR_COMBINACIONES; /*Si esta en TRUE la funcion de fitness evalua todas las posibles
                       Combinaciones de los contenedores dentro de las bodegas 1,
                       3 y 4 para encontrar la que ofrece mejores valores de fitness y
                       de penalizacion. En caso contrario utilizamos la combinaicon
                       básica ([1,2,3] [4,5] [6,7] [8]) y evaluamos la solucion...*/

LARGE_INTEGER t_inicial_ciclo, t_final_ciclo, t_inicial_evaluacion, t_final_evaluacion; /*Instantes de
tiempo iniciales y finales*/

double tiempo_ciclo, //Duaracion del ciclo
vol_total, //Volumen total de los bultos generados...
VOL_OVERLOAD, //Volumen de mas que se ha generado...
semilla, //Semilla inicial para el random del los primeros individuos...
p_cruce, //Probabilidad de cruce
p_mut, //Probabilidad de mutacion
fitness_global, //Suma de todos los fitness de la generacion (para seleccion por ruleta)...
ruleta[ MAX_INVIDUOS_EN_POBLACION]; //Ruleta de prioridades para decidir en la funcion de
ruleta...

```

```

FILE *fp_posiciones, /*Lee desde fichero las 210 combinaciones posibles y las carga en el vector
                     combinaciones*/
*fp_datos, /*Describe el proceso que se esta realizando...
*fp_mejores_generacion, /*Muestra los mejores individuos de cada generacion
*fp_mejores_ciclo, /*Muestra los mejores de cada ciclo
*fp_tiempos, /*Tiempos de cada generacion y de cada ciclo
*fp_excel; /*Resultados normalizados y demas para excel...

/*****
***** DEFINICIONES DE FUNCIONES *****/
*****/

void menu_principal();
void inicio();
void inicia_parametros();
void libera_memoria();
void abre_ficheros();
void cierra_ficheros();
void genera_semillas();
void inicia_soluciones();
void carga_combinaciones();
void genera_poblacion_inicial();
void genera_lista_inicial();
void genera_pasajeros();
void genera_fuel();
void genera_bultos(int op);
double genera_double();
int inserta_bulto(int bulto,int cont,INDIVIDUO *ind);
void genera_lista_enfrentamientos();
void calcula_estadisticas();
void copia_individuo(INDIVIDUO origen, INDIVIDUO *destino);
void genera_ruleta();
int seleccion_por_ruleta();
int seleccion_por_torneo();
void funcion_cruce_n_puntos(INDIVIDUO *hijo1, INDIVIDUO *hijo2, int padre1, int padre2,int op);
void funcion_cruce_por_contenedores(INDIVIDUO *ind1,INDIVIDUO *ind2,int p1, int p2, int op);
void funcion_de_mutacion(INDIVIDUO *ind);
void actualiza_contenedores(INDIVIDUO *ind);
void repara_soluciones(INDIVIDUO *ind);
void calcula_ocupacion(INDIVIDUO * ind);
void carga_maxima(INDIVIDUO *ind);
void genera_nueva_poblacion();
double calcula_tiempo(LARGE_INTEGER *fin, LARGE_INTEGER *init);
void ciclo();
void funcion_fitness();
void calcula_TRIM(INDIVIDUO *ind, int opcion);
void imprime_datos_iniciales(FILE *fp_destino);
void imprime_generacion();
void imprime_tiempo_ciclo();
void imprime_mejor_ciclo(FILE *fp_destino);
void imprime_mejor_generacion(FILE *fp_destino);
void imprime_resultados_normalizados();

/*****
*****
*****/

//Funcion para introducir parametros principales...
void menu_principal(){
    /*Cuantos individuos?
    num_individuos = 100;
    /*Cuantos bultos?
    num_bultos = 20;
    /*Que codificacion usas (0) Normal -- (1) Por contenedores
    op_codificacion = 0;
    /*Permites que los bultos totales escedan el maximo (Algunos no se cargaran)

```

```

// (0)--> SI || (1)--> NO
op_envio = 0;
//Metodo de seleccion usado? (0)--> Torneo (1)--> Ruleta
op_seleccion = 0;
//Si es cruce--> Cruce en un punto o en dos?
num_puntos_cruce = 1;
//Probamos todas las combinaciones posibles... (TRUE = si -- FALSE=no)
PROBAR_COMBINACIONES = TRUE;
//Reparar las soluciones inaceptables...
REPARAR = TRUE;
//Cuantos ciclos?
num_ciclos = 5;
//Cuantas generaciones?
num_generaciones = 100;
//P. Cruce?
p_cruce = 0.9;
//P. Mutacion?
p_mut = 0.1;
/*El fichero de datos se hace enorme cuando hay muchos individuos o bultos,
con esta opcion podemos imprimir este archivo o no*/
IMPRIME_DATOS = FALSE;
/*Si esta en TRUE utilizamos la funcion de carga_maxima dada para llenar el avion...*/
MAXIMIZAR_CARGA = FALSE;
/*Controlamos la posicion en la que se sientan los pasajeros (TRUE) o dejamos que se sienten
donde quieran*/
CONTROL_BOOKING = FALSE;
}

//Funcion primaria que reserva la memoria para las poblaciones, abre los ficheros y genera los bultos...
void inicio(){
    //Reservar memoria...
    actual = (INDIVIDUO *)malloc(num_individuos*sizeof(INDIVIDUO));
    nueva = (INDIVIDUO *)malloc(num_individuos*sizeof(INDIVIDUO));
    temporal = (INDIVIDUO *)malloc(num_individuos*sizeof(INDIVIDUO));

    //Abre los ficheros de escritura...
    abre_ficheros();
    //Iniciando semillas para las funciones rand()
    genera_semillas();
    //Carga la lista de combinaciones para los contenedores (Funcion de fitness)
    carga_combinaciones();
    //Genera los pasajeros del vuelo...
    genera_pasajeros();
    //Genera la cantidad de carburante...
    genera_fuel();
    //Generando cargamento...
    genera_bultos(op_envio);
    //Imprimimos los datos descriptivos del proceso
    imprime_datos_iniciales(fp_datos);
}

//Funcion para inicializar todos los datos y reservar la memoria
void inicia_parametros(){
    //Iniciando parametros
    OVERLOAD = FALSE;
    gen_actual = 0;
    pos_ruleta = 0;
    pos_torneo = 0;
    pantry_e=0;
    pantry_f=0;
    //Iniciando contenedores...
    inicia_soluciones();
    //Generando poblacion inicial
    genera_poblacion_inicial();
    //Evaluamos la poblacion para obtener los primeros datos
    funcion_fitness();
    //Calculamos las estadísticas...

```

```

    calcula_estadisticas();
    //Imprimimos la primera generacion...
    if(IMPRIME_DATOS)
        imprime_generacion();
    //Imprimimos el mejor de la generacion 0..
    imprime_mejor_generacion(fp_mejores_generacion);
}

//Funcion que libera la memoria anteriormente reservada...
void libera_memoria(){
    free(actual);
    free(nueva);
}

//Funcion que abre los ficheros...
void abre_ficheros(){
    fopen_s(&fp_datos, "Datos.txt", "w+");
    fopen_s(&fp_mejores_generacion, "MejoresGeneracion.txt", "w+");
    fopen_s(&fp_mejores_ciclo, "MejoresCiclo.txt", "w+");
    fopen_s(&fp_excel, "Excel.txt", "w+");
    fopen_s(&fp_tiempos, "Tiempos.txt", "w+");
    fopen_s(&fp_posiciones, "Combinaciones.txt", "r");
}

//Funcion que cierra los ficheros...
void cierra_ficheros(){
    fclose(fp_datos);
    fclose(fp_mejores_generacion);
    fclose(fp_mejores_ciclo);
    fclose(fp_excel);
    fclose(fp_tiempos);
}

/*Introduce las 210 combinaciones posibles de los contenedores, de momento a mano,
no se me ocurre como hacerlo de otra forma sin hacerlo muy complicado o repetir
combinaciones.
- Sin hacer esto y repetir secuencias que no aportan nada (Ej; 123 en bodega 1 es igual que
132, 231 y 213) en lugar de 210 serian 1260 posibilidades*/
void carga_combinaciones(){
    int i=0;

    for(i=0; i<NUM_COMBINACIONES;i++){
        fscanf_s(fp_posiciones, "%d %d %d %d %d %d %d",
            &combinaciones[i].distribucion[0],
            &combinaciones[i].distribucion[1],
            &combinaciones[i].distribucion[2],
            &combinaciones[i].distribucion[3],
            &combinaciones[i].distribucion[4],
            &combinaciones[i].distribucion[5],
            &combinaciones[i].distribucion[6]);
    }
    fclose(fp_posiciones);
}

//Funcion que genera una semilla aleatoria entre 0 y 1 en funcion del instante de tiempo
//para que la funcion rand() de valores diferentes cada vez.
void genera_semillas(){
    srand((long)time(NULL));
    semilla = ((double)1)/((double)(rand()%100));
}

void inicia_soluciones(){
    int i=0,j=0;

```

```

for(i=0;i<num_individuos;i++){
    actual[i].fitness = 0.0;
    actual[i].TRIM = 2.5; //Ponemos el peor posible...
    actual[i].peso_cargado = 0;
    actual[i].vol_cargado = 0.0;

    for(j=0;j<NUM_CONTAINERS+1;j++){
        actual[i].contenedores[j].completo = FALSE;
        actual[i].contenedores[j].correcto = TRUE;
        actual[i].contenedores[j].num_bultos = 0;
        actual[i].contenedores[j].peso_total = 0;
        actual[i].contenedores[j].vol_total = 0.0;
    }
    //Iniciando lista de mejores...
    mejor_absoluto.fitness = 0.0;
    generacion_del_mejor = 0;

    for(i=0;i<num_generaciones;i++){
        mejores_generacion[i].fitness = 0.0;
    }
    for(i=0;i<num_ciclos;i++){
        mejores_ciclo[i].fitness = 0.0;
    }
}

//Elige aleatoriamente el numero de pasajeros de cada zona...
void genera_pasajeros (){
    pasajeros_zona_C = rand()%MAX_PASAJEROS_ZONA;
    pasajeros_zona_B = rand()%MAX_PASAJEROS_ZONA;
    if(CONTROL_BOOKING){
        pasajeros_zona_A = pasajeros_zona_C +(rand()%10);
        if(pasajeros_zona_A>MAX_PASAJEROS_ZONA)
            pasajeros_zona_A = MAX_PASAJEROS_ZONA;
    }
    else
        pasajeros_zona_A = rand()%MAX_PASAJEROS_ZONA;
}

//Elige aleatoriamente la cantidad de fuel y el consiguiente factor de correccion...
void genera_fuel(){
    int aux=0;

    aux = rand()%12;

    FACTOR_FUEL = fuel[aux];
    peso_fuel = (26 + aux) * 500;
}

/*Funcion que genera n bultos aleatoriamente por si no quieres meterlos por teclado
Ofrece dos posibilidades:
A) Generar n bultos sin tener en cuenta si el peso o volumen exceden lo permitido
B) Generar n bultos comprobando que no se superen los maximos establecidos*/
void genera_bultos(op){
    int i=0, peso_libre=0;
    double margen_peso=0.0, margen_vol=0.0, vol_libre=0.0, num_peso=0.0, num_vol=0.0;

    BULTO aux;

    vol_total = 0.0;
    peso_total = 0;

    vol_libre = ((NUM_CONTAINERS-1) *

```

```

VOL_MAXIMO_CONTENEDOR)+VOL_MAXIMO_BODEGA_5;
peso_libre = ((NUM_CONTAINERS-1) *
              PESO_MAXIMO_CONTENEDOR)+PESO_MAXIMO_BODEGA_5;

margen_peso = PESO_MAX_BULTO - PESO_MIN_BULTO;
margen_vol = VOL_MAX_BULTO - VOL_MIN_BULTO;

for(i=0;i<num_bultos;i++){
    num_peso = genera_double();
    num_vol = genera_double();

    aux.peso = (int)(num_peso*margen_peso)+PESO_MIN_BULTO;
    aux.volumen = (num_vol*margen_vol)+VOL_MIN_BULTO;

    vol_libre = vol_libre - aux.volumen;
    peso_libre = peso_libre - aux.peso;

    if ((op==1)&&((vol_libre<0)||((peso_libre<0)))){
        num_bultos = i-1;
        vol_libre = vol_libre + aux.volumen;
        peso_libre = peso_libre + aux.peso;
        printf("ATENCION!! Para evitar OVERLOAD los bultos se reducen a %d\n",
               num_bultos);
    }else{
        cargamento[i] = aux;
        vol_total += aux.volumen;
        peso_total += aux.peso;
    }

    if(vol_libre<0 || peso_libre<0)
        OVERLOAD = TRUE;
}
//En caso de que se exceda el peso o volumen se informa y se evalua
if(OVERLOAD){
    if(peso_libre<0)
        PESO_OVERLOAD = (peso_libre * (-1));
    else
        PESO_OVERLOAD = 0;

    if(vol_libre<0)
        VOL_OVERLOAD = (double)(vol_libre * (-1));
    else
        VOL_OVERLOAD = 0.0;

    printf("OVERLOAD!!\n");

    printf("El volumen excede en %f y el peso en %d\n",
           VOL_OVERLOAD,PESO_OVERLOAD);
}
}

//Genera un numero decimal aleatorio entre 0 y 1 --> Se usa en genera_bultos()
double genera_double(){
    double num=0.0;
    num = sin(rand()%100);
    if (num <0)
        num = num +1;

    return num;
}

/*Genera la poblacion inicial de individuos:
   Selecciona un contenedor y lo va llenando cogiendo paquetes al azar hasta que se llenen
   todos. En ese punto, los paquetes que queden iran destinados al contenedor 0 que
   engloba los paquetes que no se cargan*/
void genera_poblacion_inicial(){
    int ind=0, contenedor=0, bulto=0, bultos_libres=0, contenedores_libres=0, intentos=0;

```

```

for(ind =0; ind<num_individuos;ind++){ //Para cada individuo...
    genera_lista_inicial();
    bultos_libres = num_bultos; //Cada vez que encajamos un bulto restamos uno
    contenedores_libres = NUM_CONTAINERS;
    contenedor = 0;
    intentos = 0;
    while((bultos_libres!=0)){
        //Elegimos un contenedor distinto del 0 ...
        contenedor = (contenedor +1)% (NUM_CONTAINERS+1);
        if (contenedor==0)
            contenedor++;
        //Elegimos un bulto al azar...
        bulto = rand() % bultos_libres;

        /*Probamos de colocarlo en cualquiera de los contenedores y si no cabe en
        Niguno lo ponemos en el contenedor 0*/

        if(inserta_bulto(bultos_aux[bulto],contenedor,&actual[ind])){
            bultos_aux[bulto] = bultos_aux[bultos_libres-1];
            bultos_libres--;
            intentos =0;
        }
        else{
            intentos++;
            if(intentos==NUM_CONTAINERS){
                inserta_bulto(bultos_aux[bulto],0,&actual[ind]);
                bultos_aux[bulto] = bultos_aux[bultos_libres-1];
                bultos_libres--;
                intentos =0;
            }
        }
        calcula_ocupacion(&actual[ind]);
    }
}

/*Funcion que genera una lista auxiliar que representa los paquetes y que luego se
desorganizara para dar lugar a las soluciones iniciales de los individuos...*/
void genera_lista_inicial(){
    int i=0;

    for(i=0;i<num_bultos;i++)
        bultos_aux[i]=i;
}

/*Inserta un bulto en un determinado contenedor:
NOTA: la funcion devuelve 1 si se ha insertado correctamente y 0 si no es posible
El contenedor 0 no tiene limite de capacidad...
*/
int inserta_bulto(int bulto,int cont,INDIVIDUO *ind){
    int peso = 0, PESO_LIMITE =0;
    double vol = 0.0, VOL_LIMITE=0.0;

    /*if(cont>8)
        printf("Ciclo %d - Genracion %d - Contenedor %d",ciclo_actual,gen_actual,cont);*/

    if(cont==0){ //No hace falta comprobar el peso ni volumen, no tiene limite
        ind->detalle[bulto] = cont;
        ind->contenedores[cont].contenido[ind->contenedores[cont].num_bultos] = bulto;
        ind->contenedores[cont].num_bultos++;
        ind->contenedores[cont].peso_total+=cargamento[bulto].peso;
        ind->contenedores[cont].vol_total+=cargamento[bulto].volumen;
        return 1;
    }
    else{//Comprobamos valores, si cabe se introduce, si no pues no...

```

```

        if(cont==8){ //El contenedor 8 representa la bodega de carga 5 y tiene un volumen y
                    peso maximo diferentes
                    VOL_LIMITE = VOL_MAXIMO_BODEGA_5;
                    PESO_LIMITE = PESO_MAXIMO_BODEGA_5;
                }
                else{
                    VOL_LIMITE = VOL_MAXIMO_CONTENEDOR;
                    PESO_LIMITE = PESO_MAXIMO_CONTENEDOR;
                }
                if(!ind->contenedores[cont].completo){//si no esta completo..

                    peso = (ind->contenedores[cont].peso_total + cargamento[bulto].peso);
                    vol = (ind->contenedores[cont].vol_total + cargamento[bulto].volumen);

                    if((peso<PESO_LIMITE)&&(vol<VOL_LIMITE)){//Si cabe lo ponemos dentro...
                        ind->detalle[bulto] = cont;
                        ind->contenedores[cont].contenido[ind-
>contenedores[cont].num_bultos] = bulto;
                        ind->contenedores[cont].num_bultos++;
                        ind->contenedores[cont].peso_total+=cargamento[bulto].peso;
                        ind->contenedores[cont].vol_total+=cargamento[bulto].volumen;

                        if((ind-
>contenedores[cont].num_bultos==MAX_BULTOS_POR_CONTENEDOR)||
                            ((peso+PESO_MIN_BULTO)>PESO_LIMITE)||
                            ((vol+VOL_MIN_BULTO)>VOL_LIMITE))
                            ind->contenedores[cont].completo = TRUE;

                        return 1;
                    }
                }
            }
            return 0;
        }
    }
}

```

//Busca en la generacion los mejores individuos a partir de su fitness
 //Ademas calcula el fitness global para la funcion de seleccion por ruleta...

```

void calcula_estadisticas(){
    int pos=0, best=0;
    double fit_best=0.0,fit=0.0,pen_best=0.0,pen=0.0;

```

```

    fit_best = actual[0].fitness;
    best = 0;
    fitness_global=fit_best;

```

```

    for(pos=1;pos<num_individuos;pos++){
        calcula_ocupacion(&actual[pos]);

        fit = actual[pos].fitness;

        fitness_global+=fit;

        if(fit_best<fit){//Aqui gana el que mejor fitness tenga...
            best = pos;
            fit_best = actual[pos].fitness;
        }
    }
}

```

//Ahora solo es cuestion de actualizar la lista de los mejores individuos y el mejor global...

//Primero el mejor de la geneacion
 copia_individuo(actual[best],&mejores_generacion[gen_actual]);

//Ahora el mejor absoluto...


```

        if(gen_actual == 0){
            copia_individuo(actual[best], &mejor_absoluto);
        }
        else{
            if(actual[best].fitness > mejor_absoluto.fitness){
                copia_individuo(actual[best], &mejor_absoluto);
                generacion_del_mejor = gen_actual;
            }
        }

        /*Si el metodo de seleccion elegido sea la ruleta es necesario construirla,
        sino hara falta construir la lista de enfrentamientos para el torneo...
        (0)- Torneo (1)- Ruleta*/
        if(op_seleccion == 1){
            genera_ruleta();
            pos_ruleta = 0;
        }
        else{
            genera_lista_enfrentamientos();
            pos_torneo = 0;
        }
    }

    //Copia el individuo origen en el de destino....
    void copia_individuo(INDIVIDUO origen, INDIVIDUO *destino){
        int cont=0, pos=0;

        destino->fitness = origen.fitness;
        destino->TRIM = origen.TRIM;
        destino->peso_cargado = origen.peso_cargado;
        destino->vol_cargado = origen.vol_cargado;

        for(cont=0; cont<NUM_CONTAINERS+1; cont++){
            destino->contenedores[cont].completo = origen.contenedores[cont].completo;
            destino->contenedores[cont].correcto = origen.contenedores[cont].correcto;
            destino->contenedores[cont].num_bultos = origen.contenedores[cont].num_bultos;
            destino->contenedores[cont].peso_total = origen.contenedores[cont].peso_total;
            destino->contenedores[cont].vol_total = origen.contenedores[cont].vol_total;
            for(pos=0; pos<origen.contenedores[cont].num_bultos; pos++){
                destino->contenedores[cont].contenido[pos] =
                    origen.contenedores[cont].contenido[pos];
            }
        }

        for(pos=0; pos<NUM_CONTAINERS-1; pos++){
            destino->posicion.distribucion[pos] = origen.posicion.distribucion[pos];
        }

        for(pos=0; pos<num_bultos; pos++){
            destino->detalle[pos] = origen.detalle[pos];
        }
    }

    /*Selección por torneo: entran 2 individuos y se decide el ganador en función de su penalización
    y eso no fuera suficiente en función de su fitness*/
    int seleccion_por_torneo(){
        int ind1=0, ind2=0;
        double pen1=0.0, pen2=0.0, fit1=0.0, fit2=0.0;

        ind1 = lista_enfrentamientos[pos_torneo];
        pos_torneo++;
        ind2 = lista_enfrentamientos[pos_torneo];
        pos_torneo++;

        fit1 = actual[ind1].fitness;
        fit2 = actual[ind2].fitness;
    }

```

```

        if(fit1<fit2)
            return ind2;
        else
            return ind1;
    }

```

//Genera un a lista desordenada de los individuos para poder relaizar enfrentamientos aleatorios...

```

void genera_lista_enfrentamientos(){
    int i=0, rand1=0, rand2=0, temp=0;

    for(i=0; i<num_individuos*2;i++){
        if(i<num_individuos)
            lista_enfrentamientos[i] = i;
        else
            lista_enfrentamientos[i] = i-num_individuos;
    }

    for(i=0; i < num_individuos*2; i++)
    {
        rand1 = rand() % num_individuos;
        rand2 = rand() % num_individuos;

        if(i>=num_individuos){
            rand1+=num_individuos;
            rand2+=num_individuos;
        }

        temp = lista_enfrentamientos[rand1];
        lista_enfrentamientos[rand1]=lista_enfrentamientos[rand2];
        lista_enfrentamientos[rand2]=temp;
    }
}

```

//Selección por ruleta: elige un individuo haciendo "girar" la ruleta...

```

int seleccion_por_ruleta(){
    int ok=0,res=0;
    double num=0.0;

    num = genera_double();
    ok = FALSE;

    while(!ok){
        num -= ruleta[pos_ruleta];
        if(num<0)
            ok = TRUE;
        else
            pos_ruleta = (pos_ruleta+1)%num_individuos;
    }
    res = pos_ruleta;
    pos_ruleta++;

    return res;
}

```

//Función que genera la ruleta para utilizar la función seleccion_por_ruleta...

```

void genera_ruleta(){
    int i=0;

    for(i=0;i<num_individuos;i++){
        ruleta[i] = (actual[i].fitness/fitness_global);
    }
}

```

/*Función de cruce: Selecciona uno o dos puntos al azar dentro del cromosoma de

la solución e intercambia el contenido de los dos progenitores.

El resultado (los dos descendientes) quedan guardados en la nueva generación en la misma posición en que estaban sus padres..

```

    op-> 1 - Cruce en un punto ==> Si no cruce en dos puntos...*/
void funcion_cruce_n_puntos(INDIVIDUO *hijo1, INDIVIDUO *hijo2, int padre1, int padre2, int op){
    int p1=0, p2=0, i=0;

    p1 = rand()%num_bultos;

    if(op==1){ //Cruce en un punto
        //printf("Punto de corte: %d\n",p1);
        for(i=0;i<p1;i++){
            hijo1->detalle[i]=actual[padre1].detalle[i];
            hijo2->detalle[i]=actual[padre2].detalle[i];
        }
        for(i<num_bultos;i++){
            hijo1->detalle[i]=actual[padre2].detalle[i];
            hijo2->detalle[i]=actual[padre1].detalle[i];
        }
    }
    else{ //Cruce en dos puntos
        do{
            p2 = rand()%num_bultos;
        }while(p1==p2);

        //Primero hay que saber que punto esta delante y cual detras...
        if(p1>p2){ //hay que invertirlos...
            i=p1;
            p1=p2;
            p2=i;
        }
        //printf("Puntos de corte: %d - %d\n",p1, p2);

        for(i=0;i<p1;i++){
            hijo1->detalle[i]=actual[padre1].detalle[i];
            hijo2->detalle[i]=actual[padre2].detalle[i];
        }
        for(i<p2;i++){
            hijo1->detalle[i]=actual[padre2].detalle[i];
            hijo2->detalle[i]=actual[padre1].detalle[i];
        }
        for(i<num_bultos;i++){
            hijo1->detalle[i]=actual[padre1].detalle[i];
            hijo2->detalle[i]=actual[padre2].detalle[i];
        }
    }
}

```

/*Funcion de cruce para el segundo tipo de codificacion...*/

```

void funcion_cruce_por_contenedores(INDIVIDUO *ind1, INDIVIDUO *ind2, int p1, int p2, int op){
    int i1,
        i2,
        i,
        j,
        k,
        lista_1[MAX_BULTOS_POR_VUELO],
        lista_1_aux[MAX_BULTOS_POR_VUELO],
        lista_2[MAX_BULTOS_POR_VUELO],
        lista_2_aux[MAX_BULTOS_POR_VUELO],
        eliminados_1[MAX_BULTOS_POR_VUELO],
        num_eliminados_1 = 0,
        eliminados_2[MAX_BULTOS_POR_VUELO],
        num_eliminados_2 = 0,
        pos_cruce_1 = 0,
        pos_cruce_2 = 0,
        ok1,

```

```

        ok2,
        cont1,
        cont2;

i1=0;
i2=0;
for(i=0;i<NUM_CONTAINERS+1;i++){
    for(j=0;j<actual[p1].contenedores[i].num_bultos;j++){
        lista_1_aux[i1] = actual[p1].contenedores[i].contenido[j];
        i1++;
    }
    for(j=0;j<actual[p2].contenedores[i].num_bultos;j++){
        lista_2_aux[i2] = actual[p2].contenedores[i].contenido[j];
        i2++;
    }
}

pos_cruce_1 = rand()%num_bultos;

if(op==1){
    for(i=0;i<pos_cruce_1;i++){
        lista_1[i] = lista_1_aux[i];
        lista_2[i] = lista_2_aux[i];
    }
    for(i=pos_cruce_1;i<num_bultos;i++){
        lista_1[i] = lista_2_aux[i];
        lista_2[i] = lista_1_aux[i];
    }
}
else{//Cruce en dos puntos

    do{
        pos_cruce_2 = rand()%num_bultos;
    }while(pos_cruce_1==pos_cruce_2);

    //Primero hay que saber que punto esta delante y cual detras...
    if(pos_cruce_1>pos_cruce_2){ //hay que invertirlos...
        i=pos_cruce_1;
        pos_cruce_1=pos_cruce_2;
        pos_cruce_2=i;
    }

    for(i=0;i<pos_cruce_1;i++){
        lista_1[i] = lista_1_aux[i];
        lista_2[i] = lista_2_aux[i];
    }
    for(i=pos_cruce_1;i<pos_cruce_2;i++){
        lista_1[i] = lista_2_aux[i];
        lista_2[i] = lista_1_aux[i];
    }
    for(i=pos_cruce_2;i<num_bultos;i++){
        lista_1[i] = lista_1_aux[i];
        lista_2[i] = lista_2_aux[i];
    }
}
//Buscamos los que puedan haberse eliminado...
num_eliminados_1 = 0;
num_eliminados_2 = 0;

for(i=0;i<num_bultos;i++){
    ok1 = FALSE;
    ok2 = FALSE;
    for(j=0;j<num_bultos;j++){
        if(lista_1[j]==i){
            ok1 = TRUE;
        }
    }
}

```

```

        if(lista_2[j]==i){
            ok2 = TRUE;
        }
    }
    if(!ok1){
        eliminados_1[num_eliminados_1] = i;
        num_eliminados_1++;
    }
    if(!ok2){
        eliminados_2[num_eliminados_2] = i;
        num_eliminados_2++;
    }
}

//buscamos los repetidos y los sustituimos por los eliminados...
for(i=0;i<num_bultos;i++){
    cont1=0;
    cont2=0;
    for(j=0;j<num_bultos;j++){
        if(lista_1[j]==i){
            cont1++;
            if(cont1==2){
                num_eliminados_1--;
                lista_1[j] = eliminados_1[num_eliminados_1];
            }
        }
        if(lista_2[j]==i){
            cont2++;
            if(cont2==2){
                num_eliminados_2--;
                lista_2[j] = eliminados_2[num_eliminados_2];
            }
        }
    }
}

//Traducimos cada posicion al contenedor correspondiente
i1=0;
i2=0;
for(j=0;j<NUM_CONTAINERS+1;j++){
    for(k=0;k<actual[p1].contenedores[j].num_bultos;k++){
        lista_1_aux[lista_1[i1]]=j;
        i1++;
    }
    for(k=0;k<actual[p2].contenedores[j].num_bultos;k++){
        lista_2_aux[lista_2[i2]]=j;
        i2++;
    }
}

//Finalmente pasamos la lista a los individuos...
for(i=0;i<num_bultos;i++){
    ind1->detalle[i] = lista_1_aux[i];
    ind2->detalle[i] = lista_2_aux[i];
}
}

/*Funcion de mutacion: Le pasamos un individuo y escogemos un paquete al azar
y lo cambiamos por otro tambien elegido al azar...*/
void funcion_de_mutacion(INDIVIDUO *ind){
    int num1=0, num2=0, z=0;

    num1 = rand()%num_bultos;

```

```

    num2 = rand()%num_bultos;

    z = ind->detalle[num1];
    ind->detalle[num1] = ind->detalle[num2];
    ind->detalle[num2] = z;
}

/*Una vez el individuo ha sido cruzado y mutado es necesario actualizar su lista interna
de contenedores a partir del detalle de los bultos...*/
void actualiza_contenedores(INDIVIDUO *ind){
    int cont=0,pos=0,aux=0,peso=0,num=0,PESO_LIMITE=0;
    double vol=0.0, VOL_LIMITE=0.0;

    //Primero iniciamos los contenedores como si fueran nuevos...
    for(cont = 0;cont<NUM_CONTAINERS+1;cont++){
        ind->contenedores[cont].completo = FALSE;
        ind->contenedores[cont].correcto = TRUE;
        ind->contenedores[cont].num_bultos = 0;
        ind->contenedores[cont].peso_total = 0;
        ind->contenedores[cont].vol_total = 0.0;
    }

    //Ahora repartimos los bultos en los contenedores...
    for(cont=0;cont<num_bultos;cont++){
        aux = ind->detalle[cont];
        pos = ind->contenedores[aux].num_bultos;

        ind->contenedores[aux].contenido[pos] = cont;
        ind->contenedores[aux].num_bultos ++;
        ind->contenedores[aux].peso_total+= cargamento[cont].peso;
        ind->contenedores[aux].vol_total+= cargamento[cont].volumen;
    }

    /*Finalmente evaluamos los contenedores para ver si son todos correctos o si
    han de ser reparados...*/
    //El 0 no se mira porque en ese no existen limitaciones (es el de la carga que no vuela)
    for(cont =1;cont<NUM_CONTAINERS+1;cont++){
        if(cont==8){ //Este contenedor es la bodega 5 que tiene un peso y volumen diferentes..
            VOL_LIMITE = VOL_MAXIMO_BODEGA_5;
            PESO_LIMITE = PESO_MAXIMO_BODEGA_5;
        }
        else{
            VOL_LIMITE = VOL_MAXIMO_CONTENEDOR;
            PESO_LIMITE = PESO_MAXIMO_CONTENEDOR;
        }
        num = ind->contenedores[cont].num_bultos;
        peso = ind->contenedores[cont].peso_total;
        vol = ind->contenedores[cont].vol_total;

        if((num<MAX_BULTOS_POR_CONTENEDOR)&&
            (vol<VOL_LIMITE)&&
            (peso<PESO_LIMITE))
            ind->contenedores[cont].correcto = TRUE;
        else
            ind->contenedores[cont].correcto = FALSE;

        if((num<MAX_BULTOS_POR_CONTENEDOR)&&
            ((vol+VOL_MIN_BULTO)<=VOL_LIMITE)&&
            ((peso+PESO_MIN_BULTO)<=PESO_LIMITE))
            ind->contenedores[cont].completo = FALSE;
        else
            ind->contenedores[cont].completo = TRUE;
    }
}

```

```

/*Si existen contenedores que violen alguno de los parametros de peso, vol o numero de bultos,
esta funcion redistribuye esa carga extra entre los otros contenedores para hacer
que la solucion ofertada sea valida...*/
void repara_soluciones(INDIVIDUO *ind){
    int i=0, bulto=0, cont=0, intentos=0, num=0, PESO_LIMITE=0;
    double VOL_LIMITE=0.0;

    for(i=1;i<NUM_CONTAINERS+1;i++){
        if(!ind->contenedores[i].correcto){
            if(i==8){//Bodega 5 --> tiene un peso y volumen diferentes...
                PESO_LIMITE = PESO_MAXIMO_BODEGA_5;
                VOL_LIMITE = VOL_MAXIMO_BODEGA_5;
            }
            else{
                PESO_LIMITE = PESO_MAXIMO_CONTENEDOR;
                VOL_LIMITE = VOL_MAXIMO_CONTENEDOR;
            }

            do{
                bulto = rand()%((int)ind->contenedores[i].num_bultos); //Escogemos
                                                                    un bulto al azar...

                intentos = 1;
                num = rand()%NUM_CONTAINERS;

                do{
                    if(intentos==NUM_CONTAINERS) //Si los he probado todos y
                                                no cabe lo pongo en el 0

                        cont=0;

                    else{
                        cont = (num+intentos)%(NUM_CONTAINERS+1);
                        //Pruebo en cualquier container menos en el 0 y en
                                                                    el mismo...
                        if((cont==0)||((cont==i))){
                            cont++;
                            if(cont>8)
                                cont=1;
                        }

                        intentos++;
                    }
                }while(!inserta_bulto(ind->contenedores[i].contenido[bulto],cont,ind));

                //Una vez el bulto se ha colocado en otro container lo sacamos de este
                ind->contenedores[i].vol_total-=cargamento[ind-
>contenedores[i].contenido[bulto]].volumen;
                ind->contenedores[i].peso_total-=cargamento[ind-
>contenedores[i].contenido[bulto]].peso;
                ind->contenedores[i].contenido[bulto]=ind-
>contenedores[i].contenido[ind->contenedores[i].num_bultos-1];
                ind->contenedores[i].num_bultos--;

                //Comprobamos si ahora el contenedor es correcto...
                if((ind-
>contenedores[i].num_bultos<MAX_BULTOS_POR_CONTENEDOR)&&
                    (ind->contenedores[i].vol_total<VOL_LIMITE)&&
                    (ind->contenedores[i].peso_total<PESO_LIMITE))
                    ind->contenedores[i].correcto = TRUE;
                else
                    ind->contenedores[i].correcto = FALSE;

                if((ind-
>contenedores[i].num_bultos<MAX_BULTOS_POR_CONTENEDOR)&&
                    ((ind-
>contenedores[i].vol_total+VOL_MIN_BULTO)<=VOL_LIMITE)&&
                    ((ind-
>contenedores[i].peso_total+PESO_MIN_BULTO)<=PESO_LIMITE))

```

```

        ind->contenedores[i].completo = FALSE;
    else
        ind->contenedores[i].completo = TRUE;
    }while(!ind->contenedores[i].correcto);
    }
}

if(MAXIMIZAR_CARGA)
    carga_maxima(ind);
}

//Suma el volumen y peso de los bultos que se han cargado...
void calcula_ocupacion(INDIVIDUO * ind){
    int i;

    ind->peso_cargado = 0;
    ind->vol_cargado = 0.0;

    for (i=1;i<NUM_CONTAINERS+1;i++){
        ind->peso_cargado += ind->contenedores[i].peso_total;
        ind->vol_cargado += ind->contenedores[i].vol_total;
    }
}

/*La idea de esta función es intentar recuperar la mayor cantidad de carga posible
de la que se queda en tierra y cargarla en cualquier otro contenedor*/
void carga_maxima(INDIVIDUO *ind){
    int i, j, completos;

    for(i=0;i<ind->contenedores[0].num_bultos;i++){
        completos = 0;
        for(j=1;j<NUM_CONTAINERS+1;j++){
            if(ind->contenedores[j].completo){
                completos++;
                if(completos==NUM_CONTAINERS)
                    return;
            }
            else{
                if(inserta_bulto(ind->contenedores[0].contenido[i],j,ind))
                    j=NUM_CONTAINERS+1;
            }
        }
    }
}

/*Funcion que genera una nueva poblacion de individuos: */
void genera_nueva_poblacion(){
    int i=0, p1=0, p2=0;
    double res=0;

    for(i=0;i<num_individuos;i+=2){
        //Elegimos los padres con el metodo que sea...
        if(op_seleccion==0){ //Selección por torneo...
            p1 = seleccion_por_torneo();
            p2 = seleccion_por_torneo();
        }
        else{//Selección por ruleta...
            p1 = seleccion_por_ruleta();
            p2 = seleccion_por_ruleta();
        }

        //Los cruzamos con el metodo elegido si superan la probabilidad de cruce...
        res = genera_double();
        if(res<p_cruce){//Si pasan la prueba los cruzamos...

```



```

        if(op_codificacion == 0)

funcion_cruce_n_puntos(&nueva[i],&nueva[i+1],p1,p2,num_puntos_cruce);
        else

funcion_cruce_por_contenedores(&nueva[i],&nueva[i+1],p1,p2,num_puntos_cruce);
    }
    else{//Si no pasan tal cual a la nueva generacion...
        copia_individuo(actual[p1],&nueva[i]);
        copia_individuo(actual[p2],&nueva[i+1]);
    }

    //Los mutamos si superan la probabilidad de mutacion...
    //PRIMER INDIVIDUO...
    res = genera_double();
    if(res<p_mut){
        funcion_de_mutacion(&nueva[i]);
    }
    //SEGUNDO INDIVIDUO...
    res = genera_double();
    if(res<p_mut){
        funcion_de_mutacion(&nueva[i+1]);
    }

    //Actualizamos las soluciones...
    actualiza_contenedores(&nueva[i]);
    actualiza_contenedores(&nueva[i+1]);

    //Reparamos las soluciones que no esten permitidas o no...
    if(REPARAR){
        repara_soluciones(&nueva[i]);
        repara_soluciones(&nueva[i+1]);
    }
}
//Actualizamos la generacion actual con la nueva...
temporal = actual;
actual = nueva;
nueva = temporal;
}

//Calcula el tiempo transcurrido entre dos momentos...
double calcula_tiempo(LARGE_INTEGER *fin, LARGE_INTEGER *init){
    LARGE_INTEGER freq;
    QueryPerformanceFrequency(&freq);
    return (double)((fin->QuadPart - init->QuadPart) / (double)freq.QuadPart);
}

//Esta funcion ejecuta un ciclo completo con todas las generaciones....
void ciclo(){
    //Instante inicial de tiempo en que da comienzo el ciclo...
    QueryPerformanceCounter(&t_inicial_ciclo);

    inicia_parametros();

    for(gen_actual=1;gen_actual<num_generaciones;gen_actual++){

        genera_nueva_poblacion();
        funcion_fitness();
        calcula_estadisticas();

        if(IMPRIME_DATOS)
            imprime_generacion();

        imprime_mejor_generacion(fp_mejores_generacion);
    }
}

```

```

//Instante final de tiempo para el ciclo...
QueryPerformanceCounter(&t_final_ciclo);
tiempo_ciclo = calcula_tiempo(&t_final_ciclo,&t_inicial_ciclo);
imprime_tiempo_ciclo();
imprime_mejor_ciclo(fp_mejores_generacion);
imprime_mejor_ciclo(fp_mejores_ciclo);
imprime_resultados_normalizados();
}

/*=====
===== MAIN =====
*/
void main(){

    menu_principal();
    inicio();

    printf("\nPROCESANDO ");
    for(ciclo_actual=0;ciclo_actual<num_ciclos;ciclo_actual++){
        printf(".");
        ///CODIFICACION A
        //if(ciclo_actual==0){
        //    op_codificacion = 0;
        //    op_seleccion = 0;
        //    num_puntos_cruce = 1;
        //    fprintf(fp_excel,"\nCodificacion A: TORNEO - 1 PUNTO\n");
        //}
        //if(ciclo_actual==(num_ciclos/8)*1){
        //    printf("\n");
        //    fprintf(fp_excel,"\nCodificacion A: TORNEO - 2 PUNTO\n");
        //    num_puntos_cruce = 2;
        //}
        //if(ciclo_actual==(num_ciclos/8)*2){
        //    printf("\n");
        //    fprintf(fp_excel,"\nCodificacion A: RULETA - 1 PUNTO\n");
        //    op_seleccion = 1;
        //    num_puntos_cruce = 1;
        //}
        //if(ciclo_actual==(num_ciclos/8)*3){
        //    printf("\n");
        //    fprintf(fp_excel,"\nCodificacion A: RULETA - 2 PUNTO\n");
        //    num_puntos_cruce = 2;
        //}

        ///CODIFICACION A
        //if(ciclo_actual==(num_ciclos/8)*4){
        //    printf("\n");
        //    op_codificacion = 1;
        //    op_seleccion = 0;
        //    num_puntos_cruce = 1;
        //    fprintf(fp_excel,"\nCodificacion A: TORNEO - 1 PUNTO\n");
        //}
        //if(ciclo_actual==(num_ciclos/8)*5){
        //    printf("\n");
        //    fprintf(fp_excel,"\nCodificacion A: TORNEO - 2 PUNTO\n");
        //    num_puntos_cruce = 2;
        //}
        //if(ciclo_actual==(num_ciclos/8)*6){
        //    printf("\n");
        //    fprintf(fp_excel,"\nCodificacion A: RULETA - 1 PUNTO\n");
        //    op_seleccion = 1;
        //    num_puntos_cruce = 1;
        //}
        //if(ciclo_actual==(num_ciclos/8)*7){

```

```

//      printf("\n");
//      fprintf(fp_excel, "\nCodificacion A: RULETA - 2 PUNTO\n");
//      num_puntos_cruce = 2;
//}

ciclo();
}
printf(" COMPLETADO.\n");

libera_memoria();
cierra_ficheros();
printf("\nProceso concluido...");
printf("\n   --- Pulsa enter para terminar ---");
getchar();
}
/*=====
===== END OF MAIN =====
=====*/

```

/*FUNCION OBJETIVO.....

FITNESS...

EL fitness del individuo se definira en relacion a la cantidad de carga que se queda en tierra y la penalizacion sufrida. Todo esto normalizado de manera que un valor de 1 se traduciria en CG de gravedad perfecto con el avion cargado al maximo...

$$\text{FITNESS} = \frac{\left| \frac{\text{volumen cargado}}{\text{volumen maximo del vuelo}} + \frac{\text{peso en tierra}}{\text{peso maximo del vuelo}} \right|}{2} + (2,5 - |\text{TRIM}|)$$

3,5

De esta forma se da prioridad al equilibrado del avion mejorando la seguridad y calidad del trayecto asi como el consumo de carburante. Aunque tambien se tiene en cuenta la cantidad de carga que se transporta...*/

```

void funcion_fitness(){
    int i=0;
    double vol_vuelo, peso_vuelo, res;

    for(i=0;i<num_individuos;i++){
        if(!PROBAR_COMBINACIONES){
            //Instante inicial de tiempo en que da comienzo la evaluacion...
            QueryPerformanceCounter(&t_inicial_evaluacion);
        }

        calcula_TRIM(&actual[i],PROBAR_COMBINACIONES);

        //Ahora calculamos el fitness con la funcion de arriba descrita...
        if(OVERLOAD){
            vol_vuelo = actual[i].vol_cargado/MAX_VOLUMEN_POR_VUELO;
            peso_vuelo = ((double)actual[i].peso_cargado)/MAX_PESO_POR_VUELO;
        }
        else{
            vol_vuelo = actual[i].vol_cargado/vol_total;
            peso_vuelo = ((double)actual[i].peso_cargado)/peso_total;
        }

        res = (((vol_vuelo + peso_vuelo)/2))+(2.5 - actual[i].TRIM))/3.5;

        actual[i].fitness = res;

        if(!PROBAR_COMBINACIONES){
            //Instante final de tiempo para la evaluacion
            QueryPerformanceCounter(&t_final_evaluacion);
        }
    }
}

```

```

    }
}

```

//Calcula el angulo de TRIM a partir de lo cargado en las bodegas, fuel y pasajeros...

```

void calcula_TRIM(INDIVIDUO *ind, int opcion){

    int b1 ,b3 ,b4 ,b5, limite, i, best_combinacion, ok;
    double res, best_TRIM;

    best_TRIM = 2.5;
    best_combinacion = 0;
    ok = TRUE;

    if(!REPARAR){
        for(i=1;i<NUM_CONTAINERS+1;i++){
            if(!ind->contenedores[i].correcto){
                ok = FALSE;
                i=NUM_CONTAINERS+1;
                opcion = 0;
            }
        }
    }

    if(opcion)
        limite = NUM_COMBINACIONES;
    else
        limite = 1;

    for(i=0;i<limite;i++){
        b1 = ind->contenedores[combinaciones[i].distribucion[0]].peso_total+
            ind->contenedores[combinaciones[i].distribucion[1]].peso_total+
            ind->contenedores[combinaciones[i].distribucion[2]].peso_total;

        b3 = ind->contenedores[combinaciones[i].distribucion[3]].peso_total+
            ind->contenedores[combinaciones[i].distribucion[4]].peso_total;

        b4 = ind->contenedores[combinaciones[i].distribucion[5]].peso_total+
            ind->contenedores[combinaciones[i].distribucion[6]].peso_total;

        b5 = ind->contenedores[8].peso_total;

        res = INDICE_DOW +
            (pantry_e*PANTRY_E)+
            (pantry_f*PANTRY_F)+
            (b1*FACTOR_CARGO_1)+
            (b3*FACTOR_CARGO_3)+
            (b4*FACTOR_CARGO_4)+
            (b5*FACTOR_CARGO_5)+
            (pasajeros_zona_A*FACTOR_CABIN_0A)+
            (pasajeros_zona_B*FACTOR_CABIN_0B)+
            (pasajeros_zona_C*FACTOR_CABIN_0C)+
            (FACTOR_FUEL);

        res = res - CERO_TRIM;
        res = (res * FACTOR_TRIM);

        if(res<0)
            res = res *(-1);
        if(res<best_TRIM){
            best_TRIM = res;
            best_combinacion = i;
        }
    }
}

```

```

/*una vez recorridas todas las opciones actualizamos la lista de combinaciones
del individuo con la mejor de todas...*/

if(!ok){
    ind->TRIM =2.5;
}
else{
    ind->TRIM = best_TRIM; //Se guarda el mejor valor como el TRIM...
}

for (i=0;i<NUM_CONTAINERS-1;i++){
    ind->posicion.distribucion[i] = combinaciones[best_combinacion].distribucion[i];
}
}

/*****
***** FUNCIONES PARA PRESENTACION DE DATOS *****/
*****/

/* fp_datos -- "Datos.txt"
   fp_mejores -- "MejoresIndividuos.txt"
   fp_excel -- "Excel.txt"
   fp_tiempos -- "Tiempos.txt"
*/

//Imprime datos iniciales del proceso...
void imprime_datos_iniciales(FILE *fp_destino){

    int i=0;

    fprintf(fp_destino,"-----\n");
    fprintf(fp_destino,"----- DATOS INICIALES DEL PROCESO -----
---\n");
    fprintf(fp_destino,"-----\n");
    fprintf(fp_destino,"\n\tNumero de pasajeros en la Zona A: %d ",pasajeros_zona_A);
    fprintf(fp_destino,"\n\tNumero de pasajeros en la Zona B: %d ",pasajeros_zona_B);
    fprintf(fp_destino,"\n\tNumero de pasajeros en la Zona C: %d ",pasajeros_zona_C);
    fprintf(fp_destino,"\n\tPeso del combustible: %d Kgs.",peso_fuel);
    fprintf(fp_destino,"\n\tPeso del cargamento: %d Kgs.",peso_total);
    fprintf(fp_destino,"\n\tVolumen del cargamento: %f ",vol_total);
    fprintf(fp_destino,"\n\tNumero de bultos: %d",num_bultos);

    fprintf(fp_destino,"\n\tPermite OVERLOAD: ");
    if(op_envio)
        fprintf(fp_destino,"NO");
    else{
        fprintf(fp_destino,"SI");
        fprintf(fp_destino,"\n\tExceso de peso: %d Kgs.",PESO_OVERLOAD);
        fprintf(fp_destino,"\n\tExceso de volumen: %f ",VOL_OVERLOAD);
    }

    fprintf(fp_destino,"\n\tProbamos todas las combinaciones posibles de cada solucion: ");
    if(PROBAR_COMBINACIONES)
        fprintf(fp_destino,"SI");
    else
        fprintf(fp_destino,"NO");

    fprintf(fp_destino,"\n\tNumero de ciclos: %d",num_ciclos);
    fprintf(fp_destino,"\n\tNumero de individuos: %d ",num_individuos);
    fprintf(fp_destino,"\n\tNumero de generaciones: %d",num_generaciones);

    fprintf(fp_destino,"\n\tMetodo de seleccion: ");
    if(op_envio)
        fprintf(fp_destino,"Torneo");
    else
        fprintf(fp_destino,"Ruleta");

```

```

fprintf(fp_destino, "\n\tProbabilidad de cruce: %f", p_cruce);
fprintf(fp_destino, "\n\tProbabilidad de mutacion: %f", p_mut);

fprintf(fp_destino, "\n\t-----");
fprintf(fp_destino, "\n\t----- DETALLE DEL CARGAMENTO ----- TOTAL %d BULTOS -----", num_bultos);
fprintf(fp_destino, "\n\t-----");

for(i=0; i<num_bultos; i++){
    fprintf(fp_destino, "\n\t BULTO %d --> \t PESO: %d \t VOL: %f", i+1, cargamento[i].peso, cargamento[i].volumen);
}

}

//Imprime en el fichero de datos.txt la generacion completa con todos los parametros...
void imprime_generacion(){
    int ind=0, i=0, pos=0;

    fprintf(fp_datos, "\n\t-----");
    fprintf(fp_datos, "\n\t----- CICLO %d ----- GENERACION %d -----", ciclo_actual, gen_actual);

    for(ind=0; ind<num_individuos; ind++){
        fprintf(fp_datos, "\n\t(%d - %d) ==> ", gen_actual, ind);
        for(i=0; i<num_bultos; i++){
            fprintf(fp_datos, "%d ", actual[ind].detalle[i]);
            if((i%40)==39)
                fprintf(fp_datos, "\n\t");
        }
        fprintf(fp_datos, "\n");

        for(i=0; i<NUM_CONTAINERS-1; i++){
            pos = actual[ind].posicion.distribucion[i];
            fprintf(fp_datos, "\tC%d(%d Kg. | %f m3) \t", pos, actual[ind].contenedores[pos].peso_total, actual[ind].contenedores[pos].vol_total);
            if((i==2)|| (i==4))
                fprintf(fp_datos, "\n");
        }

        fprintf(fp_datos, "\n\tC%d(%d Kg. | %f m3)\n", 8, actual[ind].contenedores[8].peso_total, actual[ind].contenedores[8].vol_total);

        fprintf(fp_datos, "\tNo Cargado(%d Kg. | %f m3)\tAngulo TRIM: %f\tFitness: %f\n", actual[ind].contenedores[0].peso_total, actual[ind].contenedores[0].vol_total, actual[ind].TRIM, actual[ind].fitness);
    }
}

/*Funcion que imprime el mejor individuo de cada generacion en el fichero que quieras...
*/
void imprime_mejor_generacion(FILE *fp_destino){
    int i=0, pos=0;

    if(gen_actual==0){
        if(ciclo_actual==0){
            fprintf(fp_destino, "-----\n");
            fprintf(fp_destino, "----- MEJORES INDIVIDUOS DE CADA GENERACION Y MEJOR ABSOLUTO DEL CICLO ----- \n");
        }
    }
}

```

```

        fprintf(fp_destino,"-----\n");
    }
    fprintf(fp_destino,"===== CICLO %d\n",ciclo_actual);
}

fprintf(fp_destino,"\n(%d) ==> ",gen_actual);

for(i=0;i<num_bultos;i++){
    fprintf(fp_destino,"%d ",mejores_generacion[gen_actual].detalle[i]);
    if((i%40)==39)
        fprintf(fp_destino,"\n\t");
}
fprintf(fp_destino,"\n");

for(i=0;i<NUM_CONTAINERS-1;i++){
    pos = mejores_generacion[gen_actual].posicion.distribucion[i];
    fprintf(fp_destino,"\tC%d(%d Kg. | %f m3) \t",
            pos,
            mejores_generacion[gen_actual].contenedores[pos].peso_total,
            mejores_generacion[gen_actual].contenedores[pos].vol_total);
    if((i==2)||i==4))
        fprintf(fp_destino,"\n");
}

fprintf(fp_destino,"\n\tC%d(%d Kg. | %f m3)\n",
        8,
        mejores_generacion[gen_actual].contenedores[8].peso_total,
        mejores_generacion[gen_actual].contenedores[8].vol_total);

fprintf(fp_destino,"\tNo Cargado(%d Kg. | %f m3)\tAngulo TRIM: %f\tFitness: %f\n",
        mejores_generacion[gen_actual].contenedores[0].peso_total,
        mejores_generacion[gen_actual].contenedores[0].vol_total,
        mejores_generacion[gen_actual].TRIM,
        mejores_generacion[gen_actual].fitness);
}
/*Imprime el mejor individuo de cada ciclo junto con la generacion a la que pertenece
en el fichero que desees...*/
void imprime_mejor_ciclo(FILE *fp_destino){
    int i=0,pos=0;
    double ocupacion = 0.0;

    fprintf(fp_destino,"\n*****");
    fprintf(fp_destino,"***** CICLO %d\n",ciclo_actual);

    fprintf(fp_destino,"\n(%d) ==> ",generacion_del_mejor);

    for(i=0;i<num_bultos;i++){
        fprintf(fp_destino,"%d ",mejor_absoluto.dDetalle[i]);
        if((i%40)==39)
            fprintf(fp_destino,"\n\t");
    }
    fprintf(fp_destino,"\n");

    for(i=0;i<NUM_CONTAINERS-1;i++){
        pos = mejor_absoluto.posicion.distribucion[i];

        fprintf(fp_destino,"\tC%d(%d Kg. | %f m3) \t",
            pos,
            mejor_absoluto.contenedores[pos].peso_total,
            mejor_absoluto.contenedores[pos].vol_total);
        if((i==2)||i==4))
            fprintf(fp_destino,"\n");
    }
}

```


D.3. Script's Matlab para el análisis de resultados

D.3.1. Script para la función A

```
%Lee los datos del fichero Datos_matlab_<A, B , C, D ,E>.txt generado por
%el código del AG y permite visualizar la gráfica en cuestión y las
%poblaciones que se han ido generando durante las ejecuciones

%La primera fila es la definición del proceso:
%<max_ciclos><max_generaciones><poblacion><total_variables><funcion>
%<p_cruce><p_mut_bin><p_mut_real><sharing><nicho><x1min><x1max><x2min><x2max>...

datos = load('Datos_matlab_A.txt');

max_ciclos = datos(1,1);
max_generaciones = datos(1,2);
poblacion = datos(1,3);
total_variables = datos(1,4);
funcion = datos(1,5);
p_cruce = datos(1,6);
p_mut_bin = datos(1,7);
p_mut_real = datos(1,8);
sharing = datos(1,9);
nicho = datos(1,10);
xmin = datos(1,11);
xmax = datos(1,12);
ymin = datos(1,13);
ymax = datos(1,14);

if(sharing==0)
    str = ['Pob: ' num2str(poblacion) ' P.Cruce: ' num2str(p_cruce)...
          ' P.Mut Bin: ' num2str(p_mut_bin) ' P.Mut.Real: ' num2str(p_mut_real)...
          ' Sin Sharing'];
else
    str = ['Pob: ' num2str(poblacion) ' P.Cruce: ' num2str(p_cruce)...
          ' P.Mut Bin: ' num2str(p_mut_bin) ' P.Mut.Real: ' num2str(p_mut_real)...
          ' Con Sharing--> nicho: ' num2str(nicho)];
end

%1er paso--> dialogo de opciones:
opcion = questdlg('Selecciona una opcion...','Menu grafico',...
    'Solo la funcion','Generacion concreta','Animacion','Solo la funcion');

switch opcion
case 'Solo la funcion'
    x = xmin:0.0001:xmax; % Lista de puntos en el eje X
    y = ((x.^2)/25)-10.123456789012344; % Función A
    h=figure;
    hold on;
    title(['Funcion A';'y = (x^2/25)-10,123456789012344']);
    plot(x,y,'b');
    text(0,-10.123456789012344,'<math>\leftarrow \min = -10,123456789012344</math>','FontSize',10)
    grid on;
    x=0;
    plot(x,((x.^2)/25)-10.123456789012344,'r*');
    hgsave(h,'Funcion_A','-v6');
    print -djpeg Funcion_A;
    hold off;
    pause(3);
    close all Force;
    break
case 'Generacion concreta'
    textos = {'Introduce numero de ciclo entre 1 y ' num2str(max_ciclos) ':'},...
        ['Introduce numero de generacion entre 0 y ' num2str(max_generaciones) ':'}];
```

```

titulo = 'Generacion concreta';
op = 0;

while (op==0)
    correcto=0;
    answer=inputdlg(textos,titulo,1);
    ciclo=str2double(answer{1});
    generacion=str2double(answer{2});

    offset = (((max_generaciones*total_variables) +(total_variables+1)) * (ciclo-1));
    puntos = datos(offset+2:offset+max_generaciones+(total_variables+1),1:poblacion+1);
    gen = puntos(generacion,:);
    h=figure;
    hold on;
    x = xmin:0.0001:xmax;
    y = ((x.^2)/25)-10.123456789012344; % Funcion A
    grid on;
    plot(x,y,'b')
    title(['Funcion A - Generacion: ' num2str(generacion)];...
        'y = (x^2/25)-10.123456789012344';str));
    plot(gen,(((gen.^2)/25)-10.123456789012344),'g*','LineWidth',2,'MarkerSize',5)
    max = gen(poblacion+1);
    plot(max,(((max.^2)/25)-10.123456789012344),'r*','LineWidth',2,'MarkerSize',5)
    text(max,(((max.^2)/25)-10.123456789012344),...
        ['\leftarrow Mejor fitness = ' num2str(((max.^2)/25)-10.123456789012344)],...
        'FontSize',10)

    print('-djpeg',['Figura A (' num2str(ciclo) '-' num2str(generacion) ')']);
    hgsave(h,['Figura A (' num2str(ciclo) '-' num2str(generacion) ')'],'-v6');
    hold off

    button = questdlg('Quieres representar otra generacion?',...
        'Otra generacion?...','Si','No','No');
    if strcmp(button,'No')
        op=1;
    end
end;
close all Force;
break
case 'Animacion'
    op=0;
    while (op==0)
        answer=inputdlg(['Que ciclo deseas ver (Entre 1 y ' num2str(max_ciclos) ')'],'Animacion',1);
        ciclo=str2double(answer{1});
        offset = (((max_generaciones*total_variables) +(total_variables+1)) * (ciclo-1));
        puntos = datos(offset+2:offset+max_generaciones+(total_variables+1),1:poblacion+1);
        %h=figure;

        for cont=1:max_generaciones+1
            newplot;
            gen = puntos(cont,:);
            hold on;
            x = xmin:0.0001:xmax;
            y = ((x.^2)/25)-10.123456789012344; % Función A
            grid on;
            plot(x,y,'b')
            title(['Funcion A - Generacion: ' num2str(cont-1)];...
                'y = (x^2/25)-10.123456789012344';str));
            plot(gen,(((gen.^2)/25)-10.123456789012344),'g*','LineWidth',2,'MarkerSize',5)
            max = gen(poblacion+1);
            plot(max,(((max.^2)/25)-10.123456789012344),'r*','LineWidth',2,'MarkerSize',5)
            text(max,(((max.^2)/25)-10.123456789012344),...
                ['\leftarrow Mejor fitness = ' num2str(((max.^2)/25)-10.123456789012344)],...
                'FontSize',10)

            %Descomentando estas dos líneas guardara todas las
            %imagenes del ciclo

```

```

    %print('-djpeg',[Figura A (' num2str(ciclo) '-' num2str(cont) ')])
    %hgsave(h,[Figura A (' num2str(ciclo) '-' num2str(cont) ')],'-v6');
    hold off
    pause(0.1);
end;

button = questdlg('Quieres ver otra animacion?',...
'Otra generacion?...','Si','No','No');
if strcmp(button,'No')
    op=1;
    close all Force;
end
end;
close all Force;
end
close all Force;

```

D.3.2. Script para la función B

%Lee los datos del fichero Datos_matlab_<A, B , C, D ,E>.txt generado por
 %el código del AG y permite visualizar la gráfica en cuestión y las
 %poblaciones que se han ido generando durante las ejecuciones

%La primera fila es la definición del proceso:

%<max_ciclos><max_generaciones><poblacion><total_variables><funcion>

%<p_cruce><p_mut_bin><p_mut_real><sharing><nicho><x1min><x1max><x2min><x2max>...

```
datos = load('Datos_matlab_B.txt');
```

```

max_ciclos = datos(1,1);
max_generaciones = datos(1,2);
poblacion = datos(1,3);
total_variables = datos(1,4);
funcion = datos(1,5);
p_cruce = datos(1,6);
p_mut_bin = datos(1,7);
p_mut_real = datos(1,8);
sharing = datos(1,9);
nicho = datos(1,10);
xmin = datos(1,11);
xmax = datos(1,12);
ymin = datos(1,13);
ymax = datos(1,14);

```

```

if(sharing==0)
    str = ['Pob: ' num2str(poblacion) ' P.Cruce: ' num2str(p_cruce)...
    ' P.Mut Bin: ' num2str(p_mut_bin) ' P.Mut.Real: ' num2str(p_mut_real)...
    ' Sin Sharing'];
else
    str = ['Pob: ' num2str(poblacion) ' P.Cruce: ' num2str(p_cruce)...
    ' P.Mut Bin: ' num2str(p_mut_bin) ' P.Mut.Real: ' num2str(p_mut_real)...
    ' Con Sharing--> nicho: ' num2str(nicho)];
end

```

%1er paso--> dialogo de opciones:

```
opcion = questdlg('Selecciona una opcion...','Menu grafico',...
'Solo la funcion','Generacion concreta','Animacion','Solo la funcion');
```

```

switch opcion
case 'Solo la funcion'
    x = xmin:0.0001:xmax; % Lista de puntos en el eje X
    y= (cos(x.^2)).*(cos(x).^2); %Funcion B
    h=figure;
    hold on;

```

```

title(['Funcion B'; 'y = cos(x^2)*cos^2(x)']);
plot(x,y,'b');
text(0,1, '\leftarrow max = 1', 'FontSize', 10)
grid on;
x=0;
plot(x,(cos(x.^2)).*(cos(x).^2), 'r*');
hgsave(h, 'Funcion_B', '-v6');
print -djpeg Funcion_B;
hold off;
pause(3);
close all Force;
break
case 'Generacion concreta'
textos = {'Introduce numero de ciclo entre 1 y ' num2str(max_ciclos) ':'}, ...
        ['Introduce numero de generacion entre 0 y ' num2str(max_generaciones) ':'}];
titulo = 'Generacion concreta';
op = 0;

while (op==0)
    correcto=0;
    answer=inputdlg(textos,titulo,1);
    ciclo=str2double(answer{1});
    generacion=str2double(answer{2});

    offset = (((max_generaciones*total_variables) +(total_variables+1)) * (ciclo-1));
    puntos = datos(offset+2:offset+max_generaciones+(total_variables+1),1:poblacion+1);
    gen = puntos(generacion,:);
    h=figure;
    hold on;
    x = xmin:0.0001:xmax;
    y =(cos(x.^2)).*(cos(x).^2); % Funcion B
    grid on;
    plot(x,y,'b')
    title(['Funcion B - Generacion: ' num2str(generacion)]; ...
        'y = cos(x^2)*cos^2(x);str});
    plot(gen,((cos(gen.^2)).*(cos(gen).^2)), 'g*', 'LineWidth', 2, 'MarkerSize', 5)
    max = gen(poblacion+1);
    plot(max,((cos(max.^2)).*(cos(max).^2)), 'r*', 'LineWidth', 2, 'MarkerSize', 5)
    text(max,((cos(max.^2)).*(cos(max).^2)), ...
        [' \leftarrow Mejor fitness = ' num2str(((cos(max.^2)).*(cos(max).^2))), ...
        'FontSize', 10)

    print('-djpeg', ['Figura B (' num2str(ciclo) '-' num2str(generacion) ')']);
    hgsave(h, ['Figura B (' num2str(ciclo) '-' num2str(generacion) ')'], '-v6');
    hold off

    button = questdlg('Quieres representar otra generacion?', ...
        'Otra generacion?...','Si','No','No');
    if strcmp(button, 'No')
        op=1;
    end
end;
close all Force;
break
case 'Animacion'
op=0;
while (op==0)
    answer=inputdlg(['Que ciclo deseas ver (Entre 1 y ' num2str(max_ciclos) ')'], 'Animacion', 1);
    ciclo=str2double(answer{1});
    offset = (((max_generaciones*total_variables) +(total_variables+1)) * (ciclo-1));
    puntos = datos(offset+2:offset+max_generaciones+(total_variables+1),1:poblacion+1);
    %h=figure;

    for cont=1:max_generaciones+1
        newplot;
        gen = puntos(cont,:);
        hold on;

```

```

x = xmin:0.0001:xmax;
y = (cos(x.^2)).*(cos(x).^2); % Funcion A
grid on;
plot(x,y,'b')
title(['Funcion B - Generacion: ' num2str(cont-1)];...
      'y = cos(x^2)*cos^2(x);str});
plot(gen,((cos(gen.^2)).*(cos(gen).^2)), 'g*', 'LineWidth',2, 'MarkerSize',5)
max = gen(poblacion+1);
plot(max,((cos(max.^2)).*(cos(max).^2)), 'r*', 'LineWidth',2, 'MarkerSize',5)
text(max,((cos(max.^2)).*(cos(max).^2)),...
      [' \leftarrow Mejor fitness = ' num2str(((cos(max.^2)).*(cos(max).^2))),...
      'FontSize',10)

%Descomentando estas dos lineas guardara todas las
%imagenes del ciclo
%print('-djpeg',['Figura B (' num2str(ciclo) '-' num2str(cont) ')]
%hgsave(h,['Figura B (' num2str(ciclo) '-' num2str(cont) ')],'-v6');
hold off
pause(0.1);
end;

button = questdlg('Quieres ver otra animacion?',...
                  'Otra generacion?...','Si','No','No');
if strcmp(button,'No')
    op=1;
    close all Force;
end
end;
close all Force;
end

```

D.3.3. Script para la función C

%Lee los datos del fichero Datos_matlab_<A, B, C, D, E>.txt generado por
 %el codigo del AG y permite visualizar la grafica en cuestion y las
 %poblaciones que se han ido generando durante las ejecuciones

%La primera fila es la definicion del proceso:
 %<max_ciclos><max_generaciones><poblacion><total_variables><funcion>
 %<p_cruce><p_mut_bin><p_mut_real><sharing><nicho><x1min><x1max><x2min><x2max>...

```
datos = load('Datos_matlab_C.txt');
```

```

max_ciclos = datos(1,1);
max_generaciones = datos(1,2);
poblacion = datos(1,3);
total_variables = datos(1,4);
funcion = datos(1,5);
p_cruce = datos(1,6);
p_mut_bin = datos(1,7);
p_mut_real = datos(1,8);
sharing = datos(1,9);
nicho = datos(1,10);
xmin = datos(1,11);
xmax = datos(1,12);
ymin = datos(1,13);
ymax = datos(1,14);

```

```

if(sharing==0)
    str = ['Pob: ' num2str(poblacion) ' P.Cruce: ' num2str(p_cruce)...
          ' P.Mut Bin: ' num2str(p_mut_bin) ' P.Mut.Real: ' num2str(p_mut_real)...
          ' Sin Sharing'];
else
    str = ['Pob: ' num2str(poblacion) ' P.Cruce: ' num2str(p_cruce)...
          ' P.Mut Bin: ' num2str(p_mut_bin) ' P.Mut.Real: ' num2str(p_mut_real)...

```

```

    ' Con Sharing--> nicho: ' num2str(nicho)];
end

%1er paso--> dialogo de opciones:
opcion = questdlg('Selecciona una opcion...','Menu grafico',...
    'Solo la funcion','Generacion concreta','Animacion','Solo la funcion');

switch opcion
case 'Solo la funcion'
    [x,y]=meshgrid(linspace(xmin,xmax,100));
    z= ((cos(x).*cos(y)).*(x+y)); % Funcion C
    h=figure;
    hold on;
    title(['Funcion C','z = (cos(x)*cos(y))*(x+y)']);
    xlabel('Eje x');
    ylabel('Eje y');
    %view([-37.5,30]);
    view([-62,32]);
    surf(x,y,z);
    text(3.29231,3.29231,6.436174,['\leftarrow max = ' num2str(6.436174)],'FontSize',10)
    grid on;
    hgsave(h,'Funcion_C','-v6');
    print -djpeg Funcion_C;
    hold off;
    pause(3);
    close all Force;
    break
case 'Generacion concreta'
    textos = {'Introduce numero de ciclo entre 1 y ' num2str(max_ciclos) ':',...
        'Introduce numero de generacion entre 0 y ' num2str(max_generaciones) ':'};
    titulo = 'Generacion concreta';
    op = 0;

    while (op==0)
        correcto=0;
        answer=inputdlg(textos,titulo,1);
        ciclo=str2double(answer{1});
        generacion=str2double(answer{2});

        offset = (((max_generaciones*total_variables) +(total_variables+1)) * (ciclo-1));
        puntos =
datos(offset+2:offset+(total_variables*max_generaciones)+(total_variables+1),1:poblacion+1);
        genx = puntos(((generacion*2)-1),:);
        geny = puntos((2*generacion),:);
        h=figure;
        hold on;
        [x,y]=meshgrid(linspace(xmin,xmax,100));
        z= ((cos(x).*cos(y)).*(x+y)); % Funcion C
        grid on;
        title(['Funcion C - Generacion: ' num2str(generacion)];...
            'z = (cos(x)*cos(y))*(x+y)'];str});
        xlabel('Eje x');
        ylabel('Eje y');
        %view([-37.5,30]);
        view([-62,32]);
        surf(x,y,z);
        colormap gray;
        plot3(genx,geny,((cos(genx).*cos(geny)).*(genx+geny)),'g*','LineWidth',3,'MarkerSize',5)
        maxx = genx(poblacion+1);
        maxy = geny(poblacion+1);
        plot3(maxx,maxy,((cos(maxx).*cos(maxy)).*(maxx+maxy)),'r*','LineWidth',3,'MarkerSize',5)

        text(maxx,maxy,((cos(maxx).*cos(maxy)).*(maxx+maxy)),...
            ['\leftarrow mejor fitness = ' num2str(((cos(maxx).*cos(maxy)).*(maxx+maxy))),...
            'FontSize',10)

        print('-djpeg',['Figura C ( ' num2str(ciclo) '- ' num2str(generacion) ' )']);

```

```

hgsave(h,['Figura C (' num2str(ciclo) '-' num2str(generacion) ')'],'-v6');
hold off

button = questdlg('Quieres representar otra generacion?',...
'Otra generacion?...','Si','No','No');
if strcmp(button,'No')
    op=1;
end
end;
close all Force;
break
case 'Animacion'
    op=0;
    while (op==0)
        answer=inputdlg(['Que ciclo deseas ver (Entre 1 y ' num2str(max_ciclos) ')'],'Animacion',1);
        ciclo=str2double(answer{1});
        offset = (((max_generaciones*total_variables) +(total_variables+1)) * (ciclo-1));
        puntos =
datos(offset+2:offset+(total_variables*max_generaciones)+(total_variables+1),1:poblacion+1);
        %h=figure;

        for cont=1:max_generaciones+1
            newplot;
            hold on;
            [x,y]=meshgrid(linspace(xmin,xmax,100));
            z= ((cos(x).*cos(y)).*(x+y)); % Funcion C
            grid on;
            title(['Funcion C - Generacion: ' num2str(cont-1)];...
                'z = (cos(x)*cos(y))*(x+y)';str));
            xlabel('Eje x');
            ylabel('Eje y');
            %view([-37.5,30]);
            view([-62,32]);
            surf(x,y,z);
            colormap gray;
            genx = puntos(((cont*2)-1),:);
            geny = puntos((cont*2),:);

            plot3(genx,geny,((cos(genx).*cos(geny)).*(genx+geny)),'g*','LineWidth',3,'MarkerSize',5)
            maxx = genx(poblacion+1);
            maxy = geny(poblacion+1);
            plot3(maxx,maxy,((cos(maxx).*cos(maxy)).*(maxx+maxy)),'r*','LineWidth',3,'MarkerSize',5)

            text(maxx,maxy,((cos(maxx).*cos(maxy)).*(maxx+maxy)),...
                ['\leftarrow mejor fitness = ' num2str(((cos(maxx).*cos(maxy)).*(maxx+maxy)))],...
                'FontSize',10)

            %Descomentando estas dos lineas guardara todas las
            %imagenes del ciclo
            %print('-djpeg',['Figura A (' num2str(ciclo) '-' num2str(cont) ')'])
            %hgsave(h,['Figura A (' num2str(ciclo) '-' num2str(cont) ')'],'-v6');
            hold off
            pause(0.1);
        end;

        button = questdlg('Quieres ver otra animacion?',...
'Otra generacion?...','Si','No','No');
        if strcmp(button,'No')
            op=1;
            close all Force;
        end
    end;
    close all Force;
end
close all Force;

```

D.3.4. Script para la función D

%Lee los datos del fichero Datos_matlab_<A, B, C, D, E>.txt generado por
 %el código del AG y permite visualizar la gráfica en cuestión y las
 %poblaciones que se han ido generando durante las ejecuciones

%La primera fila es la definición del proceso:
 %<max_ciclos><max_generaciones><poblacion><total_variables><funcion>
 %<p_cruce><p_mut_bin><p_mut_real><sharing><nicho><x1min><x1max><x2min><x2max>...

```
datos = load('Datos_matlab_D.txt');
```

```
max_ciclos = datos(1,1);
max_generaciones = datos(1,2);
poblacion = datos(1,3);
total_variables = datos(1,4);
funcion = datos(1,5);
p_cruce = datos(1,6);
p_mut_bin = datos(1,7);
p_mut_real = datos(1,8);
sharing = datos(1,9);
nicho = datos(1,10);
xmin = datos(1,11);
xmax = datos(1,12);
ymin = datos(1,13);
ymax = datos(1,14);
```

```
if(sharing==0)
    str = ['Pob: ' num2str(poblacion) ' P.Cruce: ' num2str(p_cruce)...
          ' P.Mut Bin: ' num2str(p_mut_bin) ' P.Mut.Real: ' num2str(p_mut_real)...
          ' Sin Sharing'];
else
    str = ['Pob: ' num2str(poblacion) ' P.Cruce: ' num2str(p_cruce)...
          ' P.Mut Bin: ' num2str(p_mut_bin) ' P.Mut.Real: ' num2str(p_mut_real)...
          ' Con Sharing--> nicho: ' num2str(nicho)];
end
```

%1er paso--> dialogo de opciones:

```
opcion = questdlg('Selecciona una opcion...','Menu grafico',...
    'Solo la funcion','Generacion concreta','Animacion','Solo la funcion');
```

```
switch opcion
case 'Solo la funcion'
    x = xmin:0.0001:xmax+1; % Lista de puntos en el eje X
    y = x.*sin(x); %Funcion D
    h=figure;
    hold on;
    title(['Funcion D';'y = x*sin(x)']);
    plot(x,y,'b');
    x=(30.5*pi);
    text(x,(x*sin(x)),[' max=' num2str((x*sin(x)))'],'FontSize',10)
    grid on;
    plot(x,(x.*sin(x)),'r*');
    hgsave(h,'Funcion_D','-v6');
    print -djpeg Funcion_D;
    hold off;
    pause(3);
    close all Force;
    break
case 'Generacion concreta'
    textos = ['Introduce numero de ciclo entre 1 y ' num2str(max_ciclos) ':',...
              ['Introduce numero de generacion entre 0 y ' num2str(max_generaciones) ':'];
    titulo = 'Generacion concreta';
    op = 0;

    while (op==0)
```



```

correcto=0;
answer=inputdlg(textos,titulo,1);
ciclo=str2double(answer{1});
generacion=str2double(answer{2});

offset = (((max_generaciones*total_variables) +(total_variables+1)) * (ciclo-1));
puntos = datos(offset+2:offset+max_generaciones+(total_variables+1),1:poblacion+1);
gen = puntos(generacion,:);
h=figure;
hold on;
x = xmin:0.0001:xmax+1;
y = x.*sin(x); % Funcion B
grid on;
plot(x,y,'b')
title(['Funcion D - Generacion: ' num2str(generacion)];...
    'y = x*sen(x);str});
plot(gen,(gen.*sin(gen)),'g*','LineWidth',2,'MarkerSize',5)
max = gen(poblacion+1);
plot(max,(max.*sin(max)),'r*','LineWidth',2,'MarkerSize',5)
text(max,(max.*sin(max)),...
    ['\leftarrow Mejor fitness = ' num2str(max.*sin(max))],...
    'FontSize',10)

print('-djpeg',['Figura D (' num2str(ciclo) '-' num2str(generacion) ')']);
hgsave(h,['Figura D (' num2str(ciclo) '-' num2str(generacion) ')'],'-v6');
hold off

button = questdlg('Quieres representar otra generacion?',...
    'Otra generacion?...','Si','No','No');
if strcmp(button,'No')
    op=1;
end
end;
close all Force;
break
case 'Animacion'
    op=0;
    while (op==0)
        answer=inputdlg(['Que ciclo deseas ver (Entre 1 y ' num2str(max_ciclos) ')'],'Animacion',1);
        ciclo=str2double(answer{1});
        offset = (((max_generaciones*total_variables) +(total_variables+1)) * (ciclo-1));
        puntos = datos(offset+2:offset+max_generaciones+(total_variables+1),1:poblacion+1);
        %h=figure;

        for cont=1:max_generaciones+1
            newplot;
            gen = puntos(cont,:);
            hold on;
            x = xmin:0.0001:xmax+1;
            y = x.*sin(x); % Funcion A
            grid on;
            plot(x,y,'b')
            title(['Funcion D - Generacion: ' num2str(cont-1)];...
                'y = x*sen(x);str});
            plot(gen,(gen.*sin(gen)),'g*','LineWidth',2,'MarkerSize',5)
            max = gen(poblacion+1);
            plot(max,(max.*sin(max)),'r*','LineWidth',2,'MarkerSize',5)
            text(max,(max.*sin(max)),...
                ['\leftarrow Mejor fitness = ' num2str(max.*sin(max))],...
                'FontSize',10)

            %Descomentando estas dos lineas guardara todas las
            %imagenes del ciclo
            %print('-djpeg',['Figura D (' num2str(ciclo) '-' num2str(cont) ')']);
            %hgsave(h,['Figura D (' num2str(ciclo) '-' num2str(cont) ')'],'-v6');
            hold off
            pause(0.1);
        end
    end
end

```

```

end;

button = questdlg('Quieres ver otra animacion?',...
'Otra generacion?...','Si','No','No');
if strcmp(button,'No')
    op=1;
    close all Force;
end
end;
close all Force;
end

```

D.3.5. Script para la función E

%Lee los datos del fichero Datos_matlab_<A, B , C, D ,E>.txt generado por
 %el codigo del AG y permite visualizar la grafica en cuestion y las
 %poblaciones que se han ido generando durante las ejecuciones

%La primera fila es la definicion del proceso:
 %<max_ciclos><max_generaciones><poblacion><total_variables><funcion>
 %<p_cruce><p_mut_bin><p_mut_real><sharing><nicho><x1min><x1max><x2min><x2max>...

```

datos = load('Datos_matlab_E.txt');

max_ciclos = datos(1,1);
max_generaciones = datos(1,2);
poblacion = datos(1,3);
total_variables = datos(1,4);
funcion = datos(1,5);
p_cruce = datos(1,6);
p_mut_bin = datos(1,7);
p_mut_real = datos(1,8);
sharing = datos(1,9);
nicho = datos(1,10);
xmin = datos(1,11);
xmax = datos(1,12);
ymin = datos(1,13);
ymax = datos(1,14);

if(sharing==0)
    str = ['Pob: ' num2str(poblacion) ' P.Cruce: ' num2str(p_cruce)...
        ' P.Mut Bin: ' num2str(p_mut_bin) ' P.Mut.Real: ' num2str(p_mut_real)...
        ' Sin Sharing'];
else
    str = ['Pob: ' num2str(poblacion) ' P.Cruce: ' num2str(p_cruce)...
        ' P.Mut Bin: ' num2str(p_mut_bin) ' P.Mut.Real: ' num2str(p_mut_real)...
        ' Con Sharing--> nicho: ' num2str(nicho)];
end

```

%1er paso--> dialogo de opciones:

```

opcion = questdlg('Selecciona una opcion...','Menu grafico',...
'Solo la funcion','Generacion concreta','Animacion','Solo la funcion');

```

```

switch opcion
case 'Solo la funcion'
    x = xmin:0.0001:xmax; % Lista de puntos en el eje X
    y = ((cos(x.*200)).*(x<(-pi/400))...
        +((cos(x.*200).*20).*(x>=(-pi/400))&(x<=(pi/400))))...
        +((cos(x.*200)).*(x>(pi/400)))); %Funcion E
    h=figure;
    hold on;
    title(['Funcion E','cos(200x)          -pi/400 < x',...
        'y = cos(200x)*20  -p1/400 <= x <= pi/400',...
        'cos(200x)          x > pi/400']);
    plot(x,y,'b');

```

```

text(0,20, '\leftarrow max = 20', 'FontSize', 10)
grid on;
plot(0,20, 'r*');
hgsave(h, 'Funcion_E', '-v6');
print -djpeg Funcion_E;
hold off;
pause(3);
close all Force;
break
case 'Generacion concreta'
    textos = {'Introduce numero de ciclo entre 1 y ' num2str(max_ciclos) ':',...
              'Introduce numero de generacion entre 0 y ' num2str(max_generaciones) ':'};
    titulo = 'Generacion concreta';
    op = 0;

    while (op==0)
        correcto=0;
        answer=inputdlg(textos,titulo,1);
        ciclo=str2double(answer{1});
        generacion=str2double(answer{2});

        offset = (((max_generaciones*total_variables) +(total_variables+1)) * (ciclo-1));
        puntos = datos(offset+2:offset+max_generaciones+(total_variables+1),1:poblacion+1);
        gen = puntos(generacion,:);
        h=figure;
        hold on;
        x = xmin:0.0001:xmax;
        y = ((cos(x.*200)).*(x<(-pi/400)))...
            +((cos(x.*200).*20).*((x>=(-pi/400))&(x<=(pi/400))))...
            +((cos(x.*200)).*(x>(pi/400))); %Funcion E
        grid on;
        plot(x,y,'b')
        title({'Funcion E - Generacion: ' num2str(generacion)];...
              'cos(200x)          -pi/400 < x';...
              'y = cos(200x)*20  -pi/400 <= x <= pi/400';...
              'cos(200x)          x > pi/400';str});

        plot(gen,((cos(gen.*200)).*(gen<(-pi/400)))...
              +((cos(gen.*200).*20).*((gen>=(-pi/400))&(gen<=(pi/400))))...
              +((cos(gen.*200)).*(gen>(pi/400))))...
              'g*', 'LineWidth', 2, 'MarkerSize', 5)
        max = gen(poblacion+1);
        plot(max,((cos(max.*200)).*(max<(-pi/400)))...
              +((cos(max.*200).*20).*((max>=(-pi/400))&(max<=(pi/400))))...
              +((cos(max.*200)).*(max>(pi/400))))...
              'r*', 'LineWidth', 2, 'MarkerSize', 5)
        text(max,(((cos(max.*200)).*(max<(-pi/400)))...
              +((cos(max.*200).*20).*((max>=(-pi/400))&(max<=(pi/400))))...
              +((cos(max.*200)).*(max>(pi/400))))...
              ['\leftarrow Mejor fitness = ' num2str(((cos(max.*200)).*(max<(-pi/400)))...
              +((cos(max.*200).*20).*((max>=(-pi/400))&(max<=(pi/400))))...
              +((cos(max.*200)).*(max>(pi/400)))))],...
              'FontSize', 10)

        print('-djpeg', ['Figura E (' num2str(ciclo) ' ' num2str(generacion) ')']);
        hgsave(h, ['Figura E (' num2str(ciclo) ' ' num2str(generacion) ')'], '-v6');
        hold off

        button = questdlg('Quieres representar otra generacion?',...
                          'Otra generacion?...','Si','No','No');
        if strcmp(button,'No')
            op=1;
        end
    end;
    close all Force;
    break
case 'Animacion'

```

```

op=0;
while (op==0)
    answer=inputdlg(['Que ciclo deseas ver (Entre 1 y ' num2str(max_ciclos) ')'],'Animacion',1);
    ciclo=str2double(answer{1});
    offset = (((max_generaciones*total_variables) +(total_variables+1)) * (ciclo-1));
    puntos = datos(offset+2:offset+max_generaciones+(total_variables+1),1:poblacion+1);
    %h=figure;

    for cont=1:max_generaciones+1
        newplot;
        gen = puntos(cont,:);
        hold on;
        x = xmin:0.0001:xmax;
        y = ((cos(x.*200)).*(x<(-pi/400)))...
            +((cos(x.*200).*20).*((x>=(-pi/400))&(x<=(pi/400))))...
            +((cos(x.*200)).*(x>(pi/400)))); %Funcion E
        grid on;
        plot(x,y,'b')
        title(['Funcion E - Generacion: ' num2str(cont)];...
            'cos(200x)          -pi/400 < x';...
            'y = cos(200x)*20  -p1/400 <= x <= pi/400';...
            'cos(200x)          x > pi/400';str));
        plot(gen,((cos(gen.*200)).*(gen<(-pi/400)))...
            +((cos(gen.*200).*20).*((gen>=(-pi/400))&(gen<=(pi/400))))...
            +((cos(gen.*200)).*(gen>(pi/400))));...
            'g','LineWidth',2,'MarkerSize',5)
        max = gen(poblacion+1);
        plot(max,((cos(max.*200)).*(max<(-pi/400)))...
            +((cos(max.*200).*20).*((max>=(-pi/400))&(max<=(pi/400))))...
            +((cos(max.*200)).*(max>(pi/400))));...
            'r','LineWidth',2,'MarkerSize',5)
        text(max,(((cos(max.*200)).*(max<(-pi/400)))...
            +((cos(max.*200).*20).*((max>=(-pi/400))&(max<=(pi/400))))...
            +((cos(max.*200)).*(max>(pi/400))));...
            ['\leftarrow Mejor fitness = ' num2str(((cos(max.*200)).*(max<(-pi/400)))...
            +((cos(max.*200).*20).*((max>=(-pi/400))&(max<=(pi/400))))...
            +((cos(max.*200)).*(max>(pi/400))));...
            'FontSize',10)

        %Descomentando estas dos lineas guardara todas las
        %imagenes del ciclo
        %print('-djpeg',['Figura E (' num2str(ciclo) '-' num2str(cont) ')'])
        %hgsave(h,['Figura E (' num2str(ciclo) '-' num2str(cont) ')'],'-v6');
        hold off
        pause(0.1);
    end;

    button = questdlg('Quieres ver otra animacion?',...
        'Otra generacion?...','Si','No','No');
    if strcmp(button,'No')
        op=1;
        close all Force;
    end
end;
close all Force;
end

```