UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

"Plataforma en Java para el Diseño de Circuitos Lógicos Combinatorios, Experimentando con Diferentes Operadores Genéticos"

TESIS

Para obtener el título de:

Ingeniero en Computación

Presenta:

Cruz Pérez Javier

Directores de Tesis:

M.C. Hilda Caballero Barbosa

Dr. Carlos A. Coello Coello

Huajuapan de León, Oax. Julio del 2003.

Agradecimientos

A Jehieli por confiar en mí y darme ánimos para continuar en momentos difíciles.

A mis padres, Javier y Bertha, por haberme apoyado siempre, por haber estado cuando los necesite y por lo que soy, se que lo debo a ellos.

A mi Hermano, Gabriel, por confiar en mí y por darme su apoyo.

Al Dr. Carlos A. Coello Coello, por permitirme ser su tesista, por su orientación y por haber dedicado tiempo en la elaboración de esta tesis.

A la M.I.A. Hilda Caballero Barbosa, por haberme guiado para la elaboración de este trabajo de tesis, así como por la amistad que me brindó.

A Efrén, Eduardo, Gregorio, Nareli, Judith, Armando, Acely y Margarita que me brindaron su amistad, apoyo, sugerencias y por hacer mi estancia en México más llevadera.

A mis amigos, cuyas sugerencias y comentarios hicieron posible mejorar este programa.

A la Universidad Tecnológica de la Mixteca, por permitirme utilizar sus instalaciones y por el apoyo de sus docentes.

Agradezco al CONACyT por la beca otorgada para la realización de esta tesis de ingeniería, la cual se derivó del proyecto titulado "Estudio y Desarrollo de Técnicas Avanzadas de Manejo de Restricciones para Algoritmos Evolutivos en el Contexto de Optimización Numérica" (Ref. 32999-A), cuyo responsable es el Dr. Carlos A. Coello Coello.

Índice

Capítulo 1

Introducción

El adelanto tecnológico conseguido en los últimos 50 años por la humanidad ha sido tal, que es superior a la suma de los avances alcanzados hasta antes de los años 1950s. Esto se debe en gran medida a la electrónica digital, cuya implementación ha permitido progresos importantes en la ciencia, en la industria, en las comunicaciones y en la vida cotidiana. El mayor representante de ello es la computadora digital, herramienta sin la cual es imposible imaginar los logros obtenidos, como por ejemplo el programa espacial, control de tránsito aéreo, automatización de procesos en la industria, la Internet, por mencionar algunos. Todo ello gracias al desarrollo de circuitos integrados pequeños y a bajo costo. El diseño de los circuitos ha requerido de la aplicación de varias técnicas para simplificarlos. En un principio se utilizó el algebra booleana, no obstante, al incrementarse el tamaño y complejidad de los circuitos (mayor número de variables de entrada y mayor número de salidas) fue necesario utilizar otras técnicas. Los mapas de Karnaugh representaron una solución (estos mapas son un procedimiento que facilita la reducción de los circuitos de forma gráfica), sin embargo, éstos dejan de ser prácticos para problemas mayores de seis variables. El método de Quine-McCluskey permitió superar este problema, y además, su implementación en una computadora resulta más simple. Estas son las herramientas más conocidas para el diseño de circuitos lógicos (una mayor explicación de éstas se dará en el capítulo 2).

1.1 Motivación

El diseño de los circuitos lógicos combinatorios (CLCs) ha permitido obtener circuitos pequeños a un menor costo. Las herramientas para lograrlo se han mejorado hasta conseguir sofisticadas técnicas heurísticas de búsqueda como las incorporadas en ESPRESSO [13, 14, 71]. El diseño de CLCs tiene una complejidad $O(\frac{3^n}{n})$, donde n es el número de variables del problema. Esto significa que tendrá un comportamiento exponencial dependiendo del número de entradas, y por lo tanto es considerado un problema NP-completo [14]. Debido a esto se sigue investigando en esta área, y se han propuesto nuevos métodos para diseñarlos.

Algunas de las nuevas técnicas se encuentran en lo que se conoce como *Computación Evolutiva*. Se le llama así porque éstas logran encontrar soluciones a problemas de alta complejidad simulando el proceso evolutivo. La computación evolutiva se conforma de cuatro paradigmas principales: las estrategias evolutivas, la programación evolutiva, la programación genética y los algoritmos genéticos (se explica a grandes rasgos cada uno de ellos en el capítulo 3 y se expone con detalle el algoritmo genético en el capítulo 4). En la literatura se reportan implementaciones de los dos últimos paradigmas para simplificar circuitos combinatorios, los cuales han sido aplicados con resultados aceptables.

A la utilización de métodos evolutivos para el diseño de hardware se le conoce como *Hardware Evolutivo*, el cuál se divide en dos grupos principales, la evaluación intrínseca (o en línea) que es cuando se trabaja directamente con hardware reconfigurable[1] y la extrínseca (o fuera de línea) y es cuando únicamente se labora con simulaciones [71]. Éstas a su vez pueden subdividirse en sistemas análogos y digitales.

1.2 Objetivo

El objetivo de este trabajo es el mejoramiento de la implementación (Diseño Óptimo de Circuitos Lógicos usando Algoritmos Genéticos) propuesta por Coello, Christiansen y Hernández [13, 14].

[1] La solución obtenida es implementada en dispositivos reconfigurables los cuales son evaluados en algunos equipos.

La forma de mejorarla es agregando una interfaz gráfica para facilitar la interpretación de resultados; así como la inserción de nuevos operadores genéticos para hacer un análisis de la influencia de éstos en el desempeño del algoritmo genético. Logrando así, tener una herramienta académica para visualizar su comportamiento con diferentes operadores, y poder realizar un estudio de ello al modificarse sus parámetros de entrada. Con la finalidad de ejecutar el programa vía Internet.

1.3 Alcance del trabajo de tesis

Este trabajo de tesis está situado en el ámbito extrínseco, pretendiendo construir CLCs a nivel de compuertas (AND, OR, NOT, NAND, NOR y XOR).

Se parte de un programa base (el programa está hecho en C para UNIX), el cuál se trasladó al lenguaje Java, con el propósito de poder tener un programa disponible en Internet. Se le agregaron otros operadores, ya que el programa de referencia solo trabaja con un operador de selección, uno de cruza y uno de mutación (se expone con detalle en la sección 5.2). Y se agregó una interfaz gráfica para facilitar la interpretación de los resultados, puesto que el programa inicial carecía de ésta y los resultados eran interpretados manualmente.

Debido a los tipos de representación (la binaria y la entera) con que se optó trabajar, sólo fueron ingresados al programa los operadores genéticos más conocidos de éstas, dando la posibilidad al usuario de elegir la combinación de operadores que mejor considere. También se da la oportunidad de modificar valores importantes del algoritmo genético (probabilidad de cruza, probabilidad de mutación, número de generaciones, tamaño de la población, entre otros) para visualizar el efecto de éstos en su desempeño.

En lo referente a la forma de representar el circuito, que es por medio de una matriz, los circuitos a los que se les puede hallar una solución son aquellos de 10 entradas y 10 salidas como valores máximos. Sin embargo, por la forma de implementar la interfaz, el programa solo permite insertar seis variables de entrada y ocho salidas.

1.4 Estructura de la tesis

El contenido del presente trabajo de tesis está organizado en 7 capítulos.

En el *capítulo 1* se da una introducción de la importancia de la electrónica digital, la motivación para el desarrollo de este trabajo de tesis, así como el objetivo del mismo y las limitaciones que éste tiene. Terminando con una breve descripción de su contenido por capítulos.

En el *capítulo 2* se explican las técnicas tradicionales y más conocidas para el diseño de circuitos lógicos (álgebra booleana, mapas de Karnaugh y el método de Quine-McCluskey).

La descripción de las bases teóricas de la computación evolutiva, así como de sus principales paradigmas se encuentran en el *capítulo 3*. El cual termina con una breve introducción al hardware evolutivo.

El *capítulo 4* contiene lo referente al algoritmo genético y se explican a grandes rasgos sus componentes principales.

En el *capítulo 5* se detalla la implementación del sistema desarrollado, escribiendo porqué la utilización del lenguaje Java, las características del programa base, así como las del nuevo sistema; concluyendo con una breve explicación de las ventajas que ofrece este método en comparación de las técnicas tradicionales.

El *capítulo 6* está dedicado a describir los resultados obtenidos por este nuevo sistema; haciendo un análisis de su comportamiento con los diferentes operadores implementados para circuitos de diferentes niveles de complejidad, comparando resultados con otras técnicas evolutivas ya implementadas.

En el *capítulo 7* se encuentran las conclusiones de este trabajo. En él se resume lo obtenido, remarcando los resultados mas interesantes, y se describe el trabajo que se puede desarrollar en un futuro.

Finalmente, en la sección de apéndices, el *apéndice A* contiene el manual de usuario (guía para utilizar el software implementado); mientras que el *apéndice B* contiene el pseudocódigo del nuevo operador de cruza (múltiples puntos) y en el *apéndice C* se describe el contenido del CD-ROM.

Capítulo 2

Circuitos Lógicos

El avance logrado por la humanidad tanto tecnológica como científicamente se debe en gran medida a la tecnología digital. Por mencionar algunos progresos tenemos: la telefonía satelital, sistemas de guía y navegación, sistemas de radar, control de procesos industriales, instrumentación médica, sistemas militares y la computadora, siendo esta última la más representativa de la tecnología digital [5, 49, 52].

Gracias a la computadora se han logrado los avances significativos de nuestros días. Su diseño, mantenimiento y análisis de operación se hace mediante técnicas y simbologías conocidas como álgebra de Boole [5, 49]. Lleva este nombre en honor del matemático ingles George Boole, quien publicó, en 1854 "Investigación de las leyes del pensamiento, sobre las que se basan las teorías matemáticas de la lógica y la probabilidad" [6], donde Boole realizó un análisis matemático de la lógica. Fue C.E. Shannon [72], quien en 1938, utilizó el álgebra de Boole para representar el comportamiento de los circuitos de interruptores o de conmutación [5, 49], las técnicas utilizadas por Shannon fueron adoptadas universalmente para el diseño y análisis de los circuitos de conmutación. Dada la similitud entre los circuitos antes mencionados con los actuales, las técnicas usadas por Shannon son las mismas que se utilizan para el diseño de los modernos circuitos eléctricos [49].

Las ventajas de tener una técnica matemática para describir la operación interna de una computadora son varias, pero las más significativas son [5]:

- En la realización de cálculos, es más práctico hacerlo con funciones que con esquemas o diagramas lógicos.

- Las expresiones que describen una red de circuitos puede ser simplificada igual como se realiza en el álgebra ordinaria, dando como ventaja tener una mayor economía en la construcción y una mayor confiabilidad en la operación.

Por lo tanto, es de suma importancia el conocimiento del álgebra de Boole en la electrónica digital.

A continuación se dará una breve introducción de las características del álgebra booleana, y se describirán las técnicas más comunes empleadas en la simplificación de los circuitos.

2.1 Álgebra booleana

El álgebra booleana al igual que el álgebra ordinaria se conforma de un conjunto de elementos, de un conjunto de operadores y de postulados y teoremas, pero el álgebra booleana difiere de la ordinaria en que los valores que pueden tener sus variables sólo son dos, 0 ó 1 [49, 56, 71, 75]. Estos valores representan el nivel de voltaje existente en las terminales de entrada o salida de un circuito lógico [49, 71].

2.1.1 Postulados básicos

Antes de mencionar los postulados del álgebra de Boole, se describirá una propiedad del álgebra ordinaria que se aplica con facilidad a la booleana. La propiedad se denomina **cierre** y se define de la siguiente forma [49]:

"Un conjunto de elementos (K) está cerrado con respecto a un operador binario si para cada par de elementos de K, el operador binario especifica una regla para obtener un elemento único de K."

En otras palabras, si al aplicarse un operador binario a cada par de elementos del conjunto K, y los resultados siguen perteneciendo a K, entonces se dice que K está cerrado con respecto a ese operador.

En el álgebra booleana al sólo permitirse dos posibles valores (0 ó 1) a las variables, el resultado siempre será 0 ó 1. Por lo tanto, el álgebra booleana es un sistema algebraico cerrado.

Los postulados del álgebra boolena son [56]:

Postulado 1. Definición. Un *álgebra booleana* es un sistema algebraico cerrado, formado por un conjunto K de dos o más elementos y los dos operadores $\cdot$ y $+$; de manera alternativa, para cada a y b de un conjunto K, $a \cdot b$ pertenece a K y $a + b$ pertenece a K ($+$ se llama OR y $\cdot$ se llama AND).

Postulado 2. Existencia de los elementos 1 y 0. En el conjunto K existen los elementos 1 y 0, únicos, tales que para toda a en K

(a) $a + 0 = a$,

(b) $a \cdot 1 = a$,

donde 0 es el elemento neutro para la operación OR y 1 es el elemento neutro para la operación AND.

Postulado 3. Conmutatividad de las operaciones $+$ y $\cdot$. Para toda a y b en K

(a) $a + b = b + a$,

(b) $a \cdot b = b \cdot a$.

Postulado 4. Asociatividad de las operaciones $+$ y $\cdot$. Para toda a, b y c en K

(a) $a + (b + c) = (a + b) + c$,

(b) $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.

Postulado 5. Distributividad de $+$ sobre $\cdot$ y de $\cdot$ sobre $+$. Para toda a y b en K

(a) $a + (b \cdot c) = (a + b) \cdot (a + c)$,

(b) $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

Postulado 6. Existencia del complemento. Para toda a en K existe un único elemento llamado a' (*complemento* de a) en K tal que

(a) $a + a' = 1$,

(b) $a \cdot a' = 0$.

2.1.2 Dualidad

En el álgebra booleana es de gran importancia el principio de *dualidad* [49, 56]. Este principio establece que, si la expresión es válida en el álgebra booleana, entonces su expresión dual también es válida. Se determina expresión dual al cambio de todos los operadores + por ·, todos los operadores · por +, todos los unos por ceros y todos los ceros por unos [47, 49, 56]. De esta manera, tenemos que el dual de cualquier teorema en el álgebra booleana también es un teorema. Dicho de otra manera, si cualquier enunciado es consecuencia de los postulados del álgebra de Boole, entonces el dual también es consecuencia de estos postulados [47].

2.1.3 Teoremas booleanos

La tabla 2.1 muestra ocho teoremas del álgebra booleana. Los teoremas son listados en pares; cada relación es el dual de su pareja. Por simplicidad no se utilizó el operador ·.

Teorema 1.	Idempotencia.	(a) $a + a = a$	(b) $aa = a$
Teorema 2.	Elementos neutros.	(a) $a + 1 = 1$	(b) $a0 = 0$
Teorema 3.	Involución.	$(a')' = a$	
Teorema 4.	Absorción	(a) $a + ab = a$	(b) $a(a + b) = a$
Teorema 5.		(a) $a + a'b = a + b$	(b) $a(a' + b) = ab$
Teorema 6.		(a) $ab + ab' = a$	(b) $(a + b)(a + b') = a$
Teorema 7.		(a) $ab + ab'c = ab + ac$	(b) $(a + b)(a + b' + c)$ $= (a + b)(a + c)$
Teorema 8.	Teorema de DeMorgan	(a) $(a + b)' = a'b'$	(b) $(ab)' = a' + b'$

Tabla 2.1: Teoremas del álgebra booleana [56].

Estos teoremas pueden ser demostrados haciendo uso de los postulados mencionados en la sección 2.1.1. Como este no es el objetivo de este trabajo, se le recomienda al lector consultar [49, 56, 75].

2.2 Funciones booleanas

Las variables booleanas solo pueden tener el valor de 0 ó 1, por este hecho también se pueden llamar variables binarias. A la expresión formada por variables binarias, por los operadores OR, AND y NOT, paréntesis y signo igual, se le conoce como función booleana. Por ejemplo [49]:

$$F = ab + (cd)'$$

2.2.1 Mintérminos y maxtérminos

Se denomina *mintérmino* o *producto estándar* a cada una de las 2^n posibles combinaciones de las n variables de entrada multiplicadas. Y *maxtérmino* o *suma estándar* a cada una de las 2^n combinaciones de las n entradas sumadas [49]. La tabla 2.2 muestra los mintérminos y maxtérminos cuando son tres variables de entrada. En la tabla también se muestra el símbolo que denota al mintérmino y al maxtérmino. Cada maxtérmino es el complemento de su mintérmino correspondiente.

			Mintérmino		**Maxtérmino**	
A	**B**	**C**	**Término**	**Designación**	**Término**	**Designación**
0	0	0	$A'B'C'$	m_0	$A+B+C$	M_0
0	0	1	$A'B'C$	m_1	$A+B+C'$	M_1
0	1	0	$A'BC'$	m_2	$A+B'+C$	M_2
0	1	1	$A'BC$	m_3	$A+B'+C'$	M_3
1	0	0	$AB'C'$	m_4	$A'+B+C$	M_4
1	0	1	$AB'C$	m_5	$A'+B+C'$	M_5
1	1	0	ABC'	m_6	$A'+B'+C$	M_6
1	1	1	ABC	m_7	$A'+B'+C'$	M_7

Tabla 2.2: Mintérminos y maxtérminos para tres variables [49].

2.2.2 Suma de productos

La función booleana puede ser expresada en forma algebraica mediante una tabla de verdad[1] dada [49], formando un mintérmino para cada combinación de variables que produce un 1 y después sumarlos generando así su expresión. Para visualizarlo utilizaremos la tabla 2.3.

A	B	C	Función F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Tabla 2.3: Tabla de verdad de tres variables.

De modo que la función quedará expresada de la siguiente forma:

$$F = A'B'C' + A'BC + AB'C + ABC' = m_0 + m_3 + m_5 + m_6.$$

Cuando la función booleana se encuentra expresada en suma de mintérminos también suele usarse la siguiente notación:

$$F(A, B, C) = \Sigma(0, 3, 5, 6).$$

Esta propiedad del álgebra booleana implica que cualquier función booleana puede expresarse como una suma de mintérminos o suma de productos.

2.2.3 Producto de sumas

De igual manera ocurre con el complemento. Solo que aquí se tiene el producto de las sumas. Utilizando la tabla 2.3 ejemplificaremos esto. Tomando el complemento de F que se lee:

$$F' = A'B'C + A'BC' + AB'C' + ABC = m_1 + m_2 + m_4 + m_7$$

[1]La tabla de verdad es la forma de representar el comportamiento de un circuito lógico con base en su salida, dependiendo ésta de los valores de entrada [75].

podemos obtener la función siguiente:

$$F = (A + B + C')(A + B' + C)(A' + B + C)(A' + B' + C') = M_1 M_2 M_4 M_7.$$

Esta segunda propiedad dice que cualquier función booleana puede expresarse como un producto de maxtérminos o un producto de sumas. Y de manera similar a la suma de productos suele expresarse como:

$$F(ABC) = \Pi(1, 2, 4, 7).$$

Se dice que una funcion booleana se encuentra en *forma canónica* cuando ésta es expresada como una suma de productos o producto de sumas [49].

Una razón por la cual las expresiones son construidas en forma canónica, es la manera directa en que éstas pueden convertirse en circuitos lógicos. A este tipo de circuitos se les considera circuitos de dos niveles, es decir, la señal solo tiene que atravesar dos compuertas desde que entra hasta que sale [5].

2.3 Compuertas lógicas

La compuerta lógica es el bloque de construcción básico de los sistemas digitales [56, 73, 76], que constituye un bloque de hardware que puede activarse o desactivarse al satisfacerse los requerimientos lógicos de la entrada [56, 71].

A continuación se describirán las compuertas utilizadas para diseñar CLCs.

2.3.1 Compuertas básicas

Existen tres compuertas básicas con las que se puede construir cualquier sistema digital, y estas son la compuerta OR, la compuerta AND y la compuerta NOT [49, 56, 73, 76].

La compuerta OR

Circuito lógico de dos o más entradas, cuya salida es igual a la suma lógica de las entradas [71, 75]. La figura 2.1a muestra su tabla de verdad, en donde podemos observar que la salida será 1 siempre y cuando por lo menos una de las entradas tenga un valor de 1, y será 0 cuando todas las entradas sean 0. Su símbolo gráfico se muestra en la figura 2.1b. Mientras que su función algebraica se visualiza en la figura 2.1c, donde es posible observar que el

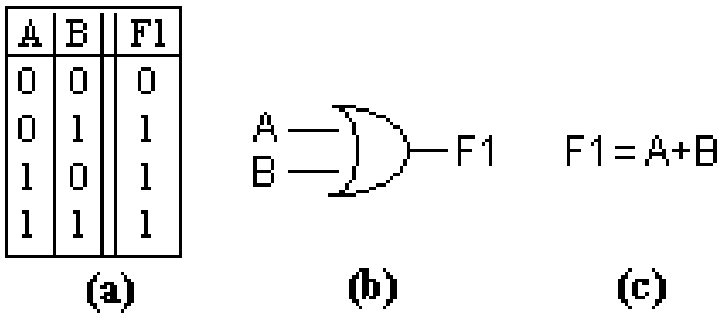


Figura 2.1: (a)Tabla de verdad de la compuerta OR. (b)Símbolo estándar. (c)Función algebraica.

símbolo + representa la operación OR.

La compuerta AND

La salida de la compuerta AND es igual al producto de sus entradas, las cuales pueden ser dos o más. Su forma símbolica se puede observar en la figura 2.2b. Su tabla de verdad (figura 2.2a) indica que la salida será 1 únicamente cuando todas las entradas sean 1. El signo · es el utilizado para representar esta operación, y como la figura 2.2c lo muestra, éste puede ser omitido.

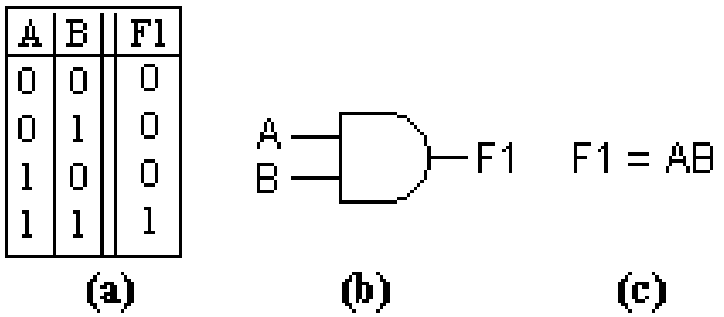


Figura 2.2: (a)Tabla de verdad de la compuerta AND. (b)Símbolo estándar. (c)Función algebraica.

La compuerta NOT o inversor

Ciruito lógico de una única entrada. La salida de este circuito siempre será el valor contrario al que entró, de ahí su nombre. La tabla de verdad, su símbolo correspondiente y su función algebraica se muestran en la figura 2.3 [56, 71, 75].

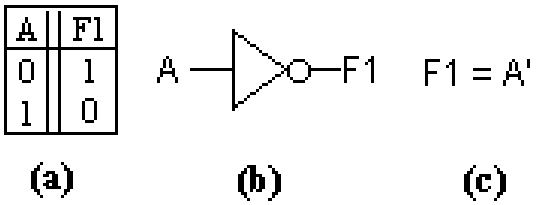


Figura 2.3: (a)Tabla de verdad de la compuerta NOT. (b)Símbolo del circuito. (c)Función booleana.

2.3.2 Compuertas universales

La compuerta NAND y la NOR son muy utilizadas para implementar diseños. El porqué de su extensa utilización, se debe a que una de estas compuertas, ya sea la compuerta NAND o la compuerta NOR, puede ser usada para representar cualquiera de las tres compuertas básicas (figura 2.4); esta característica hace que sean llamadas compuertas universales [56, 75].

Nelson et al. [56] explican que "es más fácil construir un circuito integrado si todos sus componentes son NAND (o todos NOR) en vez de combinar compuertas AND, OR y NOT". Y que la construcción de estos es más rápida y fácil que la de sus equivalentes, teniendo un efecto importante sobre el costo.

La compuerta NAND

Esta compuerta es equivalente a la compuerta AND seguida del inversor (figura 2.5b y 2.5c) [49, 56, 75]. Tiene su propio símbolo (figura 2.5d) y signo (figura 2.5e) [49]. La figura 2.5a muestra la tabla de verdad de esta compuerta, donde la salida será 1 siempre y cuando por lo menos una de las entradas tenga un valor de 0.

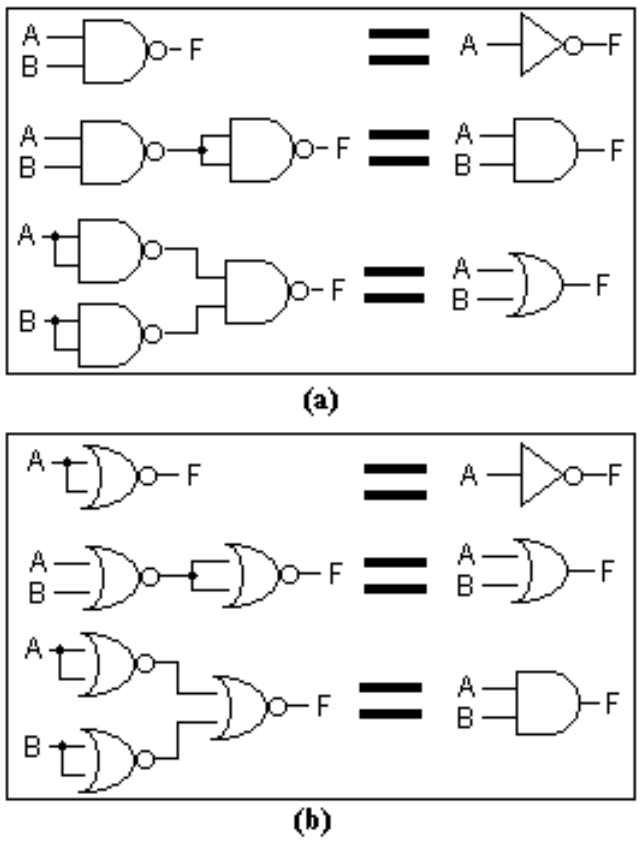


Figura 2.4: (a)Las compuertas NAND se pueden utilizar para poner en práctica cualquier función booleana. (b)Las compuertas NOR se pueden emplear para generar cualquier compuerta básica.

La compuerta NOR

La compuerta NOR es una combinación de la OR seguida de un inversor [49, 56, 75] (figura 2.6b y 2.6c). Esta compuerta es de gran utilidad y su uso es tan extenso que tiene su propio símbolo (figura 2.6d) y signo (figura 2.6e) [49]. La tabla de verdad (figura 2.6a) muestra el comportamiento de esta compuerta, como se puede observar su valor de salida será 1 únicamente cuando los valores de la entrada sean 0.

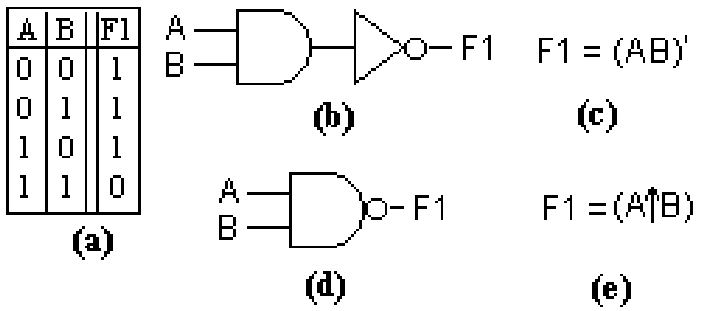


Figura 2.5: (a)Tabla de verdad de la compuerta NAND. (b)Circuito equivalente. (c)Función booleana del circuito equivalente. (d)Símbolo NAND. (e)Función algebraica de la compuerta NAND.

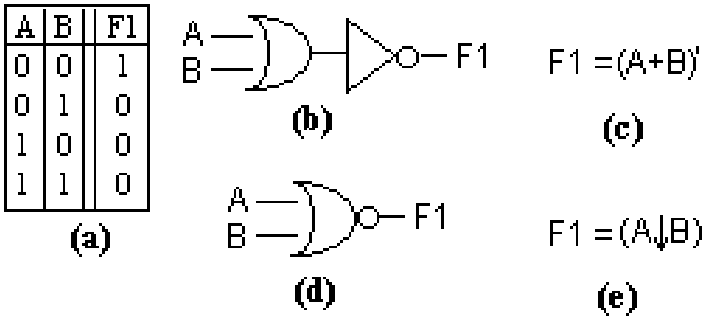


Figura 2.6: (a)Tabla de verdad de la compuerta NOR. (b)Circuito equivalente. (c)Función booleana del circuito equivalente. (d)Símbolo NOR. (e)Expresión booleana de la compuerta NOR.

Tanto la compuerta NAND como la compuerta NOR pueden ser de dos o más entradas.

2.3.3 Otras compuertas

Existen otras compuertas conocidas como OR y NOR exclusivo, cuya construcción tiene un interés práctico [49], puesto que aparecen con gran frecuencia cuando se trata de resolver problemas de operaciones aritméticas digitales y códigos de detección y corrección de errores [49, 56].

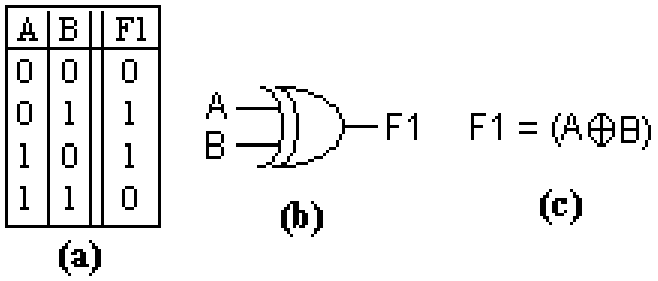
La compuerta OR excluyente (XOR)

La compuerta OR excluyente mejor conocida como XOR tiene como función booleana la siguiente expresión [49, 56, 71, 75]:

$$X = A'B + AB'$$

Tiene su propio símbolo y signo (figura 2.7b y 2.7c) [49, 56, 71, 75, 76].

En su tabla de verdad (figura 2.7a), podemos observar que su salida será 1 únicamente cuando una de sus entradas sea 1, en otras palabras, producirá una salida con valor 1 siempre que las entradas tengan valores opuestos.



A	B	F1
0	0	0
0	1	1
1	0	1
1	1	0

Figura 2.7: (a)Tabla de verdad de la compuerta XOR. (b)Símbolo del circuito. (c)Función algebraica.

La compuerta XOR puede tener más de dos entradas, pero no son comunes, puesto que la construcción en hardware de este tipo de compuertas es muy complicada [49, 71], de hecho son construidas con otros tipos de compuertas [49].

La compuerta NOR exluyente (XNOR)

La negación de la compuerta XOR (figura 2.8b y 2.8c) genera a la compuerta NOR excluyente (XNOR) o también conocida como de equivalencia [49, 56]. El porque de su nombre lo podemos observar en la tabla de verdad de esta compuerta (figura 2.8a), donde se muestra que la salida sera 1 únicamente cuando ambas entradas sean iguales. Al igual que la XOR, esta compuerta tiene su propio signo (figura 2.8e) y símbolo (figura 2.8d).

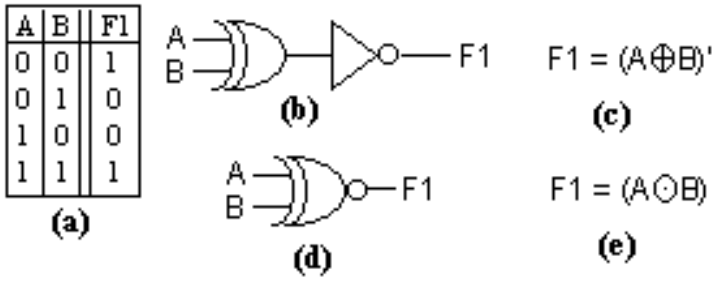


Figura 2.8: (a)Tabla de verdad. (b)Circuito equivalente. (c)Función booleana equivalente. (d)Símbolo del circuito. (e)Función algebraica.

2.4 Simplificación de las funciones booleanas

El procedimiento de diseño del circuito comprende los siguientes pasos [71, 75]:

- Establecer el problema.
- Asignar símbolos literales a las variables de entrada y salida.
- Establecer la tabla de verdad.
- Escribir la forma estándar donde la salida sea 1.
- Escribir la suma de productos.
- Simplificar la expresión de salida.
- Trazar el diagrama lógico para la expresión final.

Partiendo de que contamos con la tabla de verdad, podemos tratar de simplificar las funciones para obtener el diagrama lógico.

2.4.1 Simplificación utilizando el álgebra booleana

Los teoremas del álgebra de Boole pueden utilizarse para simplificar expresiones lógicas [49, 71, 75]. Pero no siempre es obvio el teorema a aplicar para lograr obtener una simplificación mínima. Y no se puede saber con certeza que la reducción conseguida sea la mínima. Sin embargo, con la experiencia se pueden alcanzar resultados razonablemente buenos [49, 71, 75].

Para realizar la reducción por medio del álgebra booleana se requieren de tres etapas [71, 75]:

1. Expresar la función en su forma canónica.
2. Verificar si es posible la factorización.
3. Recurrir al resto de los teoremas

Para ejemplificar lo anterior se simplificará una función a partir de la tabla 2.4 (el ejemplo se tomó de [5]).

A	**B**	**C**	**F**
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Tabla 2.4: Tabla de verdad de tres variables para ejemplificar la simplificación por medio del álgebra de Boole.

El primer paso es obtener la expresión en suma de productos, para hacerlo se toman todos los mintérminos en donde aparece un 1, y se obtiene la siguiente expresión.

$$\begin{aligned} F &= A'BC' + A'BC + ABC' \\ F &= m_2 + m_3 + m_6. \end{aligned} \tag{2.1}$$

El siguiente paso es factorizar. De m_2 y m_3, se factoriza A' quedando:

$$F = A'(BC' + BC) + ABC'.$$

Al aplicar el teorema 6.a (sección 2.1.3) al resultado anterior, se obtiene:

$$F = A'B + ABC'.$$

Es posible factorizar todavía a B logrando:

$$F = B(A' + AC').$$

Haciendo uso del teorema 5.a se consigue la siguiente expresión:

$$F = A'B + BC'$$

para la suma de productos ésta es la expresión mínima, para producto de sumas se puede factorizar otra vez B y se tiene:

$$F = B(A' + C')$$

función que ya no es posible simplificar más.

Como se puede notar esta función booleana es mucho más simple que la función 2.1, y tiene una compuerta menos que su expresión en suma de productos.

Sin embargo, esta no es la única manera de reducir la expresión, a continuación se mostraran los pasos para simplificar por otro camino.

$F = A'BC' + A'BC + ABC'$	Se factoriza $A'B$
$F = A'B(C + C') + ABC'$	Postulado 6.a
$F = A'B + ABC'$	Se factoriza B
$F = B(A' + AC')$	Teorema 5.a
$F = B(A' + C').$	

Como puede observar de esta manera se llega directamente a la función más simple. Y como se había comentado desde un inicio, no es fácil simplificar mediante el álgebra booleana, ya que encontrar el teorema o postulado adecuado a utilizar no es una tarea sencilla .

2.4.2 Mapas de Karnaugh

El mapa de Karnaugh o mapa K, es una técnica gráfica que permite simplificar una ecuación lógica de una manera sencilla y ordenada. Este método puede ser utilizado para solucionar problemas con cualquier número de variables, pero su utilidad práctica está limitada a seis variables [5, 47, 49, 71, 75].

El mapa se compone de 2^n cuadrículas, que son todas las posibles combinaciones de las n variables binarias de entrada.

Los cuadros del mapa K son marcados de modo que los cuadrados horizontales adyacentes difieren únicamente en una variable y de igual forma los cuadrados verticales. En la figura 2.9 se muestran los mapas utilizados para dos, tres y hasta seis variables de entrada.

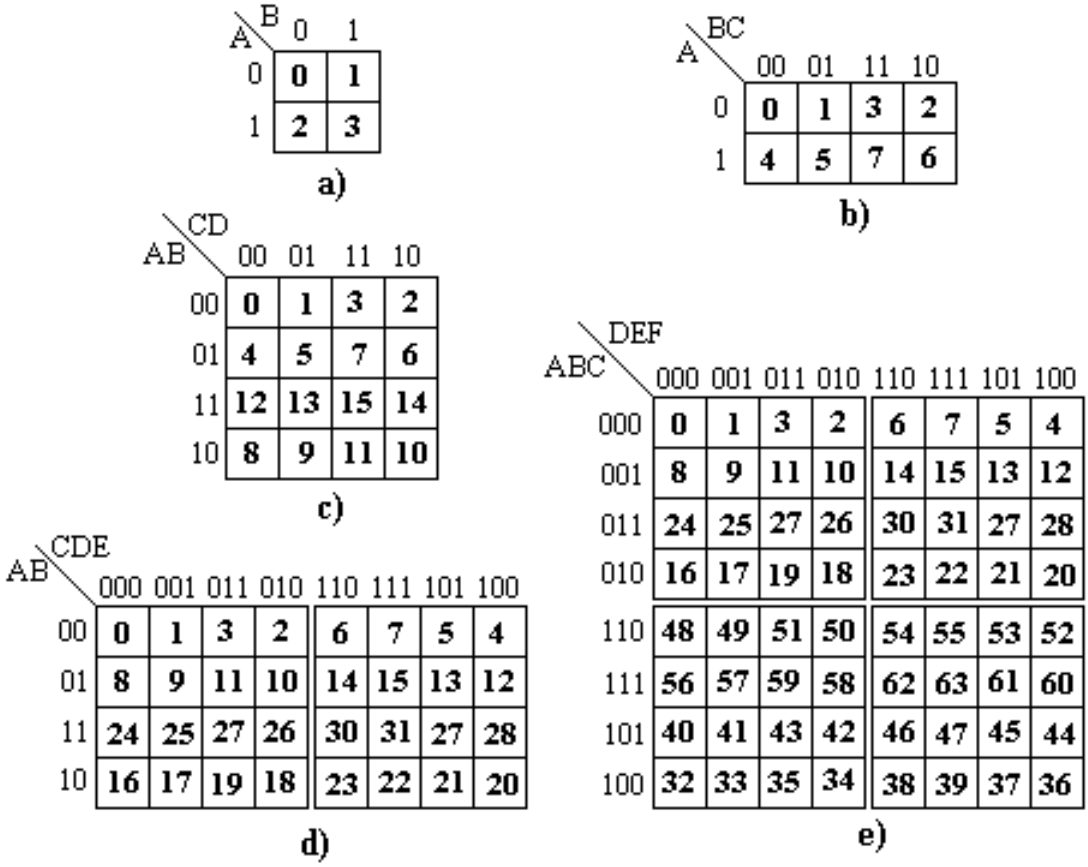


Figura 2.9: Mapas de Karnaugh para (a)dos, (b)tres, (c)cuatro, (d) cinco y (e) seis variables.

Los números dentro de las casillas representan al mintérmino al que pertenece cada cuadro. Como puede ver en la figura 2.9 es sencillo hacer el mapa hasta cuando tiene cuatro varibles de entrada, para los mapas de cinco y seis entradas se debe tener más cuidado, y el proceso de simplificación se complica.

Para simplificar una función se deben seguir los siguientes pasos [75]:

- Construir el mapa K y colocar unos en sus respectivos cuadros de acuerdo a la tabla de verdad. Y el resto se rellena con ceros.
- Examinar el mapa para ver si hay unos adyacentes y repetir aquellos unos que no sean adyacentes a ningún otro uno. A éstos se llaman *unos aislados*.
- Buscar los unos adyacentes y agruparlos ya sea en pares, en grupos de cuatro o de ocho unos, asegurando utilizar el menor número de agrupaciones.
- Formar la suma OR de los términos encontrados por cada agrupamiento.

Los unos se agrupan únicamente de manera horizontal o vertical.

Para ejemplificar mejor este método se usará la función 2.1.

Primero se dibujará la cuadrícula y se llenará de acuerdo a la tabla de verdad, después se agruparán los unos. La figura 2.10 muestra encerrados los unos adyacentes y la función resultante de la reducción.

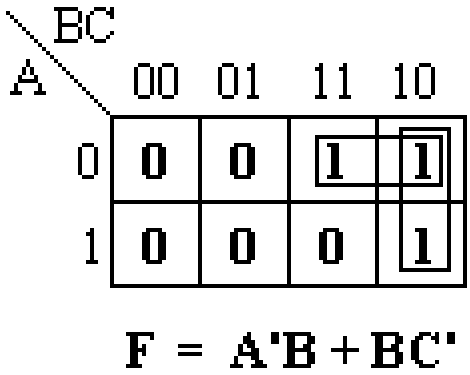


Figura 2.10: Simplificación con mapas de Karnaugh.

Como puede observar simplificar en este método es muy sencillo. Sin embargo, el resultado que da siempre es en forma canónica (en este caso suma de productos, también existe el proceso para obtener la solución en forma de producto de sumas) y éste no siempre será el mínimo. El resultado del ejemplo todavía se puede reducir como fue visto en la sección anterior.

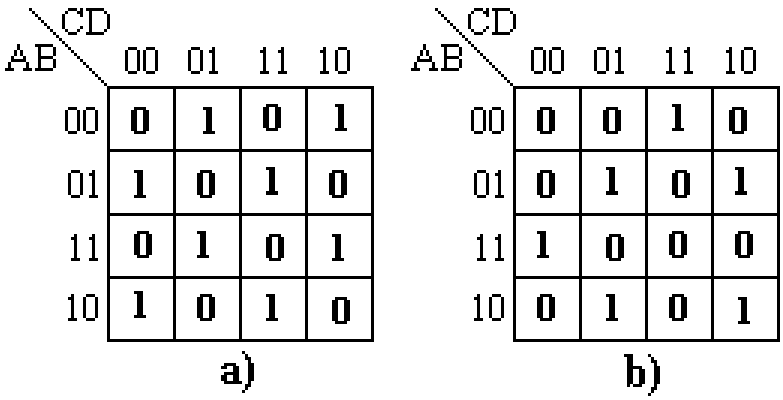


Figura 2.11: Problemas que los Mapas de Karnaugh no simplifican.

Otra desventaja de este método es que es imposible utilizarlo para resolver funciones como las que se muestran en las figuras 2.11a y 2.11b. Como es fácil notar no existe algún uno adyacente para simplificarse. Existe un método llamado anillo de mintérminos en el mapa K, el cuál permite reducir aún más las funciones, sin embargo, es complicado. Este método se creó con el propósito de utilizar la compuerta XOR dentro de las funciones booleanas [50].

Este método por ser gráfico es de difícil programación, y se utiliza más el método que explicaremos a continuación para este fin. Sin embargo, puede revisar una implementación del método del mapa de Karnaugh en [82].

2.4.3 Método de Quine-McCluskey

Como se ha hecho notar el método del mapa de Karnaugh ya no es fácil de utilizar cuando se exceden las seis variables, cuando esto ocurre el mapa se convierte en un procedimiento de ensayo y error, que depende de la habilidad del usuario para reconocer ciertos patrones [49].

El método de Quine-McCluskey es una técnica tabular que garantiza encontrar la expresión mínima en forma canónica para una función, por medio de un conjunto de pasos específicos, superando la limitante del mapa K (respecto al número de variables con las que puede trabajar) y su mayor ventaja es que puede ser programado de manera sencilla [49].

El método consta de dos partes, la primera es hallar los términos candidatos para la reducción, éstos se denominan implicantes primos. La segunda fase consiste en elegir de entre los implicantes primos, a aquellos que puedan dar la expresión con el menor número de variables.

Para explicar este método se usará la siguiente ecuación (el ejemplo se extrajo de [71]):

$$F = \Sigma(1, 5, 6, 7, 11, 12, 13, 15).$$

Lo primero que se tiene que hacer es construir una tabla en la que cada fila es un mintérmino.

Los mintérminos son agrupados de acuerdo al número de variables complementadas. Es decir, se comenzará con el mintérmino que esté formado por n variables complementadas (m_0), claro, si existe. Se continuará con el mintérmino con $n-1$ variables complementadas (m_1), y así sucesivamente hasta llegar al mintérmino m_{n-1}. La tabla 2.5 muestra lo anterior.

	A	**B**	**C**	**D**	**F**
1	0	0	0	1	$\surd$
5	0	1	0	1	$\surd$
6	0	1	1	0	$\surd$
12	1	1	0	0	$\surd$
7	0	1	1	1	$\surd$
11	1	0	1	1	$\surd$
13	1	1	0	1	$\surd$
15	1	1	1	1	$\surd$

Tabla 2.5: Tabla para reducción de términos.

El siguiente paso consiste en comparar el implicante primo de un bloque con todos los del siguiente bloque, y elegir aquellos que solo difieren en una variable y con esos términos generar otra tabla, los términos usados son marcados. La tabla 2.6 lo ejemplifica.

	A B C D	**F**
1,5	0 - 0 1	
5,7	0 1 - 1	√
5,13	- 1 0 1	√
6,7	0 1 1 -	
12,13	1 1 0 -	
7,15	- 1 1 1	√
11,15	1 - 1 1	
13,15	1 1 - 1	√

Tabla 2.6: Tabla para reducción de términos, segundo paso.

Se repite la selección de mintérminos que solo difieran en un solo valor, generando la tabla 2.7.

	A B C D
5,7,13,15	- 1 - 1
5,13,7,15	- 1 - 1

Tabla 2.7: Tabla para reducción de términos, tercer paso.

Como se puede observar ya no es posible reducir más (los términos que quedan difieren entre de sí por más de un elemento), lo que se hace es escribir en una tabla los términos que no fueron marcados para la simplificación (tabla 2.8).

	A B C D	Término
1,5	0 - 0 1	$A'C'D$
6,7	0 1 1 -	$A'BC$
11,15	1 - 1 1	ACD
12,13	1 1 0 -	ABC'
5,7,13,15	- 1 - 1	BD

Tabla 2.8: Implicantes primos.

En este caso se llegó a utilizar tres tablas, pero en general, el proceso se continúa hasta donde ya no haya emparejamientos (cuando los términos tengan más de un elemento diferente entre de sí).

El segundo paso consiste en elegir de entre los implicantes primos, a aquellos que puedan dar la expresión con el menor número de variables, así que para tal efecto se construye una matriz como la que muestra la tabla 2.9 donde cada columna corresponde a un mintermino, y cada renglón contiene a todos los implicantes primos encontrados.

		1	5	6	7	11	12	13	15
$A'C'D$	1,5	X	X						
$A'BC$	6,7			X	X				
ACD	11,15					X			X
ABC'	12,13						X	X	
BD	5,7,13,15		X		X			X	X

Tabla 2.9: Tabla de implicantes primos, último paso del método Quine-McCluskey.

Se seleccionan a todas aquellas columnas que solo tengan un elemento en ellas, así se tiene la certeza de que estos valores son representados por la función, en este caso es m_1, m_6, m_{11} y m_{12}, como estos por renglón cubren a m_5, m_7, m_{13} y m_{15}, el término que los representa es innecesario y se obtiene la siguiente función:

$$F = A'C'D + A'BC + ACD + ABC'.$$

Esta es la función mínima para éste problema, sin embargo, si se hace uso de la factorización, se puede reducir el número de compuertas obteniendo lo siguiente:

$$F = D(A'C' + AC) + B(A'C + AC'),$$

se reducen de 11 operaciones (ocho ANDs y tres ORs) a nueve (seis ANDs y tres ORs), y si todavía se aplica el operador XOR obtenemos:

$$F = D(A \oplus C)' + B(A \oplus C),$$

con lo cual se reduce a seis operaciones (dos ANDs, dos XORs, un OR y un NOT, en este caso se contabiliza el inversor por negar una operación y no a una variable).

Como se muestra la expresión obtenida por el método de Quine-McCluskey no siempre es la menor (para formas canónicas sí) en lo referente al número de compuertas u operaciones. Generalmente la implementación del método de Quine-McCluskey se realiza con los pasos que se explicaron en esta sección. Como puede observar no es posible implementar la reducción de circuitos por medio de la compuerta XOR, sin embargo, en [78] se explica la manera de hacerlo.

Capítulo 3

Computación Evolutiva

El término Computación Evolutiva[1] (CE) se refiere al conjunto de técnicas de búsqueda, optimización y aprendizaje de máquina que emulan el proceso de evolución de las especies [19, 37, 53, 58, 71, 77].

A continuación se proporcionarán algunos antecedentes de la teoría de la evolución, así como una breve explicación de los diversos paradigmas que existen en la CE. Se concluirá con una corta introducción al hardware evolutivo.

3.1 Bases biológicas, antecedentes históricos

Una de las primeras teorías que pretendían explicar la evolución de las especies, fue la propuesta por el zoólogo francés Jean Baptiste Antoine de Monet (Caballero de Lamarck). Lamarck argumentaba que las características adquiridas por los individuos a lo largo de su vida, son transferidas genéticamente y heredadas a sus descendientes de manera directa. Es decir, que las variaciones en el ambiente influyen directamente sobre las características de los individuos, provocando un cambio en su comportamiento, pues cuanto más se usa una estructura más se acentúa y viceversa [45].

El científico alemán Augusto Weismann demostró que la teoría de Lamarck estaba equivocada. Weismann explica que una vez iniciada la fertilización existen dos procesos de división celular. Uno que lleva al cuerpo o "soma"[2]

[1] También es conocido como Algoritmos Evolutivos o Técnicas Evolutivas.
[2] Conocido en la actualidad como fenotipo.

(quien no transmite información hereditaria), y el otro proceso que genera los gametos[3] (quienes transmiten la información hereditaria), los cuales forman el punto de inicio de la siguiente generación; esta idea es conocida como teoría del germoplasma. Weismann afirma que la selección natural es el único mecanismo capaz de alterar la composición genética de un organismo [79].

La mayor aportación a la teoría de la evolución fue hecha por el biólogo inglés Charles Darwin, con su libro "El Origen de las Especies", publicado en 1859, en donde explica que una especie que no sufriera cambios se volvería incompatible con su medio ambiente, puesto que éste tiende a cambiar con el tiempo. También se afirma que las similitudes entre hijos y padres se debían a que ciertas características eran heredadas de generación en generación, pero que ocurrían cambios cuya finalidad era hacer a los nuevos individuos más aptos para sobrevivir. Darwin afirmaba que la selección natural era el proceso de mayor importancia dentro de la evolución [22].

Otra aportación significativa la realizó el monje austriaco Gregor Mendel[4]. Mendel al experimentar con plantas de chícharos descubrió tres leyes básicas de la herencia: la Ley de Segregación, la Ley de Independencia y la Ley de la Uniformidad. Esta última establece que las características de los padres heredadas a los hijos dependen de los genes dominantes o recesivos. Mendel concluyó que no existen mezclas de genes, sino que sólo se combinan en la reproducción conservando la individualidad de generación en generación [51].

Las bases teóricas propuestas por Weismann, Darwin y Mendel, son el sustento del paradigma conocido como Neo-Darwinismo, el cual establece que el proceso evolutivo es un proceso de adaptación al medio ambiente, donde los individuos más aptos sobrevivirán heredando las características de sus padres [19, 71]. En otras palabras, la evolución de las especies es guiada por los siguientes procesos: reproducción, mutación, competencia y selección [53, 58, 60, 77].

Dependiendo del punto de vista con que se observe a la evolución, ésta puede ser tomada como un proceso de aprendizaje (W. E. Cannon [9]) o como un proceso de optimización (H. J. Bremermann [7, 33]).

[3]Llamado actualmente genotipo.

[4]Considerado el padre de la genética.

Fogel lo ve como un proceso de optimización [25] y lo explica de la siguiente manera:

"Si consideramos a la evolución natural como un proceso de optimización, debido a que la evolución ha sido capaz de optimizar organismos hasta hacerlos aptos para sobrevivir, entonces, de modelarse ésta adecuadamente, puede emplearse para encontrar la mejor solución de un problema, es decir, la solución óptima."

3.2 Computación evolutiva

La CE se basa en el paradigma del Neo-Darwinismo y pretende simular el proceso evolutivo en la computadora [19, 58, 71, 77]. Para conseguir su objetivo requiere los siguientes elementos [54]:

- Una representación para las soluciones potenciales al problema.
- Una manera de crear una población inicial de dichas soluciones potenciales.
- Una función de evaluación que juega el papel del ambiente, comparando las soluciones en términos de su aptitud.
- Operadores genéticos que alteran la composición de la descendencia.
- Valores para los parámetros que usa la técnica (tamaño de la población, probabilidades de aplicar los operadores genéticos, entre otros).

La CE se ha aplicado en problemas de búsqueda, de optimización y aprendizaje de máquina, donde las soluciones son difíciles de hallar por medio de técnicas convencionales, debido a que los espacios de búsqueda son extremadamente grandes, complejos y frecuentemente con restricciones difíciles de satisfacer [19, 58, 71].

Los algoritmos empleados en la CE manipulan un conjunto de soluciones potenciales, lo que implica un alto grado de paralelismo, puesto que se exploran varias regiones del espacio de búsqueda a la vez. Sus operadores son probabilísticos y no determinísticos, lo que evita que queden atrapados en óptimos locales fácilmente [19, 71, 77].

Dentro de la inteligencia artificial la CE es considerada como un conjunto de técnicas heurísticas sub-simbólicas, es decir, representan el conocimiento de manera numérica y no simbólica (a diferencia de los sistemas expertos que utilizan la representación simbólica) [19, 37, 71, 77].

Las principales variantes de las técnicas evolutivas son [18, 37, 53, 60, 71, 77]:

- Programación evolutiva.
- Estrategias evolutivas.
- Programación genética.
- Algoritmos genéticos.

3.2.1 Programación evolutiva

Lawrence J. Fogel es quien propuso la Programación Evolutiva (PE) en 1964 [27]. En esta técnica se ve a la inteligencia como un comportamiento adaptativo [18, 53, 77].

Fogel propone la PE para resolver problemas de predicción de secuencias. Esta técnica consistía básicamente en evolucionar autómatas de estados finitos, con la finalidad de que éstos fueran capaces de predecir los símbolos que recibirían. Los autómatas tenían un valor de aptitud de acuerdo a su facultad de predecir símbolos [27]. Lawrence utilizó un modelo de mutación para alterar los estados y las transiciones de los autómatas. Sin embargo, como se pretendía modelar la evolución de las especies, no usó ningún operador de recombinación (y sigue siendo la principal característica de la PE), ya que diferentes especies no se cruzan entre sí [19, 53, 58, 60, 71, 77].

El algoritmo básico de la PE es [27]:

- Generar una población inicial en forma aleatoria.
- Aplicar el operador de mutación (en esta técnica este operador es el principal [53]). Cada padre genera sólo un hijo.
- Calcular la aptitud de los individuos y realizar el proceso de selección (generalmente se utiliza el torneo estocástico [18, 60]).
- Reemplazar la población actual con la obtenida en la selección.

Originalmente la PE se utilizó para resolver problemas de predicción. Actualmente su uso se ha extendido a problemas de optimización continua de parámetros, planeación de rutas, control automático, diseño y entrenamiento de redes neuronales, reconocimiento de patrones, entre otros [26, 28].

3.2.2 Estrategias evolutivas

Las Estrategias Evolutivas (EEs) son un método que fue desarrollado por Paul Bienert, Ingo Rechenbergf y Hans-Paul Schwefel en la Universidad Técnica de Berlín en el año de 1963 [70]. Las estrategias fueron concebidas para solucionar problemas hidrodinámicos con un alto grado de complejidad [53, 58, 71, 77].

En este paradigma existen los operadores de recombinación, dado que se simula la evolución a nivel de los individuos. Sin embargo, el operador de mutación es el más importante, ya que el algoritmo progresa según la aplicación de este operador en los individuos más aptos. Las variables de control son usadas para producir nuevos cambios aleatorios en las variables objetivo y después las propias variables de control son mutadas. La forma original (1+1)-EE usaba sólo un padre que generaba un solo hijo. El hijo se mantenía si era mejor que el padre, de lo contrario era eliminado (selección extintiva) [18, 19, 60, 71].

En la (1+1)-EE, un nuevo individuo se genera usando [18, 19, 71]:

$$X^{t+1} = X^t + N(0, \sigma)$$

donte t se refiere a la generación (o iteración) en la que se encuentra, y $N(0, \sigma)$ es un vector de números Gaussianos independientes, con una media de cero y desviación estándar σ.

El concepto de población es introducido por Rechenberg [59], al proponer la estrategia $(\mu + 1) - EE$, en la cual hay μ padres y se genera un solo hijo (a este tipo de cruza se le conoce como panmítica [18, 19]), el cual puede reemplazar al peor padre de la población.

Mientras que Schwefel [68, 69] introduce el uso de múltiples hijos en las $(\mu + \lambda) - EE$ y $(\mu, \lambda) - EE$. La notación representa la manera en que será utilizada la selección [18]:

- "En el primer caso (selección "+"), los μ mejores individuos obtenidos de la unión de padres e hijos sobreviven."
- "En el segundo caso (selección ","), sólo los μ mejores hijos de la siguiente generación sobreviven."

Sus operadores de selección son determinísticos [19, 53, 60, 71, 77].

Este paradigma se ha aplicado a problemas de ruteo y redes, bioquímica, óptica, diseño de ingeniería y magnetismo [18, 29], por mencionar algunos ejemplos.

3.2.3 Programación genética

La Programación genética (PG) es una técnica que propusieron N. L. Cramer y John Koza (de manera independiente). Ellos sugirieron una estructura de árbol para representar un programa en un genoma [37, 52, 71, 77]. El trabajo de Koza se diferencia del de Cramer, en que Koza logra automatizar la función de aptitud, siendo ésta la principal razón por la cual esta propuesta se ha popularizado [18, 71, 77].

Los individuos en la PG son programas de computadora estructurados jerárquicamente. Los individuos se forman mediante conjuntos de términos y funciones, los cuales actúan como primitivas que sirven de base para la construcción de programas. El conjunto de términos se compone de las variables, constantes o funciones de aridad cero que sirven como argumentos de las funciones. Los términos son considerados como hojas en la estructura de árbol. Mientras que el conjunto de funciones está compuesto por los operadores aritméticos, los operadores binarios o funciones de dominio específico, y en el árbol se les conoce como nodos de tipo función [42, 43, 44].

El algoritmo básico para la PG [4, 42, 43, 44] se muestra a continuación:

- Inicializar la población.
- Evaluar los programas en la población existente y asignar un valor de aptitud a cada individuo.
- Hasta que la nueva población no sea completada:
  - Seleccionar uno o varios individuos en la población aplicando un proceso de selección.

  - Ejecutar los operadores genéticos en el o los individuos seleccionados de la población.
  - Insertar a los nuevos individuos en la nueva población.

- Reemplazar la población existente con la nueva población, hasta cumplir el criterio de terminación.

- Presentar al mejor individuo de la población.

Los métodos de selección usados son: selección proporcional, selección mediante torneo, selección de estado uniforme (los algoritmos genéticos utilizan los mismos métodos, los cuales se explicarán con más detalle en el capítulo siguiente) [42, 43, 44].

Para implementar el operador de cruza se deben seguir los siguientes pasos [18, 42, 43, 44]:

- Seleccionar dos individuos como padres.

- Seleccionar aleatoriamente un subárbol o segmento de instrucciones.

- Intercambiar los subárboles o segmentos de código entre los dos padres.

- Evitar sustituciones de nodo terminal en el nodo raíz.

Dentro de la PG existen cuatro operadores secundarios (los cuales son aplicados en un porcentaje bajo a la población) que son [42, 43, 44]:

1. Mutación. Se selecciona un nodo al azar y el subárbol es cambiado por uno nuevo generado aleatoriamente.

2. Permutación. Se selecciona un nodo al azar y se reordenan los argumentos del subárbol.

3. Edición. Se elige un punto al azar y se reduce de acuerdo a un conjunto de reglas. Por ejemplo: (A * A) = A.

4. Encapsulamiento. Se identifican los subárboles potencialmente reutilizables.

La PG ha sido aplicada en el procesamiento de señales e imágenes, en el diseño de circuitos electrónicos (tando digítales [43, 44], como analógicos [44]), control y robótica, entre otros [77].

3.2.4 Algoritmos genéticos

En el siguiente capítulo se tratará con más detalle este tema. Por el momento sólo se comentará que fué desarrollado por John Holland y sus estudiantes en la universidad de Michigan a principios de los 1960s [35] e inicialmente fueron denominados "planes reproductivos".

Este algoritmo tiene dos principales características: utiliza una representación de cadena binaria de longitud fija (cromosoma) y utiliza en gran medida la recombinación. Al igual que las EEs, el Algoritmo Genético (AG) realiza la simulación de la evolución a nivel de los individuos [19, 60, 71].

El algoritmo básico de un AG es el siguiente [8]:

- Generar aleatoriamente una población inicial de cromosomas.
- Calcular la aptitud de cada individuo.
- Seleccionar (por lo general en forma probabilística) a los individuos con base en su aptitud.
- Aplicar los operadores genéticos de cruza y mutación para generar la siguiente población.
- Reemplazar la población actual con la obtenida en la selección.
- Repetir el ciclo nuevamente hasta que cierta condición se satisfaga.

El AG ha sido utilizado en aprendizaje de máquina, en bases de datos, en el reconocimiento de patrones, en predicción, en la planeación de movimientos de robots, en optimización estructural, entre otros [18, 33, 53, 58, 60].

3.3 Hardware evolutivo

El propósito del Hardware Evolutivo es el diseño de circuitos eletrónicos por medio de un proceso de selección natural [55].

George J. Friedman fue uno de los primeros en aplicar técnicas evolutivas a la robótica. Propuso un mecanismo para construir, probar y evaluar circuitos en forma automática, utilizando mutaciones aleatorias y procesos de selección [30].

La idea se basa en codificar los circuitos en un cromosoma, usar un proceso de ensamble y prueba junto a un proceso evolutivo, permitiendo diseñar circuitos con distintos grados de complejidad [36, 40, 43, 44].

Cuando el proceso de evolución emplea sólo compuertas lógicas básicas se dice que está dirigido a nivel de compuertas, pero cuando se utilizan las compuertas y a su vez se construyen nuevos módulos compuestos con ellas, entonces se encuentra dirigido a nivel de funciones [36, 40, 55].

En la actualidad el hardware evolutivo también es dividido de acuerdo al proceso de evaluación [40]:

- Extrínseca. Las soluciones se construyen y evalúan como modelos (simulaciones).
- Intrínseca. Las soluciones son implementadas en dispositivos reconfigurables, los cuales son evaluados en algunos equipos.
- Mixtrínseca. Es una población compuesta de modelos y dispositivos reconfigurables.

En el diseño de circuitos lógicos a nivel de compuertas, existen implementaciones en Sistemas de Hormigas [16, 17, 52], también en la PG con distintas representaciones [40, 42, 43, 44, 55, 61, 71] y en AG [11, 13, 14, 15, 37].

Capítulo 4

Algoritmo Genético

Este capítulo está dedicado al AG, que es probablemente el paradigma más conocido de la CE [10, 18, 19, 60, 71, 77].

4.1 ¿Qué es un AG?

John Koza define un AG de la siguiente manera [42, 43, 44]:

"El algoritmo genético es un algoritmo matemático altamente paralelo que transforma un conjunto de objetos matemáticos (típicamente cadenas de caracteres de longitud fija que se ajustan al modelo de las cadenas de cromosomas), cada uno de los cuales se asocia con una aptitud[1], en una población nueva (es decir, la siguiente generación) usando operaciones modeladas de acuerdo al principio Darwiniano de reproducción y sobrevivencia del más apto y tras haberse presentado de forma natural una serie de operaciones genéticas (notablemente la recombinación sexual)."

Mientras que Goldberg [33] lo define como:

"Algoritmos de búsqueda basados en el mecanismo de selección natural y genética natural. Combinan la supervivencia de la más apta entre una estructura dada de cadenas, con un intercambio aleatorio de información para conformar un algoritmo de búsqueda con algo del talento de la búsqueda

[1]La aptitud es el valor asignado a cada individuo y que indica qué tan bueno es éste para la solución de un problema.

humana."

El AG, al ser una técnica heurística estocástica no requiere de información específica para guiar la búsqueda. Para implementar un AG que resuelva cualquier problema, en general se requieren de los siguientes componentes [12, 54]:

- Una representación de soluciones potenciales al problema.
- Una forma de crear una población inicial de soluciones potenciales (esta población inicial suele generarse de forma aletaoria).
- Una función de evaluación.
- Operadores genéticos que alteran la composición de los descendientes (normalmente se usan la cruza y la mutación).
- Valores para los diversos parámetros utilizados por el algoritmo genético (tamaño de la población, probabilidad de cruza y mutación, número máximo de generaciones, entre otros).

El algoritmo básico para un AG, fue descrito en el capítulo anterior y ese algoritmo es conocido como AG simple. Existe otra variante del AG: en él se pasa intacto a la siguiente generación al mejor individuo de la población, es decir, no se elige para cruzarse ni mutarse, a esta variante se le conoce como AG con elitismo [10, 18, 60]. Se ha demostrado que este tipo de algoritmo (AG con elitismo) converge al óptimo [18, 60, 63], mientras que el simple no; de ahí la utilización de esta variante en el programa desarrollado como parte de esta tesis.

La tabla 4.1 muestra las similitudes y diferencias que tiene el AG con los otros paradigmas de la CE.

	Algoritmo Genético	**Estrategias Evolutivas**	**Programación Evolutiva**	**Programación Genética**
Representación	Binaria	Real	Real	Árbol
Selección	Probabilística, basada en la preservación	Determinística extintiva o basada en la preservación	Probabilística Extintiva	Probabilística, basada en la preservación
Recombinación	Cruza uniforme, cruza de 1 y de 2 puntos. Es únicamente sexual y es el operador principal	Discreta e intermedia, sexual y panmítica, importante para la autoadaptación	Ninguna	Selección de subárboles
Mutación	Mutación uniforme y aleatoria. Es el operador secundario	Gaussiana. Es el operador principal	Gaussiana. Es el único operador	Selección de un nodo al azar y se modifica por un nuevo árbol

Tabla 4.1: Tabla comparativa de los principales paradigmas de la CE. [18]

4.2 La representación

Un cromosoma es una estructura de datos que representa a un individuo de la población, donde cada posición es llamada gen, y el valor que puede tomar cada gen se denomina alelo [10, 18, 48, 53, 77].

La representación usada tradicionalmente para codificar el conjunto de soluciones es el esquema binario, en la cual un cromosoma es una cadena de la forma (b_1,b_2,...,b_m), donde b_1,b_2,...,b_m pueden tomar un valor de cero o uno (figura 4.1) [12, 18, 57].

Existen varias razones a favor de utilizar la representación binaria, pero la mayoría de ellas se encuentran basadas en el trabajo desarrollado por Holland. En su trabajo, Holland comparó dos representaciones, las cuales tenían aproximadamente la misma capacidad de acarreo de información. La diferencia radicaba en que una tenía pocos alelos y cadenas largas, mientras que la otra contenía cadenas cortas y un número elevado de alelos. Holland argumentó que la primera opción tenía un mayor grado de *paralelismo implícito*, puesto que permite obtener un mayor número de *esquemas* que la otra representación [35].

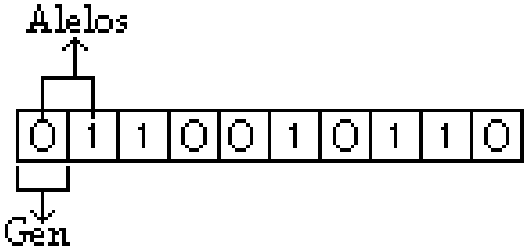


Figura 4.1: Cromosoma con representación binaria [53].

Un *esquema* es una plantilla que describe a un subconjunto de cadenas que comparten ciertas similitudes en algunas posiciones a lo largo de su longitud [12, 18, 33, 35, 57]. Para calcular el número de esquemas dentro de una cadena se usa la fórmula $(c+1)^L$, donde c es la cardinalidad[2] del alfabeto y L es la longitud de la cadena, mientras que el 1 se suma a la cardinalidad debido a que en los *esquemas* se utiliza un símbolo extra, el cual indica que no importa el valor de esa posición [18]. La importancia de que la cadena contenga más esquemas radica en que se favorece la diversidad y se incrementa la posibilidad de formar buenos *bloques constructores* (sección de un cromosoma que permite tener una aptitud elevada a la cadena en la que se encuentra presente) en cada generación, dando como consecuencia un mejor desempeño del AG con el paso del tiempo, de acuerdo al teorema de los esquemas[3] [18, 33, 35].

En lo referente al *paralelismo implícito* del AG, demostrado por Holland [18, 35], se refiere al hecho de que mientras el AG calcula las aptitudes de los individuos en una población, calcula de forma implícita las aptitudes promedio de un número mucho más alto de cadenas cromosómicas, a través del cálculo de las aptitudes promedio observadas en los *bloques constructores* que se detectan en la población.

Por lo tanto es preferible tener muchos genes con pocos alelos posibles que pocos genes y muchos alelos posibles. Esto no sólo se encuentra sustentado por razones teóricas, sino también por justificaciones biológicas, ya que en la

[2]La cardinalidad es el número de posibles valores que puede tener el gen, por ejemplo de un alfabeto binario $c = 2$ que corresponde al cero y al uno.

[3]El teorema de los esquemas es la sustentación teórica que da una idea del funcionamiento del AG [53].

genética es usual encontrar cromosomas con muchas posiciones y pocos alelos por posición, que pocas posiciones y muchos alelos por posición [12, 18]. Holland [35] también demostró que el paralelismo implícito del AG no impide usar alfabetos de mayor cardinalidad, aunque no se debe olvidar que el alfabeto binario ofrece el mayor número de esquemas posibles por bit de información que cualquier otra representación [12, 18, 33]. Sin embargo, este tipo de codificación (la binaria) tiene varias desventajas, las cuales han permitido la utilización de otras representaciones. Algunas de las desventajas son [18]:

- La alta dimensionalidad de los problemas genera cadenas extremadamente largas, lo que da como consecuencia que el AG tenga problemas para generar resultados aceptables en la gran mayoría de los casos.
- La representación binaria no mapea adecuadamente el espacio de búsqueda cuando se trata de números adyacentes en dicho espacio.

Ronald [62] resume las principales razones por las que una codificación binaria puede no resultar adecuada a un determinado problema:

- *Epístasis*: el valor de un bit puede suprimir las contribuciones de aptitud de otros bits en el cromosoma.
- *Representación natural*: algunos problemas (tal es el caso del problema del agente viajero), se prestan de manera natural para la utilización de representaciones de mayor cardinalidad que la binaria.
- *Soluciones ilegales*: los operadores genéticos utilizados pueden producir con frecuencia (e incluso todo el tiempo) soluciones ilegales.

Algunas de las representaciones utilizadas para resolver los problemas que tiene la representación binaria son [18, 53]:

- Binaria con Códigos de Gray.
- Entera.
- Real.
- Listas binarias de Longitud variable.
- De árbol.
- Híbridos.

4.3 La población inicial

La población inicial se forma a partir de un conjunto de m (tamaño de la población) individuos, donde m es un parámetro de entrada al AG. Para generar la población inicial se crean m cadenas aleatoriamente. El procedimiento consiste en asignar un valor aleatorio (cero o uno) a cada gen del cromosoma [58]. Para hacerlo se procede de la siguiente manera [58]:

- Generar un número real aleatorio r entre cero y uno.
- Si $r \leq 0.5$ entonces el alelo valdrá cero; de lo contrario el valor será uno.

Aunque generalmente la población es creada de manera aleatoria, también existen métodos determinísticos para hacerlo [18].

El tamaño ideal de la población para encontrar la solución a un problema, ha sido un tema de estudio al cual no se ha logrado encontrar una respuesta definitiva (la regla empírica que se utiliza comúnmente es generar una población de al menos 2 veces el tamaño del cromosoma), pero se han logrado hacer algunas observaciones importantes al respecto [18].

De Jong realizó algunos análisis del impacto de los parámetros de un AG en su desempeño, y algunas de sus conclusiones son [39]:

- El incremento del tamaño de la población reduce los efectos estocásticos del muestreo aleatorio en una población finita, como consecuencia se mejora el desempeño del algoritmo a largo plazo, pero el precio a pagar es una mayor lentitud del programa.
- Reducir el porcentaje de cruza mejora la media de desempeño, lo que sugiere que producir una generación de individuos completamente nuevos no es bueno.

Grefenstette usó un AG para optimizar los parámetros de otro, y algunas de sus observaciones son [34]:

- La ausencia de mutación tiene como consecuencia un desempeño pobre del AG, es decir, la importancia de este operadar es mayor de lo que se tiene pensado, y su importancia radica en que permite refrescar valores perdidos del espacio de búsqueda.

- El porcentaje óptimo de cruza parece decrementarse conforme se aumenta el tamaño de la población.

Schaffer realizó experimentos para hallar los parámetros óptimos de un AG con codificación de Gray, usando muestreo estocástico universal. Algunas de sus observaciones fueron [66]:

- El uso de tamaños grandes de población (> 200) con porcentajes altos de mutación(> 0.05) no mejora el desempeño de un AG.
- Con poblaciones pequeñas (< 20) con bajos porcentajes de mutación (< 0.002) no mejora el desempeño de un AG.
- Los operadores genéticos pueden muestrear eficientemente el espacio de búsqueda sin necesidad de usar poblaciones excesivamente grandes.
- Conforme se incrementa el tamaño de la población, el efecto de la cruza parece diluirse.

4.4 La función de evaluación

Después de crearse la población inicial se tiene que asignar la aptitud a cada uno de los individuos. A esta tarea es a lo que se le llama función de evaluación, la cual juega el papel del ambiente, y la que indicará la utilidad de cada individuo de acuerdo a la aptitud que se le haya asignado [10, 48, 58].

Una característica indispensable de la función de evaluación, es que debe ser rápida, ya que evalúa individuo por individuo en las sucesivas generaciones.

El buen comportamiento del AG depende de la cantidad de óptimos locales que existan en el problema a resolver, y a qué tan aislado se encuentre el óptimo global, puesto que si este último se haya en una región muy pequeña del espacio de búsqueda, será necesario "jugar" con los operadores genéticos del AG, así como con sus parámetros para hacer que tenga un buen desempeño y se logre hallar la solución.

4.5 La selección

La selección es el operador que elige a los individuos que se utilizarán en la recombinación dependiendo de su aptitud (la sobrevivencia del más apto). Los

individuos escogidos heredarán sus características a la siguiente generación de posibles soluciones.

Para seleccionar a los mejores individuos existen varios métodos, pero pueden clasificarse en:

- *Selección Proporcional*, propuestos por Holland [35]. Los individuos se eligen en forma estocástica de acuerdo a su contribución de aptitud con respecto al total de la población. Existen diversos métodos, siendo los más comunes: Selección de Ruleta, Universal Estocástica, Sobrante Estocástico y Muestreo Determinístico [32].
- *Selección Mediante Torneo*, propuesta por Wetzel [80].
- *Selección de estado Uniforme*, propuesta por Whitley [81]. Este método es utilizado en los AGs no generacionales, en donde sólo son reemplazados algunos individuos en cada generación (los menos aptos).

4.5.1 Selección de ruleta

Este método consiste en crear una ruleta en la que cada individuo tiene asignado un valor, el cual es una fracción proporcional de su aptitud. Y la forma de calcularlo es dividiendo la aptitud del individuo entre la suma de aptitudes de la población. La ruleta generada es girada tantas veces como individuos tenga la población, para así determinar qué individuos serán seleccionados [10, 18, 48, 57].

4.5.2 Selección mediante torneo

Esta técnica consiste en redistribuir de forma aleatoria a los individuos. Después se eligen a los individuos que competirán en el torneo, (generalmente la competencia se realiza en grupos de dos individuos) en donde se seleccionarán con base en comparaciones directas a los individuos que serán utilizados en la cruza, siendo elegido aquél que tenga la aptitud más alta [19, 37, 48, 71].

4.6 Los operadores genéticos

Los operadores genéticos son aquellos métodos que afectan la forma de pasar la información genética de padres a hijos, los más utilizados son la recombinación y la mutación, existe un tercero al que se llama operador de reordenamiento, cuyo trabajo consiste en cambiar el orden de los genes de un cromosoma, con la finalidad de unir a los genes que tengan relaciones entre ellos y permita así, facilitar la producción de bloques constructores [18].

4.6.1 La recombinación

La recombinación o cruza, es el operador más importante del AG, ya que éste es quien genera nuevos individuos o soluciones para resolver el problema [18, 38].

En la CE para simular la cruza lo que se hace es intercambiar segmentos de cadenas lineales de longitud fija. Las técnicas básicas para la cruza son [18, 19, 53, 71, 77]:

- Cruza de un punto.
- Cruza de dos puntos.
- Cruza uniforme.

La cruza dentro del AG es manejada mediante un porcentaje, al que se denomina porcentaje de cruza (Pc), el cual indica si se va a efectuar la recombinación o no, es decir, que habrá algunos individuos que pasarán a la siguiente generación intactos [18]. Al igual que m, Pc es un parámetro de entrada al AG y no existe un valor único para asegurar el buen desempeño del AG, lo que generalmente se utiliza es [10]:

$$Pc \geq 0.6$$

Cruza de un punto

Este método fue propuesto por Holland [35], y consiste en elegir un punto de manera aleatoria dentro del cromosoma de cada padre, y a partir de éste se intercambia el material genético. Esta técnica fue muy utilizada por varios años, pero las desventajas que tiene hace que su utilización sea menos

frecuente [18].

El problema fundamental de la cruza de un punto es que presupone que los bloques constructores son esquemas cortos y de bajo orden, y cuando esto no sucede (por ejemplo, con cadenas largas), suele no proporcionar resultados apropiados [18].

La cruza de un punto trata también preferencialmente algunas posiciones del cromosoma, como por ejemplo los extremos de una cadena [18].

La figura 4.2 muestra cómo se realiza este tipo de cruza.

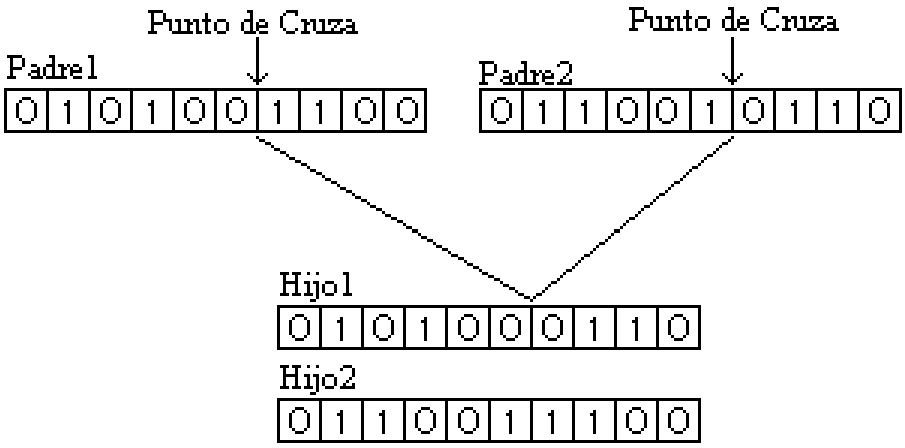


Figura 4.2: Cruza de un punto [18].

Cruza de dos puntos

El primero en implementar este tipo de cruza fue De Jong [39].

Esta cruza es una generalización de la de un punto, en donde en lugar de generar un punto de cruza se crean dos [10, 19, 71, 77].

La ventaja de esta ténica comparada con la anterior es que los efectos disruptivos[4] son menores en comparación con la de un punto, de ahí que sea la más utilizada.

[4]Los efectos disruptivos o destructivos es el nombre que se le dá al fenómeno de destrucción de esquemas que ocurre al momento de efectuarse la cruza [18].

La figura 4.3 muestra cómo se realiza este tipo de cruza.

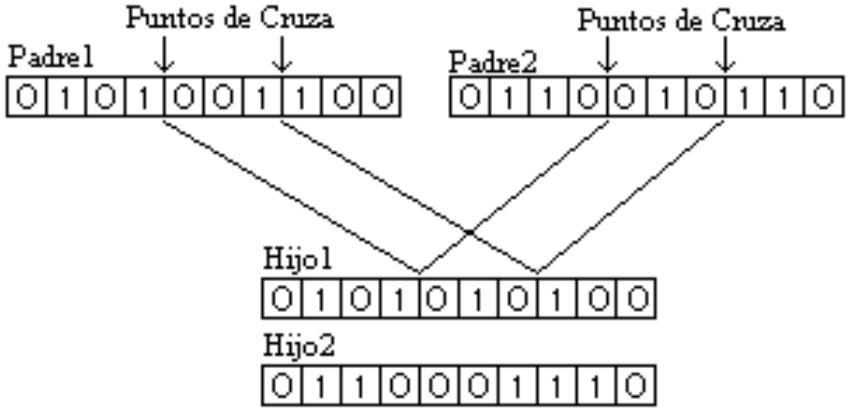


Figura 4.3: Cruza de dos puntos [18].

Cruza Uniforme

Esta técnica la propuso inicialmente Ackley [2], aunque se atribuye a Syswerda [18, 74].

En este tipo de cruza se habla de una recombinación de n puntos, donde el número de puntos de cruza no es fijado previamente [18, 19, 71].

Cabe mencionar que la cruza uniforme no es muy utilizada, y es debido a que tiene un mayor efecto disruptivo que cualquiera de las antes citadas. Para evitar la destrucción de esquemas, suele usarse una probabilidad de cruza de 0.5. En la práctica se sugiere que se usen valores más pequeños [18].

La figura 4.4 muestra cómo se realiza este tipo de cruza.

4.6.2 La mutación

Dentro del AG la mutación es considerada un operador secundario, es decir, no es utilizada con frecuencia.

Como en la cruza, la mutación se utiliza dependiendo del porcentaje asignado para ello, pero como se había mencionado antes, se utiliza menos que la cruza, así que el porcentaje de mutación nunca será mayor del 5% [10].

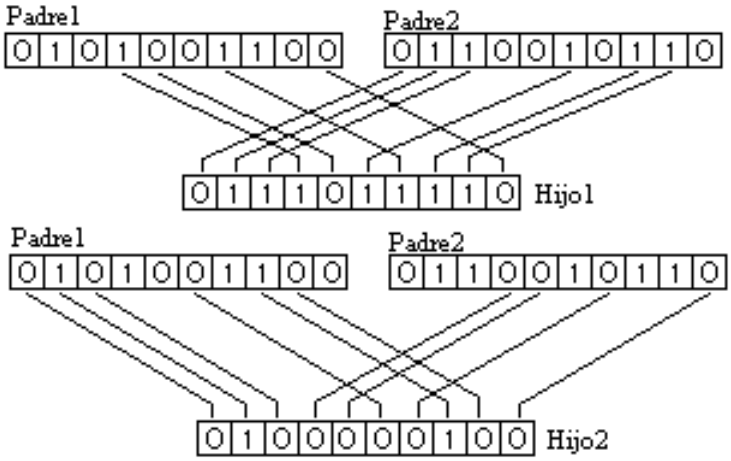


Figura 4.4: Cruza Uniforme [18].

En la práctica, se recomienda utilizar porcentajes de mutación de entre 0.001 y 0.01 para la representación binaria [18], aunque existen autores que recomiendan calcularla con base en la fórmula $p_m = \frac{1}{L}$ (donde L es la longitud del cromosoma) [3, 18].

Existen diferentes tipos de mutaciones dependiendo de la representación que se haya elegido, pero en este caso sólo son de interés las utilizadas para la representación binaria y entera. Los dos tipos de mutación principales para la codificación binaria son:

1. *La mutación uniforme*, la cual consiste en elegir un gen al azar del cromosoma a mutar, para después cambiar el valor del gen. Por ejemplo, si el alelo es uno se modificará a cero, y viceversa [18].

2. *La mutación aleatoria*, igual que en la anterior se elige un gen al azar. La diferencia radica en que no se cambia el valor por un cero o un uno dependiendo del alelo, sino que se elige un número: ya sea cero o uno para sustituir el alelo del gen a mutar [38]. El inconveniente de esta técnica es que el alelo que sustituirá al actual puede tener el mismo valor.

Para la representación entera se puede utilizar la mutación aleatoria, la forma de hacerlo es incrementando el rango, en vez de que se elija un cero o un uno, se permite seleccionar cero, uno, dos o n, siendo n el número entero

máximo del rango a elegir. Mientras que la mutación uniforme no es posible implementarla para la codificación entera.

En la figura 4.5 se ejemplifica la mutación uniforme, mientras que un ejemplo de la mutación aleatoria se muestra en la figura 4.6.

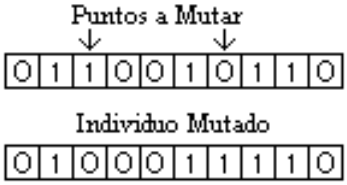


Figura 4.5: Mutación Uniforme.

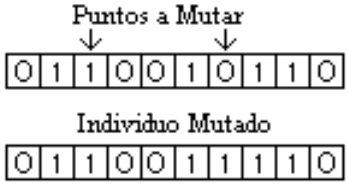


Figura 4.6: Mutación Aleatoria.

4.7 Mecanismo de paro

Al no ser posible conocer la respuesta de antemano a la que se debe de llegar, el problema de detener el AG no es trivial [10]. Generalmente se utilizan dos formas para detener el AG [10, 18, 42, 43, 44]:

1. Una de ellas es ejecutar el AG hasta un número máximo de generaciones (definidas por el usuario).

2. La otra consiste en ejecutar el AG hasta que la población se haya estabilizado (cuando todos o la mayoría de los individuos dentro de la población tengan la misma aptitud).

Capítulo 5

Desarrollo

El programa implementado en este trabajo de tesis se encuentra ubicado dentro del hardware evolutivo, en el área extrínseca, diseñando a nivel de compuertas (ver sección 3.3). Fue desarrollado en el lenguaje Java sobre el sistema operativo Linux (Mandrake 8.1), con el propósito de tener un programa disponible en Internet.

En las siguientes líneas se abordará el porqué del lenguaje de programación, también se explicarán de manera breve algunas de las características del programa base y las del nuevo sistema.

5.1 ¿Por qué Java?

Java fue creado por James Gosling, Oatric Naughton, Chris Warth, Ed Frank y Mike Sheridan en Sun Microsystems en 1991 [58, 67]. En un principio se llamó "Oak" y fue renombrado como "Java" en 1995 [23, 24, 31, 67].

Java surge por la necesidad de contar con un lenguaje de programación que fuera independiente de la plataforma, permitiendo generar software para diversos dispositivos electrónicos. La World Wide Web hace que Java se convierta en el lenguaje de programación más importante para el desarrollo de software para la Internet, ya que la Web demandaba programas portables [23, 24, 31, 46, 67].

Aunque la Internet fue la carta de presentación que permitió a Java ser tomado en consideración [41], este lenguaje tiene otras características im-

portantes que han prolongado su existencia, y por las cuales fue considerado para la implementación del software desarrollado en el presente trabajo.

A continuación se describirán algunas de ellas, pero antes de comenzar a hablar de las particularidades del lenguaje, se dará una definición de él.

La empresa Sun lo define como [1]:

"Java es un lenguaje de programación simple, distribuido, interpretado, orientado a objetos, robusto, seguro, neutro con respecto a las arquitecturas, portátil, de alto rendimiento, de múltiples subprocesos, dinámico, compatible con las tecnologías de moda y de propósitio general. Brinda soporte de programación para Internet con applets de Java independientes de las plataformas."

5.1.1 Simple

Java fue creado con la finalidad de que su aprendizaje y utilización resultaran sencillos [31, 46, 67]. Por tal motivo, fue modelado a partir de C y C++, debido a que estos lenguajes son los más difundidos [31, 41, 46].

Aunque su sintaxis es similar a C y C++, y hereda la estructura de orientación a objetos del último, Java excluye varios de los conceptos más complejos de sus predecesores, entre los que destacan [24, 31, 41, 46, 64, 67]:

- Apuntadores y la aritmética de apuntadores.
- Necesidad de liberar memoria.
- Herencia múltiple.
- Definición de tipos (typedef).
- Macros.

Como consecuencia se tiene una mayor seguridad en los programas y se reducen los errores más comunes de programación.

5.1.2 Independiente de la plataforma

La independencia de la plataforma se refiere a que un programa en Java puede ejecutarse sin importar el sistema operativo de la máquina en particular [24].

La independencia de plataforma es una de las ventajas más representativas de Java sobre otros lenguajes de programación [46]. La meta de Java era "escribir una vez, ejecutarse en cualquier sitio, en cualquier momento y para siempre", el objetivo se alcanzó en gran medida [67].

El que sea independiente de la plataforma hace posible su portabilidad.

Para lograr la independencia, Java maneja estándares que facilitan la construcción de los sistemas [24, 31, 46], y maneja las interfaces por medio de un sistema abstracto que permite sean implementadas en entornos Unix, Pc o Mac [31].

Y en lugar de generar un código ejecutable, crea un bytecode[1]. Este código puede ser mostrado en cualquier sistema que tenga un intérprete Java. El intérprete se denomina máquina virtual Java (JVM) (figura 5.1) [1, 24, 31, 46, 64, 67].

El ser interpretado lo ha hecho flanco de ataques, ya que es más lento que los programas que son compilados y ejecutados [31, 41, 46]; sin embargo, el bytecode fue diseñado para ser fácilmente traducido a código nativo, dando como consecuencia un alto rendimiento sin perder las ventajas de independencia de plataforma [67].

5.1.3 Dinámico

Dado que los programas en Java son interpretados, provoca que sean dinámicos [64]. Puesto que Java se beneficia de la tecnología orientada a objetos, no intenta conectar todos los módulos que comprenden la aplicación hasta el mismo tiempo de ejecución [31, 67]. Esto da la oportunidad de agregar nuevos métodos o actualizar el código cuando el sistema se encuentre ejecutando sin incurrir en errores [31, 64, 67].

5.1.4 Seguro

La seguridad de Java consta de dos partes [31]. En primer lugar los apuntadores son eliminados, con esto se previene el acceso ilegal a la memoria [31, 41].

[1]El bytecode es un conjunto de instrucciones parecidas al código máquina, pero no están ligadas a un hardware en específico [46, 67].

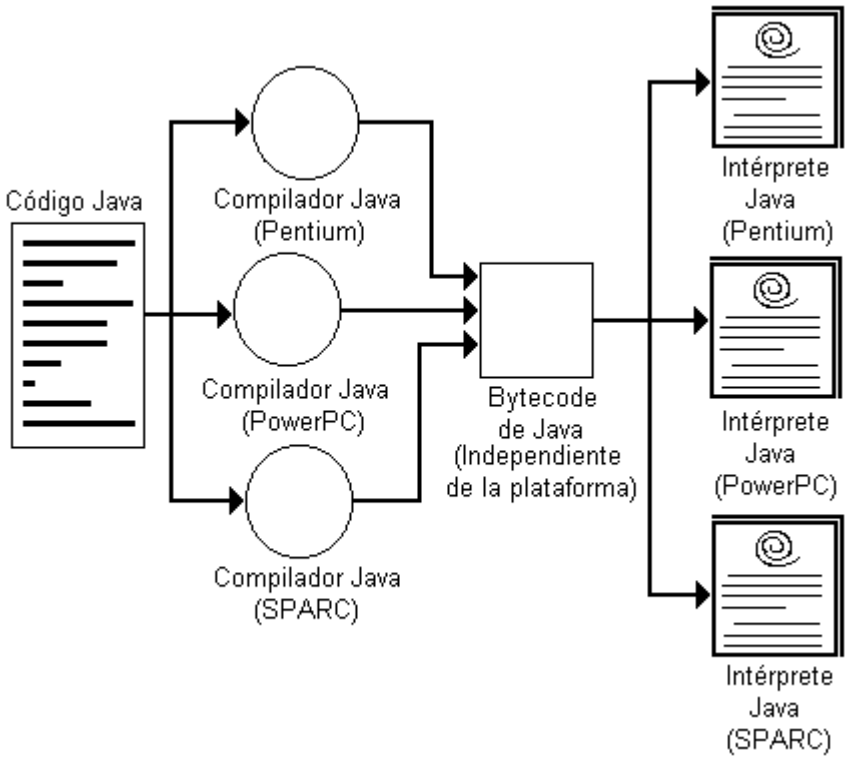


Figura 5.1: Esquema de la Portabilidad de los Programas hechos en Java [46].

Y en segundo lugar, el código es analizado por un verificador de ByteCode, que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal antes de ser ejecutado por el intérprete (figura 5.2) [23, 31, 67].

Con esto se tiene la certeza de que el programa en ejecución no tendrá virus y el usuario podrá utilizarlo con toda tranquilidad.

El *Cargador de Clases* también ayuda a Java a mantener su seguridad, esto lo hace separando el espacio de nombres del sistema de archivos locales de los recursos procedentes de la red. Esto permite limitar cualquier aplicación del tipo Caballo de Troya [31].

Para imposibilitar que una clase suplante a una predefinida, las clases son almacenadas en un espacio privado de nombres, asociado con el origen.

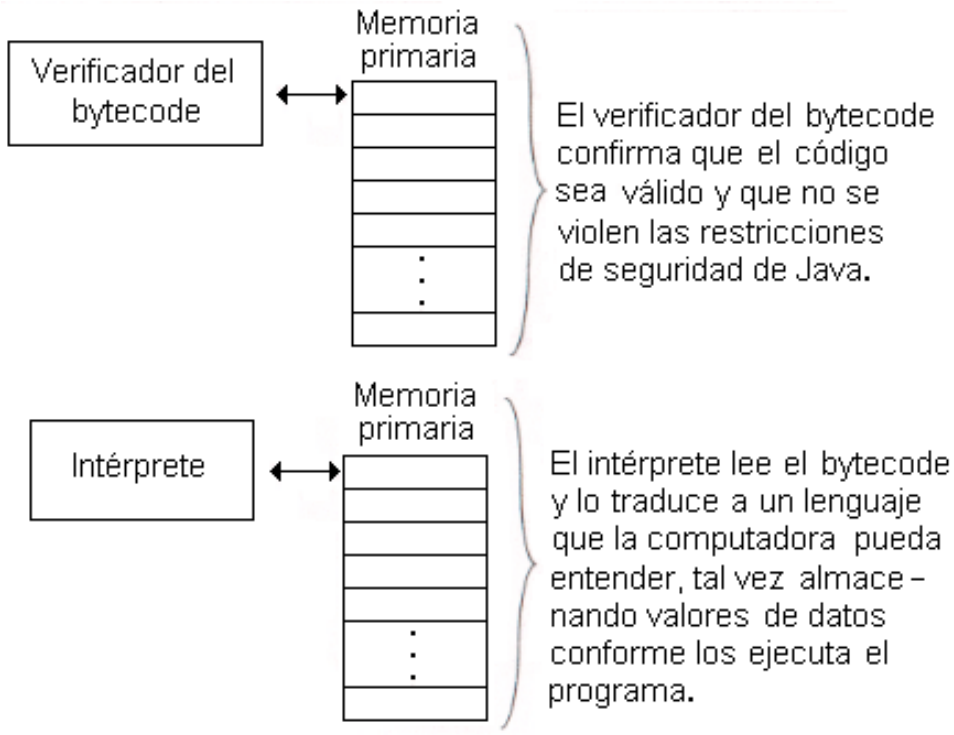


Figura 5.2: Manejo de la seguridad en Java [23].

Java no permite abrir archivos de la máquina local, impide ejecutar aplicaciones nativas de una plataforma y evita que se utilicen otras computadoras como puente; es decir, no se puede utilizar una máquina para hacer peticiones o hacer operaciones con otra [1, 23, 31, 67]. En consecuencia, esto limita las operaciones de impresión y de almacenamiento. Sin embargo, pueden ser incorporadas solicitando al usuario que las habilite. La forma de hacerlo es que el programador firme el applet, con ello se revela la identidad de quien lo generó, posibilitando al usuario conocer la procedencia del programa. Al ser firmado el applet, éste pregunta al usuario si admite que se ejecute o no. Lo mismo ocurre con cada una de las acciones (imprimir, abrir y almacenar) que requieran utilizar los recursos de la computadora local. Esto funciona únicamente cuando el programa proviene de la red, puesto que, para los que son locales, el applet no pregunta al usuario y las actividades se realizan de manera automática [1].

5.1.5 Robusto

Java es considerado un lenguaje robusto y confiable, gracias a las siguientes características [24, 31, 67]:

- *Validación de tipos.* Los objetos de tipos compatibles pueden ser asignados a otros objetos sin necesidad de modificar sus tipos. Los objetos de tipos potencialmente incompatibles requieren un modificador de tipo (cast). Si la modificación de tipo es imposible, el compilador reportará un error en tiempo de compilación, pero si resulta legal, el compilador lo permitirá e insertará en tiempo de ejecución una validación.

- *Validación del apuntador NULL.* Todos los programas en Java usan apuntadores para referenciar a un objeto. Esto no genera inestabilidad porque una validación del apuntador NULL ocurre cada vez que un apuntador deja de referenciar a un objeto.

- *Límites de un arreglo.* Se verifica en tiempo de ejecución que un programa no use arreglos para tratar de acceder a áreas de memoria que no le pertenecen.

- *Manejo de memoria.* Muchos de los errores que se cometen en la programación son ocasionados por no liberar la memoria que se debería, o se libera la misma memoria más de una vez. Java lo hace automáticamente realizando una recolección de basura (garbage collector), liberando memoria que no se utiliza y evitando así que el programador se ocupe de esta tarea.

Aunque este último punto es considerado inapropiado porque el proceso de recuperar memoria ya no utilizada puede llevar mucho tiempo, y además puede no ser aceptable si se invoca por el sistema en un momento crítico de la aplicación; es posible tener un control al respecto, puesto que Java al incluir soporte para la programación concurrente, le permite al programador la oportunidad de decidir cuando disparar la recolección de basura [41].

5.2 Programa base

El presente trabajo de tesis se desarrolló con base al programa de Coello et al. [13, 14]. Este sistema se encuentra desarrollado sobre la plataforma UNIX,

utilizando el lenguaje C. Una de las desventajas (bien conocida) que tiene este lenguaje, es que el programa no es de fácil transportación (refiriéndose al sistema operativo en el que se trabaje), ya que el código tiene que manipularse y compilarse de nuevo para poder ser ejecutado. De ahí que haya surgido la inquietud de usar Java para desarrollar una nueva versión de este programa.

5.2.1 El AG

Con respecto al AG, Coello et al. [13, 14] eligieron una versión con elitismo.

A continuación se describirán algunas de las características del programa base de acuerdo a los componentes del AG.

La representación

La representación usada por el programa base es la binaria y la entera, esto se hace para demostrar la utilidad de ésta última. En palabras de Coello et al. [13, 14], "aunque se ha argumentado que una representación binaria proporciona el número máximo de esquemas, existe evidencia de que en algunos dominos tales como el de la optimización numérica, los alfabetos de cardinalidad más alta proporcionan mejores resultados en un período de tiempo más corto que sus contrapartes binarias. Con este precedente en mente, decidimos experimentar con un alfabeto de cardinalidad n, donde n es un valor definido por el usuario. Esta representación permite la manipulación de cadenas más cortas, decrementando la complejidad de la decodificación, y al mismo tiempo permite explorar regiones del espacio de búsqueda que la representación binaria no parece cubrir apropiadamente cuando la distancia fenotípica es muy pequeña, como en este caso."

La población inicial

La población inicial es generada de manera aleatoria.

La función de evaluación

Coello et al. explican en [13, 14] que utilizan una comparación bit por bit del resultado producido para una cierta combinación de entradas con respecto a las salidas esperadas. Se penaliza por cada error hayado. Si el individuo representa un circuito funcional, entonces la función de aptitud "premia" la

solución por cada WIRE[2] que contenga, a fin de alentar los diseños que usen menos compuertas. En otras palabras, la función de evaluación trabaja en 2 etapas. Primero maximiza el número de aciertos (con respecto a la tabla de verdad). Una vez que se producen circuitos válidos, se maximiza su cantidad de WIREs, de manera que se minimice su número de compuertas.

Coello et al. [13, 14] resumen que esta función de aptitud proporciona resultados altamente satisfactorios.

La selección

Para el operador de selección, el programa base tiene implementada únicamente la selección por torneo binaria (realiza la competencia únicamente entre dos individuos).

Los operadores genéticos

Para los operadores genéticos, fueron implementadas la cruza de dos puntos y la mutación aleatoria.

Mecanismo de paro

Respecto a la forma de detener el AG, Coello et al. [13, 14] optaron hacerlo por medio del número de generaciones, ya que no fue posible definir el umbral que podría asegurar que no existe una mejor solución [14].

5.2.2 La representación del circuito

La forma en que se representa un circuito lógico es por medio de una matriz bidimensional, en la que cada elemento es una compuerta[3] que recibe en sus dos entradas la salida de cualquier compuerta de la columna anterior, en la figura 5.3 se visualiza la representación.

La codificación de una compuerta requiere de tres genes: uno indica el tipo de compuerta, y los otros dos corresponden a las entradas.

[2]Cable de unión entre columnas, es decir, indica la ausencia de compuertas.

[3]Las compuertas que utiliza el Dr. Coello en su programa son: la AND, OR, NOT, XOR y WIRE.

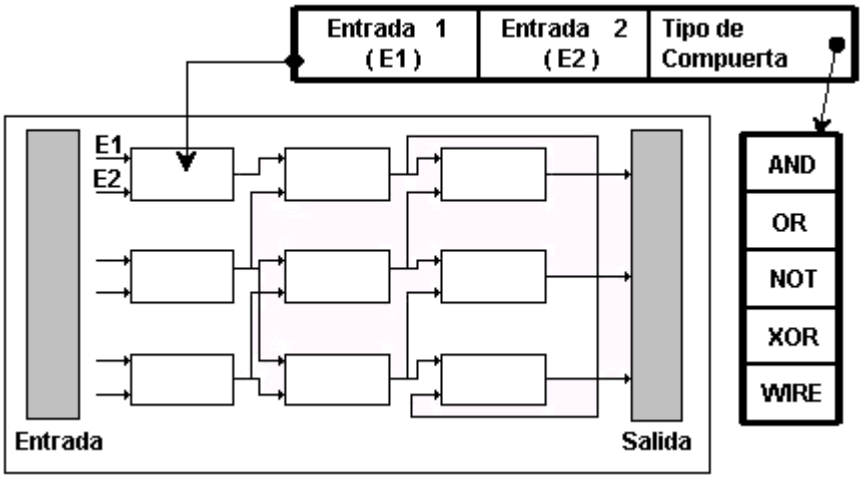


Figura 5.3: Forma de representar el circuito en el programa base [13, 14, 52].

Ahora es posible calcular la longitud del cromosoma (L):

$$L = renglones * columnas * 3,$$

donde $renglones * columnas$ determina el número de componentes que tiene la matriz, y el número tres, la cantidad de genes necesarios por componente. Esto es cierto, siempre y cuando se trabaje con la representación entera, puesto que, para la binaria es necesario utilizar un cierto número de genes para cada uno de los tres elementos mencionados (tipo de compuerta y las dos entradas). Por tal motivo, la longitud del cromosoma ahora se calcula de la siguiente manera:

$$L = renglones * columnas * 3 * reprebinaria,$$

donde $reprebinaraia$ es el número de genes necesarios para manipular la representación binaria.

Por ejemplo si la matriz es de 7*7, siete es el número máximo a representar, así que para este caso se necesitan tres posiciones más, ya que $111 = 7$. Por lo tanto, la longitud del cromosoma será:

$$L = 7 * 7 * 3 * 3 = 441$$

mientras que para la entera será:

$$L = 7 * 7 * 3 = 147$$

para este ejemplo, la longitud del cromosoma con codificación entera es tres veces menor a la longitud obtenida por la binaria, por tal motivo las ejecuciones de ésta serán en menor tiempo. Ahora que si la matriz es de 8*8 se requiere de cuatro posiciones (1000 = 8) para generar a cada uno de los elementos del cromosoma con representación binaria, mientras que, la entera sigue constante, así que ahora ésta será cuatro veces menor que la binaria, de ahí la importancia de utilizar la codificación entera.

5.2.3 Trabajando con el programa

Para poder interactuar con el software es necesario trabajar en la línea de comandos y la sintaxis es:

nombreEjecutable Archivo_de_Entrada Archivo_de_Salida Semilla_Aleatoria Tamaño_de_la_Población Número_de_Generaciones Probabilidad_de_Mutación Probabilidad_de_Cruza

Donde *nombreEjecutable*, es el nombre del archivo después de compilarse el programa.

El *Archivo_de_Entrada* corresponde al nombre de un archivo de texto, donde se especifica la configuración del problema, y tiene los siguientes datos:

- Tipo de representación (binaria o entera).
- Número de variables de entrada.
- Número de salidas.
- Cardinalidad.
- Número de compuertas a utilizar.
- Número de renglones.
- Número de columnas.
- Umbral de impresión.
- Posibles entradas 1.

- Posibles entradas 2.
- Tabla de verdad de entradas.
- Tabla de verdad de salidas.

A continuación se muestra un pequeño ejemplo de un archivo de entrada (la tabla de verdad introducida es la de una compuerta XOR).

```
BINARY
2
1
5
5
5
5
5
5
29
0 0
0 1
1 0
1 1
0
1
1
0
```

El *Archivo_de_Salida* es creado por el programa. Este archivo contendrá la información del comportamiento del AG, así como la matriz del mejor individuo por generación. Como se puede intuir, este archivo llega a ser muy extenso para estudiarlo, puesto que los datos son guardados por generación.

Las siguientes líneas muestran un ejemplo del archivo de salida. Cabe aclarar que sólo es una pequeña sección.

```
Fecha y tiempo de inicio: Mon Mar 10 13:50:40 2003
------------------------------------------------------------------
                                Population Report
Generation 367   Old Strings
Generation 368   New Strings
Generation 367 Accumulated Statistics:
Total Crossovers= 9163, Total Mutations = 18359
min= 0.000000   max= 27.000000   avg= 24.360000   sum= 2436.000000
Global Best Individual so far, Generation 367:
Fitness= 27.000000
------------------------------------------------------------------
                                Population Report
Generation 423   Old Strings
Generation 424   New Strings

WIRE1(4,2) WIRE1(2,4) WIRE1(1,3) WIRE1(3,1) WIRE1(5,5)
  XOR(2,3) WIRE1(3,2) WIRE1(4,1) WIRE1(3,4) WIRE1(5,1)
WIRE1(3,2) WIRE1(3,3) WIRE1(1,2) WIRE1(2,4) WIRE1(5,4)
WIRE1(1,3) WIRE1(2,2) WIRE1(4,4) WIRE1(3,2) WIRE1(4,2)
WIRE1(3,1) WIRE1(1,3) WIRE1(1,2) WIRE1(1,2) WIRE1(2,4)

f[1]=4
f[2]=24
fitness=28.000000
violations=0
String= 1 1 0 1 0 0 1 1 0 1 0 0 1 1 1 0 0 1 0 1 0 0 1 1 1 1 0 0 0 0
0 1 0 1 1 0 1 1 1 1 0 1 1 1 0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 0 1 1 0 1
1 1 1 1 1 1 1 0 0 1 1 1 0 0 1 1 0 1 0 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1
0 1 1 0 1 0 1 1 1 0 0 0 0 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 0 1 0 1 1
1 1 0 0 1 0 1 0 1 1 1 0 1 1 1 1 1 0 1 1 0 1 0 0 1 1 0 1 1 0 1 1 1 1
0 0 1 1 0 1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 0 1 1 0 1 1 1 0 0 0
1 1 1 0 1 1 0 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0 1 1 0
------------------------------------------------------------------
Generation 423 Accumulated Statistics:
Total Crossovers= 10554, Total Mutations= 21075
min= 0.000000   max= 28.000000   avg= 24.120000   sum= 2412.000000
Global Best Individual so far, Generation 423:
Fitness= 28.000000
```

Como se puede observar se tiene que interpretar la matriz de manera manual para obtener el diseño del circuito. Cuando son problemas con un número reducido de compuertas, como en este caso, no es difícil hacerlo, pero cuando el número de compuertas llega a ser considerable es fácil equivocarse.

Tampoco indica el número de compuertas óptimas o las utilizadas por el circuito, lo que también el usuario es el encargado de contabilizar.

Los siguientes datos de la línea de comandos son opcionales, el programa considera para ellos ciertos valores por omisión:

- Probabilidad de cruza = 0.5
- Probabilidad de mutación = $\frac{0.5}{L}$
- Generaciones = 1000
- Tamaño de la población = 1000
- Semilla inicial = 0.27

Ahora que si desea probar el programa con valores que no sean los definidos por omisión, solo tiene que agregarlos a la línea de comandos.

5.3 El sistema nuevo

El nuevo sistema se desarrolló en el lenguaje de programación Java, y se puede utilizar accesando a la página `http://nuyoo.utm.mx/~jcruz/Tesis.html`, donde hay una liga al applet en el cual trabaja el programa.

Las caraterísticas de este programa son descritas a continuación.

5.3.1 El AG

Al igual que el programa de referencia se implementa el AG con elitismo. No hay que olvidar que se ha demostrado que este algoritmo converge al óptimo global de un problema arbitrario.

La representación

Este programa trabaja con representación binaria y representación entera.

La población inicial

Al igual que en el programa en el que se encuentra basado este trabajo de tesis, la población inicial del AG es generada de manera aleatoria.

La función de evaluación

Se retomó la función de aptitud desarrollada en el programa base, puesto que la función proporciona mayor información al AG.

La función objetivo es:

$$f(x) = aciertos + bonos,$$

donde *aciertos* es el número de salidas de la tabla de verdad que se cumplen y *bonos* es un valor que se hace cero cuando aciertos es menor que el número exacto de salidas de la tabla de verdad. En caso contrario (o sea, si se cumplen todos los bits de salida), *bonos* toma un valor igual a la cantidad de WIREs del circuito.

Como *aciertos* es el número de bits de salida de la tabla de verdad, entonces:

$$aciertos = 2^{ne} * ns,$$

donde ne es el número de entradas y ns es el número de salidas. Mientras *bonos* (número de WIREs que tiene el individuo) se puede expresar de la siguiente manera:

$$bonos = r * c - CO,$$

siendo r renglones, c columnas y CO el número de compuertas que no son WIREs. Por lo tanto la función objetivo es:

$$f(x) = 2^{ne} * ns + r * c - CO.$$

La selección

Se implementó únicamente la selección por torneo binaria.

Los operadores genéticos

Con respecto a los operadores genéticos se utilizan los mismos con los que trabaja la versión de Coello et al. [13, 14], y se agregaron otros más.

Para la mutación, se agregó la uniforme y la aleatoria. Esta última puede trabajar tanto para la representación entera como para la binaria, la uniforme sólo trabaja para la representación binaria (ver sección 4.6.2).

El porcentaje de mutación puede ser $P_m = \frac{1}{L}$, que es una de las formas recomendadas [3, 18]. También se puede usar $P_m = \frac{0.5}{L}$ (L representa la longitud del cromosoma) que es la utilizada en la implementación de Coello et al. [13, 14], o alternativamente, el usuario puede insertar otro valor, el cual se encuentra limitado por el rango (0.001-0.01) que es recomendado en la bibliografía (ver sección 4.6.2).

Además de la cruza de dos puntos, se implementaron la cruza uniforme y la cruza de un punto. La cruza al cambiar subcadenas del cromosoma y no los valores (alelos) de los genes, puede utilizarse de manera indiferente para cualquiera de las dos representaciones.

Cabe mencionar que se generó un cuarto operador de cruza, al que se le llamó cruza de *Múltiples Puntos*[4], el cual consiste en generar dos rangos aleatoriamente, y realizar el intercambio de material genético según los rangos. Este nuevo método tiene la característica de fijar los puntos de intercambio, como la cruza de uno y dos puntos. Sin embargo, el número de puntos de cruza es aleatorio y se generan *múltiples puntos* de cruza como en la cruza uniforme. La figura 5.4 muestra un ejemplo del funcionamiento de este operador de cruza.

Mecanismo de paro

La manera de detener el AG, es al igual que el programa de referencia, mediante el número de generaciones que se desea que el AG trabaje, ya que no es posible determinar la aptitud máxima en donde se encuentre el mínimo número de compuertas para un problema en particular.

[4]El pseudocódigo de este nuevo operador de cruza se encuentra en el apéndice B.

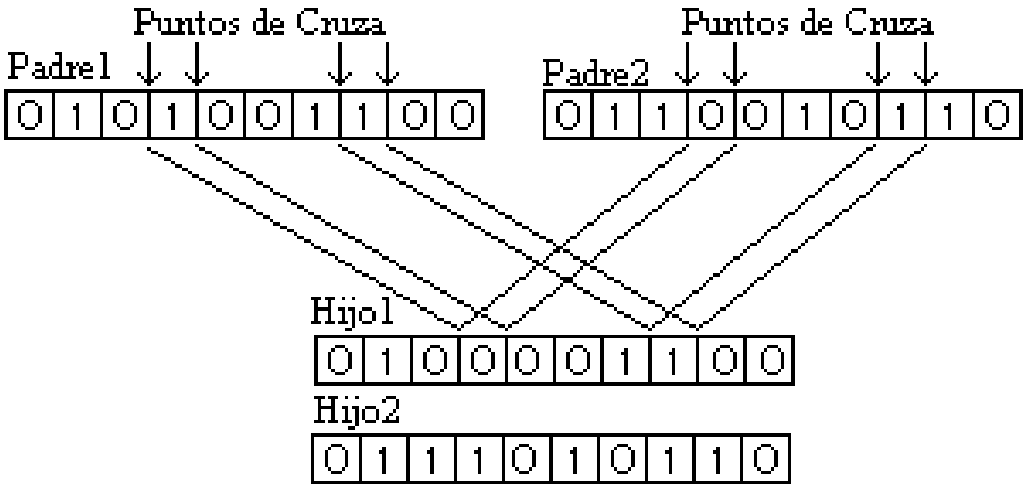


Figura 5.4: Cruza de Múltiples Puntos.

5.3.2 La representación del circuito

Se utilizó la misma representación seleccionada por Coello et al. [13, 14] para visualizar el diseño del circuito. Además, se agregaron dos compuertas más (la NAND y la NOR), dando la oportunidad al usuario de elegir con que compuertas desea resolver el problema.

5.3.3 Trabajando con el programa

Otra diferencia con el programa base, es que la inserción de datos ya no se realiza desde la línea de comandos con un archivo de texto, ahora se hace por medio de un menú a través de una interfaz gráfica y el usuario puede modificar los datos (tanto parámetros del AG, como el tipo de operador a utilizar) en tiempo de ejecución. La figura 5.5 muestra la interfaz del programa.

Se resuelve el problema de la interpretación de la matriz, ya que el diseño se muestra de manera gráfica al término de la ejecución del AG (figura 5.6).

Al programa se añadió la manera de mostrar la función booleana. Con el AG, primero se obtiene el diseño y después la función. Aprovechando esto, se agrega el teorema de la involución para que la expresión booleana no inserte NOTs continuos.

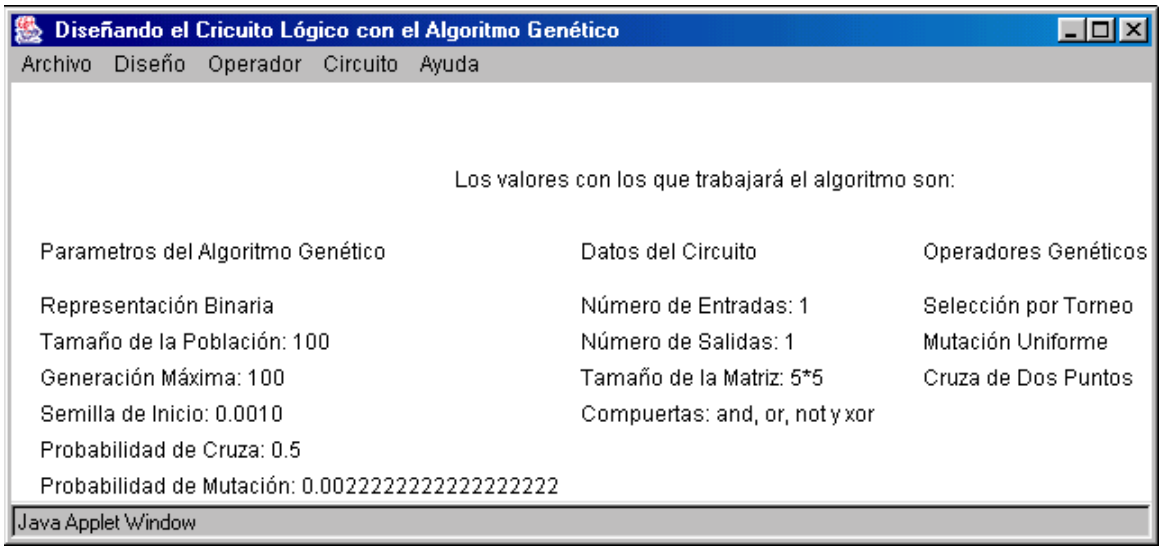


Figura 5.5: Interfaz de entrada del programa desarrollado para esta tesis.

Para representar la función booleana se usaron los siguientes símbolos:

Símbolo	**Compuerta**
+	OR
′	NOT
@	XOR
&	NAND
|	NOR

Tabla 5.1: Símbolos utilizados para representar a cada una de las compuertas.

Para la AND no se utilizó símbolo alguno.

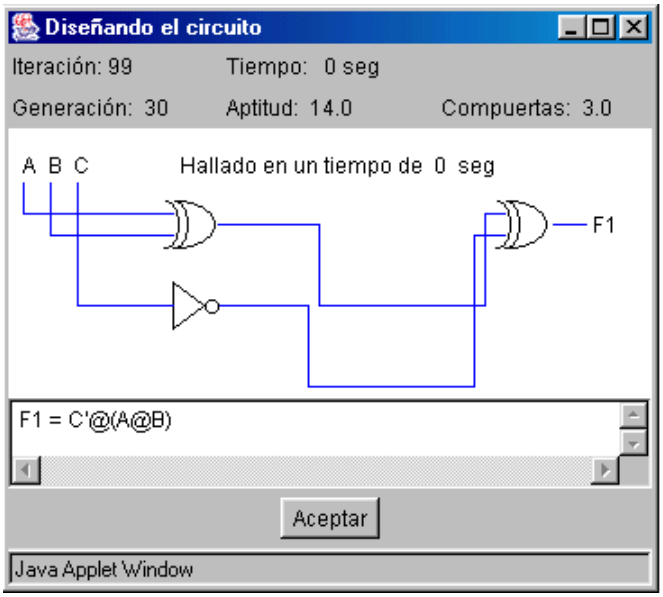


Figura 5.6: El resultado que muestra el programa.

5.4 Ventajas de esta implementación sobre las técnicas tradicionales

Una de las ventajas más significativas es que cada cromosoma representa una posible solución, lo que hace que tenga más de una solución por ejecución.

La manera en que se implementaron las compuertas permite trabajar con compuertas XOR, en la bibliografía revisada ni el método de Quine-McCluskey, ni los mapas de Karnaugh la consideran para hacer reducciones, y las técnicas que la consideran son complejas.

No hay que olvidar que estas técnicas sólo simplifican a dos niveles (suma de productos o producto de sumas), y que este resultado no es siempre el mínimo (refiriéndose al número de operaciones que utiliza para expresar la función).

Las pruebas realizadas solo se hicieron probando con la compuerta XOR, con la finalidad de poder comparar los resultados con los obtenidos por el programa base y otras implementaciones (se expecificará cuáles en el siguiente capítulo), pero puede ser utilizado sin ella o con las compuertas NAND y

NOR como lo muestra la figura 5.7. Como se muestra en las figuras 5.7b y 5.7c la forma de representar a las compuertas NAND y NOR fue insertando una *N* a las compuertas AND y OR, en lugar de agregar un círculo al frente de ellas.

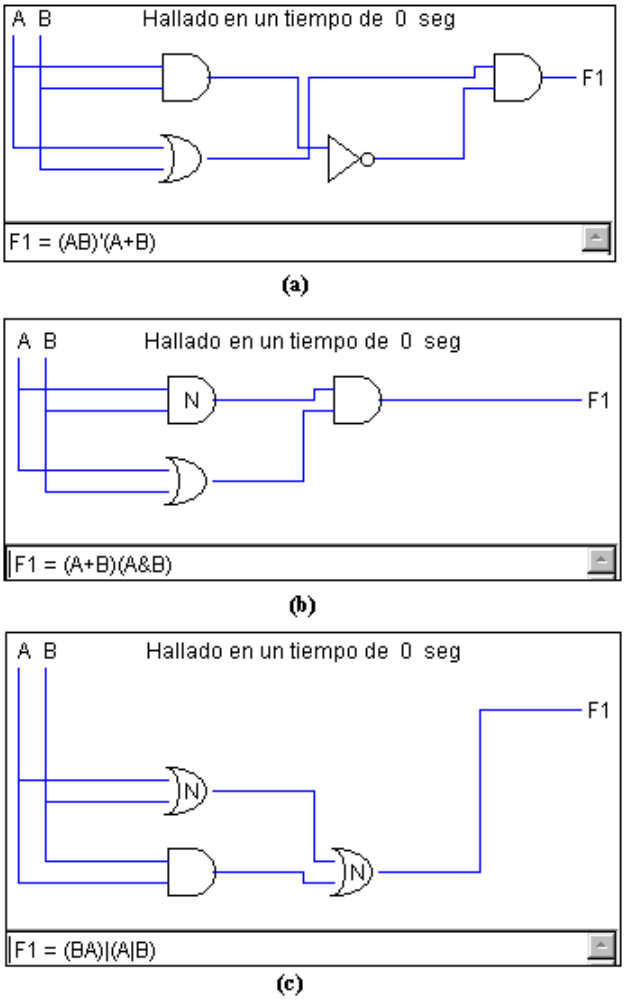


Figura 5.7: Diseños de la compuerta XOR utilizando: (a)compuertas básicas, (b) y (c) compuertas básicas y universales.

En el *apéndice A* se encuentra el manual de usuario, donde se explica como trabajar con el software.

5.5 ¿Como trabaja la función de aptitud?

Para ejemplificar la manera de trabajar de la función de aptitud se utiliza la tabla de verdad que se muestra a continuación (tabla 5.2).

A	**B**	**C**	**F1**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Tabla 5.2: Tabla de verdad del primer circuito

Para este ejemplo se utilizan los siguientes parámetros:

- Representación binaria.
- Cruza de dos puntos.
- Mutación uniforme.
- Probabilidad de cruza = 0.5.
- Probabilidad de mutación = $\frac{1}{L}$.
- Semilla inicial = 0.273.
- Con 100 individuos.
- Usando las compuertas básicas y la XOR.
- Y un tamaño de matriz de 5*5.

El AG genera la población inicial de forma aleatoria, y la población es evaluada para encontrar al mejor individuo de la misma. El individuo es una cadena de ceros y unos (por el tipo de representación utilizada), con una

longitud L = 5*5*3*3 = 225 genes, considerando que el proporcionar la cadena del mejor individuo no daría mucha información, se optó por mostrar su diseño en la figura 5.8. Cabe aclarar que en el programa no se genera el diseño cuando el circuito no es factible, sólo se indica al usuario que no se encontró diseño alguno.

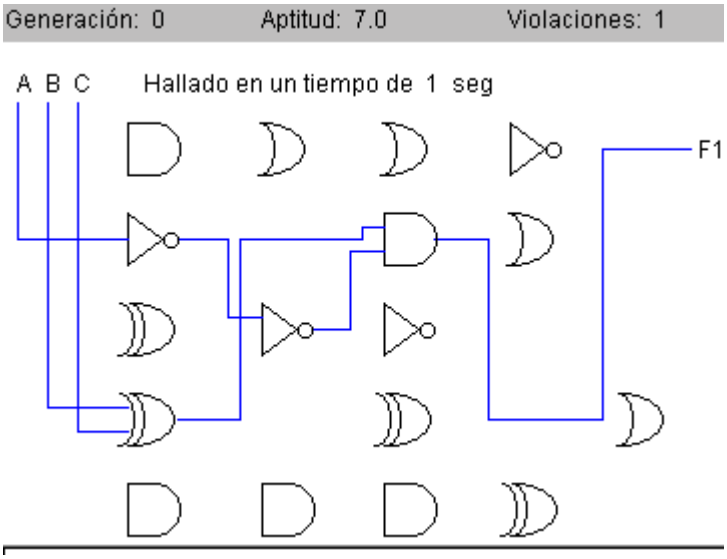


Figura 5.8: Diseño del mejor individuo en la generación 0.

Se evalúa al individuo generando una tabla de verdad después de cada columna iniciando por la que se encuentra más a la izquierda. La tabla de verdad que se tiene de entrada se utiliza con la primera columna, de la cual se obtiene la tabla que se utilizará en la siguiente columna y así sucesivamente. Al terminar de recorrer las columnas (en este caso cinco) la última tabla generada es comparada con la tabla de verdad de salida. Hay que recordar que la función objetivo es:

$$f(x) = aciertos + bonos,$$

$$f(x) = 2^{ne} * ns + r * c - CO,$$

donde ne es el número de entradas, ns el número de salidas, r el número de renglones, c el número de columnas y CO el número de compuertas que no son WIREs. Y *bonos* será cero, siempre que existan violaciones en el circuito.

En este caso una de las combinaciones no se cumple, por lo tanto $bonos = 0$ y la aptitud de este individuo será siete.

El AG sigue corriendo y es hasta la generación 200 donde encuentra un individuo que tiene un circuito factible, el cual se muestra en la figura 5.9.

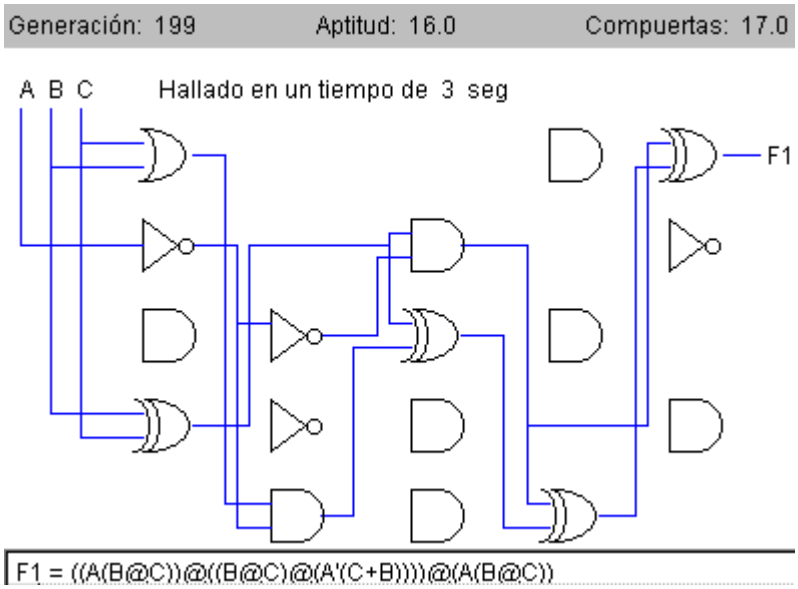


Figura 5.9: Diseño del circuito factible en la generación 200.

La función de aptitud vuelve a trabajar de la misma manera, sin embargo, ahora al ser un circuito factible se contabilizan las compuertas WIREs y se agregan a la aptitud. En este caso es una aptitud de 16, ocho aciertos y ocho compuertas WIREs.

El AG incrementará la aptitud del individuo en proporción al número de compuertas WIREs contenidas en él, como se observa en la figura 5.10.

Es en la generación 266 donde se alcanza un circuito que tiene cinco compuertas, como se puede observar desaparecen las compuertas que no utiliza el circuito.

Sin embargo, es hasta la generación 766 donde encuentra el diseño óptimo que consta de cuatro compuertas y la figura 5.11 muestra el diseño de este circuito.

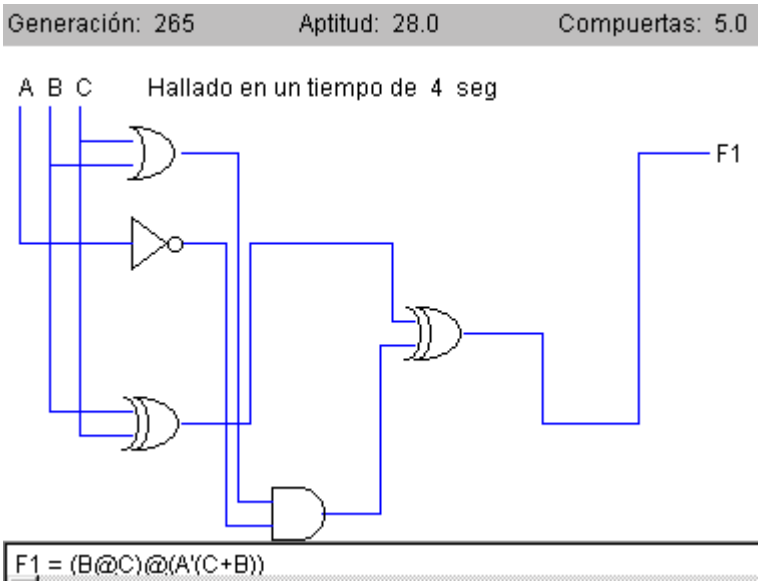


Figura 5.10: Diseño del mejor individuo en la generación 266.

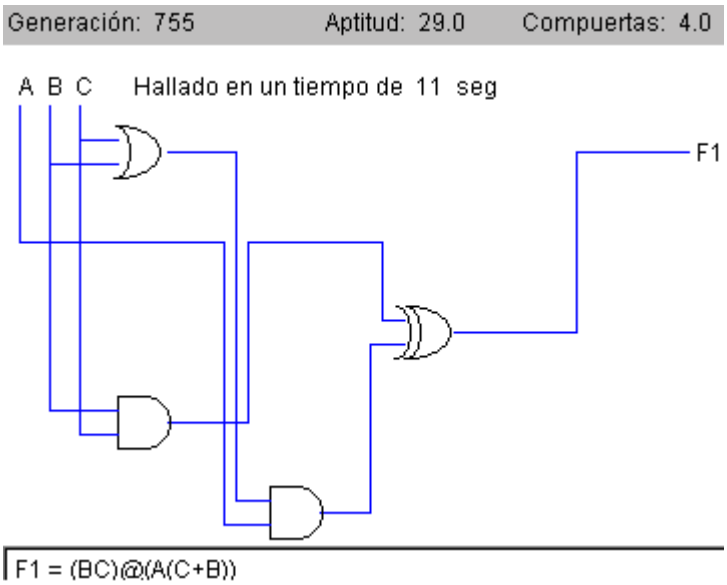


Figura 5.11: Diseño del circuito óptimo.

Capítulo 6

Resultados

Como se ha visto, el AG tiene diferentes parámetros (tamaño de población, número máximo de generaciones, probabilidad de cruza y mutación, entre otras), así como también diferentes operadores genéticos, y que en combinación modifican el desempeño del AG. El desempeño se puede medir de varias formas:

- Con respecto al tiempo (lo que tarda en descubrir la solución).
- Referente a la convergencia en el menor número de iteraciones posibles.
- Considerando el mayor número de soluciones óptimas (entendiendo por óptimo, para este caso de estudio, el diseño que utiliza el menor número de compuertas) y el mayor número de soluciones factibles (entendiendo por factible, a la solución dada por el AG que cumple con el comportamiento de la tabla de verdad) encontradas.

Para demostrar la efectividad del método se tomó como medida de evaluación, la producción de soluciones óptimas y de soluciones factibles. Por solución óptima se considerará aquel circuito con el menor número de compuertas obtenidas por el programa implementado en este trabajo de tesis comparado con las otras técnicas desarrolladas para el diseño de circuitos.

Los diseños obtenidos por la implementación desarrollada en esta tesis (AG en Java) son comparados con el programa base (PB) [14], la implementación de Islas (AG Basado en Casos) la cual tiene memoria [37], el sistema de Serna (PG prefija) que emplea otra técnica de la CE [71] y con el programa de Mendoza (Colonia de Hormigas) [52], con la finalidad de demostrar que

puede obtener soluciones similares o mejores a las ya obtenidas por estas técnicas.

También se realizó una comparación de las soluciones obtenidas al combinar los diferentes operadores, con el propósito de observar la influencia de ellos para encontrar las soluciones (tanto óptimas, como factibles) a los problemas propuestos.

Los resultados aquí mostrados fueron hallados después de ejecutar el programa 10 veces (cada una con una semilla inicial diferente, comenzando con 0.091 e incrementando cada vez en 0.091). Estas 10 veces se ejecutaron con cada una de las cruzas (una y dos puntos,uniforme y múltiples puntos), con los dos tipos de mutación (uniforme y aleatoria), con sus dos probabilidades de mutación[1] y con cinco diferentes tamaños de población (100, 500, 1000, 1500 y 2000 individuos). Por tanto, el programa fue ejecutado 800 veces con la representación binaria. Para la codificación entera, al no aplicársele la mutación uniforme sino únicamente la aleatoria, sólo se ejecutó el programa 400 veces. De modo que el número total de ejecuciones fue de 1200 para cada uno de los problemas propuestos.

Algunos de los parámetros se mantuvieron fijos, sus valores fueron tomados de [14]: la probabilidad de cruza = 0.5 y el tamaño de la matriz = 5*5. Esto debido a que la cruza de dos puntos presenta sus mejores resultados haciendo uso de esta probabilidad de cruza (P_c). Y el tamaño de matriz se deriva de los resultados obtenidos por el programa base. El tamaño de la matriz no es un parámetro del AG, pero sí un valor importante, ya que forma parte de la representación de la solución para resolver el diseño del circuito [14, 37, 52].

Ahora se puede calcular el tamaño del cromosoma. Entonces L para la codificación binaria será:

$$L = 5 * 5 * 3 * 3 = 225$$

mientras que para la entera será de:

$$L = 5 * 5 * 3 = 75$$

[1]Como se mencionó en el capítulo anterior, la probabilidad de mutación (Pm) que se utiliza en el programa base (misma que se retoma en esta aplicación) es de $\frac{0.5}{L}$ (donde L es la longitud del cromosoma), y se implementó también la $P_m = \frac{1}{L}$.

Otros datos como el número de generaciones, el número de entradas y el número de salidas se especificarán en cada uno de los problemas.

Se tomaron tiempos de ejecución del programa; al no ser la medida usada para la evaluación del desempeño del AG, debido a que el tiempo varía según las características de la máquina, sólo se puede comentar que se pudo observar que el tiempo se incrementa en forma proporcional con respecto al número de individuos (por ejemplo, si con 100 individuos tarda 10 segundos, con 500 individuos tardará 50 segundos). Con respecto al tipo de representación, la entera es tres veces más rápida que la binaria. Y considerando las combinaciones entre los operadores genéticos (mutación y cruza), no existen cambios significativos. El AG trabaja con mayor velocidad sobre la plataforma Windows que en las plataformas Linux y Solaris (sólo se probó sobre diferentes versiones de Windows, y sobre los sistemas Red Hat 7.2 y Mandrake 8.1 de Linux y Solaris 8.0).

A continuación se muestran los circuitos obtenidos (el diseño y la función booleana) y el análisis comparativo de las diferentes variables estudiadas. Primero se hace un análisis de los datos conseguidos (porcentaje de circuitos factibles y óptimos) con la probabilidad de mutación $(P_m) = \frac{0.5}{L}$, los cuales son mostrados en dos tablas, después, se realiza la comparación de las soluciones encontradas con la $P_m = \frac{1}{L}$, también en otras dos tablas. Se continúa comparando las funciones halladas por el AG en Java con las otras técnicas. Y para terminar se proporcionan los parámetros necesarios para que pueda reproducir el diseño mostrado como resultado.

6.1 Circuitos de una salida

Los circuitos lógicos combinatorios expuestos en esta sección constan de varias entradas y una salida, los resultados expuestos se obtuvieron después de ejecutar el AG durante 1000 generaciones.

6.1.1 Ejemplo 1

La tabla de verdad de este circuito se visualiza en la tabla 6.1.

A	**B**	**C**	**F1**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Tabla 6.1: Tabla de verdad del primer circuito

Como se puede observar, este problema no es fácil de solucionar si se utilizan algunas de las técnicas tradicionales (mapas de Karnaugh o el método de Quine-McCluskey).

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	70%	100 %	100 %	100 %	100 %
		Uniforme	80 %	100 %	100 %	100 %	100 %
	Un Punto	Aleatoria	80 %	100 %	100 %	100 %	100 %
		Uniforme	90 %	100 %	100 %	100 %	100 %
	Uniforme	Aleatoria	90 %	100 %	100 %	100 %	100 %
		Uniforme	100 %	100 %	100 %	100 %	100 %
	Múltiples Puntos	Aleatoria	90 %	100 %	100 %	100 %	100 %
		Uniforme	90 %	100 %	100 %	100 %	100 %
Entera	Dos Puntos	Aleatoria	90 %	100 %	100 %	100 %	100 %
	Un Punto	Aleatoria	90 %	100 %	100 %	100 %	100 %
	Uniforme	Aleatoria	100 %	100 %	100 %	100 %	100 %
	Múltiples Puntos	Aleatoria	100 %	100 %	100 %	100 %	100 %

Tabla 6.2: Porcentaje de soluciones factibles halladas con una $P_m = \frac{0.5}{L}$.

En esta primera tabla (tabla 6.2) se puede observar que este problema no presenta mayores dificultades para el AG, ya que a paritr de los 500 individuos se obtienen en todos los casos soluciones factibles.

Algo que llama la atención es que para un tamaño de 100 individuos, la cruza uniforme y la cruza de múltiples puntos producen más soluciones factibles que la cruza de dos puntos y la de un punto. Es importante recordar que la cruza uniforme no es muy utilizada, y que la cruza de múltiples puntos es un nuevo operador de cruza desarrollado en este trabajo de tesis.

No hay que perder de vista que la representación entera, en este caso, también produce valores muy aceptables de soluciones factibles.

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	0%	50 %	60 %	50 %	50 %
		Uniforme	30%	20 %	40 %	60 %	80 %
	Un Punto	Aleatoria	10%	20 %	20 %	40 %	60 %
		Uniforme	20%	20 %	60 %	50 %	90 %
	Uniforme	Aleatoria	40%	10 %	0 %	20 %	10 %
		Uniforme	50%	20 %	20 %	10 %	40 %
	Múltiples Puntos	Aleatoria	30%	40 %	40 %	40 %	40 %
		Uniforme	10%	70 %	60 %	70 %	60 %
Entera	Dos Puntos	Aleatoria	10%	30 %	20 %	20 %	40 %
	Un Punto	Aleatoria	40%	40 %	20 %	70 %	30 %
	Uniforme	Aleatoria	10%	40 %	30 %	40 %	20 %
	Múltiples Puntos	Aleatoria	40%	40 %	10 %	20 %	20 %

Tabla 6.3: Porcentajes de soluciones óptimas obtenidas con una $P_m = \frac{0.5}{L}$.

Ahora se revisa la producción de circuitos óptimos (tabla 6.3). Como se puede observar, conforme se incrementa el tamaño de la población, la cruza de dos puntos y un punto incrementan su producción de soluciones óptimas; mientras que la recombinación uniforme las decrementa y la de múltiples

puntos la mantiene.

Se puede notar que la representación binaria trabaja mejor con mutación uniforme que con mutación aleatoria, y que la producción de soluciones óptimas supera a las obtenidas con la representación entera.

Es interesante notar que para este problema, la cruza de dos puntos aporta menos resultados óptimos que la cruza de un punto y la cruza uniforme. Esto es importante porque se ha dicho que la cruza de dos puntos es mejor que la cruza de un punto, y la uniforme no es muy usada porque tiene un alto efecto disruptivo.

Ahora se analizará el porcentaje de soluciones factibles con $P_m = \frac{1}{L}$ (tabla 6.4).

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	90 %	100 %	100 %	100 %	100 %
		Uniforme	100 %	100 %	100 %	100 %	100 %
	Un Punto	Aleatoria	100 %	100 %	100 %	100 %	100 %
		Uniforme	100 %	100 %	100 %	100 %	100 %
	Uniforme	Aleatoria	100 %	100 %	100 %	100 %	100 %
		Uniforme	100 %	100 %	100 %	100 %	100 %
	Múltiples Puntos	Aleatoria	90 %	100 %	100 %	100 %	100 %
		Uniforme	100 %	100 %	100 %	100 %	100 %
Entera	Dos Puntos	Aleatoria	90 %	100 %	100 %	100 %	100 %
	Un Punto	Aleatoria	90 %	100 %	100 %	100 %	100 %
	Uniforme	Aleatoria	100 %	100 %	100 %	100 %	100 %
	Múltiples Puntos	Aleatoria	90 %	100 %	100 %	100 %	100 %

Tabla 6.4: Porcentaje de soluciones factibles halladas con una $P_m = \frac{1}{L}$.

Como es notorio, con esta P_m, el porcentaje de soluciones factibles es mayor que el porcentaje proporcionado por la P_m vista con anterioridad. A partir de los 500 individuos el porcentaje de soluciones factibles es del 100 %, mientras que es del 90 % como mínimo con poblaciones de 100 individuos.

Es de llamar la atención que la cruza uniforme es la única que produce en todos los casos (tanto en los dos tipos de representación como con los diferentes operadores de mutación) circuitos factibles.

La tabla 6.5 muestra los circuitos óptimos alcanzados por el AG haciendo uso de esta probabilidad de mutación.

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	20 %	50 %	40 %	30 %	60 %
		Uniforme	30 %	60 %	50 %	60 %	60 %
	Un Punto	Aleatoria	40 %	70 %	80 %	70 %	70 %
		Uniforme	30 %	60 %	80 %	100 %	80 %
	Uniforme	Aleatoria	50 %	30 %	20 %	10 %	10 %
		Uniforme	40 %	40 %	50 %	20 %	0 %
	Múltiples Puntos	Aleatoria	30 %	60 %	60 %	40 %	50 %
		Uniforme	30 %	70 %	80 %	80 %	50 %
Entera	Dos Puntos	Aleatoria	10 %	40 %	70 %	60 %	50 %
	Un Punto	Aleatoria	20 %	30 %	50 %	70 %	90 %
	Uniforme	Aleatoria	50 %	10 %	10 %	20 %	60 %
	Múltiples Puntos	Aleatoria	10 %	40 %	20 %	40 %	40 %

Tabla 6.5: Porcentaje de soluciones óptimas encontradas con una $P_m = \frac{1}{L}$.

Como se puede observar en la tabla 6.5, el AG en general entrega más soluciones óptimas con este tipo de probabilidad.

Hablando de la representación, se puede comprobar que en promedio la representación binaria entrega más soluciones óptimas que la entera.

Es posible agregar que la cruza uniforme, por su efecto disruptivo, es mejor que trabaje con tamaños de población no mayores de 500 individuos cuando la probabilidad de la cruza es igual a 0.5.

Ahora se compara la función booleana obtenida por el AG en Java con las

encontradas por las otras técnicas evolutivas (tabla 6.6).

Programa	**Expresión Obtenida**	**Compuertas Usadas**
PB [14]	$F = ((A + B)C) \oplus (AB)$	2 ANDs, 1 OR, 1 XOR Total = 4
AG Basado en casos [37]	$F = (A + B)((AB) \oplus C)$	2 ANDs, 1 OR, 1 XOR Total = 4
PG prefija [71]	$F = (A + B)((AB) \oplus C)$	2 ANDs, 1 OR, 1 XOR Total = 4
Colonia de hormigas [52]	$F = (A + B)((AB) \oplus C)$	2 ANDs, 1 OR, 1 XOR Total = 4
AG en Java	$F = ((A + C)B) \oplus (AC)$	2 ANDs, 1 OR, 1 XOR Total = 4

Tabla 6.6: Comparación de la solución obtenida por el AG en Java con la de otros programas (Ejemplo 1).

Como se puede observar, se obtuvo el mismo número y tipo de compuertas para representar el diseño de este problema. Sin embargo, no es la misma función, puesto que difiere en el tipo de variables que utiliza para hacer las operaciones, pero el resultado es el mismo. Este es un ejemplo claro de que una función es posible expresarla de varias maneras, aún si se usa el mismo número y tipo de compuertas.

El resultado que se muestra en la tabla 6.6 del PB fue hallado en la generación 81, con una población de 300 individuos y haciendo uso de la representación entera (no se menciona con qué semilla se logra este resultado). Mientras que el de Islas [37] encuentra la solución en la generación 174 en la segunda ejecución del programa, usando una semilla de 0.85, con una población de 1000 individuos, y las mismas P_c y P_m que las que utiliza Coello [14], pero emplea la codificación binaria en lugar de la entera.

El diseño conseguido por el AG en Java lo muestra la figura 6.1, y para reproducirlo se requiere de los siguientes datos:

- Múltiples puntos como cruza.
- Con mutación aleatoria.
- Representación binaria.
- Con 500 individuos.
- $P_m = \frac{0.5}{L}$
- Semilla = 0.273
- 100 generaciones, 100 es el valor mínimo permitido por el programa.

Para este problema el óptimo fue encontrado en la generación 49. Se observa que este programa encontró el óptimo en menos generaciones que el programa base y el AG basado en casos, y es la nueva cruza propuesta en este trabajo de tesis la que lo logra.

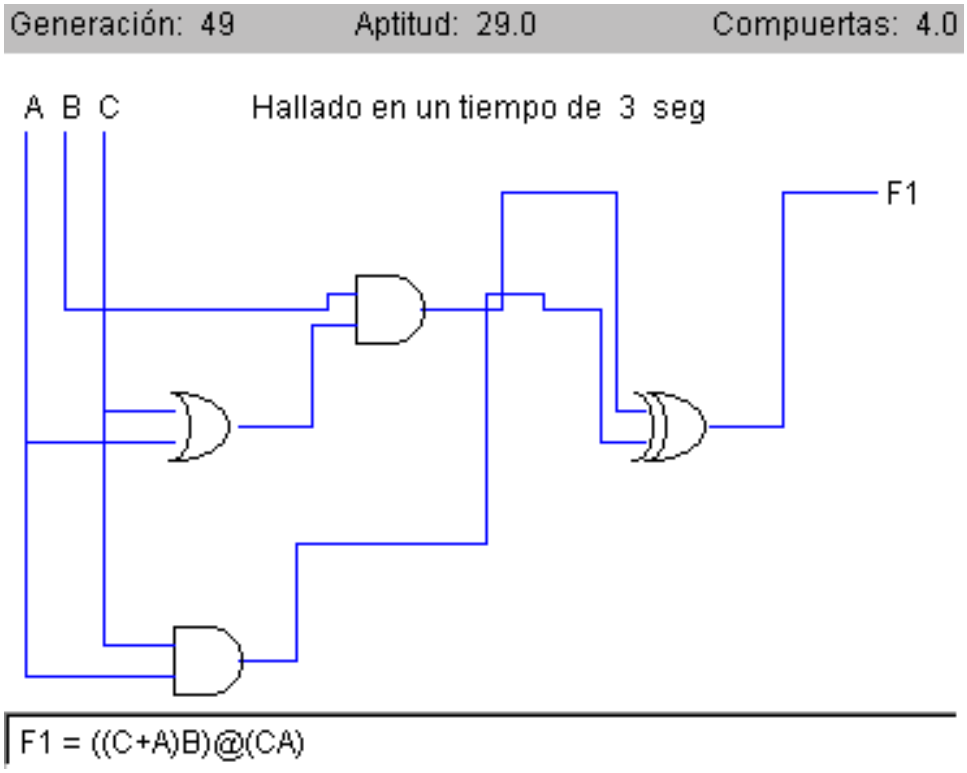


Figura 6.1: Diseño obtenido por el AG.

Para este problema puede ver otros resultados conseguidos por el AG en [20].

6.1.2 Ejemplo 2

La tabla 6.7 muestra la tabla de verdad de otro circuito de prueba.

A	B	C	D	F1
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1

A	B	C	D	F1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Tabla 6.7: Tabla de verdad del circuito de Sasao

La particularidad de este circuito es que se encuentra documentado en la literatura [13, 14, 52, 71]. Sasao [65] es quien lo utiliza para demostrar su técnica de simplificación, haciendo uso de compuertas ANDs y XORs. La función hallada por esta técnica se muestra en la tabla 6.12.

El simplificar este circuito por las técnicas tradicionales no es sencillo.

Como se presenta en la tabla 6.8, este problema resulta complicado para el AG. Como se puede ver el porcentaje de soluciones factibles es considerablemente baja ya que no sobrepasa el 40 % en la mayoría de las combinaciones.

En este ejemplo se observa la poca aportación de resultados factibles por parte de la cruza uniforme, siendo nula la aportación cuando se utiliza la representación entera. Se reafirma que en la representación binaria se trabaja mejor con una mutación uniforme.

El nuevo operador de cruza trabaja similar a la cruza de dos puntos y un punto, inclusive tiene un mejor comportamiento en la representación entera que la cruza de dos puntos y un punto, a lo que se refiere a la producción de resultados factibles.

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	10 %	20 %	30 %	20 %	20 %
		Uniforme	20 %	10 %	50 %	40 %	20 %
	Un Punto	Aleatoria	10 %	30 %	20 %	40 %	30 %
		Uniforme	10 %	10 %	40 %	30 %	60 %
	Uniforme	Aleatoria	10 %	0 %	0 %	10 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	10 %
	Múltiples Puntos	Aleatoria	0 %	10 %	30 %	40 %	10 %
		Uniforme	20 %	10 %	30 %	40 %	20 %
Entera	Dos Puntos	Aleatoria	20 %	0 %	30 %	40 %	20 %
	Un Punto	Aleatoria	10 %	40 %	0 %	40 %	40 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	10 %	30 %	30 %	30 %

Tabla 6.8: Porcentaje de soluciones factibles halladas con una $P_m = \frac{0.5}{L}$.

En la tabla 6.9 se muestra la producción de soluciones óptimas.

Es curioso observar que la cruza de dos puntos no produce circuitos óptimos en ninguna de las codificaciones (binaria y entera), aún cuando el número de soluciones factibles es alto. Lo mismo sucede con la cruza uniforme, que en este problema tiene un rendimiento deficiente.

Mientras que la nueva propuesta de cruza produce la gran parte de soluciones óptimas encontradas para este problema con $P_m = \frac{0.5}{L}$, aún teniendo un porcentaje menor al de la cruza de dos puntos en soluciones factibles.

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	0%	0 %	0 %	0 %	0 %
		Uniforme	0%	0 %	0 %	0 %	0 %
	Un Punto	Aleatoria	0%	0 %	0 %	0 %	0 %
		Uniforme	0%	0 %	10 %	0 %	0 %
	Uniforme	Aleatoria	0%	0 %	0 %	0 %	0 %
		Uniforme	0%	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0%	0 %	10 %	0 %	0 %
		Uniforme	10%	10 %	10 %	0 %	0 %
Entera	Dos Puntos	Aleatoria	0%	0 %	0 %	0 %	0 %
	Un Punto	Aleatoria	0%	0 %	0 %	0 %	0 %
	Uniforme	Aleatoria	0%	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0%	0 %	10 %	0 %	0 %

Tabla 6.9: Porcentajes de soluciones óptimas obtenidas con una $P_m = \frac{0.5}{L}$.

La tabla 6.10 muestra los resultados de soluciones factibles conseguidas por la otra P_m.

La cruza de dos puntos y la uniforme mejoran el rendimiento del AG con este tipo de P_m, ya que se ve un incremento en la producción de soluciones factibles, mientras que la cruza de un punto y la de múltiples puntos decrementa el porcentaje de soluciones factibles. Sin embargo, el porcentaje sigue siendo bajo, ya que la producción no supera el 40 % en la mayoria de los casos mostrados en la tabla 6.10.

La cruza de un punto tiene un buen comportamiento con esta probabilidad de mutación cuando se utiliza la representación entera, al igual que la cruza de dos puntos. En este caso ocurrío lo contrario a lo visto en la tabla 6.8.

Mientras que la cruza uniforme en combinación con la mutación uniforme en la representación binaria tiene un comportamiento mas continuo de producción de soluciones factibles que con la mutación aleatoria.

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	10 %	10 %	20 %	40 %	40 %
		Uniforme	30 %	20 %	20 %	40 %	40 %
	Un Punto	Aleatoria	0 %	30 %	20 %	20 %	20 %
		Uniforme	10 %	10 %	40 %	60 %	0 %
	Uniforme	Aleatoria	0 %	0 %	0 %	10 %	0 %
		Uniforme	10 %	10 %	10 %	10 %	20 %
	Múltiples Puntos	Aleatoria	0 %	10 %	20 %	30 %	30 %
		Uniforme	10 %	10 %	10 %	10 %	10 %
Entera	Dos Puntos	Aleatoria	10 %	30 %	30 %	40 %	10 %
	Un Punto	Aleatoria	20 %	30 %	30 %	40 %	20 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	10 %	0 %	10 %	20 %	20 %

Tabla 6.10: Porcentaje de soluciones factibles obtenidas con una $P_m = \frac{1}{L}$.

Sin embargo, el AG para resolver el circuito de Sasao no trabaja muy bien con esta P_m ($\frac{1}{L}$), puesto que no encuentra el mismo número de soluciones óptimas que con la P_m anterior como se observa en la tabla 6.11.

En la representación entera no se encuentra ningún circuito óptimo, mientras que en la codificación binaria con una población de 500 individuos sólo la cruza de un punto con mutación aleatoria y la cruza de múltiples puntos con mutación uniforme encuentran el 10 % de soluciones cada una; por otro lado con una población de 1500 individuos, la cruza de un punto con mutación uniforme consigue obtener también el mismo porcentaje de circuitos óptimos.

Estos resultados son muy interesantes, ya que es de llamar la atención que la cruza de dos puntos no genera ni un circuito óptimo, y la cruza de un punto y la de múltiples puntos son las únicas que los generán, dentro de los datos utilizados. Son importantes porque la de cruza de un punto no es muy utilizada debido al efecto destructivo de esquemas que tiene, mientras que la cruza de multiples puntos es un operador nuevo.

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Un Punto	Aleatoria	0 %	10 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	10 %	0 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	10 %	0 %	0 %	0 %
Entera	Dos Puntos	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Un Punto	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	0 %	0 %	0 %	0 %

Tabla 6.11: Porcentajes de soluciones óptimas obtenidas con una $P_m = \frac{1}{L}$.

La tabla 6.12 muestra las comparaciones de la funcion booleana conseguida por el AG en Java contra las encontradas por otras técnicas evolutivas.

Se puede observar que la técnica de Sasao encuentra un circuito con un número elevado de compuertas, cabe mencionar que el diseño se realiza en dos niveles.

El programa base encuentra un circuito con 10 compuertas, mientras que la PG prefija y la colonia de hormigas consiguen reducir el número de compuertas a siete. El AG en Java logra encontrar el circuito con siete compuertas también.

La función obtenida por la colonia de hormigas es similar a la lograda por el AG en Java. La diferencia radica en que en la primera se niega toda la operación XOR, mientras que en la segunda sólo se niega un elemento de la operación XOR, (la operación XOR negada es XNOR). Se puede demostrar que negar toda la operación XOR es igual a negar sólo un elemento de ésta.

Es de llamar la atención el poco uso de la compuerta NOT en los diseños

encontrados por las técnicas evolutivas.

Programa	**Expresión Obtenida**	**Compuertas Usadas**
Sasao [14, 71, 52]	$F = B' \oplus C'A' \oplus BC'D' \oplus B'C'A$	5 ANDs, 3 XORs, 4 NOTs. Total = 12
PB [14]	$F = (BDC' \oplus ((B + D) \oplus A \oplus (C + D + A)))'$	2 ANDs, 3 ORs, 3 XORs, 2 NOTs. Total = 10
PG prefija [71]	$F = (A + (C + D)') \oplus ((D \oplus B) + CD)$	1 ANDs, 3 ORs, 2 XORs, 1 NOT. Total = 7
Colonia de hormigas [52]	$F = (((D + C)(C \oplus A)) \oplus ((CD) + B)))'$	2 ANDs, 2 ORs, 2 XORs, 1 NOT. Total = 7
AG en Java	$F = ((C \oplus A)(C + D)) \oplus (B + (DC))'$	2 ANDs, 2 ORs, 2XORs, 1 NOT. Total = 7

Tabla 6.12: Comparación de la solución obtenida por el AG en Java con las de otros programas.

Al ser utilizadas únicamente las compuertas básicas y la compuerta XOR, se puede afirmar que el diseño de siete compuertas es el óptimo, ya que tiene el menor número de compuertas encontradas.

Para reproducir el diseño conseguido por el AG en Java (figura 6.2), se requieren los siguientes datos:

- Cruza de un punto.
- Con mutación aleatoria.
- Representación binaria.
- Con 500 individuos.
- $P_m = \frac{1}{L}$.
- Semilla = 0.273
- Con 102 generaciones.

Se eligió este resultado porque converge en la generación 102, y es el diseño que converge en menos generaciones que cualquier otro diseño alcanzado.

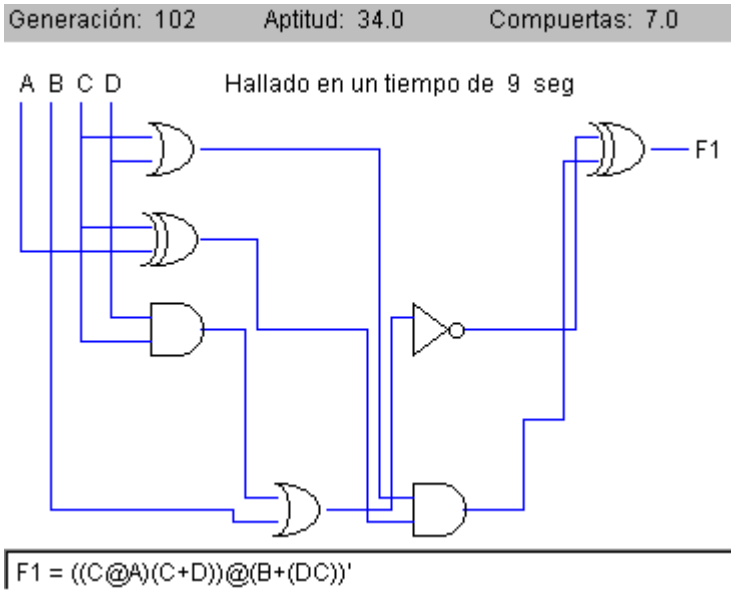


Figura 6.2: Diseño del circuito de Sasao obtenido por el AG.

6.1.3 Ejemplo 3

El tercer ejemplo es un circuito de seis variables de entrada con una salida.

Este ejemplo se visualiza en la tabla 6.13.

Para solucionar este problema se requiere de una matriz de 6*6, y se utilizan los mismos datos usados para resolver los circuitos de una salida.

Este problema es un circuito de paridad impar, y n es el número mínimo de compuertas para diseñarlo usando XOR, donde n es el número de entradas. El diseño se conforma de $n-1$ compuertas XORs y una NOT. Al ser el ejemplo un problema de paridad impar de seis variables, el diseño óptimo debe ser de cinco compuertas XORs y una NOT, dando un total de seis compuertas.

Es posible determinar la aptitud que puede alcanzar el individuo, que es:

$$f(x) = 2^6 * 1 + 6 * 6 - 6 = 94.$$

Por la interfaz implementada este es el problema con mayor número de entradas al cual se le puede buscar una solución.

A	**B**	**C**	**D**	**E**	**F**	**F1**
0	0	0	0	0	0	1
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	1
0	0	0	1	0	0	0
0	0	0	1	0	1	1
0	0	0	1	1	0	1
0	0	0	1	1	1	0
0	0	1	0	0	0	0
0	0	1	0	0	1	1
0	0	1	0	1	0	1
0	0	1	0	1	1	0
0	0	1	1	0	0	1
0	0	1	1	0	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	1
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	1
0	1	0	0	1	1	0
0	1	0	1	0	0	1
0	1	0	1	0	1	0
0	1	0	1	1	0	0
0	1	0	1	1	1	1
0	1	1	0	0	0	1
0	1	1	0	0	1	0
0	1	1	0	1	0	0
0	1	1	0	1	1	1
0	1	1	1	0	0	0
0	1	1	1	0	1	1
0	1	1	1	1	0	1
0	1	1	1	1	1	0

A	**B**	**C**	**D**	**E**	**F**	**F1**
1	0	0	0	0	0	0
1	0	0	0	0	1	1
1	0	0	0	1	0	1
1	0	0	0	1	1	0
1	0	0	1	0	0	1
1	0	0	1	0	1	0
1	0	0	1	1	0	0
1	0	0	1	1	1	1
1	0	1	0	0	0	1
1	0	1	0	0	1	0
1	0	1	0	1	0	0
1	0	1	0	1	1	1
1	0	1	1	0	0	0
1	0	1	1	0	1	1
1	0	1	1	1	0	1
1	0	1	1	1	1	0
1	1	0	0	0	0	1
1	1	0	0	0	1	0
1	1	0	0	1	0	0
1	1	0	0	1	1	1
1	1	0	1	0	0	0
1	1	0	1	0	1	1
1	1	0	1	1	0	1
1	1	0	1	1	1	0
1	1	1	0	0	0	0
1	1	1	0	0	1	1
1	1	1	0	1	0	1
1	1	1	0	1	1	0
1	1	1	1	0	0	1
1	1	1	1	0	1	0
1	1	1	1	1	0	0
1	1	1	1	1	1	1

Tabla 6.13: Tabla de verdad del primer circuito

Como se puede observar, este problema no es fácil de solucionar si se utilizan algunas de las técnicas tradicionales.

En esta primera tabla (tabla 6.14) se puede observar que este problema no presenta mayores dificultades para el AG, ya que a partir de los 500 individuos se obtienen en todos los casos soluciones factibles.

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Entera	Dos Puntos	Aleatoria	70 %	100 %	100 %	100 %	100 %
	Un Punto	Aleatoria	80 %	100 %	100 %	100 %	100 %
	Uniforme	Aleatoria	90 %	100 %	100 %	100 %	100 %
	Múltiples Puntos	Aleatoria	90 %	100 %	100 %	100 %	100 %

Tabla 6.14: Porcentaje de soluciones factibles halladas con una $P_m = \frac{0.5}{L}$.

Algo que llama la atención es que para un tamaño de 100 individuos, la cruza uniforme y la cruza de múltiples puntos producen más soluciones factibles que la cruza de dos puntos y la de un punto. Es importante recordar que la cruza uniforme no es muy utilizada, y que la cruza de múltiples puntos es un nuevo operador de cruza desarrollado en este trabajo de tesis.

Ahora se revisa la producción de circuitos óptimos (tabla 6.15).

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Entera	Dos Puntos	Aleatoria	30%	100 %	100 %	100 %	100 %
	Un Punto	Aleatoria	30%	90 %	100 %	100 %	100 %
	Uniforme	Aleatoria	30%	0 %	10 %	0 %	10 %
	Múltiples Puntos	Aleatoria	40%	100 %	100 %	100 %	100 %

Tabla 6.15: Porcentajes de soluciones óptimas obtenidas con una $P_m = \frac{0.5}{L}$.

Como se puede observar, la cruza de dos puntos, un punto y múltiples puntos generan a partir de los 500 individuos el 100 % de soluciones óptimas;

mientras que la aportación de la recombinación uniforme es mínima.

Ahora se analizará el porcentaje de soluciones factibles con $P_m = \frac{1}{L}$ (tabla 6.16).

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Entera	Dos Puntos	Aleatoria	100 %	100 %	100 %	100 %	100 %
	Un Punto	Aleatoria	100 %	100 %	100 %	100 %	100 %
	Uniforme	Aleatoria	100 %	100 %	100 %	100 %	100 %
	Múltiples Puntos	Aleatoria	90 %	100 %	100 %	100 %	100 %

Tabla 6.16: Porcentaje de soluciones factibles halladas con una $P_m = \frac{1}{L}$.

Como es notorio, con esta P_m, el porcentaje de soluciones factibles es mayor que el porcentaje proporcionado por la P_m vista con anterioridad. A partir de los 500 individuos el porcentaje de soluciones factibles es del 100 %, mientras que es del 90 % como mínimo con poblaciones de 100 individuos.

La tabla 6.5 muestra los circuitos óptimos alcanzados por el AG haciendo uso de esta probabilidad de mutación. Como se puede observar el AG en general entrega más soluciones óptimas con este tipo de probabilidad.

Este problema no ha sido utilizado por las otras técnicas evolutivas, por lo tanto no es posible hacer una comparación. Sin embargo, se puede concluir que el AG en Java tiene un buen desempeño, en lo referente a este problema.

El diseño encontrado en menos iteraciones (converge en 66 generaciones) lo muestra la figura 6.3. Para reproducirlo se requiere de los siguientes datos:

- Cruza de un punto.
- Con mutación aleatoria.
- Representación entera.
- Con 2000 individuos.
- $P_m = \frac{0.5}{L}$.
- Semilla = 0.364

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Entera	Dos Puntos	Aleatoria	60 %	100 %	100 %	100 %	100 %
	Un Punto	Aleatoria	80 %	100 %	100 %	100 %	100 %
	Uniforme	Aleatoria	10 %	20 %	0 %	10 %	10 %
	Múltiples Puntos	Aleatoria	70 %	100 %	100 %	100 %	100 %

Tabla 6.17: Porcentaje de soluciones óptimas encontradas con una $P_m = \frac{1}{L}$.

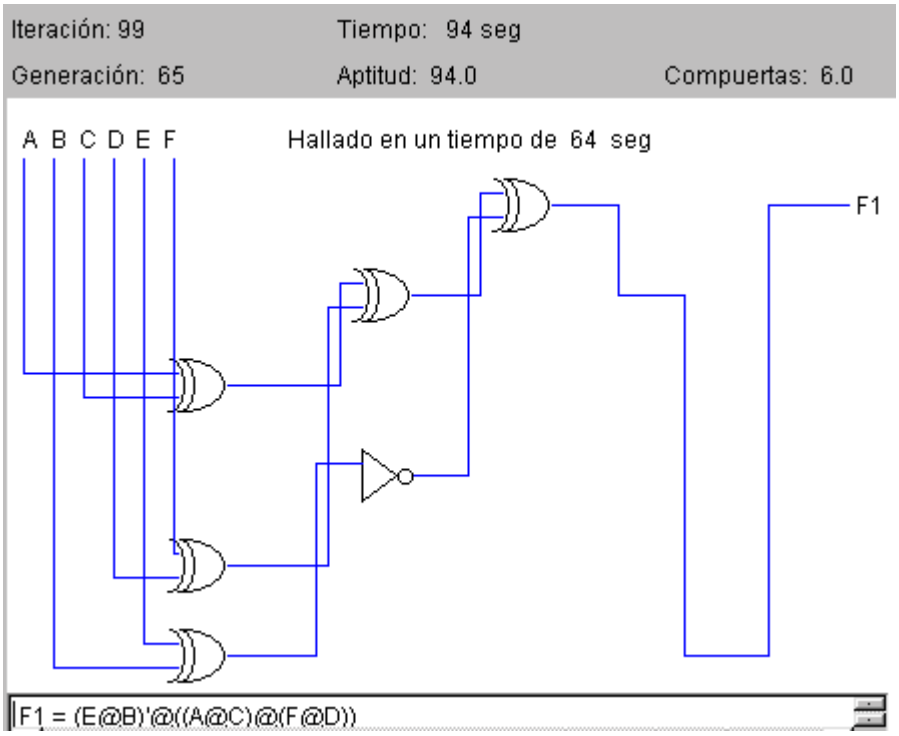


Figura 6.3: Diseño del circuito de paridad impar seis.

6.2 Circuitos de Múltiples Salidas

En esta sección se mostrarán los resultados obtenidos al probar el programa con circuitos de múltiples salidas. Al igual que los anteriores ejemplos, se

manejaron los mismos datos para la P_c y para el tamaño de la matriz, sólo que esta vez se utilizaron 3000 generaciones.

El valor de la longitud del cromosoma es el mismo que se calculó para los ejemplos anteriores. Sin embargo, para el segundo circuito que se mostrará en esta sección, fue necesario cambiar el tamaño de la matriz de 5*5 a 7*7. Esto se debe a que sólo es posible hallar la solución con este tamaño (tamaño mínimo) [14, 52]. Por lo tanto la longitud del cromosoma en el segundo ejemplo será de 147 para cuando la representación utilizada sea la entera, mientras que para la binaria será de 441.

Otros diseños obtenidos por el AG en Java para estos problemas se encuentran en [20, 21].

6.2.1 Sumador de dos bits

La tabla de verdad de este problema se muestra en la tabla 6.18.

A	B	C	D	F3	F2	F1
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0

A	B	C	D	F3	F2	F1
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Tabla 6.18: Tabla de verdad del sumador de dos bits

Es notorio que al incrementar el número de salidas, la efectividad del AG para generar soluciones factibles se ve reducida (como se observa en la tabla 6.19). Sin embargo, para las cruzas de dos puntos, un punto y la de múltiples puntos no es tan significativo, no así para la cruza uniforme, ya que su aportación de circuito factibles es mínima y se puede remarcar que esta cruza trabaja mejor cuando el número de individuos no sea mayor de 100.

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	10 %	80 %	20 %	90 %	90 %
		Uniforme	40 %	80 %	90 %	90 %	90 %
	Un Punto	Aleatoria	20 %	50 %	70 %	60 %	100 %
		Uniforme	30 %	50 %	90 %	80 %	100 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	40 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	20 %	20 %	80 %	60 %	50 %
		Uniforme	0 %	80 %	80 %	70 %	80 %
Entera	Dos Puntos	Aleatoria	50 %	80 %	90 %	100 %	70 %
	Un Punto	Aleatoria	20 %	70 %	70 %	90 %	90 %
	Uniforme	Aleatoria	50 %	0 %	10 %	0 %	0 %
	Múltiples Puntos	Aleatoria	60 %	50 %	70 %	70 %	90 %

Tabla 6.19: Porcentaje de soluciones factibles halladas con una $P_m = \frac{0.5}{L}$.

Se repite la tendencia a incrementar el número de soluciones factibles al incrementarse la población, aunque no hay que olvidar que al incrementar el número de individuos se incrementa el tiempo de ejecución.

Ahora hay que verificar el número de soluciones óptimas obtenidas (tabla 6.20).

Los resultados obtenidos no son muy buenos. Lo que llama la atención es que la cruza de dos puntos trabaja mejor con representación binaria que con representación entera (en este caso), mientras que los otros dos tipos de cruza que producen un número considerable de circuitos óptimos lo hacen con la representación entera en lugar de la binaria.

El nuevo operador de cruza sigue produciendo un número considerable de soluciones óptimas, inclusive, llegando a superar a la cruza de dos puntos (cuando utiliza la codificación entera).

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	0%	0 %	0 %	20 %	20 %
		Uniforme	0%	10 %	20 %	0 %	10 %
	Un Punto	Aleatoria	0%	20 %	10 %	0 %	10 %
		Uniforme	0%	0 %	10 %	0 %	10 %
	Uniforme	Aleatoria	0%	0 %	0 %	0 %	0 %
		Uniforme	0%	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0%	0 %	10 %	0 %	0 %
		Uniforme	0%	10 %	10 %	0 %	10 %
Entera	Dos Puntos	Aleatoria	0%	10 %	10 %	0 %	10 %
	Un Punto	Aleatoria	0%	30 %	10 %	40 %	20 %
	Uniforme	Aleatoria	10%	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0%	0 %	30 %	10 %	30 %

Tabla 6.20: Porcentajes de soluciones óptimas obtenidas con una $P_m = \frac{0.5}{L}$.

La tabla (tabla 6.21) que se muestra a continuación expone los porcentajes de soluciones factibles cuando la $P_m = \frac{1}{L}$. Este valor de P_m favorece, en algunos casos, el incremento de las soluciones factibles, ya que en otros se observa un comportamiento poco constante.

Aunque se ha visto que la mutación uniforme es la que entrega un mayor número de soluciones factibles que la mutación aleatoria. Cuando la representación usada es la binaria, con este tipo de P_m ocurre lo contrario.

No hay que perder de vista a la cruza de un punto. En los ejemplos que se han visto hasta el momento, es el tipo de cruza que se mantiene constante en lo que se refiere al porcentaje de soluciones (entrega un número considerable de soluciones factibles), el cual inclusive llega a ser más alto que el de la cruza de dos puntos.

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	10 %	90 %	60 %	90 %	80 %
		Uniforme	60 %	30 %	60 %	50 %	60 %
	Un Punto	Aleatoria	30 %	60 %	90 %	100 %	90 %
		Uniforme	40 %	70 %	80 %	90 %	100 %
	Uniforme	Aleatoria	20 %	0 %	0 %	0 %	0 %
		Uniforme	10 %	10 %	10 %	0 %	0 %
	Múltiples Puntos	Aleatoria	40 %	50 %	60 %	70 %	70 %
		Uniforme	30 %	0 %	30 %	0 %	20 %
Entera	Dos Puntos	Aleatoria	70 %	50 %	80 %	90 %	90 %
	Un Punto	Aleatoria	50 %	100 %	90 %	100 %	90 %
	Uniforme	Aleatoria	20 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	60 %	40 %	50 %	90 %	70 %

Tabla 6.21: Porcentaje de soluciones factibles obtenidas con una $P_m = \frac{1}{L}$.

La tabla 6.22 presenta los porcentajes de soluciones óptimas.

Si bien la P_m llega a incrementar el número de soluciones factibles, no ocurre lo mismo con la cantidad de circuitos óptimos. Como muestra la tabla 6.22, la aportación de éstos es mucho menor que la que se entrega con la $P_m = \frac{0.5}{L}$. Por lo tanto se puede concluir que esta última P_m es mejor.

Es de llamar la atención que para la representación binaria la cruza de múltiples puntos en combinación con la mutación aleatoria y con este tipo de P_m ($\frac{1}{L}$) generan más circuitos óptimos que cualquiera de las otras combinaciones. Ya que genera por lo menos un 10 % de soluciones óptimas apartir de los 500 individuos, mientras que para las otras combinaciones el porcentaje de soluciones óptimas no es muy constante. En el caso de la representación entera, es la cruza de un punto la que tiene un comportamiento mas constante.

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	0 %	10 %	0 %	20 %	0 %
		Uniforme	0 %	10 %	0 %	10 %	0 %
	Un Punto	Aleatoria	0 %	0 %	0 %	30 %	10 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	10 %	30 %	10 %	20 %
		Uniforme	10 %	0 %	0 %	0 %	0 %
Entera	Dos Puntos	Aleatoria	0 %	0 %	10 %	0 %	40 %
	Un Punto	Aleatoria	0 %	20 %	0 %	20 %	30 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	10 %	0 %	0 %	0 %	0 %

Tabla 6.22: Porcentaje de soluciones óptimas encontradas con una $P_m = \frac{1}{L}$.

La función booleana lograda por el AG en Java se compara con las encontradas por las otras técnicas evolutivas (tabla 6.23).

Es curioso observar que los resultados obtenidos por los otros algoritmos y el AG en Java sean iguales, exceptuando la función (F3) encontrada por la colonia de hormigas. Lo que se puede concluir es que el número mínimo de compuertas que pueden representar a este circuito con la ayuda de la compuerta XOR es de siete.

En la figura 6.4 se muestra el diseño encontrado por el AG en Java. Si se desea repetirlo requerirá de los siguientes datos:

- Cruza de un punto.
- Con mutación aleatoria.
- Representación entera.
- Con 1500 individuos.
- $P_m = \frac{0.5}{L}$.
- Semilla = 0.273
- Con 481 generaciones.

Programa	**Expresión Obtenida**	**Compuertas Usadas**
PB [52]	$F1 = B \oplus D$ $F2 = BD \oplus (A \oplus C)$ $F3 = AC + BD(A \oplus C)$	3 ANDs, 1 OR, 3 XORs Total = 7
PG prefija [71]	$F1 = B \oplus D$ $F2 = BD \oplus (A \oplus C)$ $F3 = AC + BD(A \oplus C)$	3 ANDs, 1 OR, 3 XORs Total = 7
Colonia de hormigas [52]	$F1 = B \oplus D$ $F2 = BD \oplus (A \oplus C)$ $F3 = BD(A \oplus C) \oplus AB$	3 ANDs, 4 XORs, Total = 7
AG en Java	$F1 = B \oplus D$ $F2 = BD \oplus (A \oplus C)$ $F3 = AC + BD(A \oplus C)$	3 ANDs, 1 OR, 3 XORs Total = 7

Tabla 6.23: Comparación de la solución obtenida por el AG en Java con las de otros programas.

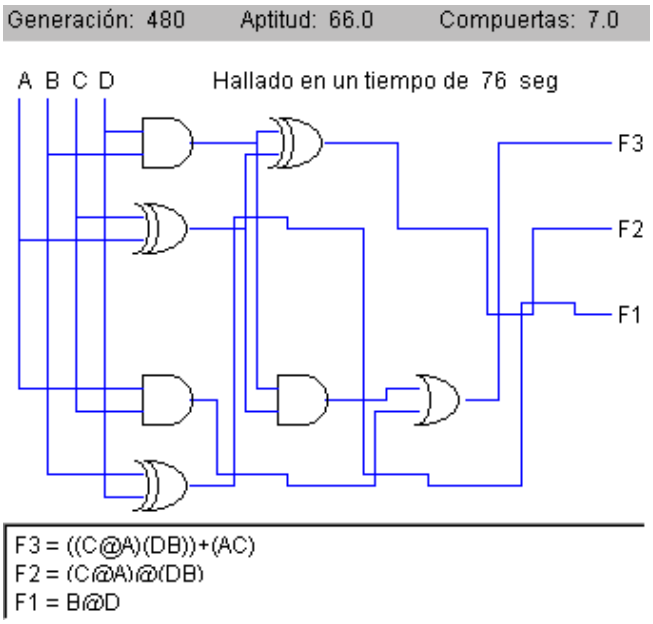


Figura 6.4: Diseño obtenido por el AG.

6.2.2 Comparador de dos bits

La tabla de verdad que muestra la tabla 6.24, es la del comparador de dos bits. A continuación se verá que la complejidad de este circuito hace que el desempeño del AG sea pobre.

A	B	C	D	F3	F2	F1
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0

A	B	C	D	F3	F2	F1
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

Tabla 6.24: Tabla de verdad del comparador de dos bits.

La tabla 6.25 muestra los porcentajes de circuitos factibles hallados por el AG con la $P_m = \frac{0.5}{L}$. Se puede observar que el número de soluciones factibles es alto.

Es de llamar la atención que en la representación entera, la cruza de dos puntos y un punto tienen el 100 % de los circuitos factibles con 2000 individuos.

La cruza de uno, dos y múltiples puntos en combinación con la mutación uniforme en la representación entera, tienen un porcentaje constante y alto de soluciones factibles después de 500 individuos generando un 60 % de circuitos factibles como mínimo. No así la cruza uniforme, ya que su producción de este tipo de circuitos es casi nula. Estos resultados reafirman que el AG tiene un buen desempeño al usar la representación binaria siempre que se utilize la mutación uniforme.

En la representación entera sucede algo similar, aunque el número de soluciones que entrega es un poco mayor.

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	10%	50 %	50 %	60 %	60 %
		Uniforme	30 %	70 %	80 %	70 %	90 %
	Un Punto	Aleatoria	10 %	30 %	70 %	80 %	50 %
		Uniforme	30 %	60 %	80 %	100 %	90 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	20 %	0 %	0 %	10 %	0 %
	Múltiples Puntos	Aleatoria	0 %	20 %	70 %	60 %	60 %
		Uniforme	20 %	80 %	70 %	80 %	70 %
Entera	Dos Puntos	Aleatoria	70 %	50 %	90 %	80 %	100 %
	Un Punto	Aleatoria	30 %	70 %	70 %	100 %	100 %
	Uniforme	Aleatoria	20 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	20 %	80 %	90 %	80 %	80 %

Tabla 6.25: Porcentaje de soluciones factibles halladas con una $P_m = \frac{0.5}{L}$.

Ahora se analizarán los porcentajes de soluciones óptimas alcanzadas por el AG en Java, para ello se observará la tabla 6.26.

Se nota que para este problema el AG sólo encuentra una solución. Para reproducirla se requiere de los siguientes datos:

- Representación entera.
- Cruza de un punto.
- Con mutación aleatoria.
- Con 2000 individuos.
- $P_m = \frac{0.5}{L}$
- Semilla = 0.6369999999999999.
- Tamaño de la matriz 7*7.
- En 1432 generaciones.

El resultado se encuentra en la iteración 1432.

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Un Punto	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
Entera	Dos Puntos	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Un Punto	Aleatoria	0 %	0 %	0 %	0 %	10 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	0 %	0 %	0 %	0 %

Tabla 6.26: Porcentajes de soluciones óptimas obtenidas con una $P_m = \frac{0.5}{L}$.

La tabla 6.27 expone los porcentajes de las soluciones factibles halladas con la otra P_m.

Como se mencionó con anterioridad esta P_m empeora el rendimiento del AG cuando la cruza que se utiliza es la de múltiples puntos, e incrementa el número de soluciones factibles cuando la cruza usada es la de dos puntos o la de un punto.

Con la $P_m = \frac{1}{L}$ ocurre lo contrario, ya que como se puede observar el AG con representación binaria tiene un mejor desempeño cuando se utiliza la mutación aleatoria en lugar de la uniforme. Y con la representación entera disminuye su aportación de circuitos factibles.

La aportación de la cruza uniforme continua siendo mínima.

Para este problema es mejor la utilización de la $P_m = \frac{0.5}{L}$, debido a que tiene un porcentaje mas alto al encontrar un diseño factible. Al igual que con la P_m anterior sólo se encuentra un circuito óptimo.

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	20 %	40 %	80 %	90 %	100 %
		Uniforme	50 %	60 %	50 %	70 %	50 %
	Un Punto	Aleatoria	10 %	60 %	60 %	70 %	90 %
		Uniforme	30 %	70 %	70 %	40 %	30 %
	Uniforme	Aleatoria	20 %	0 %	0 %	0 %	0 %
		Uniforme	20 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	30 %	20 %	50 %	50 %	70 %
		Uniforme	10 %	10 %	10 %	20 %	10 %
Entera	Dos Puntos	Aleatoria	50 %	80 %	50 %	50 %	40 %
	Un Punto	Aleatoria	60 %	70 %	70 %	80 %	90 %
	Uniforme	Aleatoria	0 %	10 %	0 %	0 %	10 %
	Múltiples Puntos	Aleatoria	40 %	40 %	40 %	50 %	50 %

Tabla 6.27: Porcentaje de soluciones factibles obtenidas con una $P_m = \frac{1}{L}$.

El diseño conseguido por el AG en Java lo muestra la figura 6.5.

Para poder reproducirlo se necesita de los siguientes datos:

- Múltiples puntos como cruza.
- Con mutación aleatoria.
- Tamaño de la matriz 7*7.
- Representación binaria.
- Con 2000 individuos.
- $P_m = \frac{1}{L}$.
- Semilla = 0.6369999999999999
- Con 217 generaciones.

Este resultado es el que se muestra en [21] y en la figura 6.5, porque como se puede apreciar, se encuentra la solución en mucho menos generaciones que la que se mencionó con anterioridad.

La tabla 6.28 expone los porcentajes de circuitos óptimos logrados por el AG en Java con la $p_m = \frac{1}{L}$.

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Un Punto	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	0 %	0 %	0 %	10 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
Entera	Dos Puntos	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Un Punto	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	0 %	0 %	0 %	0 %

Tabla 6.28: Porcentajes de soluciones óptimas obtenidas con una $P_m = \frac{1}{L}$.

Lo que que no hay que olvidar es que el AG en Java logra encontrar la solución óptima con la cruza de un punto y con la cruza de múltiples puntos, mientras que la cruza de dos puntos no encuentra el óptimo. Se hace énfasis en esto porque el programa base sólo tiene el operador de cruza de dos puntos.

La tabla 6.29 muestra las comparaciones de las funciones obtenidas por el AG en Java con las ya mencionadas.

Como se puede ver, en este ejemplo es difícil hallar la solución que contenga el número mínimo de compuertas. Es decir, el espacio donde se encuentra la representación mínima es muy pequeño, y de ahí la dificultad de encontrar esta solución.

Es de llamar la atención que los algoritmos que están más cerca de la solución óptima representan la función F3 de la misma manera, con lo que podemos concluir que esa es la única manera de representar esta función.

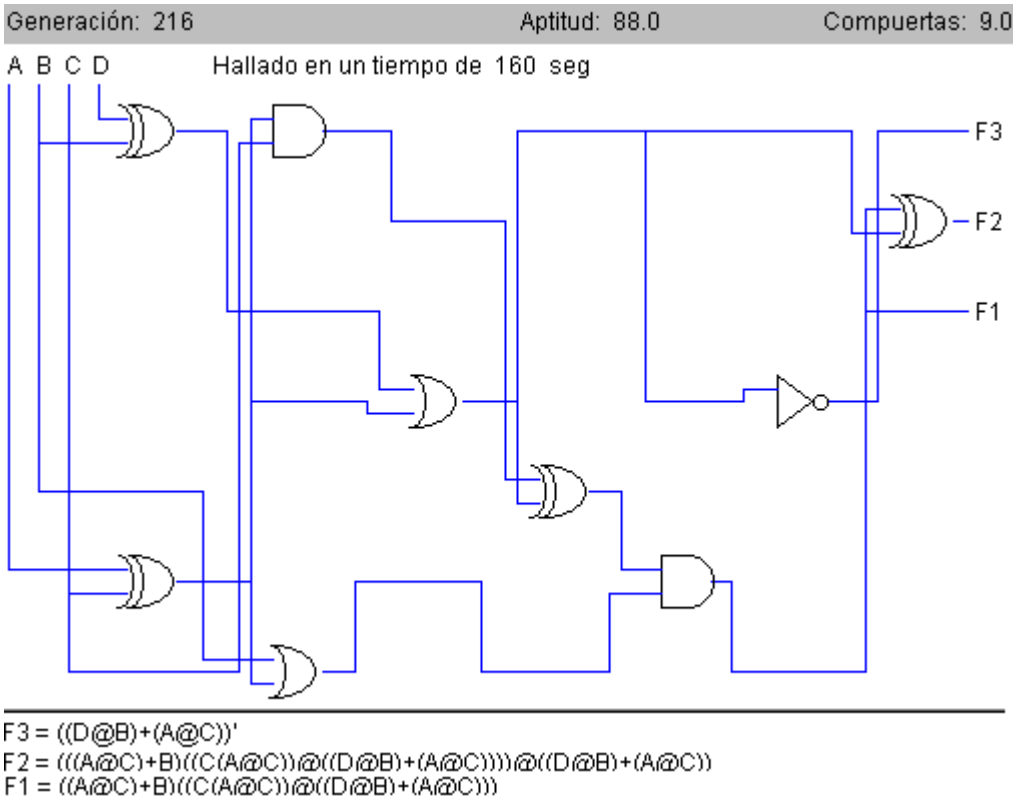


Figura 6.5: Diseño obtenido por el AG.

El resultado que muestra la tabla 6.29 del programa base fue encontrado con un tamaño de población de 2800 individuos y la encuentra en la generación 1059, haciendo uso de una matriz de 6*7. Sin embargo, no indica con qué semilla inicial fue hallado. Como puede verse, el AG en Java logra encontrar una solución óptima con una cantidad menor de individuos y con un menor número de generaciones.

Mientras que el AG basado en casos, encuentra la solución que se muestra en la tabla 6.29 con un tamaño de población de 490 individuos, con un tamaño de matriz de 6*7, con una semilla inicial de 0.139, y lo encuentra en la generación 1586.

Programa	**Expresión Obtenida**	**Compuertas Usadas**
PB [14]	$F1 = (F3 + W)'$ $F2 = F3 + ((B \oplus D) + (A \oplus C))W$ $F3 = ((A \oplus C) + (B \oplus D))'$ Donde: $W = (A \oplus C)A \oplus (D + (A \oplus C)$	2 ANDs, 3 ORs, 3 XORs, 2 NOTs Total = 10
AG Basado en casos [37]	$F1 = NM$ $F2 = M'$ $F3 = N'$ Donde: $N = ((A \oplus B)) + (B \oplus D))$ $M = (((D + ((B \oplus D)') + (A \oplus B)) \oplus (A(A \oplus B)))$	2 ANDs, 3 ORs, 3 XORs, 3 NOTs Total = 11
PG prefija [71]	$F1 = ((D'B \oplus A) + (A \oplus C)) \oplus C$ $F2 = ((B + ((A \oplus C) + D')) \oplus (C' + A)')'$ $F3 = ((A \oplus C) + D') \oplus ((A \oplus C) + B)$	1 AND, 5 ORs, 5 XORs, 4 NOTs Total = 15
Colonia de hormigas [52]	$F1 = F2' \oplus F3$ $F2 = ((CA) \oplus A) \oplus ((A \oplus C) + (B \oplus D))((A \oplus C) + D)$ $F3 = ((A \oplus C) + (B \oplus D))'$	2 ANDs, 2 ORs, 5 XORs, 2 NOTs Total = 11
AG en Java	$F1 = ((A \oplus C) + B)((C(A \oplus C)) \oplus ((A \oplus C) + (B \oplus D)))$ $F2 = F1 \oplus ((A \oplus C) + (B \oplus D))$ $F3 = ((A \oplus C) + (B \oplus D))'$	2 ANDs, 2 ORs, 4 XORs, 1 NOT Total = 9

Tabla 6.29: Comparación de la solución obtenida por el AG en Java con las de otros programas.

6.2.3 Multiplicador de dos bits

Otro circuito probado fue el multiplicador de dos bits y su tabla de verdad se muestra en la tabla 6.30.

La tabla 6.31 muestra los porcentajes de las soluciones factibles obtenidas con la $P_m = \frac{0.5}{L}$.

A	B	C	D	F4	F3	F2	F1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1

A	B	C	D	F4	F3	F2	F1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Tabla 6.30: Tabla de verdad del multiplicador de 2 bits.

En la tabla 6.31 se puede observar que al incrementar el número de salidas el desempeño del AG disminuye. Sin embargo, se sigue produciendo un número considerable de circuitos factibles.

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	0%	10 %	80 %	40 %	70 %
		Uniforme	10 %	30 %	70 %	90 %	40 %
	Un Punto	Aleatoria	20 %	40 %	50 %	50 %	40 %
		Uniforme	30 %	70 %	40 %	70 %	60 %
	Uniforme	Aleatoria	10 %	10 %	0 %	0 %	0 %
		Uniforme	50 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	20 %	20 %	40 %	20 %
		Uniforme	40 %	50 %	50 %	70 %	50 %
Entera	Dos Puntos	Aleatoria	30 %	50 %	70 %	30 %	80 %
	Un Punto	Aleatoria	50 %	20 %	60 %	50 %	50 %
	Uniforme	Aleatoria	20 %	20 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	20 %	30 %	30 %	80 %	40 %

Tabla 6.31: Porcentaje de soluciones factibles halladas con una $P_m = \frac{0.5}{L}$.

La cruza uniforme es la que aporta menos, y se ratifica que da más soluciones cuando se tiene un número pequeño de individuos.

Podemos ver que la cruza de un punto es más constante en el número de soluciones factibles con la representación entera.

La tabla 6.32 muestra que para este problema es necesario un número considerable de individuos para poder encontrar soluciones óptimas.

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	0 %	0 %	10 %	0 %	0 %
		Uniforme	0 %	20 %	10 %	20 %	0 %
	Un Punto	Aleatoria	0 %	0 %	10 %	0 %	10 %
		Uniforme	0 %	20 %	0 %	20 %	10 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	10 %	10 %	20 %	10 %
		Uniforme	0 %	10 %	20 %	10 %	20 %
Entera	Dos Puntos	Aleatoria	0 %	0 %	20 %	10 %	20 %
	Un Punto	Aleatoria	0 %	0 %	10 %	10 %	30 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	10 %	0 %	30 %	0 %

Tabla 6.32: Porcentajes de soluciones óptimas obtenidas con una $P_m = \frac{0.5}{L}$.

Como podemos observar en la tabla 6.32 la cruza de múltiples puntos es la única que devuelve soluciones óptimas a partir de los 500 individuos, sin importar la mutación que se utilizó. Por otro lado, la cruza de dos puntos y un punto son más constantes en lo que se refiere al número de soluciones óptimas.

Es interesante observar que aunque la cruza de múltiples puntos genera menos soluciones factibles que la cruza de uno y dos puntos, devuelve un mayor número de circuitos óptimos (en esta P_m).

Ahora se analizan los resultados obtenidos con la $P_m = \frac{1}{L}$. La tabla 6.33 muestra estos resultados, y como es visible, el número de soluciones factibles se reduce. Sin embargo, no hay que perder de vista que la cruza uniforme se ve favorecida por este tipo de P_m, porque devuelve circuitos factibles aún cuando el número de individuos supere a los 500.

Representación	Cruza	Mutación	Soluciones Factibles por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	50 %	40 %	60 %	70 %	50 %
		Uniforme	50 %	10 %	40 %	40 %	20 %
	Un Punto	Aleatoria	40 %	60 %	30 %	70 %	60 %
		Uniforme	40 %	20 %	50 %	60 %	30 %
	Uniforme	Aleatoria	20 %	10 %	20 %	20 %	10 %
		Uniforme	10 %	20 %	10 %	10 %	0 %
	Múltiples Puntos	Aleatoria	10 %	30 %	50 %	40 %	50 %
		Uniforme	60 %	30 %	0 %	0 %	0 %
Entera	Dos Puntos	Aleatoria	30 %	70 %	60 %	70 %	40 %
	Un Punto	Aleatoria	10 %	50 %	40 %	50 %	40 %
	Uniforme	Aleatoria	20 %	0 %	0 %	0 %	10 %
	Múltiples Puntos	Aleatoria	10 %	20 %	20 %	40 %	50 %

Tabla 6.33: Porcentaje de soluciones factibles obtenidas con una $P_m = \frac{1}{L}$.

Esto ocurre con la representación binaria, ya que para la entera, la cruza uniforme hace que el AG tenga una pobre aportación de circuitos factibles.

También es visible que este tipo de P_m desfavorece a la cruza de múltiples puntos con mutación uniforme, ya que no produce soluciones factibles a paritr de los 1000 individuos.

Al utilizar la $P_m = \frac{1}{L}$ en este problema podemos observar que la mutación aleatoria permite un mayor número de soluciones factibles.

Ahora se revisan las soluciones óptimas, presentadas en la tabla 6.34.

Representación	Cruza	Mutación	Soluciones Óptimas por Población				
			100	500	1000	1500	2000
Binaria	Dos Puntos	Aleatoria	10 %	0 %	0 %	30 %	20 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Un Punto	Aleatoria	10 %	0 %	10 %	10 %	20 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Uniforme	Aleatoria	10 %	0 %	0 %	0 %	0 %
		Uniforme	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	0 %	10 %	10 %	0 %	10 %
		Uniforme	30 %	0 %	0 %	0 %	0 %
Entera	Dos Puntos	Aleatoria	0 %	10 %	0 %	0 %	0 %
	Un Punto	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Uniforme	Aleatoria	0 %	0 %	0 %	0 %	0 %
	Múltiples Puntos	Aleatoria	10 %	0 %	0 %	0 %	0 %

Tabla 6.34: Porcentaje de soluciones óptimas encontradas con una $P_m = \frac{1}{L}$.

Como se puede observar, el número de soluciones óptimas es menor a la revisada anteriormente, aunque la mutación aleatoria se ve favorecida por esta P_m para generar soluciones factibles, no ocurre lo mismo con las soluciones óptimas, ya que como es notorio en la tabla, sólo la cruza de múltiples puntos y la de un punto generan soluciones de una manera constante.

También es notoria la poca productividad de este tipo de soluciones cuando se utiliza la representación entera, ya que sólo la cruza de dos puntos y la de múltiples puntos logran devolver soluciones factibles. Lo que llama la atención es que las soluciones óptimas devueltas por cada una de estas cruzas se generaron únicamente con una población de 100 y 500 individuos.

En la tabla 6.35 se presentan los resultados obtenidos por otras técnicas evolutivas, y es posible observar, que aunque el problema tiene mayor complejidad (tiene una función más) estas técnicas han devuelto el número mínimo de compuertas que conforman este diseño.

Programa	**Expresión Obtenida**	**Compuertas Usadas**
PB [14]	$F1 = BD$ $F2 = BC \oplus AD$ $F3 = AC \oplus (ABCD)$ $F4 = ABCD$	5 ANDs, 2 XORs Total = 7
PG prefija [71]	$F1 = BD$ $F2 = BC \oplus AD$ $F3 = AC \oplus (ABCD)$ $F4 = ABCD$	5 ANDs, 2 XORs Total = 7
Colonia de hormigas [52]	$F1 = BD$ $F2 = BC \oplus AD$ $F3 = AC \oplus (ABCD)$ $F4 = ABCD$	5 ANDs, 2 XORs Total = 7
AG en Java	$F1 = BD$ $F2 = BC \oplus AD$ $F3 = AC \oplus (ABCD)$ $F4 = ABCD$	5 ANDs, 2 XORs Total = 7

Tabla 6.35: Comparación de la solución obtenida por el AG en Java con las de otros programas.

Todos lo programas de la tabla 6.35 encuentran la solución a este problema con las mismas compuertas, y lo que llama aún más la atención es que usen las mismas funciones booleanas, con lo que podemos concluir que los algoritmos evolutivos tienden a sesgar la búsqueda hacia esta solución (la cual podría ser única) en particular.

El diseño conseguido por el AG en Java lo muestra la figura 6.6. Para reproducirlo sólo se necesitan los siguientes datos:

- Cruza de un punto.
- Con mutación aleatoria.
- Tamaño de la matriz 5*5.
- Representación binaria.
- Con 2000 individuos.
- $P_m = \frac{1}{L}$.
- Semilla = 0.8189999999999998
- Con 200 generaciones.

Se eligió este resultado porque la solución se encuentra en un número pequeño de generaciones.

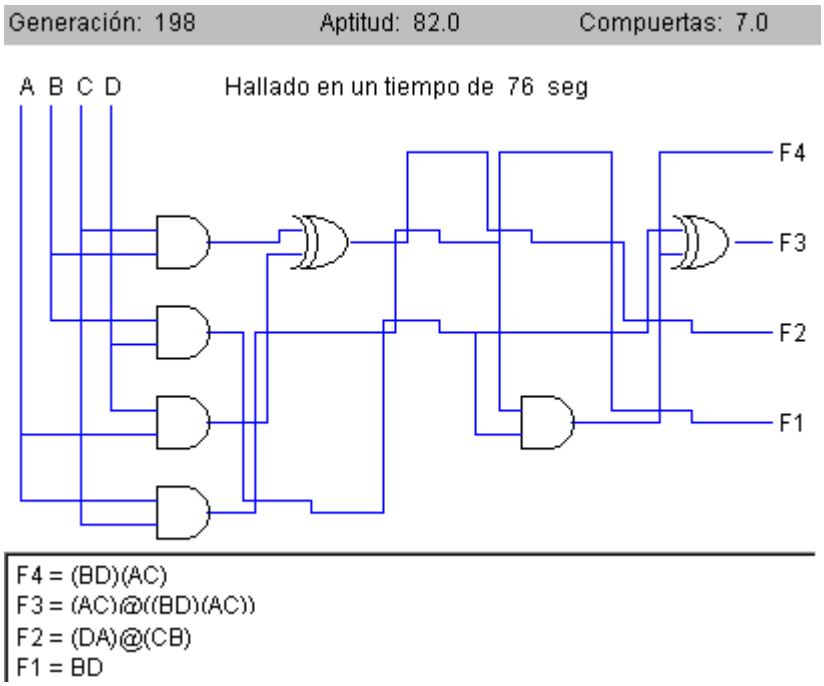


Figura 6.6: Diseño obtenido por el AG.

Capítulo 7

Conclusiones y trabajo futuro

Se cumplió con el objetivo de crear una herramienta académica, con la cual vía Internet se pueden manipular parámetros y operadores para visualizar el comportamiento del AG con la combinación de cada uno de ellos, y observar los diseños que éste encuentra. Se realizaron varias pruebas donde se demuestra que la implementación iguala e inclusive mejora algunos resultados obtenidos por otras técnicas evolutivas.

Se aportó en este trabajo de tesis un nuevo operador de cruza, al que se llamó múltiples puntos. El nuevo operador compite tanto en la producción de soluciones factibles como de soluciones óptimas contra los dos operadores más comunes (cruza de uno y dos puntos) utilizando los parámetros con los que mejor se desempeñan éstos, y en el ejemplo del comparador de dos bits, llega a obtener un diseño con menos compuertas que las técnicas evolutivas que se han utilizado para el diseño de circuitos lógicos.

Los resultados observados en las pruebas realizadas permiten concluir que la representación binaria trabaja mejor en combinación con la mutación uniforme.

La cruza uniforme da mejores resultados cuando se utilizan poblaciones de 100 individuos, lo cual podría ser ventajoso, ya que con poblaciones pequeñas el AG trabaja en un tiempo reducido. Por otro lado, si se desea tener una solución rápida (con un número menor de iteraciones), con poblaciones de entre 500 y 1500 individuos, se pueden obtener circuitos óptimos con un número pequeño de generaciones, haciendo uso de los otros operadores de cruza.

La cruza de dos, uno y múltiples puntos mejoran el desempeño del AG al incrementar el tamaño de población, pero no hay que olvidar que el tiempo invertido para obtener la solución llega a ser considerable.

También es posible concluir que la P_m aporta mejores resultados con un valor de $\frac{0.5}{L}$ (donde L es la longitud del cromosoma).

Se observa que el desempeño del AG en la búsqueda de soluciones factibles depende de la complejidad del problema a resolver. Es notorio que al incrementar el número de salidas disminuye el rendimiento del AG. Sin embargo, si llega a encontrar circuitos óptimos. El ejemplo de la sección 6.1.2 muestra que aunque el circuito es de una sola salida, su complejidad hace que el comportamiento del AG sea raquítico.

Aunque la representación binaria supera a la representación entera en cuanto al número de soluciones fáctibles y óptimas, la codificación entera le permite trabajar al AG tres veces más rápido que si se utilizara la binaria.

Como trabajo futuro se pretende insertar otro u otros operadores de selección y agregar otros operadores genéticos ya sea de cruza y/o mutación. Es necesario además, realizar un estudio respecto a los parámetros que permiten un mejor desempeño del nuevo operador de cruza, y también, realizar un análisis estadístico que permita visualizar las diferencias (sustanciales o no) entre las combinaciones de parámetros y de operadores; esto referente al AG. Con respecto al programa se podrían agregar las siguientes funciones: imprimir y guardar el diseño obtenido.

En algunas ocasiones el resultado que se muestra tiene compuertas que no se utilizan, compuertas que tienen el comportamiento de otra (por ejemplo, la NAND y la NOR que se utilice como NOT, o la AND y la OR que se utilice como WIRE) o que se tengan varios NOTs consecutivos, para evitar ejecutar el AG por más generaciones, se podría agregar una función que permita eliminar las compuertas no utilizadas (tanto las que no usa el diseño como los NOTs extras) y sustituya la compuerta con otro comportamiento por la adecuada.

Por la forma en que se implementó el programa, este tiene un límite visual para diseños que utilizen matrices mayores a 8*8 (este depende de la resolución que se utilice), se pretende implementar la manera de poder visualizar el CLC sin importar el tamaño de la matriz empleada (ni la resolución

que tenga el monitor).

La manera en que se inserta la tabla de verdad hasta el momento no permite que sean ingresados problemas con mas de seis variables de entrada, para modificar esto se requiere cambiar la interfaz actual por una que sea auxiliada por una barra de desplazamiento.

Apéndice A

Manual de usuario

Índice

Introducción

Bienvenido a *Diseñando el Circuito Lógico con el Algoritmo Genético*, este sistema le permitirá encontrar el diseño del circuito con un número mínimo de compuertas. Observar el comportamiento del algoritmo genético cuando cambie los parámetros de entrada (probabilidad de cruza, probabilidad de mutación, número de generaciones, entre otros). Así como también, probar algunos de los operadores de cruza y mutación más comunes.

Para trabajar con este software, no se requiere que lo instale en su máquina, solo necesita tener Internet para poder accesar a la página que contiene el programa.

Usted puede entrar a `http://www.utm.mx/~jcruz/Tesis.html`, donde podrá encontrar esta tesis en formato pdf y una liga al programa, la cuál lo llevará a un manual en formato html (por si necesita ayuda al momento de utilizarlo), esta página le indicará como usar el programa y le permitirá ejecutarlo. O si lo prefiere, la página `http://www.utm.mx/~jcruz/programa/programa.html` lo lleva directamente a *Diseñando el Circuito Lógico con el Algoritmo Genético*.

Diseñando un circuito

La mejor manera de aprender a usar un sistema es con ejemplos, así que trabajaremos con uno. Trataremos de encontrar el diseño de un circuito cuya tabla de verdad se muestra en la tabla 1.

A	**B**	**C**	**F1**
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Tabla 1: Tabla de verdad del circuito de ejemplo.

La tabla de verdad corresponde a un circuito de paridad impar, en este caso de tres variables de entrada.

Antes que nada tenemos que entrar al programa, así que lo haremos de la forma directa. En la parte de dirección de su navegador escriba `http://www.utm.mx/~jcruz/programa/programa.html`, como se señala en la figura 1, después de que oprima enter deberá aparecer la siguiente pantalla (figura 2).

Figura 1: Pantalla del navegador.

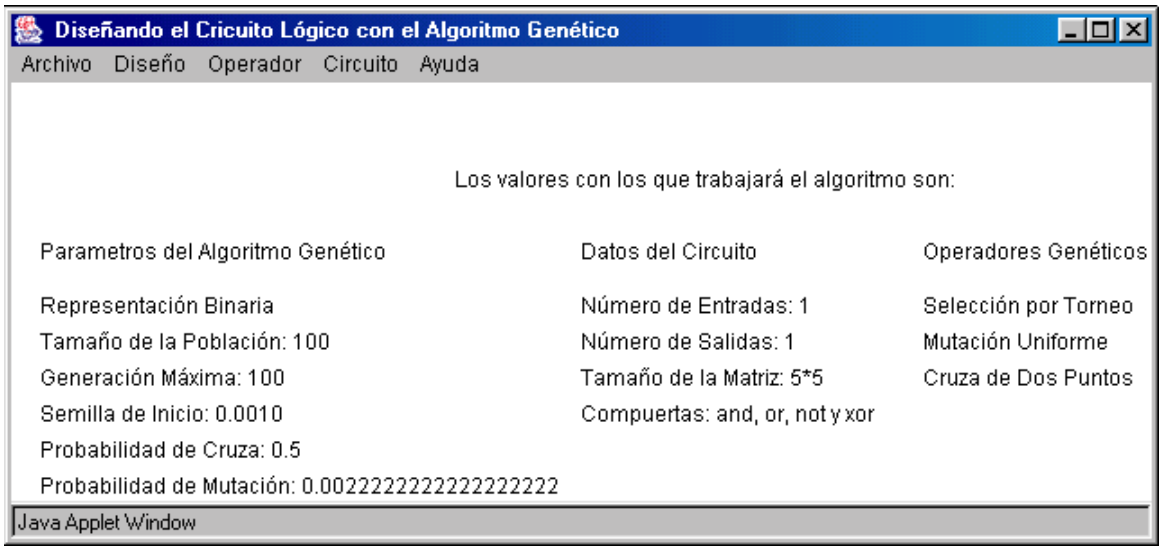


Figura 2: Pantalla principal del programa.

Lo primero que tiene que hacer es insertar el tamaño de la tabla de verdad que utilizará, para ello deberá elegir **Archivo** y seleccionar **Nuevo** (figura 3) y aparecerá la ventana que muestra la figura 4.

Figura 3: Opción para insertar un nuevo circuito.

Figura 4: Ventana para insertar el tamaño de la tabla de verdad.

Nuestro ejemplo requiere de tres entradas y una salida, así que seleccione de la lista de opciónes *Número de entradas*, el valor de tres y oprima aceptar.

Ahora necesitamos insertar los valores del circuito de ejemplo en la tabla de verdad. Primero tiene que seleccionar el menú **Diseño**, del cual escogerá la opción **Tabla de verdad** (figura 5), al hacerlo aparecerá la ventana que muestra la figura 6.

Cuando aparezca la tabla de verdad, ésta solo contendrá botones con 0s, así que para insertar los datos que desea, oprima el botón que requiera cambiarse al valor de 1, repita este paso las veces necesarias para tener la tabla de verdad correspondiente. Para nuestro ejemplo (tabla 1) este proceso será realizado cuatro veces, puesto que necesitamos cuatro 1s.

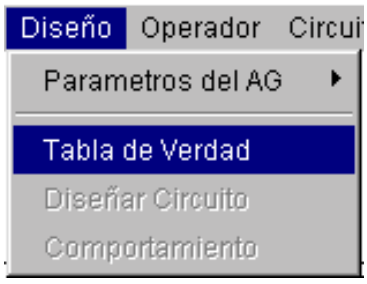


Figura 5: Opción para insertar los valores a la tabla de verdad.

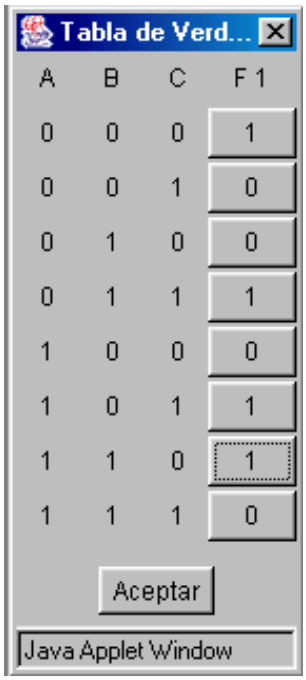


Figura 6: Tabla de verdad con los datos del ejemplo.

Una vez insertada la tabla de verdad, oprima aceptar para regresar a la pantalla principal.

El insertar la tabla de verdad solo lo necesitará hacer una vez, ya que si modifica los parámetros de entrada o los operadores, esto no afecta el contenido de la tabla. Pero si introduce otro tamaño de tabla, entonces necesitará repetir este proceso.

Lo siguiente que necesita hacer es ejecutar el algoritmo genético para diseñar el circuito lógico combinatorio, para ello vuelva a escoger la opción **Diseño**, pero en esta ocasión optará por la opción *Diseñar circuito* (como pudo notar, esta opción estaba deshabilitada antes de introducir la tabla de verdad, esto ocurrirá cada vez que inserte un nuevo circuito, el cual tenga un número de entradas o/y salidas diferentes al circuito con el que se encuentre trabajando) la figura 7 muestra la elección.

Figura 7: Opción para iniciar el diseño del circuito por medio del algoritmo genético.

Mientras la figura 8 expone la ventana donde se diseña el circuito.

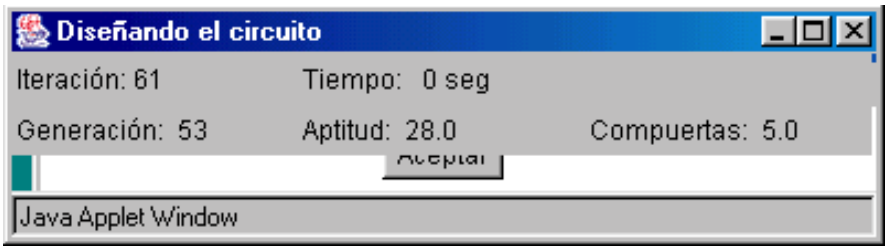


Figura 8: Ventana donde muestra que el programa se encuentra en ejecución.

Como podrá notar en esta ventana (figura 8) se muestra la iteración (ejecución o generación) en la que se encuentra, el tiempo que le lleva realizar las operaciones, y otros tres datos (*generación*, *aptitud* y *compuertas*), los cuales indican la *generación* en la que se encontró al mejor individuo, la *aptitud* que tiene y el número de *compuertas* con las que cuenta, cabe mencionar que este último dato solo aparece cuando se ha encontrado algún diseño valido, cuando esto no ocurre, es decir, cuando los valores no cumplen con los de la tabla de verdad, entonces, en lugar de *compuertas* dirá *violaciones*.

Al terminar de ejecutarse el programa aparecerá el diseño que se muestra en la figura 9.

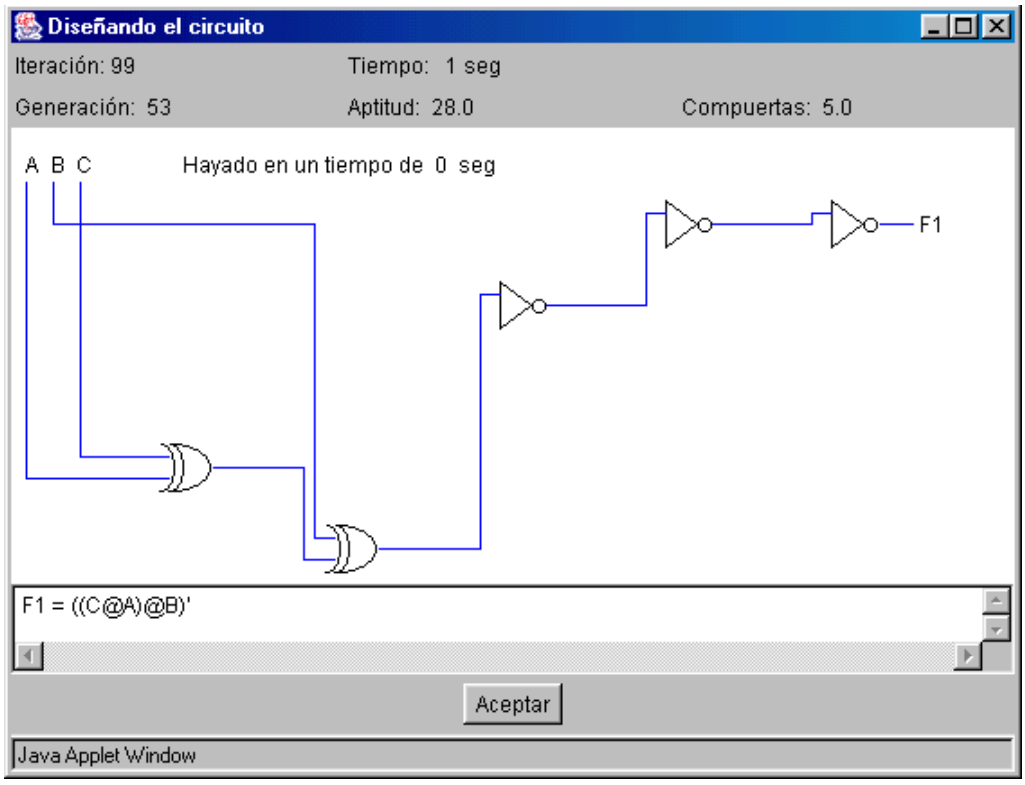


Figura 9: Diseño y función del circuito de ejemplo.

Como puede observar, en el diseño aparecen tres NOTS continuos, mientras que en la función solo aparece uno, lo que ocurre es que se aplica el teorema de involución (ver sección 2.1.3), esto aprovechando de que primero se obtiene el diseño y de éste se genera la función.

Nota: Si el programa termina de ejecutarse, y éste no encuentra ningún diseño, se lo indicará con el siguiente mensaje:

```
No se encontró una solución factible
```

Los pasos que se siguieron hasta el momento, son los mínimos que necesita realizar para comenzar a diseñar un circuito, esto se debe a que varios valores de los parámetros del algoritmo genético son manejados por omisión, así como

también los operadores genéticos con los que desarrollará. Los datos que se manejan por omisión son:

- Probabilidad de cruza = 0.5.
- Probabilidad de mutación = $\frac{0.5}{L}$.
- Tamaño de la población = 100.
- Número de generaciones = 100.
- Semilla inicial = 0.0010.
- Tamaño de la matriz 5*5.
- Compuertas a usar: AND, OR, NOT y XOR.
- Representación binaria.
- Cruza de dos puntos.
- Mutación uniforme.
- Selección por torneo.

Cuando cambie los valores y vuelva a ejecutar el programa, podrá darse cuenta que puede encontrar otros diseños, así como encontrar el circuito de tres compuertas como muestra la función del ejemplo que se desarrolló. La forma de modificar cada uno de estos valores serán explicados en las siguientes secciones, sólo que se hará de manera general, es decir, no se explicará con el ejemplo que se desarrolló en esta sección, el propósito de ello es que no olvide que los pasos a seguir para cambiar tanto los parámetros como los operadores son independientes del circuito que desee diseñar.

Modificar parámetros

Los parámetros que puede modificar son:

- El número de generaciones.
- El tamaño de la población.
- La semilla inicial.
- La probabilidad de cruza.
- Y la probabilidad de mutación.

Comenzaremos con los tres primeros. Para cambiarlos tiene que escoger el menú **Diseño** de la barra de menús de la pantalla principal, elegir **Párametros del AG** y seleccionar **Elegir valores** como se expone en la figura 10.

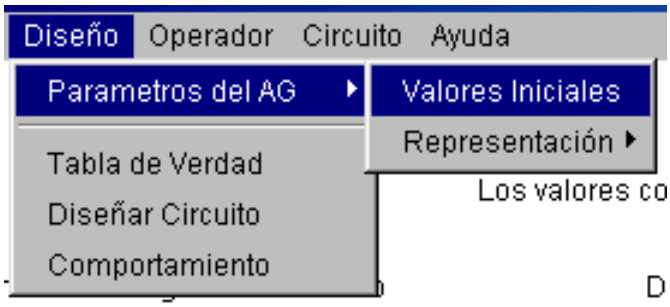


Figura 10: Opción para cambiar los valores por omisión.

Al momento de seleccionarla aparecerá la ventana que muestra la figura 11. En donde podrá modificar los valores que desee (número de generaciones, tamaño de la población y la semilla inicial). Para lograr esto tiene que mover la barra de desplazamiento, mientras lo hace, en el campo de edición que aparece al lado derecho de la barra, se le indicará el valor que se está introduciendo.

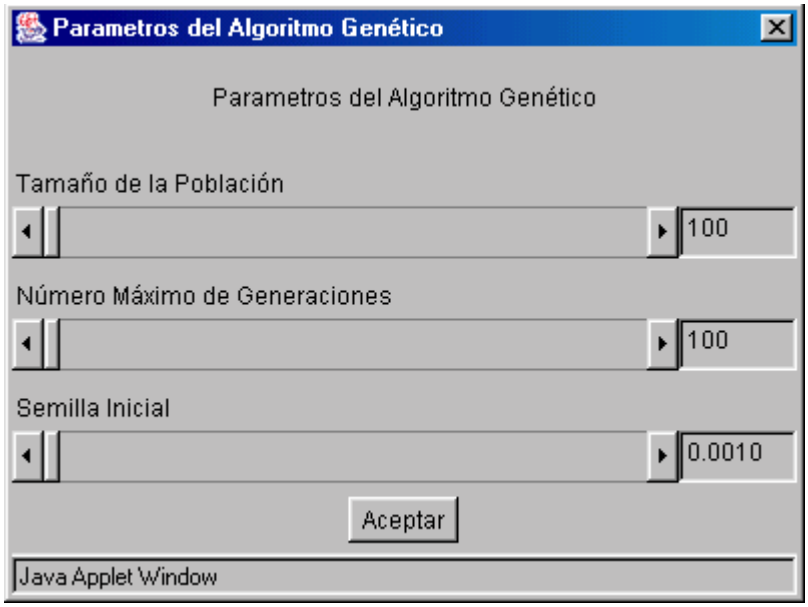


Figura 11: Pantalla donde se pueden cambiar los datos iniciales.

Solo puede introducir los datos moviendo la barra de desplazamiento para generaciones y población, pero podrá insertar por teclado la semilla con la

que desea trabajar, para ello solo seleccione el cuadro que aparece al lado derecho de la barra de desplazamiento de la semilla inicial, teclee el valor que desea y oprima enter (solo si oprime enter se insertará el valor), si inserta valores mayores que uno o caracteres que no sean números el programa insertará de nueva cuenta el último valor guardado.

Para retornar a la ventana principal tiene que oprimir el botón *Aceptar*.

En lo referente a los otros dos parámetros, lo que necesita hacer es elegir el menú **Operador**, donde seleccionará del menú **de Cruza** la opción **Probabilidad de Cruza** (figura 12), al hacerlo aparecerá la ventana que expone la figura 13.

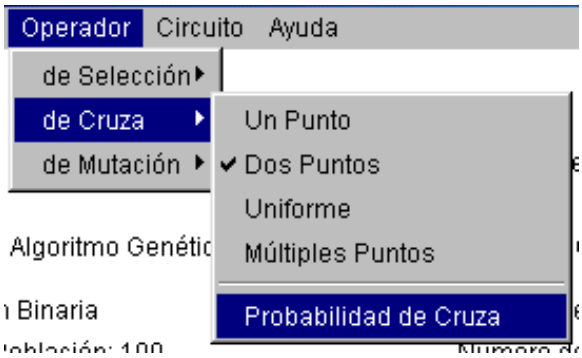


Figura 12: Opción para insertar la probabilidad de cruza.

La manera de insertar el valor con el que desea trabajar es únicamente por medio de la barra de desplazamiento, y para cerrar la ventana oprima el botón *Aceptar*.

Figura 13: Ventana para modificar el valor inicial de la probabilidad de cruza.

Para modificar la probabilidad de mutación tiene que elegir el menú **de Mutación**, y de éste, seleccionar **Probabilidad de mutación**, aparece el menú que muestra tres opciones como se presenta en la figura 14.

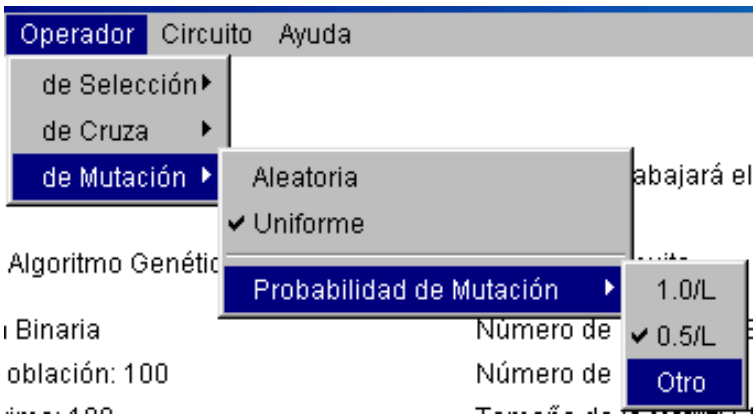


Figura 14: Opción para insertar la probabilidad de mutación.

Donde podrá elegir entre si utilizar una de las probabilidades que calcula el programa, o insertar otro valor. Esta última opción le mostrará una pantalla como se expone en la figura 15. La que le permitirá insertar una probabilidad de mutación a su gusto (de 0.001 a 0.01).

Al igual que en la probabilidad de cruza, el valor solo puede ser insertado por medio de la barra de desplazamiento, y para salir de esta ventana oprima el botón *Aceptar*.

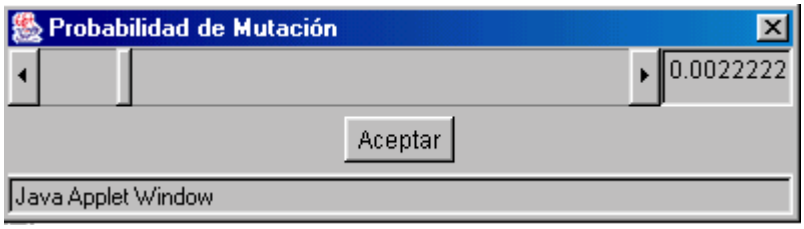


Figura 15: Opción para cambiar la probabilidad de mutación.

Usar otros operadores genéticos

Para modificar los operadores con los que desea trabajar tiene que seleccionar el menú **Operador**. En este aparecerán tres opciones.

Si desea cambiar el operador de mutación o el de cruza seleccione el menú **de Mutación** o **de Cruza** respectivamente. En seguida aparecerá una lista con los operadores disponibles según sea el caso.

Cambiar el tipo de representación

Para modificar la codificación con la que va a trabajar, lo único que tiene que hacer es elegir el menú **Diseño**, escoger **Parámetros del AG** y del menú **Representación** seleccionar la opción que desee, como en este caso comenzamos con la representación binaria cambiaremos a la entera como se observa en la figura 16.

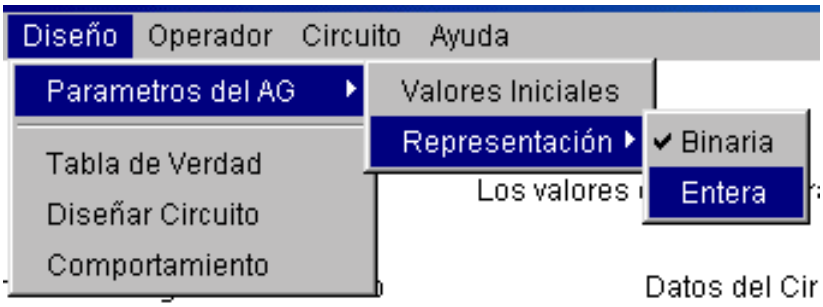


Figura 16: Eligiendo otro tipo de representación.

Trabajar con otras compuertas

Si desea utilizar otro tipo de compuertas, vaya al menú **Circuito** de la pantalla principal y de **Compuertas a usar** elija la opción con la que quiera probar el programa. La figura 17 muestra las alternativas posibles.

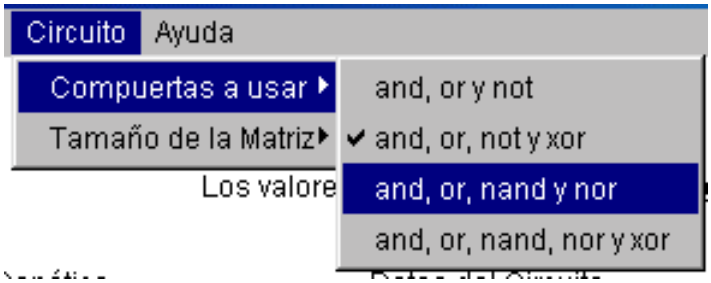


Figura 17: Seleccionando el tipo de compuertas con las que se trabajará.

Cambiar el tamaño de la matriz

Para modificar el tamaño de la matriz, necesita seleccionar la opción de la matriz con la que desea trabajar del menú **Tamaño de la matriz**, que se encuentra dentro de **Circuito** (figura 18).

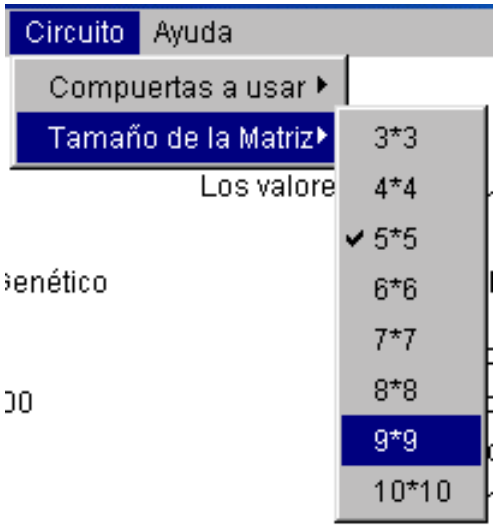


Figura 18: Estableciendo un nuevo tamaño de la matriz.

El tamaño de 3*3 hasta 8*8 es visible utilizando una resolución de 800 por 600 píxeles. Para poder ver los resultados en las matrices de 9*9 y 10*10 se le recomienda tener una resolución de 1024 por 768 píxeles.

Para la utilización del tamaño de la matriz se le recomienda que el número mínimo permitido sea de acuerdo al número máximo de entradas o de salidas.

Por ejemplo, si el problema tiene cinco entradas y una salida, cinco es el número máximo, así que el número mínimo del tamaño de la matriz debe ser 5*5, ahora que si su problema tiene cuatro entradas y ocho salidas, el número máximo es ocho, así que el número mínimo permitido es una matriz de 8*8. De no hacer esto, cuando usted trate de diseñar el circuito, el programa le mostrará el siguiente mensaje:

```
La matriz es insuficiente para el número de entradas
      y salidas del circuito que desea diseñar.
              Cambie la matriz a n*n
```

donde n es el tamaño recomendado para trabajar.

Símbolos para representar la función booleana

Para que pueda interpretar la función booleana, la tabla 2 le muestra que símbolo es utilizado para representar cada una de las compuertas.

Símbolo	**Compuerta**
+	OR
′	NOT
@	XOR
&	NAND
|	NOR

Tabla 2: Símbolos utilizados para representar a cada una de las compuertas de esta implementación.

Para la compuerta AND no se utilizó ningun símbolo.

Comportamiento del algoritmo genético

Otra opción que se tiene, es el poder ver la gráfica de comportamiento del algoritmo genético, en ella se observa el poder destructivo de las cruzas utilizadas.

Para visualizar la gráfica de comportamiento, debe elegir de la pantalla principal el menú **Diseño**, y seleccionar la opción **Comportamiento**, al hacerlo

aparecerá la ventana que muestra la gráfica del comportamiento del circuito que se haya diseñado la última vez.

La gráfica mostrará tres colores: azul, gris y rojo. El color azul indica el valor de la mejor aptitud por generación, es decir, muestra cuál es la aptitud mas alta encontrada en una determinada iteración. El color gris muesta la aptitud promedio por generación, esto es, representa el cálculo de la suma de todas las aptitudes de la población, entre el número de individuos en una iteración. Mientras que el rojo señala la aptitud mínima que se encuentra en un circuito factible.

Ayuda

Para la ayuda se encuentra este manual o el que se encuentra en la página `http://www.utm.mx/~jcruz/ayuda/manual.html`. Mientras que en el programa en el menú **Ayuda** encontrará la opción **Acerca de ...** (figura 19), donde al ser escogida aparecerá una ventana con los siguientes datos: nombre del programa, autor, fecha y versión (figura 20).

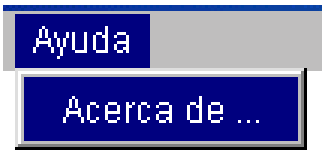


Figura 19: Opción para visualizar datos acerca del programa.

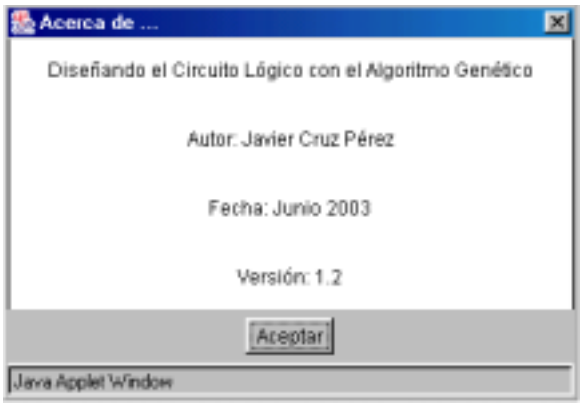


Figura 20: Ventana donde se muestran los datos del programa.

Contratiempos con el programa

Cuando intente ejecutar el programa puede ocurrirle lo siguiente:

- Que no se abra el applet, esto lo sabrá cuando aparezca el siguente mensaje:

  ```
  No puede trabajar con el programa porque su navegador no
  tiene instalado la máquina virtual de Java. Pero puede
  instalarla de manera gratuita, accesando desde aquí.
  ```

- Que abra el applet, sin embargo, no se visualiza la barra de menú.
- Que no aparezca el programa, ni el mensaje. Esto lo sabrá cuando en su navegador se muestre un rectángulo gris o un rectángulo blanco con un cuadrado en su esquina superior izquierda.

Esto sucede cuando su computadora no tiene instalado Java, o la versión que tiene no puede visualizar applets. Para solucionar ésto, solo requiere instalar la máquina virtual de Java, y para lograrlo existen dos formas, la primera es utilizar las ligas que permiten accesar a la página para actualizar el programa, y éstas se encuentran en la página del manual en la sección de *Contratiempos con el programa*, y la segunda es visitar la página `http://java.sun.com/getjava/`. Ambas opciones le permiten actualizar su software de Java de manera gratuita.

Apéndice B

Pseudocódigo

1 Inicio
2 **Si** cruzar verdadero **entonces**
2.1 El primer punto de cruza = *aleatorio*(0, límite superior)
2.2 **Si** El primer punto de cruza == 0 **entonces**
2.2.1 El segundo punto de cruza = *aleatorio*(1, límite superior)
2.3 **De lo contrario**
2.3.1 El segundo punto de cruza = *aleatorio*(0, límite superior)
2.4 temporal = longitud del cromosoma
2.5 c = al modulo de la división de la longitud del cromosoma entre la suma de los puntos de cruza
2.6 **Mientras** temporal > (0+c)
2.7 Los genes desde temporal hasta el primer punto de cruza se *copian* del *padre 1 al hijo 1 y del padre 2 al hijo 2*
2.8 A temporal se le resta el primer punto de cruza
2.9 Los genes desde temporal hasta el segundo punto de cruza se *copian* del *padre 1 al hijo 2 y del padre 2 al hijo 1*
2.10 A temporal se le resta el segundo punto de cruza
2.11 **Fin del ciclo mientras**
2.12 **Si** temporal > 0 **entonces**
2.12.1 Se *copian* los últimos genes del *padre 1 al hijo 1 y del padre 2 al hijo 2*
3 **De lo contrario**
3.1 Se dejan intactos los cromosomas, los hijos son una copia exacta de los padres.
4 Fin

El algoritmo inicia preguntando si se va cruzar (paso 2) o no (paso 3), este paso es, al igual que en la cruza de un punto y de dos puntos, el que permite dejar intactos algunos individuos. Este nuevo operador solo ha sido probado con una $P_c = 0.5$ obteniendo buenos resultados. Sin embargo, se podría hacer un estudio empírico para determinar con que P_c se favorece o se empeora el desempeño del AG.

En el punto 2.1 se busca el primer punto de cruza, *aleatorio* es una función que devuelve un número aleatorio entre un rango dado, en este caso se utiliza 0 como límite inferior, mientras que el límite superior[1] utilizado en esta implementación fue el número de renglones de la matriz que haya introducido el usuario. Este valor es otra variable que puede influir en este nuevo operador de cruza, lo que motiva a realizar experimentos con el propósito de determinar cual sería el límite superior idóneo, con el cual el AG se desempeña mejor.

En el paso 2.2 se pregunta si el punto de cruza es 0, de serlo se modifica el rango donde se buscará el siguiente punto de cruza, ahora será de 1 al límite superior, de no ser así (paso 2.3) el segundo punto de cruza puede ser 0.

Se entrega el valor de la longitud del cromosoma a una variable temporal, para evitar perder su valor (paso 2.4).

Se utiliza una variable para tener el módulo de la división realizada entre la longitud del cromosoma y la suma de los dos puntos de cruza (por eso se cuida de que solo un punto de cruza pueda ser 0, ya que no es posible realizar la división entre 0), esto para determinar la cantidad de genes que faltará copiar a los hijos (paso 2.5).

Se inicia el ciclo para copiar la información genética de los padres a los hijos (paso 2.6), y este no parará hasta que la varible temporal sea menor que el módulo.

En el punto 2.7 y 2.8 se copian los genes de los padres a los hijos, y después de ejecutar cada uno de estos pasos, se resta a la variable temporal los puntos de cruza, esto se hace para ir avanzando en el cromosoma (pasos 2.8 y 2.10).

[1]El límite superior se puede fijar de dos maneras, una por medio del programador (esto es insertando un número fijo por omisión), o el usuario puede insertarlo.

Al terminar el ciclo se pregunta si la variable temporal es mayor que 0, esto se debe a que si es mayor que 0 significa que faltan genes por copiar (paso 2.12), de lo contrario no se necesita hacer nada más y termina el algoritmo.

Apéndice C

Contenido del CD-ROM

El CD-ROM contiene la tesis en formato PDF y varias carpetas cuyo contenido se describe a continuación.

En la carpeta *codigofuente* encontrará una manual técnico en formato PDF y el código fuente para que pueda manipularlo y realizar el trabajo futuro que se planteo en el capítulo 7.

El formato de la página html se encuentra en la carpeta *Paginashtml*, para activarla necesita ejecutar el archivo *Tesis.html*, el cual tiene la liga al programa para poderlo ejecutar si no dispone de Internet.

Mientras que las carpetas *CIDCSVER2003* y *COMCEV2003* tienen las memorias de los siguientes congresos: *Conferencia Internacional de Dispositivos, Circuitos y Sistemas Veracruz 2003* y *Congreso Mexicano de Computación Evolutiva Guanajuato 2003* respectivamente. Eventos donde se expusieron algunos resultados obtenidos en este trabajo de investigación [20, 21]. Para visualizarlos sólo se necesita ejecutar el archivo *index.html* que se encuentra en cada una de las carpetas.

Bibliografía

[1] Daniel Amor. *La Revolución E-business: Claves para vivir y trabajar en un mundo interconectado.* Prentice Hall, Argentina, 2000.

[2] David H. Ackley. *A Connectionist Machine for Genetic Hillclimbing.* Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[3] Thomas Bäck. Optimal mutation rates in genetic search. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp 2–8, San Mateo, California, USA, July 1993. Morgan Kaufmann Publishers.

[4] Wolfang Banzhaf, Peter Nordin, Robert E. Keller and Frank D. Francone. Genetic programming an introduction. In *On the Automatic Evolution of Computer Programs and Its Applications*, San Francisco, California, USA, 1998. Morgan Kaufmann Publishers.

[5] Thomas C. Bartee. *Fundamentos de Computadores Digitales.* Mc Graw Hill, México, quinta edición, 1984. Primera edición en español.

[6] George Boole. *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probability.* Dover Publisher, New York, 1954.

[7] Hans J. Bremermann. The evolution of intelligence. Technical Report 17, Departmente of Mathematics, University of Washington, Seattle, USA, July 1958.

[8] Bill P. Buckles and Fred E. Petry, editors. *Genetic Algorithms.* IEEE Computer Society Press, 1992.

[9] W. D. Cannon. *The Wisdom of the Body.* Norton & Company, New York, 1932.

[10] Carlos A. Coello Coello. Introducción a los algoritmos genéticos. *Soluciones Avanzadas*, Año 3(17):5–11, 1995.

[11] Carlos A. Coello Coello, Alan D. Christiansen and Arturo Hernández Aguirre. Using genetic algorithms to design combinational logic circuits. In C. L. Philip Chen Benito R. Fernandez Cihan H. Dagli, Metin Akay and Joydeep Ghosh, editors, *Smart Engineering Systems: Neural Networks, Fuzzy Logic and Evolutionary Programming*, volume 6, pp 391–396. Intelligent Engineering through Artificial Neural Networks, November 1996.

[12] Carlos A. Coello Coello. La importancia de la representación en los algoritmos genéticos (parte I). *Soluciones Avanzadas*, Año 7(69):50–56, Mayo 1999.

[13] Carlos A. Coello Coello, Alan D. Christiansen y Arturo Hernández Aguirre. Diseño Óptimo de circuitos lógicos usando algoritmos genéticos. En *Primer Encuentro de Computación, Taller de Aprendizaje*, pp 1–10, Querétaro, Querétaro, México, Septiembre 1997. Sociedad Mexicana de Ciencia de la Computación.

[14] Carlos A. Coello Coello, Alan D. Christiansen and Arturo Hernández Aguirre. Use of evolutionary techniques to automate the design of combinational circuits. In *International Journal of Smart Engineering System Design*, pp 299–314, Malaysia, June 2000. Overseas Publishers Association.

[15] Carlos A. Coello Coello, Arturo Hernández Aguirre and Bill P. Buckles. Evolutionary multiobjective design of combinational logic circuits. In Didier Keymeulen Jason Lohn, Adrian Stoica and Silvano Colombano, editors, *Proceedings of the Second NASA/Dod Workshop on Evolvable Hardware*, pp 161–170, Los Alamitos, California, USA, July 2000. IEEE Computer Society.

[16] Carlos A. Coello Coello, Zavala G. Rosa Laura, Benito Mendoza G. and Arturo Hernández Aguirre. Ant colony system for the design of combinational logic circuits. In Peter Thomson Julian Miller, Adrian Thompson and Terence C. Fogarty, editors, *Evolvable Systems: From Biology to Hardware*, pp 21–30, Edinburgh, Scotland, April 2000.

[17] Carlos A. Coello Coello, Zavala G. Rosa Laura, Benito Mendoza G. and Arturo Hernández Aguirre. Automated design of combinational logic circuits using the ant system. Technical report, Laboratorio Nacional de Informática Avanzada, Xalapa, Veracruz, México, 2000. (Artículo publicado en la revista "Engineering Optimization").

[18] Carlos A. Coello Coello. *Introducción a la Computación Evolutiva*. Departamento de Ingeniería Eléctrica, Sección de Computación del CINVESTAV–IPN, D.F. México, Mayo 2001. Notas de Curso.

[19] Nareli Cruz Cortés. Uso de emulaciones del sistema inmune para manejo de restricciones en algoritmos evolutivos. Tesis de maestría, maestría en inteligencia artificial, UV/LANIA, Xalapa, Veracruz, México, Diciembre 2000.

[20] Javier Cruz Pérez, Hilda Caballero Barbosa y Carlos A. Coello Coello. Plataforma en Java para el diseño de circuitos lógicos combinatorios, experimentando con diferentes operadores genéticos. En *Congreso Mexicano de Computación Evolutiva (COMCEV'03)*, pp 119–129. Guanajuato, Guanajuato, México, Mayo 2003. CIMAT.

[21] Javier Cruz Pérez, Hilda Caballero Barbosa y Carlos A. Coello Coello. Diseño de circuitos lógicos con algoritmos genéticos en java. En *Conferencia Internacional de Dispositivos, Circuitos y Sistemas Veracruz 2003 (CIDCSVER'03)*. Veracruz, Veracruz, México, Junio 2003. Disponible únicamente en CD.

[22] Charles Darwin. *The Origin of Species by Means of Natural Selection or the Preservation of Favored Races in the Struggle for life*. Random House, New York, 1993. Publicado originalmente en 1859.

[23] Harvey M. Deitel. *Como programar en Java*. McGraw Hill, México, 2000.

[24] Carlos Alberto Fernández y Fernández. *Apuntes: Programación Orientada a Objetos con C++ y Java*. Universidad Tecnológica de la Mixteca, Huajuapan de León, Oaxaca, México, 2001.

[25] David B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on neural networks*, 5(1):3–14, January 1994.

[26] David B. Fogel. *Evolutionary Computation. Toward a New Philosophy of Machine Intelligence*. The Institute of Electrical and Electronic Engineers, New York, 1995.

[27] Lawrence J.Fogel. *On the organization of intellect*. PhD thesis, University of California, Los Angeles, California, 1964.

[28] Lawrence J. Fogel. *Artificial Intelligence Through Simulated Evolution. Forty years of Evolutionary Programming*. John Wiley & Sons, New York, 1999.

[29] Carlos M. Fonseca and Peter J. Fleming. Genetic algorithms for multiobjective optimization: formulation, discussion and generalization. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp 416–423, San Mateo, California, 1993. University of Illinois at Urbana-Champaign, Morgan Kauffman Publishers.

[30] George J. Friedman. Selective feedbackcomputers for engineering synthesis and nervous system analogy. Master's thesis, University of California, Los Angeles, USA, February 1956.

[31] Agustín Froufe Quintas. *Java 2: Manual de usuario y tutorial*. Alfaomega Ra-Ma, México, segunda edición, 2000.

[32] David E. Goldberg and Kalyanmoy Deb. A comparison of selection schemes used in genetic algorithms. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pp 69–93, San Mateo, California, USA, 1989. Morgan Kaufmann.

[33] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley Longman Inc., Massachusetts, 1989.

[34] John J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 16(1):122–128, Janury/February 1986.

[35] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, Massachusetts, second edition, 1992.

[36] Hitoshi Iba, Masaya Iwata and Tetsuya Higuchi. Gate-level evolvable hardware: Empirical study and application. In Dipankar Dasgupta and Zbigniew Michalewicz, editors, *Evolutionary Algorithms in Engineering Aplications*, pp 260–275, Springer–Verlag, Berlín, Alemania, 1997.

[37] Eduardo Islas Pérez. Development of a learning plataform using case–based reasoning and genetic algorithms. case study: Optimization of combinational logic circuits. Tesis de maestría, maestría en inteligencia artificial, UV/LANIA, Xalapa, Veracruz, México, Noviembre 2000.

[38] Christian Jacob. Illustrating evolutionary computation with mathematica. Technical report, University of Calgary, 2001.

[39] A. K. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptative Systems*. PhD thesis, University of Michigan, Michigan, 1975.

[40] Tatiana G. Kalganova. *Evolvable Hardware Design of Combinational Logic Circuits*. PhD thesis, Napier University, Edinburgh, Scotland, 2000.

[41] Miguel Katrib Mora. Java más allá de la farándula. *Soluciones Avanzadas*, pp 34–43, 1996.

[42] John R. Koza. *Genetic Programing. On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992.

[43] John R. Koza. *Genetic Programing II. Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, Massachusetts, 1998.

[44] John R. Koza. *Genetic Programing III. Darwinian Invention and Problem Solving*. Morgan Kaufmann Publishers, San Francisco, California, USA, 1999.

[45] Jean Baptiste Lamarck. Zoological philosophy: An exposition with regard to the natural history of animals. Technical report, Chicago University, Chicago, USA, 1984. Publicado originalmente en 1809 en francés, como Philosophie Zoologique.

[46] Laura Lemay y Charles L. Perkins. *Aprendiento Java en 21 días*. Prentice Hall, México, 1996.

[47] Seymour Lipschutz. *Matemáticas para computación*. Schaum. McGraw Hill, México, Marzo 1986.

[48] Ignacio Gabriel López Licona. Obtención de la forma de una superficie esférica a partir de radios de curvatura locales utilizando algoritmos genéticos. Tesis de ingeniería, ingeniería en computación, Universidad Tecnológica de la Mixteca, Huajuapan de León, Oaxaca, México, Febrero 2000.

[49] M. Morris Mano. *Diseño Digital*. Prentice Hall, México, 1987.

[50] Thomas Richard McCalla. *Digital Logic and Computer Design*. Maxwell Macmillan International Editions, Singapur, 1992.

[51] Gregor Johann Mendel. *Experiments in plant hybridisation*. Journal of the Royal Horticultural Society, 1901. Traducción al inglés de un artículo publicado originalmente en 1865.

[52] Benito Mendoza García. Uso del sistema de la colonia de hormigas para optimizar circuitos lógicos combinatorios. Tesis de maestría, maestría en inteligencia artificial, UV/LANIA, Xalapa, Veracruz, México, Mayo 2001.

[53] Efrén Mezura Montes. Uso de la técnica multiobjetivo npga para el manejo de restricciones en algoritmos genéticos. Tesis de maestría, maestría en inteligencia artificial, UV/LANIA, Xalapa, Veracruz, México, Septiembre 2001.

[54] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer–Verlag, New York, USA, third edition, 1996.

[55] Julian F. Miller, Dominic Job and Vesselin K. Vassilev. Principles in the evolutionary design of digital circuits–part i. *Genetic Programing and Evolvable Machines*, 1(1/2):7–35, April 2000.

[56] Victor P. Nelson, H. Troy Nagle, Bill D. Carroll y J. David Irwin . *Análisis y Diseño de Circuitos Lógicos Digitales*. Prentice Hall, México, 1996.

[57] Carla Leninca Pacheco Agüero. Distribución óptima de horarios de clases utilizando la técnica de algoritmos genéticos. Tesis de ingeniería,

ingeniería en computación, Universidad Tecnológica de la Mixteca, Huajuapan de León, Oaxaca, México, Agosto 2000.

[58] Jimmy Josué Peña Koo. Desarrollo de un applet en java del micro algoritmo genético usando optimización multiobjetivo. Tesis de licenciatura, licenciatura en ciencias computacionales, Universidad Autónoma de Yucatán, Mérida, Yucatán, Mayo 2002.

[59] Ingo Rechenberg. *Evolutions Strategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, Stuttgart, Alemania, 1973.

[60] María Margarita Reyes Sierra. Estudio de algunos aspectos teóricos de los algoritmos genéticos. Tesis de maestría, maestría en inteligencia artificial, UV/LANIA, Xalapa, Veracruz, México, Octubre 2002.

[61] Katya Rodríguez Vázquez. *Multiobjective Evolutionary Algorithms in Non–linear System Identification*. PhD thesis, University of Sheffield, Sheffield, UK, 1999.

[62] S. Ronald. Robust encodings in genetic algorithms. In D. Dasgrupta & Z. Michalewicz, editor, *Evolutionary Algorithms in Engineering Applications*, pp 30–44. Springer-Verlag, Berlín, Alemania, 1997.

[63] Günter Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Networks*, 5:96–110, January 1994.

[64] Constantino Sánchez Ballesteros. Java: Aprende a programar. *Sólo Programadores*, Año VIII(90):3, 2002.

[65] Tsutomu Sasao. *Logic Synthesis and Optimization*. Kluwer Academic Press, 1993.

[66] J. David Schaffer, Larry J. Eshelman, Richard A. Caruana and Raharshi Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pp 51–60, San Mateo, California, USA, 1989.

[67] Herbert Schildt. *Java 2: Manual de referencia*. McGraw Hill, España, cuarta edición, 2001.

[68] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutions Strategie*. Birkhäuser, Basel, Alemania, 1977.

[69] Hans-Paul Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, Wiley, Chichester, UK, 1981.

[70] Hans-Paul Schwefel. *Evolution and Optimization Seeking*. John Wiley & Sons, New York, 1995.

[71] Eduardo Serna Pérez. Diseño de circuitos lógicos combinatorios utilizando programación genética. Tesis de maestría, maestría en inteligencia artificial, UV/LANIA, Xalapa, Veracruz, México, ENERO 2001.

[72] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the AIEE*, 57:713–723, 1938.

[73] Mizrahi Sullivan. *Matemáticas Finitas. Con Aplicaciones a la Administración y Economía*. Limusa, México, 2001.

[74] Gilbert Syswerda. Uniform crossover in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pp 2–9, San Mateo, California, 1989. George Mason University, Morgan Kaufmann Publishers.

[75] Ronald J. Tocci. *Sistemas Digitales principios y aplicaciones*. Pearson Educación, México, sexta edición, 2000.

[76] Roger L. Tokheim. *Principios Digitales*. Mc Graw Hill, Madrid, España, tercera edición, 1995.

[77] Gregorio Toscano Pulido. Optimización multiobjetivo usando un micro algoritmo genético. Tesis de maestría, maestría en inteligencia artificial, UV/LANIA, Xalapa, Veracruz, México, Septiembre 2001.

[78] B. C. H. Turton. Extending Quine-McCluskey for Exclusive-Or logic synthesis. *IEEE Transactions on Education*, 39(1):81–85, 1996.

[79] August Weismann, editor. *The Germ Plasm: A Theory of Heredity*. Scott, London, UK, 1893.

[80] A. Wetzel. Evaluation of efectiveness of genetic algorithms in combinational optimization. No Publicado, 1983.

[81] Darrel Whitley. The GENITOR algorithm and selection pressure: why rank–based allocation of reproductive trials is best. In J. David Schaffer, editor, *Proceedings of the Third Conference on Genetic Algorithms*, pp 116–121, San Mateo, California, USA, July 1989. George Mason University, Morgan Kaufmann Publishers.

[82] Michael Keppler.

`http://maui.theoinf.tu-ilmenau.de/~sane/projekte/karnaugh/embed_karnaugh.html`

, June 1997. This applet was developed for the seminar "Integrated hardware and Software systems". Última visita el 5 de mayo del 2003.